5-2024

# Enhancing security and usability in password-based web systems through standardized authentication interactions

Anuj Gautam
*University of Tennessee, Knoxville*, agautam1@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Anuj Gautam entitled "Enhancing security and usability in password-based web systems through standardized authentication interactions." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Scott Ruoti, Major Professor

We have read this dissertation and recommend its acceptance:

Adam Aviv, Kent Seamons, Jinyuan Sun

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

# Improving Security and Usability in Password-based Authentication Systems using Standardized Interactions

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Anuj Gautam

May 2024

*I would like to dedicate this work to my wife and my family, whose unwavering love and support have been a constant source of encouragement throughout the process.*

# Acknowledgments

I would like to express my gratitude to my advisor, Dr. Scott Ruoti, for his guidance and support. This work would not have been possible without him. I would like to express my gratitude to my dissertation committee members: Dr. Adam Aviv, Dr. Kent Seamons, and Dr. Jinyuan Sun, for their valuable feedback and guidance. I would also like to acknowledge the invaluable support of my collaborators, Sean Oesch and Tarun Kumar Yadav, on my papers. Furthermore, I would like to appreciate my labmates for the enriching experiences we shared in the lab. I am grateful to my friends in Knoxville for the enjoyable times we had together. Lastly, I extend my heartfelt thanks to my wife for her unwavering support during the long hours of my work, and to my family for their constant encouragement.

# Abstract

Password-based authentication is the predominant method for securing access on the web, yet it is fraught with challenges due to the web's lack of inherent design for authentication. Password managers have emerged as auxiliary tools to assist users in generating, storing, and inputting passwords more securely and efficiently. But both the browser and the server are oblivious of the password manager's presence, leading to usability and security issues. However, because the web wasn't originally built to accommodate password-based authentication, password managers serve as a temporary fix and encounter several usability and security problems that limit their widespread use. This dissertation proposes a novel approach to enhance the usability and security of password-based authentication by integrating authentication as a core component of the web infrastructure, through the introduction of standardized interfaces for the interaction among browsers, password managers, and websites.

To achieve this, the dissertation introduces four implementations as an exploration: (1) the development of a Password Composition Policy (PCP) language designed to standardize and enhance password generation processes; (2) the creation of a Secure Browser Channel (SBC) aimed at bolstering security of passwords against prevalent web threats such as cross-site scripting (XSS) attacks and malicious browser extensions; (3) implementing the concept of SBC in FIDO2 passwordless authentication to show that the concept is important to more than just passwords; and (4) the application of SBC in different context than credential entry – the detection and auditing of browser-based attacks. We implemented and performed real-world evaluations, demonstrating their practical viability and effectiveness in improving web authentication. The dissertation concludes with reflections on the lessons learned from these implementations and outlines future research directions that could further

cement authentication as an integral, first-class component of the web, thereby substantially improving the security and usability landscape of web authentication.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Authentication is a critical aspect of software security, especially on the web, where users must prove their identity to access personal accounts. This necessity arises from the web's inherently complex and hostile environment, contrasting with local systems where physical access may imply access to the system. Despite the ubiquity of password-based authentication as a primary security measure, it is fraught with issues [21], such as vulnerability to theft and brute-force attacks—problems exacerbated by the common practice of password reuse. Furthermore, creating and managing strong, unique passwords for each account poses a significant hassle for users.

Password managers offer a solution to some of these challenges by enabling users to generate, store, and autofill strong, unique passwords. ' However, they are not without their own set of problems. Usability issues can frustrate users, leading to the underutilization of available features [139, 108, 144, 151, 6, 14], while some security vulnerabilities of passwords still persist. For example, inserted passwords are susceptible to cross-site scripting (XSS) and phishing attacks. This situation highlights the need for improvements in authentication methods to effectively balance the competing priorities of convenience and security.

Password managers currently serve as a stopgap solution to the web's authentication challenges. They are implemented in two main forms i) standalone applications, where users must juggle interactions between the browser and the password manager (such as manually copying and pasting credentials), or ii) browser extensions, which offer a more integrated experience by autofilling credentials directly. Despite the convenience of browser extensions,

their interactions are limited to the browser's DOM—home to many security vulnerabilities—and operate without the browser's active recognition of their presence. Moreover, website servers remain indifferent to whether a password manager is being used. This situation presents an opportunity to enhance both the security and usability of password management. By encouraging standardized interactions among browsers, servers, and password managers, there's a chance to significantly improve the authentication landscape, making it safer and more user-friendly for everyone involved.

In this dissertation, we explore the enhancement of security and usability in password-based authentication by providing standardized interactions between the web entities, with password manager acting as an intermediary. We start by suggesting a standardized way for websites to tell password managers exactly what kind of passwords they need, using something we call a Password Composition Policy (PCP) language. This helps password managers create passwords that fit each website's rules. Furthermore, we explore vulnerabilities in the credential entry process, particularly the risks associated with credentials entered into the browser, such as exposure to cross-site scripting (XSS) attacks and malicious browser extensions. To counteract these threats, we develop a Secure Browser Channel (SBC) for the safe entry of credentials. We implement a proof of concept secure browser channel for passwords as well as passwordless(FIDO2) systems. Finally, we utilize the secure browser channel to implement a system that detects and audits credential attacks in the browser, thus enhancing the protective measures against potential security breaches. Through this research, we aim to substantially improve the security and usability of web authentication, advocating for a more harmonized interaction among browsers, servers, and password managers, ultimately benefiting users and web services alike.

Leveraging existing research that highlights user challenges with password managers due to inconsistent website requirements, in Chapter 3 we introduce a Password Composition Policy (PCP) language. We developed this language after analyzing password policies from 270 websites. It enables the creation of passwords that meet various website standards and incorporate user preferences, improving both security and usability. Our proof of concept demonstrates its practicality through website and password manager implementations, showing how it simplifies compliant password generation with minimal changes needed by websites

and password managers. A study with 25 participants proves the language's ease of use for even novice developers. Additionally, our analysis uncovers that many current web PCPs fail to protect against offline attacks due to user preferences for certain character classes.

In Chapter 4, we transition from addressing usability issues in password generation to enhancing the security of credential entry into the browser. Recognizing vulnerabilities identified in past research [160], particularly those exploiting scripts' access to the browser's Document Object Model (DOM), we introduce a secure browser channel(SBC) that allows to securely insert credentials into the browser. For this, we introduce read-only API that allows password managers to securely input credentials into the browser. It operates by initially entering a dummy password into the DOM, which is subsequently replaced with the actual password through the secure channel, effectively blocking scripts and extensions from accessing the DOM during this critical exchange. This mechanism, designed to be compatible with existing password managers with minimal changes and without necessitating website modifications, significantly bolsters autofill security against threats like XSS attacks and malicious extensions. Our evaluation shows its effectiveness, enhancing security for 97% of the Alexa top 1000 websites without affecting their operation, providing a strong defense against widespread browser-based security risks.

To demonstrate that the concept of secure browser channel extends beyond password-based systems, in Chapter 5 we explore its application to the passwordless FIDO2 system. While FIDO2 is robust against remote attacks, it remains susceptible to local threats. In this chapter, we highlight how browser extensions can potentially intercept and alter FIDO2 communications between the server and the device, compromising the credential registration process. To counter this, we develop a secure browser channel tailored for FIDO2. Through this channel, servers can transmit FIDO2 requests in the request header, allowing the browser to safeguard the request until it's needed by the WebAuthn API. This method ensures that FIDO2 requests remain unaltered by malicious browser extensions, thereby securing the FIDO2 registration process against local attacks.

Finally, we take the secure browser channel's capabilities even further by exploring how it can do more than just secure credential entry. In Chapter 6, we introduce a system that detects credential theft and facilitates audit of the attacks on the browser, significantly

bolstering defenses against potential intrusions. This system utilizes the nonces from the secure browser channel to spot credential theft attempts. Additionally, it makes use of browser audit logs, allowing for a retrospective analysis of attacks through cooperation between users and servers. By boosting both the detection and auditing of browser-based attacks, this approach acts as a strong safeguard against security breaches. This approach has the dual purpose of detection and deterrence, enhancing the overall security of web authentication landscape.

In another sense, dissertation begins by addressing the usability of password generation through standardizing interactions between the password manager and the server in Chapter 3. We then move on to standardizing the interaction between the password manager and the browser to improve the security of credential entry in Chapters 4 and 5. Finally, we leverage the properties of secure browser channel – standardizing the interactions among the password manager, browser, and server – to introduce the ability to detect and audit attacks on credentials in the browser. Finally, with the enhanced understanding of the security and usability of password managers, we close with a discussion of key lessons learned and suggest directions for future research in Chapter 7.

# Chapter 2

# Background and Related Works[1]

This section provides the background information necessary to supplement the understanding of the work presented in this dissertation. We start with general authentication systems common in the web and the role of password managers in securing user credentials. We then discuss security issues associated with web authentication systems, such as phishing, cross-site scripting attacks, and malicious browser extensions. We conclude with a discussion of related works that have been conducted in improving the security and usability of password-based authentication systems on the web.

## 2.1 Authentication

Authentication is the process or method of verifying the identity of a user or system in order to allow them access to a resource, data or service. This process is important in ensuring that only authorized individuals or systems can access the restricted system. Authentication involves i) identification, where the user or system provides a unique identifier, ii) verification, where the system validates the identity of the user or system, and iii) authorization, where the system determines the level of access the user or system has. Authentication is a critical component of security systems, and it is used to protect sensitive information and resources from unauthorized access.

[1]This chapter incorporates content from previously published works or preprints related to Chapters 3, 4, 5, and 6. Please refer to these chapters for detailed references.

Authentication in computer systems usually utilizes a secret credential or identifier, only known to the user or system, to verify the identity of the user or system. The complete authentication process involves the steps of registration, access, and management. During registration, the user and the system agree on a secret credential, most commonly the user chooses the credential and registers it with the system. During access, the user proves with the system that they actually possess the correct pre-registered credential. During management, the user can manage their credentials, including changing, updating, or deleting their credentials.

The most common method of authentication is the use of passwords, where the user provides a secret password to the system to verify their identity. The user registers the password with the system during the registration process. The user supplies the password during the access phase, and the system verifies that the password is correct. The user can manage their password by changing, updating, or deleting it.

A factor for authentication is a category of credentials that can be used to authenticate the identity of a user. There are three main categories of factors of authentication [128]: something you know, something you have, and something you are.

- Something you know: This factor of authentication is knowledge-based and involves information that only the user knows such as a password, PIN, or answers to personal security questions.

- Something you have: This factor of authentication involves something the user possesses, such as a physical token (e.g., a security key fob that generates a one-time password) or a soft token (an app that generates a one-time password).

- Something you are: This factor of authentication involves something the user is, such as biometric characteristics like fingerprints, facial recognition, iris scans, or voice recognition.

**Local vs Web Authentication** Even though local systems use authentication, the security of local systems also depend on the proximity to the system, i.e. the adversary needs to be near the system to attempt to authenticate. In comparison, authentication in the web is

more challenging because it is done over the network, and the adversary can be anywhere in the world. So, local system authentication requires strictly more privileged adversaries than web authentication. Therefore, for the rest of this document, we focus on web authentication as the issues on the web is a strictly more difficult problem than local system authentication.

## 2.2 Password-based Authentication

Use of passwords is the most common method of authentication. The figure 2.1 shows the process of password-based authentication in the web. Typically, the process of password-based authentication in the web works as follows:

1. The user chooses a password.
2. The user inputs the password in the web browser.
3. The web browser sends the password to the web server.
4. The web server stores the password securely for future use.

After this, the user can utilize their registered password to access the system with the following steps:

1. The user inputs the password in the web browser.
2. The web browser sends the password to the web server.
3. The web server verifies that the password is correct.
4. The web server grants access to the user.
5. The user can then access the secure resources.

## 2.3 Attacks on Passwords

Password-based authentication is susceptible to various attacks that can compromise the security of the system.

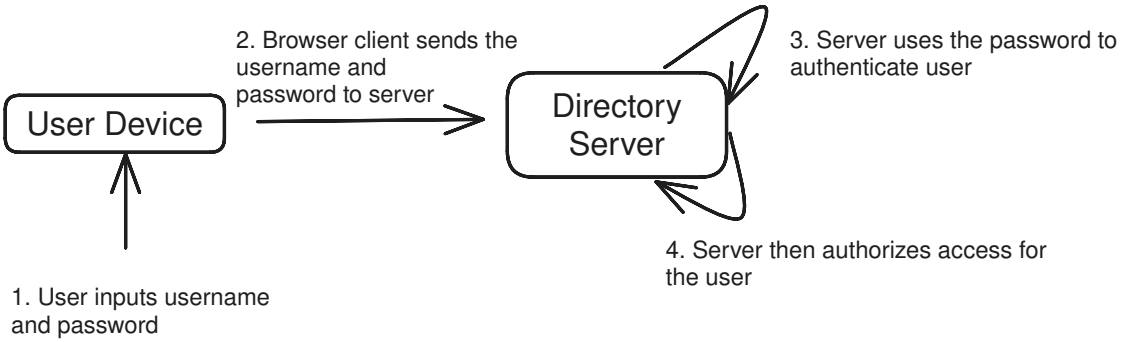



Figure 2.1: Password-based authentication process

### 2.3.1 Retrieval of plain text passwords

Some attacks on passwords result in the adversary gaining direct access to the plain text passwords. In cases of phishing attacks, an adversary tricks the user into revealing or sending them their plaintext passwords. Shoulder surfing attacks involve an adversary looking over the user's shoulder to see the password being typed. In both these cases, the adversary gains direct access to the plaintext password. These attacks are difficult to prevent as they rely on the users' actions and are not dependent on the security of the system.

Another case in which the adversary can directly gain access to plaintext passwords is when they are inserting the credentials into a compromised system. Many layers of software or hardware could be compromised for this attack to work. In terms of hardware, the adversary could compromise the keyboard or the screen to capture the passwords.

In terms of software, the adversary can compromise any layer of software that is involved. Such as the network for man-in-the-middle attacks on credentials, where they can capture the plaintext passwords as they are transmitted over the network. The adversary can also compromise the operating system or the browser to capture the plaintext passwords as they are entered by the user.

Another case of direct access to plaintext passwords is in the case of cross-site scripting attacks(XSS). XSS attacks involve the adversary injecting malicious scripts into the web page that the user is visiting. These scripts can capture the plaintext passwords as they are entered by the user. These attacks are difficult to prevent as they rely on the security of the web page that the user is visiting. XSS attacks are very common in the web (is in the top 10 OWASP vulnerabilities) and are difficult to prevent.

In cases of both hardware and software compromise, hardware and software security could be hardened to prevent the adversary from gaining access to the system. In case of hardware, some works provide secure hardware components that can be used to enter credentials securely and provide indications when the system is secure. In case of software, work has been done to either secure software systems or provide signals to the user so that they can identify if the system is secure.

### 2.3.2 Online and offline guessing attacks

Online guessing attack is the type of attack where the adversary tries to guess the password by sending multiple login attempts to the system. There is only limited number of guesses that the adversary can make as most systems, following secure practices, will throttle the amount of login requests that can be sent for an account and lock an account if too many incorrect login attempts are made. Usual method of online guessing attack is to use a list of common passwords and try them for multiple user accounts. Furthermore, if the adversary has compromised users' passwords from another website, they can use the same passwords to try to login to other websites. As can be seen, these attacks stem from the users' choice of weak passwords and password reuse.

Offline guessing attack is the type of attack where the adversary has access to the password database and can try to guess the passwords offline. In most cases, if the system is following best practices, the password database is hashed and salted. So, the adversary brute-forces the hashed passwords to try to guess the plaintext passwords. This attack is more difficult than online guessing attack as the adversary needs to have access to the password database. Modern techniques of password cracking such as dictionary attacks, rainbow tables, and brute force attacks are used to crack the hashed passwords. This attack stems from the use of weak passwords which are easy to guess.

Due to both these attacks, it is important for users to choose unique and strong passwords for each of their accounts that is hard to guess. To make sure that the users are choosing strong passwords, many websites require users to create passwords that are compliant to a strong password composition policies(PCP) and that are not present in the list of common passwords(blocklists).

Previous research has shown that users find it difficult to create strong passwords and remember them. Strong password composition policies are a pain point for users, as remembering strong passwords is difficult. So, users perform various shortcuts to create composition policy compliant passwords which are not difficult for the adversary to guess.

## 2.4 Password Managers

To alleviate the burden of password management for users, password managers have been developed. Password managers are tools that help users generate, store, and autofill passwords. Password managers help user against various attacks against passwords. They help users generate strong passwords to prevent offline guessing attacks. They help users store unique passwords for each account to prevent password reuse, to prevent online guessing attacks. Password managers help users autofill passwords to prevent phishing attacks, and shoulder surfing attacks.

Even though password managers provide these benefits, users underutilize the password managers. Specially in generation. Password managers also have issues during entry that is from passwords but hasn't been solved.

## 2.5 Passwordless Authentication

In order to move away from passwords, researchers have proposed passwordless authentication systems [80]. Instead of passwords, passwordless authentication systems utilize public-key cryptography to authenticate users. The system sends the user a challenge that the user signs with their private key. The system then verifies the signature with the user's public key.

### 2.5.1 FIDO2 protocol

The FIDO2 protocol facilitates user authentication to a web service through the use of public-key cryptography. Users register their public key with a web service and authenticate themselves by using their private key to sign challenges issued by the service.

**Entities** The FIDO2 protocol involves three main entities:

- The *Relying Party (RP)*, such as a web application like "facebook.com" that utilizes FIDO2 for authentication. This application interacts with the authenticator via the WebAuthn client. The *RP* adopts FIDO2 as an option for two-factor authentication (2FA) or for passwordless sign-in.

- The *Authenticator* holds the user's private key and is tasked with generating a login credential for the RP upon receiving user input, such as a PIN or a button press.

  FIDO2 accommodates various types of client-side authenticators, including external (roaming) authenticators like hardware security keys (*HSK*) and integrated platform authenticators like biometric systems. While this paper specifically discusses *HSK*, the concepts are applicable to all types of FIDO2 authenticators.

- The *WebAuthn Client*, often embedded within a web browser, serves as the intermediary that facilitates communication between the authenticator and the *RP*. To safeguard against phishing, the WebAuthn client conveys the origin (URL) of the relying party to the authenticator, which then uses the credentials associated with that origin to generate a response.

The FIDO2 framework is built around the Web Authentication (WebAuthn) browser API and the Client-to-Authenticator Protocol (CTAP). The CTAP allows secure interactions between the WebAuthn client and external or roaming authenticators using Bluetooth, USB, or Near Field Communication (NFC). The WebAuthn API provides an interface for the *RP* in the client.

**Registration and Authentication** Users enroll their *HSK* for either 2FA or passwordless authentication. Figure 2.2 illustrates the FIDO2 registration process. Upon the user initiating registration via the Register button, a registration request is dispatched to the *RP*. The registration sequence is as follows: (1) The *RP* initiates by dispatching a challenge to the webAuthn client, along with user and *RP* details. (2) The webAuthn client relays these details to the *HSK*, appending the *RP*'s origin or URL and the type of request. (3) Upon receiving user consent, usually via a button press, the *HSK* generates a new set of asymmetric keys. (4) The *HSK* then transmits the credential id, the public key, a SHA-256 hash of the *RP*'s domain (known as the *RP* ID hash), a counter to track authentication attempts, and an attestation signature—a signed object detailing the public key credentials along with the *HSK*'s make and model—back to the webAuthn client as part of the attestation object. (5) The webAuthn client forwards this information to the *RP*, which then verifies the signatures

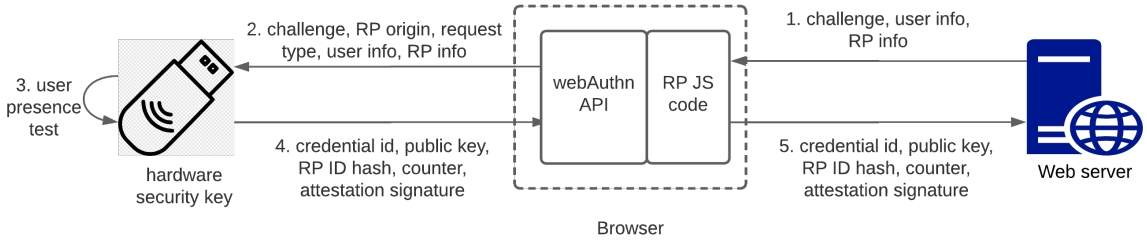



Figure 2.2: FIDO2 registration protocol

and essential components of the response. If verification is successful, the *HSK* is registered to the user's account.

The authentication process mirrors registration but with two key distinctions: (1) it omits the requirement for user data, and (2) rather than generating an attestation, the *HSK* performs an assertion by signing the response using the private key associated with that particular credential id.

## 2.6 Local malicious agents in the browser

### 2.6.1 Cross Site Scripting (XSS)

Cross-Site Scripting (XSS) [132] attacks are a prevalent security vulnerability in web applications, where attackers inject malicious scripts into content that appears on a user's browser. This type of attack exploits the trust that a user has for a particular site, allowing the attacker to execute scripts in the user's browser in the context of the trusted website. The consequences can range from stealing cookies and session tokens, to defacing websites, and even launching phishing campaigns. XSS attacks are categorized into three types: stored, reflected, and DOM-based, each differing by the method in which the malicious script is delivered to the user. Addressing XSS vulnerabilities is crucial for web security, requiring both developers and security professionals to implement measures such as input validation, encoding output, and using security headers to mitigate the risks associated with these attacks.

### 2.6.2 Browser Extensions

Browser extensions or add-ons are small software programs that extend the functionality of a web browser [117]. They are designed to enhance the user experience by adding new features or modifying existing ones. Browser extensions can be given access to a wide range of browser features, such as tabs, bookmarks, and browsing history, allowing them to customize the browser's behavior and appearance. Important extensions such as ad blockers and password

managers need these permissions to function properly, and provide users with valuable features and services.

However, browser extensions can also pose security risks, as they have access to sensitive information such as browsing history, cookies, and passwords. Malicious browser extensions can exploit this access to steal user data, inject ads, or redirect users to phishing sites. During installation, users are often required to grant permissions to the extension, but research has shown that users might not fully understand the implications of these permissions, leading to potential security risks [86].

## 2.7 Related Works

In this section, we discuss various works related to ours and how our work fits into the larger context of prior work.

### 2.7.1 PCP Languages

There have been previous proposals for building PCP languages, with each providing a different subset of the features used in our PCP language. Two proposals involve adding additional HTML attributes to input fields to specify PCP requirements [19, 109], though they only cover a small subset of the most common PCP features.

Horsch et al. [74] developed an XML-based PCP language by automatically scanning and extracting PCPs for 72,125 services. Based on a sample of 200 manually verified PCPs, they estimated that their algorithm correctly extracted PCPs in just over four out of five cases, with the remaining cases evenly split between mostly correct and incorrect. Their resulting PCP language has most of the features found in our language. However, it is missing support for multiple rules, requiring a subset of character classes, limiting maximum consecutive characters from the same character class, and set required and prohibited locations based on distance from the end of the password. This demonstrates the limitation of this type of automated PCP extraction—i.e., it can only find PCP features that the automated tool expects to find. We extend the works of previous PCP langauges by providing a more comprehensive set of features that can be used to specify PCP requirements. Due to usability

issues with encoding PCPs with existing languages, we provide a more user friendly syntax, along with a user study to evaluate its usability.

### 2.7.2 Web PCP Analysis

In 2010, Florêncio and Herley [56] retrieved PCPs for 75 websites in the US. They found that contrary to their intuition, the importance of a website had little correlation to the PCP used on that website. In many cases, the largest, most important websites had the weakest PCPs. They suggested that the reason for this was that due to market economics, these larger websites needed to be more concerned with usability than security, being able to absorb the security cost of weak PCPs more readily than smaller sites.

In 2016, Mayer et al. [107] replicated and extended the work of Florêncio and Herley. In addition to re-examining 70 of the websites used in the original study (five did not work), they also analyzed 67 German websites. They find that overall, PCP strength has been increasing, though German PCPs, on average, are weaker than US PCPs. In our work, we replicate and extend these works by including a larger dataset of 270 websites which is more geographically diverse. Compared to this prior work, we gather more features of the PCPs used on these websites and develop a more fine-grained estimation of PCP strength.

### 2.7.3 PCP Usability

Several studies have examined the effect of password policies on user behavior. These studies have shown that while strong PCPs make passwords harder to crack, they also make passwords harder for users to remember [143]. Furthermore, as the number of passwords a user needs increases, their ability to remember them decreases [163, 2]. This helps explain why when Florêncio and Herley [55] studied password behavior of half a million users, they found that users had on average 25 passwords and reused any given password on an average of 6.5 different websites.

Other research explores what PCP features make passwords harder to remember, with most research finding that it is complex character class requirements that cause the most difficulty [90, 155, 156]. In contrast, minimum length is not nearly as significant of an

impediment, leading researchers to suggest favoring longer but less complicated passwords. More recently, we have seen these suggestions reflected in NIST guidelines [62]. Our research finds that length has the greatest impact on PCP strength for both passwords generated at random and using an alphabetic-first approach. As such, we echo prior recommendations for PCPs to focus on length as opposed to complexity.

### 2.7.4 Browser-Based Password Exfiltration

Examining the literature, we identify three avenues by which passwords can be exfiltrated from the browser. This does not include threats outside of the browser such as phishing [3], man-in-the-middle network attacks [130], or compromised execution environments [53, 16]. While each threat has extensive related work, a high-level understanding of them (as held by most readers) is sufficient to understand the remainder of this paper. As such, we omit discussing them in this section for brevity.

#### Web Trackers

Web trackers are scripts used to track users across different websites, primarily to more effectively serving advertisements. Trackers collect various information and interactions of the user without the user even knowing about it. Senol et al. [153] analyzed form submissions on 100,000 popular websites, investigating whether web trackers on those pages would exfiltrate user passwords. They find that at least 2–3% of those pages include web trackers that exfiltrate passwords. While the motivations for this exfiltration are largely unknown, in many cases, it likely represents an honest-but-curious attacker model—i.e., web trackers are simply grabbing whatever information they can and are not targeting passwords specifically. Regardless, as shown by Dambra et al. [40], users encounter these web trackers frequently, indicating a need to protect user passwords against these honest-but-curious trackers.

### Malicious Client-Side Scripts

Malicious scripts running within a web page can steal user passwords by extracting them from the document object model (DOM) after they have been typed into a form by the user or autofilled by a password manager. There are several sources of malicious client-side scripts.

The most common source of malicious client-side scripts is cross-site Scripting (XSS) attacks. In these attacks, an adversary coerces a website to include attacker-controlled scripts in a web page's DOM, whether by tricking users into clicking a link with the malicious script (reflected XSS attack) or uploading the malicious script to the website (stored XSS attack). While defenses for XSS attacks are well known, the OWASP foundation consistently ranks them in its top 10 web application security risks [133], and hundreds of XSS attacks have been reported in January of 2024 alone [39].

Another common source of malicious client-side scripts are websites that use third-party libraries, with such libraries being ubiquitous [97]. While libraries produced by an adversary are transparently dangerous, more concerning are supply chain attacks [129, 49]. In these attacks, an adversary will compromise an otherwise benevolent software library; then, when websites relying on this library are updated, they will also become compromised. These attacks are already somewhat common [129, 49], with WhiteSource [170] identifying 1300 malicious JavaScript libraries in 2021.

While supply chain attacks are a problem for client-side and server-side libraries, in this paper we are primarily concerned with client-side supply chain attacks. Web client-side supply chain attacks are especially concerning as it is a common practice to load libraries from external sources.[2] If a website adopts this practice, the website will be immediately compromised as soon as the library is compromised, without needing to wait for the website to explicitly update to the compromised version of the library.

### Malicious Browser Extensions

The final avenue for password theft is malicious browser extensions. Browser extensions have two avenues for exfiltrating user passwords. First, they can inject client-side scripts into web

[2]For example, see usage instructions for Bootstrap at `https://getbootstrap.com/`.

pages, stealing passwords in the same way as other malicious client-side scripts. Alternatively, they can inspect the body of outgoing web requests, stealing passwords found in those bodies.

There are many instances of malicious browser extensions used by millions of users on official Chrome/Firefox extension stores. In 2020, 500 Chrome browser extensions were discovered secretly uploading private browsing data to attacker-controlled servers and redirecting victims to malware-laced websites [131]. Also, Awake has identified 111 malicious Chrome extensions that take screenshots, read the clipboard, harvest password tokens stored in cookies or parameters, grab user keystrokes (like passwords), etc. [15]. These extensions have been downloaded over 32 million times. Finally, in 2021, Cato's analysis of network data showed that 87 out of 551 unique Chrome extensions used on customer networks were malicious [24].

In addition to inherently malicious extensions, extensions can also be compromised through supply chain attacks.

#### Relation to Our Work

The above research demonstrates that there is a critical need to protect passwords from being exfiltrated by honest-but-curious or malicious entities. In this work, we demonstrate how password managers can be modified to prevent password exfiltration for web trackers, malicious content scripts, and malicious browser extensions.

### 2.7.5 Password Managers

Password managers serve to help users (a) create random, unique passwords, (b) store the user's passwords, and (c) fill in those passwords. On desktops, password managers are implemented as browser extensions. The browser does not provide any password management APIs for these extensions to use [123]. On mobile, there is first-party support for password managers, though this support has significant security issues [122].

### Security

Password managers have the potential to provide strong security benefits, but also have the potential to act as a single point of failure for users' accounts [160, 102, 157, 123, 122]. In particular, when passwords are autofilled into websites they are vulnerable to theft by JavaScript and extensions. In the most alarming case, if the password manager fails to require user interaction before autofilling passwords and allows passwords to be filled into iframes, it opens users to password harvesting attacks that can surreptitiously steal many if not all their passwords [160, 123]. This problem can be made even worse when the operating system enforces incorrect behavior, such as in mobile devices [122]. Similarly, care has to be taken so that other concurrent programs cannot steal sensitive information when it is being processed by the manager, such as when a manager copies information to the system clipboard [53, 16]

### Usability

Simmons et al. [158] systematized password manager use cases. They found that today's managers poorly supported many password manager use cases and that even when supported, they were often targeted at experts rather than the lay users the tools claimed to support.

Huaman et al. [78] investigated integration problems between desktop password managers and websites, finding that managers often struggle to support modern web standards and that websites are highly heterogeneous in their implementations, leading to difficulties. Seiler-Hwang et al. [152] conducted a laboratory user study of four smartphone password managers, finding significant usability issues, particularly in the integration of the manager with apps and browsers, with users rating the managers as having barely acceptable usability.

Lyastani et al. [104] instrumented a password manager to collect telemetry data regarding password manager usage. Their results show that users underutilized password generation. This phenomenon was partly explained in research by Oesch et al. [125] that surveyed users and showed that many users avoid the security-critical functionality of password managers, such as password generation or password audits because they felt these features were too difficult to use. Instead, they focused on features with the highest usability, such as autofill.

Work by Karole et al. [88] and Ciampa et al. [35] have shown that users prefer different password manager implementations, with non-technical users preferring phone and browser-based managers, whereas technical users are more likely to prefer a standalone manager.

#### Relation to Our Work

Research into the security of password managers over the last decade has consistently shown that the autofill process is a key component of security issues with password managers [160, 102, 157, 123, 122]. In this paper, we explore how the autofill process can be transformed from a weakness of password managers to one of their strengths, providing benefits not available for manual entry. Critically, this security benefit is available without any change to user behavior, which is not the case for other password manager security benefits [158, 125].

### 2.7.6 FIDO2/WebAuthn

Previous research has shown that FIDO2 is prone to local attacks. The initial investigations into the security of the early FIDO protocols—the Unified Authentication Framework (UAF) and Unified 2nd Factor (U2F)—revealed vulnerabilities. UAF operates as a passwordless authentication system, and U2F serves as a standardized 2FA system. Hu et al. conducted the first informal evaluation of UAF, highlighting three potential attacks [77], including a re-binding attack where an attacker links their authenticator with a *RP* instead of the user's authenticator. Subsequent analysis by Panos et al. [136] identified further attack vectors that could lead to system compromise, assuming that attackers could gain access to the authenticator and control the UAF client. Pereira et al. carried out the first formal analysis of FIDO1 [141], considering threats from network attackers and those capable of compromising the client or server. They affirmed that the protocol remains secure as long as the *RP* is properly authenticated. Their analysis, however, was limited to authentication and did not cover registration.

Later, UAF and U2F were combined into the W3C standardized protocol, FIDO2, which supports both passwordless authentication and 2FA. Guirat et al. provided a formal verification of FIDO2's security against both passive and active network attacks [65]. Jacomme

et al. formally analyzed several multi-factor authentication schemes, including FIDO2 [79], with a threat model encompassing malware, fingerprint spoofing, and human errors, indicating that FIDO2 is not definitively secure against malware.

Additional studies [37, 42, 105] have investigated the usability issues and perceived advantages of FIDO2 device usage. Alqubaisi et al. evaluated the threat of password attacks in comparison to single-factor FIDO2, noting that single-factor FIDO2 fares better against the password-based threat model, although it does not address targeted attacks on the FIDO2 protocol itself [9].

Chang et al.[26] highlighted vulnerabilities of U2F to side-channel and Man-In-The-Middle (MITM) attacks and proposed enhancements to the U2F protocol to mitigate side-channel threats . O'Flynn extended the threat model by demonstrating an attack on *HSK* through Electromagnetic Fault Injection, which led to the compromise of confidential data [127]. Dauterman et al. addressed the risks posed by hardware backdoors in *HSK* and proposed True2F [43], an improved version of U2F designed to defend against malicious *HSK* and enhance resistance to token fingerprinting. While considering a compromised browser, they suggested that as long as the True2F token functions correctly, it is no less secure than traditional U2F. Our work complements these efforts, enhancing the security of *HSK* beyond traditional U2F, assuming the presence of an adversary in the browser.

### Browser Modification Proposals to Secure Authentication

In the works of Stock and Johns [160], they evaluated the security of password manager autofill and discovered that passwords being autofilled were prone to being exfiltrated by malicious scripts. As a solution to this, they proposed a mechanism that involved the generation of a random placeholder called a "*password nonce* during autofill operations. The proposed solution involved four steps: generating a password nonce, automatically filling the nonce into the web page, monitoring outgoing web requests for the nonce, and substituting the nonce with the actual password if the request is intended for the correct origin. Our mitigation approach, called *sbc-FIDO2*, employs a similar strategy for replacement.

**Trusted Execution Environments (TEEs)**

Another line of research is the use of TEEs to secure authentication credentials from untrusted systems and/or browsers [38, 171, 135, 20]. In addition, there are other studies that suggest the implementation of internal FIDO2 devices or alternative multi-factor authentication methods [13, 25, 177, 154, 145]. Krypton [1] implements FIDO2 securely by utilizing mobile TEEs, Apple's Secure Enclave, and Android Keystore. In the Fidelius system[51], the authors suggest the integration of SGX enclaves into web browsers as a means to safeguard users' data from a compromised browser. However, Fidelius needs either two extra hardware components or keyboards and displays that are modified to include built-in processors capable of performing encryption and decryption. In chapter 5, even though using TEEs to secure authentication is a promising body of research, we decided not to pursue it as it requires additional hardware, making it harder to deploy to most systems, and secures against a threat model that is difficult to achieve in practice.

### 2.7.7 Detecting attack on passwords

There have been systems proposed that detect attacks on the passwords, but only when the whole password database is compromised. Most of them rely on something called honeywords. Juels and Rivest [84] introduce the conecept of adding dummy passwords in the passwords file. When the passwords file gets stolen, the server can detect that an attack has occured, and do the required steps to protect against them. But this depended on a secure storage on the server, that is unbreachable, to store which ones are honeywords and which ones are actual passwords. This is an unreasonable assumption as one can store the whole password file in this secure storage server. Wang and Reiter [166] solve the issue of secure secret state by monitoring the entry of passwords and probabilistically marking the passwords. Dionysiou and Athanasopoulos [47] also try to solve this problem by deteministic methods, using a synchronized random number generator between the server and a separate checking server. The checking server replays the login events, and checks if the correct sweetword is selected using the RNG. All of these works have the concept of honeywords, but are primarily based on when the passwords file is breached.

### 2.7.8 Provenance based intrusion detection

Zipperle et al. [179] perform a survey of providence based intrusion detection system. Common methods of creating providence in existing systems is by either utilizing audit logs, either existing or additional, to create a directed acyclic graph.

There has been a lot of work in attack detection in lower-level systems such as the Linux kernel using complete system provenance data and a posteriori audit [18, 66, 69, 70, 75, 82, 103, 110, 167, 174].

ProvDetector [166] collects kernel-level data provenance by monitoring system call activities of different processes. It uses neural embedding techniques to learn representations of the provenance data, enabling the system to effectively identify stealthy malware. Custos [134] utilizes similar audit methods to uncover tampering of system logs in the kernel.

Mnemosyne [7] attempt to postmortem analyze for watering hole attacks by creating a causality graph based on browser audit logs and extra information collected using an auditor daemon. They utilize auditor daemon as opposed to modifications in the browser, which helps in it's wider deployment. OutGuard [54] utilizes an instrumented browser to collect detailed data for different web pages, such as: request and response traces, and javascript execution traces. It then utilizes machine learning model trained on the data to detect cryptocurrency mining happening in the wild.

### 2.7.9 Browser Provenance

Usage of data provenance is common in biological sciences domain where document access logs are used to link scientific data documents and verifying scientific processes. Zhao et al [178] use semantics logs and analysis to link documents in biology experiments, so that researchers can in future examine the connection of different documents. Anand et al [12] present a provenance browser that allows visualization and navigation of provenance log data.

Margo and Seltzer [106] argue that the browser history metadata can be used as a browser provenance. They propose different applications of such lineage, such as finding the path of origin of a downloaded malware. They propose storing browser provenance as graph structure for efficient storage and lookup to find relationships a posteriori.

Yu et al [175] suggest using instrumentation, i.e. modification of existing javascript codes run in the browser, for security analysis. Jia et al. [83] insert javascript to fingerprint the different attackers in the browser using user agent, ip, etc. This work involves actual attackers instead of scripts as in our work.

# Chapter 3

# Improving Usability of Generated Passwords[1]

Despite their problems [146, 55, 44, 41, 162, 138, 165], passwords remains the dominant form of authentication [21]. Password managers strengthen password-based authentication by helping users generate, store, and enter passwords, making it easier to adopt strong, unique passwords [138, 104]. Still, research has shown that password manager users underutilize password generation [140, 104]. One potential explanation for this phenomenon is that websites' password composition policies (PCPs) can reject generated passwords, decreasing the usability and utility of the generator. [78, 126].

To address this issue, we design a PCP language that websites can use to encode and publish their PCP, with password managers downloading the PCP to ensure that they only generate compliant passwords. To inform the design of this PCP language, we extract 270 PCPs from a geographically diverse set of 626 popular websites. Using this dataset, we build an initial PCP language, then iteratively refine it as we encode the gathered PCPs, stopping once all PCPs in our data set can be efficiently and useably encoded. Our final PCP language is more feature-rich than previous efforts and is the first PCP language that can represent the full range of PCPs found in our dataset.

[1]This chapter is adopted from my publication: Gautam, Anuj and Lalani, Shan and Ruoti, Scott. "Improving Password Generation Through the Design of a Password Composition Policy Description Language" Symposium on Usable Privacy and Security. 2020.

To demonstrate the feasibility of our proposed language, we (i) build proof-of-concept websites that publish their PCP using our language; (ii) modify BitWarden, a popular password manager, to download these PCPs and generate compliant passwords; and (iii) create Python and JavaScript libraries that make it easy to use our PCP language in server- and client-side code. Next, we conduct an online usability study with 25 participants, measuring their ability to author PCPs using our language and tools. Our results show that most participants can rapidly comprehend our language and author PCP descriptions, even for complex policies.

Finally, we replicate and extend prior work analyzing Web PCPs [56, 107]. In contrast to prior efforts that use a simple heuristic that only considers the minimum length and allowed characters for measuring PCP strength, our analysis takes into account all requirements of the PCP. Additionally, our analysis includes both upper- and lower-bound estimates for PCP strength that take into account how users select passwords [101, 164]. This improved analysis shows that most PCPs in our dataset fail to require passwords that resist offline attacks. Furthermore, for users that prefer passwords comprised primarily of digits [101], nearly half of the evaluated PCPs fail to require passwords that resist online attacks.

**Research Artifacts:** Our data, scripts, and prototypes are available at `https://userlab.utk.edu/publications/gautam2022improving`.

## 3.1 PCP Dataset

To inform the design of our PCP language, we gathered an extensive corpus of PCPs deployed on the Web. Our sample is demographically diverse, including websites from highly-populated countries in each of the six inhabited continents: Africa—Nigeria, Asia—India, Europe—Germany and the United Kingdom (UK), Oceania—Australia, North America—United States (US), South American—Brazil. We also measured PCPs from China, Iran, and Russia to see if their high levels of Internet censorship [76] impacted PCP selection.

### 3.1.1 Sources

We used the Alexa and Quantcast lists of the most popular websites to select websites for each country. In January 2019, we downloaded the Alexa lists of the 250 most popular US websites and the top 50 lists for the remaining nine countries we examined. As we began to analyze these websites, we noticed a high overlap between the websites listed for each country. To obtain more unique websites for each country, in February 2019, we downloaded the Quantcast lists of the top 50 most popular websites for each country. We selected Quantcast as its country-specific lists had minimal overlap with global and US-specific websites from Alexa. We also analyzed the websites listed in the Quantcast top 50 global lists. In total, these lists identified 626 unique websites.

Next, we removed websites that do not support account creation, delegate all authentication to single sign-on (SSO) providers, or require resources we do not have to create an account (e.g., a bank account). For the remaining 320 websites, we identify websites that use the same authentication backend (e.g., google.com and youtube.com), keeping only a single representative website. We then extracted PCPs from the remaining 270 websites.

### 3.1.2 Analysis

To extract the PCP for each website, we took the following steps. First, we would look for PCP components described textually on the account creation web page or elsewhere on the domain. Second, we would examine the HTML form, looking for validation attributes that restricted what users could enter for their password. Third, we evaluated any JavaScript used to validate the password, identifying restrictions enforced therein. Fourth and finally, we manually tried to enter various passwords of different lengths and compositions.

### 3.1.3 Limitations

While our data collection resulted in a large and rich corpus, we recognize there are limitations to our methodology. First, while covering more features than past efforts [56, 74, 107], our data is not comprehensive. Still, we believe our dataset is sufficient for our purposes as we

achieved saturation [4]—i.e., we stopped discovering new PCP features at the latter end of our analysis.

Second, it is likely that we missed some PCP edge cases. Only by investigating the server-side code would it be possible to identify the exact PCP definitively. Automating the process to check more password combinations would be problematic as this would involve flooding the website with passwords.

## 3.2 PCP Description Language

Using our PCP dataset, we design a language for describing PCPs. Our language has two key design goals: (1) describe the PCPs in our dataset and (2) be simple to read and write for administrators and machines. To achieve these goals, we followed an iterative design process:

First, we created a draft version of our PCP language based on prior research (§3.8) and PCP features in our dataset. Second, we encode the PCPs in our data set using this language. When we encountered a PCP that was onerous to encode, we modified our draft PCP language to address pain points. We would then re-encode all prior PCPs to ensure that our change did not cause a usability regression. Third, after encoding all PCPs, we reviewed our language with others from our research group, focusing on improving the language's readability and identifying PCP features they had encountered in the wild but are absent in our PCP dataset. Based on their feedback, we updated our language and re-encoded the PCPs in our dataset (continuing to look for usability issues). After making a full pass encoding PCPs without changing our language, we considered it finished.

### 3.2.1 PCP Language

A PCP in our language is composed of two components: (a) a set of characters allowed in a password and (b) rules about password composition.

The allowed characters are grouped into named, disjoint sets of characters—a *charset*. By default, the PCP uses the following four default charsets: lowercase English letters (`lower`), uppercase English letters (`upper`), Arabic numerals (`digits`), and the OWASP password symbols [132] (`symbols`). Our language allows these default charsets to be modified, new

charsets to be added, and default ones to be removed. Our language also provides an `alphabet` charset that, if used, merges and replaces the default `lower` and `upper` charsets.

A PCP composition rule is a set of *requirements* that passwords must comply with to be valid. If a PCP contains multiple rules, a password need only satisfy the requirements for a single rule to be valid (the overwhelming majority of PCPs only have one rule). For example, if one rule specified that passwords must be eight characters long and contain lowercase letters and symbols and another rule specified that passwords must be fifteen characters long, fifteen character passwords of only digits would be valid, whereas fourteen character passwords of only digits would not.

The possible requirements in each rule are as follows:

- `min_length` is a positive integer specifying the password's minimum length (inclusive). All rules require that `min_length` is set, with all other requirements optional.
- `max_length` is a positive integer specifying the password's maximum length (inclusive).
- `max_consecutive` is a positive integer indicating the maximum number of times the same character can appear consecutively in a password. For example, to prevent passwords such as `AAA` or `ZZZ`, `max_consecutive` would be set to 2.
- `prohibitted_substrings` is a set of strings that may not appear anywhere in the password. When used, this commonly includes the website name and other related words. For example, to prohibit the string "google", `prohibited_substrings` would be set to `["google"]`.
- `require` is a list of charsets that must appear in the password. For example, to require that a password must have letters and digits, `require` would be set to `["alphabet", "digits"]`.
- `require_subset` is an object containing a list of charsets (`options`) from which `count` of those options must appear in the password. For example, to require that a password must have digits and symbols, but not necessarily both, `require_subset` would be set to `{"options": ["digits", "symbols"], "count": 1}`. If not set, `options` defaults to using all the PCP's charsets; `count` defaults to one.

- `charset_requirements` is a map between charset names and requirements for the named charset. For example, to add additional requirements for digits, `charset_requirements` would be set as such: `{"digits": {requirements}}`. Possible requirements include:

  - `min_required` is a positive integer specifying the minimum number of times this charset must appear in the password.
  - `max_allowed` is a positive integer specifying the maximum number of times this charset may appear in the password. For example, if set to two for the digits charset, passwords containing `111` or `123` would be rejected.
  - `max_consecutive` is a positive integer indicating the maximum number of times this charset can appear consecutively in a password. For example, if set to two for the alphabet charset, passwords containing `abc` or `ddd` would be rejected.
  - `required_locations` is a list of indices for the password at which this charset must appear. Passwords are zero-indexed and negative indices are supported (i.e., reverse string indexing). For example, to require a password that starts and ends with a symbol, `required_locations` for the symbols charset would be set to `[0, -1]`.
  - `prohibited_locations` is a list of indices for the password at which this charset must *not* appear. Passwords are zero-indexed and negative indices are supported (i.e., reverse string indexing). For example, to prevent a password from having the last two characters as digits, `prohibited_locations` for the digits charset would be set to `[-1, -2]`.

A JSON schema for our final PCP language is given in Listing 3.1. Examples of real-world PCPs encoded using our language are given in Listing 3.2.

Examining the JSON-encoded PCPs in our dataset, we find that they are 17–205 characters long, with a median length of 36 characters. These small sizes are evidence that our PCP efficiently encodes passwords. Lastly, we note that while we used JSON to encode policies,

```
{
  "charsets": {
    "name": "characters", ...
  },
  "rules": [{
    "min_length": ℤ⁺,
    "max_length": ℤ⁺,
    "max_consecutive": ℤ⁺,
    "prohibited_substrings": ["substring", ...],

    "required": ["charset_name", ...],
    "require_subset": {
      "options": ["charset_name", ...],
      "count": ℤ⁺
    },

    "charset_requirements": {
      "charset_name": {
        "min_required": ℤ⁺,
        "max_allowed": ℤ⁺,
        "max_consecutive": ℤ⁺,
        "required_locations": [ℤ⁺, ...],
        "prohibited_locations": [ℤ⁺, ...],
      },...
    }
  },...]
}
```

Listing 3.1: JSON schema for our PCP language

```
# Passwords of length six to twleve (walmart.com)
{"min_length": 6,"max_length": 12}

# Password must include at least one digit, symbol, and alphabetic character (facebook.com)
{
  "min_length": 6,
  "require": ["digits", "alphabet", "symbols"]
}

# Custom definition for symbols that are allowed (macys.com)
{
  "charsets": {"symbols": "!\"#\$%&'()*+:;<>?@[]^`{}~"},
  "rules": [{"min_length": 7,"max_length": 16}]
}

# Password must have at least one alphabetic character and either a digit or a symbol (bbc.com)
{
  "min_length": 8,
  "max_length": 50,
  "require": ["alphabet"],
  "require_subset": {
    "count": 1,
    "options": ["digits", "symbols"]
  }
}

# Password can be eight characters if it contains a lowercase character and a digit. Otherwise, it must be fifteen characters long. (github.com)
{
  "rules": [
    {"min_length": 8,"require": ["lower", "digits"]},
    {"min_length": 15}
  ]
}
```

Listing 3.2: PCP examples encoded in our language

they could also easily be encoded in a wide range of data-interchange formats (e.g., YAML, protobuf).

## 3.3 PCP-Compliant Password Generation

To demonstrate the feasibility of our proposed language, we (1) created libraries for using our PCP language, (2) built proof-of-concept websites that publish their PCP using our language, and (3) modified a password manager to generate PCP-compliant passwords.

### 3.3.1 Library Implementations

We constructed Python[2] and JavaScript[3] libraries to support our PCP language. These libraries enable the programmatic creation of PCPs, encoding PCPs to JSON, and parsing PCPs from JSON. They also automatically validate PCPs to ensure they are both semantically correct—e.g., that `min_length` is appropriately set and that character sets do not overlap—and logically consistent—e.g., that a policy does not simultaneously require and prohibit a character class.

These libraries also support checking passwords against a PCP. Finally, they can evaluate the strength PCPs, giving administrators an idea of how likely a PCP is to result in passwords that resist online and offline guessing attacks (see Appendix B for more details).

### 3.3.2 Website Implementation

We built five proof-of-concept websites, each with a PCP of varying complexity. We implemented these websites using Flask (Python) on the backend and JavaScript on the frontend. Each website publishes its PCP and provides a form where passwords can be generated, submitted, and verified.

We identified three approaches for publishing PCPs:

[2] `https://pypi.org/project/password-policy/`
[3] `https://www.npmjs.com/package/password-composition-policy`

1. **HTML:** A new attribute could be added to the password field, which would be set to the JSON-encoded PCP. Alternatively, the PCP could be encoded as XML within the HTML, adjacent to the password field.

2. **HTTP header:** An HTTP header (e.g., `X-PCP`) can specify the JSON-encoded PCP for relevant pages.

3. **File:** The JSON-encoded PCP could be available at a known URL (e.g., `domain.tld/pcp.json`). If there are multiple PCPs for a domain, this file could contain a mapping between URLs and PCPs.

Our websites use the third approach as it is the easiest to implement and the only approach which can work with non-browser-integrated managers. We checked the validity of submitted passwords on the client-side using our JavaScript library and on the server-side using our Python library. *A significant benefit of publishing PCP and using our tool to validate them is that if the PCP is ever updated, there is no need to separately update the validation code, simplifying developer workloads and preventing situations where the client- and sever-side validation may become out of sync.*

### 3.3.3 Password Manager Implementation

We modified BitWarden, a popular open-source password manager, to check if a domain hosts a `/pcp.json` file, and if so, to use it to generate PCP-compliant passwords. The actual generation is handled by our JavaScript library and occurs over three phases:

In the first phase, we set the password length to the smallest `min_length` (if there are multiple rules). Next, we use our JavaScript library to check if passwords of this length using this PCP will be offline-resistant password [57]. If not, we choose the smallest length that would result in an offline-resistant password.

In the second phase, we create an array of length equal to our calculated minimum length. Each position within the array contains an (initially empty) list of which charsets can appear at that position. To fill these lists, we first satisfy `required_locations` by setting the list at the specified index to its respective charset. Next, we set the remaining empty lists as

necessary to satisfy `min_required` and `required`. Lastly, the remaining empty lists are set to include all allowed character sets unless doing so would violate `max_allowed`.

In the third phase, we shuffle all indices not set due to `required_locations`. We then generate a password by randomly selecting a character at each index from the charsets in the list at that index. We then check the generated password against the other requirements in the PCP. If it is not, we repeat phase three until we generate a valid password. In addition to ensuring that generated passwords are PCP-compliant, we also follow recommendations by Oesch et al. [124] and ensure that generated passwords are not randomly weak. This is done by checking passwords using zxcvbn and ensuring that the generated passwords receive the highest strength rating (4).

## 3.4 Usability Study

To evaluate the usability of our developed language and libraries, we conducted an IRB-approved user study wherein participants authored five PCPs of varying complexity using our PCP language. This section gives an overview of the study and describes the tasks and study questionnaire. In addition, we discuss the development and limitations of the study. The study instrument is given in Appendix A.

### 3.4.1 Study setup

The study ran for three weeks starting Friday, January 28, 2022, and ending Tuesday, February 15, 2022. In total, 25 participants completed the study. The study was designed to take about thirty to forty minutes and participants were compensated with a $25 Amazon gift card. Participants were required to have Python 3.6.1 or higher installed on their system. The study was administered online using Qualtrics.

Participants were recruited from the EECS department at our local university using posters, email invitations, and class announcements. We also asked researchers at other universities to share the study with their students. We chose to use EECS students as we felt they were a good representation of novice developers, and we hypothesized that our language and libraries would be sufficiently usable to support novice developers.

### 3.4.2 Study tasks

Participants started by reading and accepting an informed consent statement. Next, participants installed our Python library and executed a Python instruction that allowed us to confirm that the library was correctly installed. They then entered basic demographic information (class standing, major, gender).

Participants were told that in the study they would be authoring five PCPs. They were given a link to documentation for the Python library and informed that this link would also be provided with each task. The documentation included a description of our language, source code examples, and JSON-encoded PCPs.

Participants encoded five PCPs:

1. The password must be at least 8 characters.

2. The password must be at least 8 characters and contain at least two of the following: uppercase, lowercase, digits, symbols.

3. The password must be at least 12 characters, contain a letter and a number, and not contain whitespace.

4. The password must be at at least 8 characters long and contain a letter and a number. Alternatively, the password must be at least 15 characters.

5. The password must be at least 8 characters, contain at least two symbols, contain either an upper or lowercase letter, not contain the string "mywebsite", and none of the following characters: `^'";/\`

Upon submitting a PCP, the survey checked whether the submitted PCP was parsed correctly. It also verified that the PCP was correct by checking two valid and two invalid passwords. Participants were allowed to continue when they submitted a correct PCP description or once two minutes had passed (to prevent participants from becoming stuck). After submitting their policy, participants completed an After-Scenario Questionnaire [149] (ASQ) about their experience.

Upon completing all five policies, participants were asked to fill out the System Usability Scale [23] (SUS) regarding their overall experience. They were also asked what they liked most and least about the system and library. Finally, they were asked to provide any other feedback they had.

### 3.4.3 Demographics

Participants were largely male: male (19; 76%), female (6, 24%). All students studied computer science (23; 92%) or electrical engineering (2; 8%). Participants were all more senior students: juniors (2; 8%), seniors (10, 40%), graduate students(13, 52%).

### 3.4.4 Study Design

Initially, we structured study compensation as a raffle, where five participants would receive a $50 Amazon gift card. Under this incentive scheme, only two participants completed our study. This led us to revise our study to compensate every participant (including the two who had already completed it). After making this revision, re-obtaining IRB approval, and re-launching the study, we quickly gathered our remaining 23 participants.

We also changed our documentation between the two iterations of our study. Initially, the survey provided a link to the documentation explaining how to author policies in JSON, with that documentation providing a link the Python library's documentation. However, after looking at the first two participants' results, it became clear that they lacked proficiency in JSON. To encourage participants to use the Python library, we changed the survey's documentation link to point to the Python library's documentation, with that documentation providing a link to the JSON documentation. Participants could still directly author JSON, and eight (40%) did for at least one task.

### 3.4.5 Limitations

Our students do not have the same experience as the administrators responsible for authoring PCPs. Similarly, participants had less incentive to learn and correctly enter policies than administrators trying to use these tools. As such, our results may not fully represent the

usability of our tooling for the target audience. However, past research has shown that students can serve as a reasonable approximation for developers [119, 118].Lastly, our study only measured the ability of participants to author policies, not to read them.

## 3.5 Study Results

In this section, we report the significant findings of our user study. Quantitative results for each policy are given in Table 3.1. Mean completion times use the geometric mean [149].

### 3.5.1 Success Rates

Overall, participants did very well at encoding policies. Two participants struggled at nearly all tasks, only correctly encoding a single PCP. Excluding them from our data, completion rates move to 100%, 100%, 96%, 100%, and 68%, respectively.

In policies, we detected three types of errors. First, incorrectly formatted JSON (6 total), likely stemming from unfamiliarity with JSON. Second, minor errors (10 total), such as forgetting to include a prohibited character or including a rule from a previous policy. We only classify errors as minor if users showed comprehension of the tested language and library features but made an error with the values used. Third, major errors (4 total) resulting in an entirely incorrect submission. These errors indicate that participants failed to understand how to use the language and library.

Looking at Policy 5's results more closely, we see that three errors (12%) arose due to incorrectly encoded JSON, with the remaining seven (28%) arising due to participants forgetting to include one or more of the prohibited characters. This happened even though these same participants had properly excluded characters in Policy 3.

### 3.5.2 Completion Times

Participants generally completed tasks quickly, with (geometric) mean times ranging between 36 seconds and 4 minutes. However, we note that these times are lower bounds as they do not include time participants may have spent reading documentation between tasks and before

Table 3.1: Quantitative results by policy

| Policy | Correct | JSON mistakes | Minor errors | Major errors | Mean time in minutes | Mean ASQ |
|---|---|---|---|---|---|---|
| 1 | 92% | 0 | 0 | 2 | 1.5 | 7.0 |
| 2 | 92% | 1 | 0 | 1 | 1.4 | 6.7 |
| 3 | 88% | 1 | 2 | 1 | 4.6 | 5.7 |
| 4 | 96% | 1 | 1 | 0 | 0.6 | 6.3 |
| 5 | 64% | 3 | 7 | 0 | 4.0 | 6.0 |

they started interacting with the task. Still, these times suggest that it is easy to pick up and use our language and library with no prior experience.

Using a two-way ANOVA, we find that while there is a statistically significant difference between how long each policy took to create ($F(4, 170) = 8.731$, $p < 0.001$), though this is not surprising given the difference in difficulty between policies. We do not find a statistically significant difference between time taken to author PCPs using JSON or our library ($F(1, 170) = 0.109$, $p = 0.74$), nor for the interaction effect ($F(4, 170) = 0.027$, $p = 1.00$). This is a surprising result as, based on our first two respondents, we expected participants to struggle authoring JSON.

### 3.5.3 Perceived Usability

Overall, policies received good ASQ scores (see Table 3.1), indicating that it was easy and relatively quick to author policies. The mean SUS score was 65, which can be interpreted as "Good" usability [17], receives a C grade [149], and is just above the 40th percentile of systems studied with SUS. While this is an acceptable score for our language and library to be used in the wild [17], it still fell short of our initial expectations.

Looking into the qualitative feedback, we discovered three primary critiques of our tooling. First, many participants felt that JSON was confusing. Second, participants wanted additional documentation. While we provided one example for every PCP feature, they wanted even more. Third, participants were confused by our library providing two ways to create PCPs: (a) a class exactly matching the JSON schema and (b) a simplified class that could be used to encode simple PCPs more directly. While we created this second method to reduce the amount of code participants needed to write for simple PCPs, it ended up causing unneeded confusion and is a prime candidate to remove from our library.

### 3.5.4 Takeaways

Overall, our results show that our proposed language is promising, though it has room for improvement. Other than the two participants who failed all but one task, every other participant correctly encoded Policies 1–4, except for one mistake in Policy 3.

However, of these 23 participants, nine (39%) submitted incorrect solutions for Policy 5. One-third of these errors (3) arose from improperly encoded JSON. This suggests that in line with participant feedback, it might be worthwhile to consider other more developer-friendly encodings (e.g., YAML) or supporting multiple encodings, allowing developers to choose which they will use. Alternatively, pushing for programmatic specification of PCPs could be used to avoid encoding issues entirely.

Two-thirds of the errors (6) for Policy 5 arose from minor issues with the PCP. Half of these issues (3) involved the participants removing some but not all of the prohibited characters from the symbols list. This may have arisen as the textual policy described a denylist for restricted characters, whereas participants chose to create an allowlist of symbols. To address this, the library could allow users to specify a denylist for characters and then have the library generate the appropriate character set, though further research would be needed to measure the efficacy of this approach.

The other issues with Policy 5 (3) arose from participants failing to include the list of restricted characters, even though the other requirements for this policy were included. This happened even though these same participants had properly excluded characters in Policy 3. It is unclear whether this issue stems from something in the design of our language, the general challenge of remembering all the requirements in a complex policy, or study fatigue.

## 3.6 Website Analysis

Using the PCP dataset we collected to build our language, we replicated and extended prior work analyzing website PCPs [56, 107]. Our analysis covers (1) the strength of PCPs, (2) the requirements used in PCPs, and (3) additional non-PCP authentication-related details.

To estimate PCP strength, we calculate the average number of guesses an adversary would need to discover a password that (a) complies with the PCP and (b) is of the smallest allowed length. In contrast to previous work [56, 107] which calculates strength based only on the smallest allowed length and count of allowed characters (i.e., $\#characters^{length}$), our estimates take into account all PCP features. First, we create a canonical representation of the PCP. Second, we enumerate all unique password compositions—a password composition

specifies the number of characters from each character class that makes up a password. Third, for each password composition, we calculate the number of unique passwords that exist for that composition, reducing this number to account for passwords that fail to meet the various `chartset_requirements`. Finally, we sum these counts. A more detailed description of this algorithm is given in Appendix B.1.

In addition to estimating PCP strength based on password chosen entirely at random (as is done in previous research [56, 107]), we also consider PCP strength under conditions where users prefer characters from certain character sets: (a) preferring alphabetic (particularly lowercase) characters over non-alphabetic characters (as commonly seen in the US [101]) and (b) preferring numeric characters (as commonly seen in China [101, 164]). These changes help our analysis to more accurately measure the strength of PCPs under a range of usage scenarios. These calculations are performed by modifying our enumeration of password compositions only to include compositions that use the most preferred character classes unless the PCP specifically requires another character class. A more detailed description is given in Appendix B.2.

Throughout our analysis, we categorize PCPs by (i) the country where they are popular, (ii) their Alexa global rank, (iii) their use case, (iv) whether they generate revenue by displaying ads, (v) whether usernames on the website were publicly available or easily guessed, and (vi) whether a data breach had been reported for the website. All categorizations are mutually exclusive, with PCPs popular in multiple countries categorized as "Global". Table 3.2 lists these categories and the number of PCPs in each.

### 3.6.1 PCP Strength

Figure 3.1 gives the distribution of password strengths. If passwords are generated entirely at random, nearly all PCPs are strong enough to resist online attacks ($10^6$ guesses [57]), though only about 40% are strong enough to resist offline attacks. For passwords where alphabetic characters are preferred, nearly all PCPs fall into the online-offline chasm [55]—strong enough to resist online attacks but not offline attacks(surviving $10^{14}$ guesses [57]). This chasm is problematic because PCPs in it impose a usability burden to pick more complex passwords than necessary to resist online attacks, but which are still too weak to resist offline attacks. For

Table 3.2: Number of PCPs in each category

| Country | Count |
|---|---|
| Global | 65 |
| Australia | 13 |
| Brazil | 14 |
| Germany | 17 |
| India | 9 |
| Nigeria | 13 |
| UK | 8 |
| US | 72 |
| China | 28 |
| Iran | 12 |
| Russia | 19 |

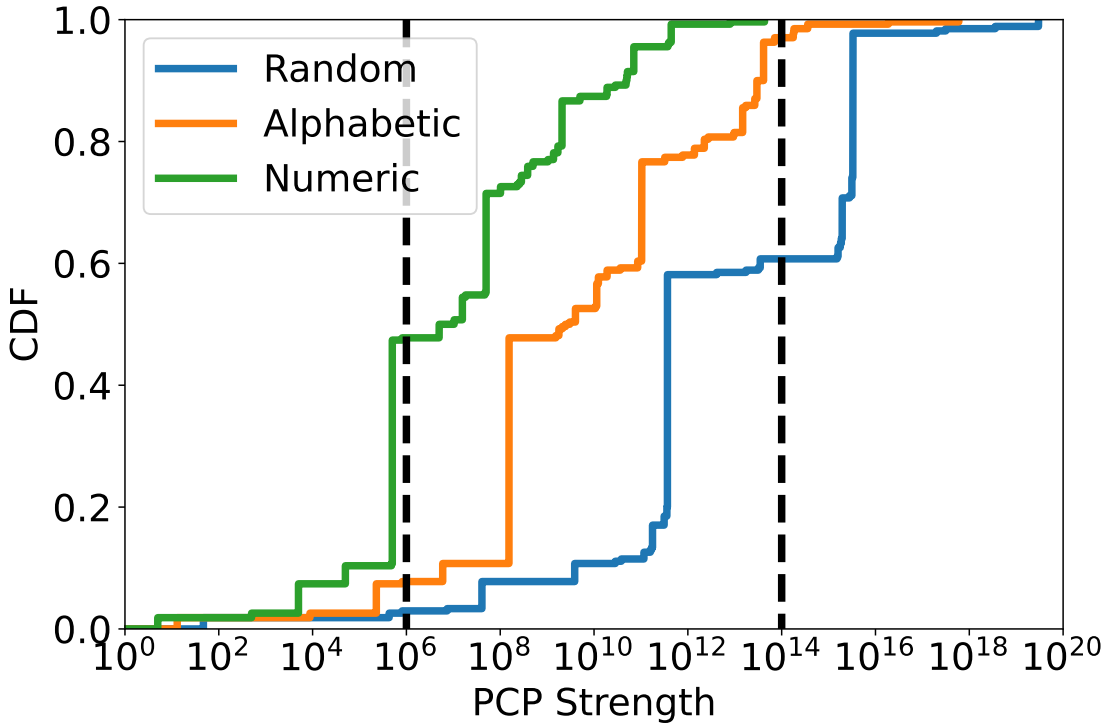
| Popularity | Count |
|---|---|
| Top 10 | 8 |
| Top 50 | 24 |
| Top 100 | 25 |
| Top 500 | 59 |
| Top 1000 | 25 |
| Top 5000 | 79 |
| 5000+ | 50 |

| Use case | Count |
|---|---|
| E-commerce | 58 |
| Finance | 10 |
| News | 72 |
| Social media | 55 |
| Software | 13 |
| Streaming | 28 |
| Other | 34 |

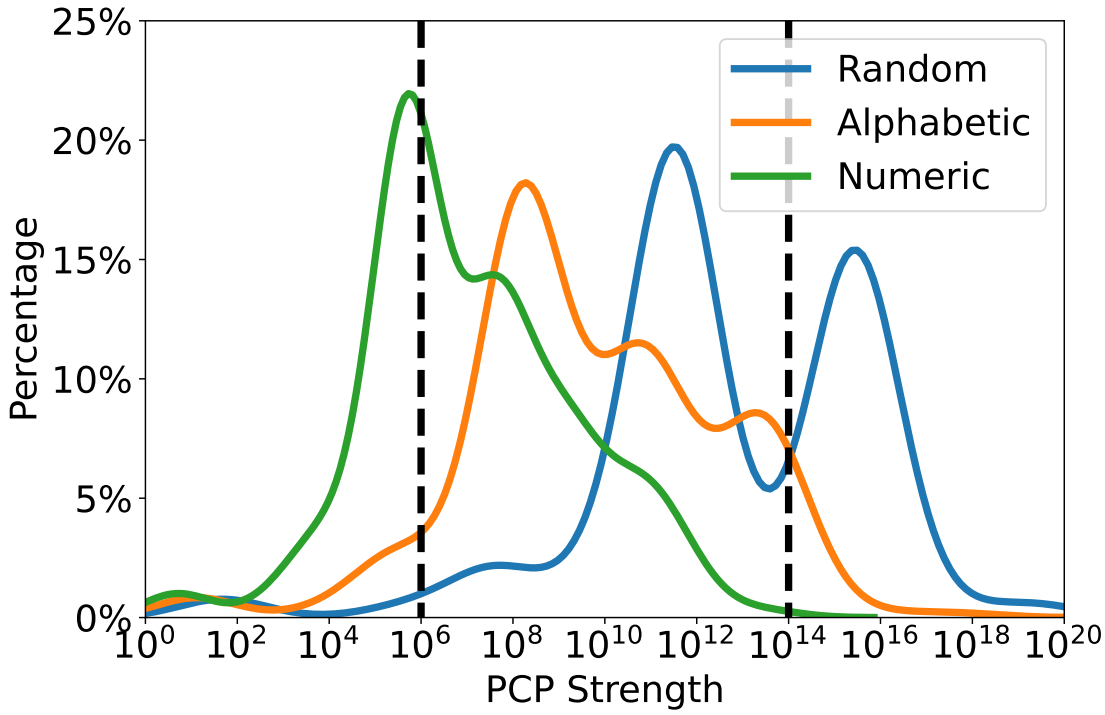
| Ad Provider | Count |
|---|---|
| Yes | 158 |
| No | 112 |

| Public username | Count |
|---|---|
| Yes | 43 |
| No | 227 |

| Past breach | Count |
|---|---|
| Yes | 51 |
| No | 219 |




(a) CDF of PCP strengths




(b) Distribution of PCP strengths

$10^6$ and $10^{14}$ are estimates of the number of guesses a password should resist to survive online and offline attacks, respectively [57].

Figure 3.1: PCP Strengths

passwords where numeric characters are preferred, half of the analyzed PCPs are insufficient to prevent online attacks, and none are strong enough to resist offline attacks.

Comparing mean PCP strength under different password generation strategies, we find that passwords generated at random ($3.5 * 10^{17}$) are roughly two orders of magnitude stronger than alphabetic-preferred passwords ($2.3 * 10^{15}$) and six orders of magnitude stronger than numeric-preferred passwords ($2.1 * 10^{11}$). This highlights the benefits of using a password generator to create passwords. It also demonstrates why it is crucial to consider generation strategy when estimating PCP strength, as assuming passwords are selected entirely at random can significantly overestimate the protectiveness of PCPs.

### Strength by Category

Figure 3.2 shows the correlation between PCP strength and a website's Alexa global ranking. In general, we find that higher-ranked websites have stronger PCPs. Using Pearson's $r$ and log scales for both rank and PCP strength, we find a medium effect size for entirely random ($r = -0.30$, $p < 0.001$), alphabetic-first ($r = -0.34$, $p < 0.001$), and numeric-first ($r = -0.34$, $p < 0.001$) strengths.

We found a statistically significant difference between strengths based on country for generation at random and alphabetic first generation, but not for numeric-first generation (one-way ANOVA—entirely random—$F(10, 259) = 1.87$, $p < 0.05$; alphabetic-first—$F(10, 259) = 2.05$, $p < 0.05$; numeric-first—$F(10, 259) = 0.29$, $p = 0.98$), We did not find any meaningful pairwise differences for the statistically significant results using Tukey's test. There was no significant difference based on use case (entirely random—$F(5, 263) = 1.04$, $p = 0.40$; alphabetic-first—$F(5, 263) = 0.59$, $p = 0.74$; numeric-first—$F(5, 263) = 0.40$, $p = 0.88$).

Figures showing strength differences based on country, global rank, and use case can be found in Appendix D. We also tested whether (i) ads, (ii) public usernames, (iii) or data breach history impacted PCP strength, finding no statistically significant differences.

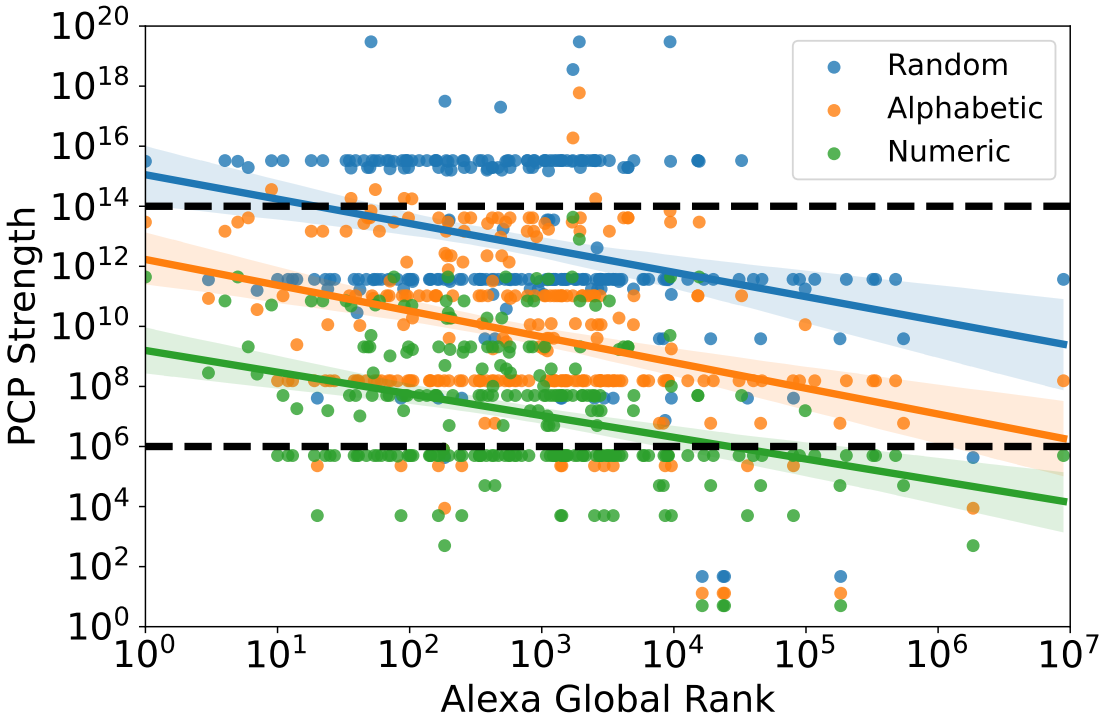



Figure 3.2: PCP strength by Alexa global rank

### 3.6.2 PCP Features

The most common minimum lengths for PCPs are 6 (128; 47%) and 8 (100; 37%) (see Figure 3.3a). Just over a tenth of PCPs (29; 11%) allowed passwords with fewer than 6 characters, with five (5; 2%) allowing passwords with a single character. These low length requirements are not only problematic for user-generated passwords but also for password generators, which are known to occasionally generate random but weak passwords at shorter password lengths [124].

Most PCP rules (195; 72%) set a maximum length for passwords, with a wide range of values (see Figure 3.3b). Just over a tenth (28; 10%) limit passwords to 16 or fewer characters, with four (4; 1%) limited to 12 or fewer characters.

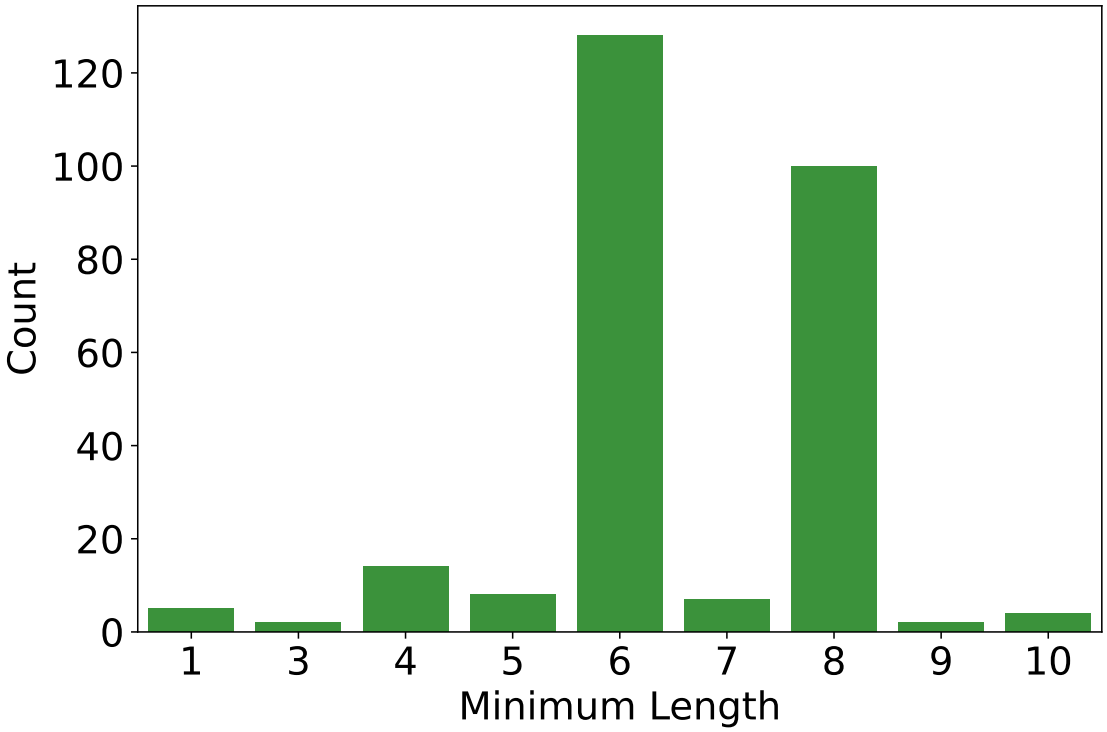
The next most common requirement was having required character classes (51; 19%): digits (42/51; 82%), alphabet (37/51; 73%), lower (12/51; 24%), upper (10/51; 20%), and symbols (4/51; 8%). This was followed by requiring a subset of character classes (43; 16%): at least one (5/43; 12%), two (14/43; 33%), or three (13/43; 30%) characters from all character classes; at least one symbol or digit character (9/43; 21%); at least one upper or symbol character (1/43; 2%); or at least one upper, digit, or symbol character (1/43; 2%).

The remaining requirements only appeared rarely. For prohibited substrings (11; 4%), websites primarily restriction personal information (10/11; 91%): name (6/11; 55%), email (5/11; 45%), username311, birthday211, website name (1/11; 9%). Rules also included max consecutive characters (9; 3%) with values of one (1/9; 11%), two (2/9; 22%), three (4/9; 44%), and seven (1/9; 11%). Finally, one PCP (1; 0%) required two lower case letters and two digits.

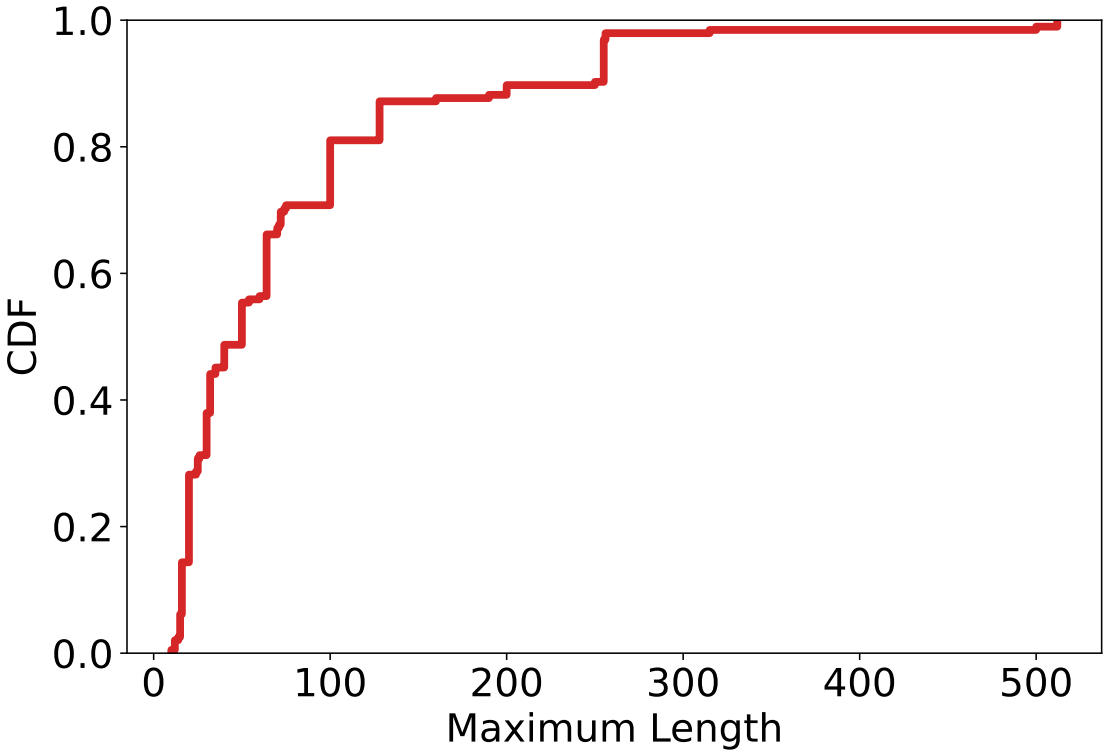
#### Multi-Rule PCPs

Of particular interest, we discovered three PCPs (3; 1%) that had more than one rule.

**gumtree.com.au** Required twenty-character passwords unless the password included an alphabetic character and either a digit or symbol, in which case ten-character passwords were allowed.




(a) Histogram of minimum lengths




(b) CDF of maximum lengths

Figure 3.3: PCP lengths

**github.com** Required fifteen-character passwords unless the password included both a lowercase character and a digit, in which case eight-character passwords were allowed.

**yy.com** Required nine-character passwords unless the password included an alphabetic character, in which case an eight-character password could be used. This could be to encourage Chinese users to pick non-digit-only passwords, which is common in that culture [101, 164].

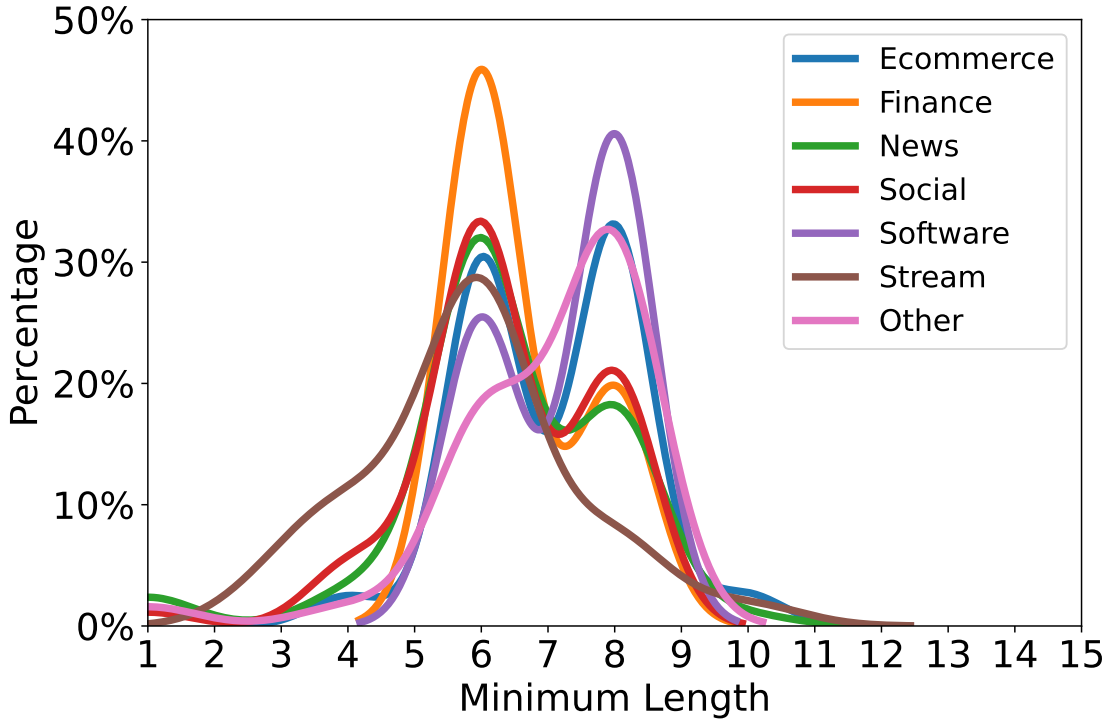
Ignoring specific requirements, these PCPs all share a common goal: allow users to choose between short but complex or long but simple passwords.

### Features by Category

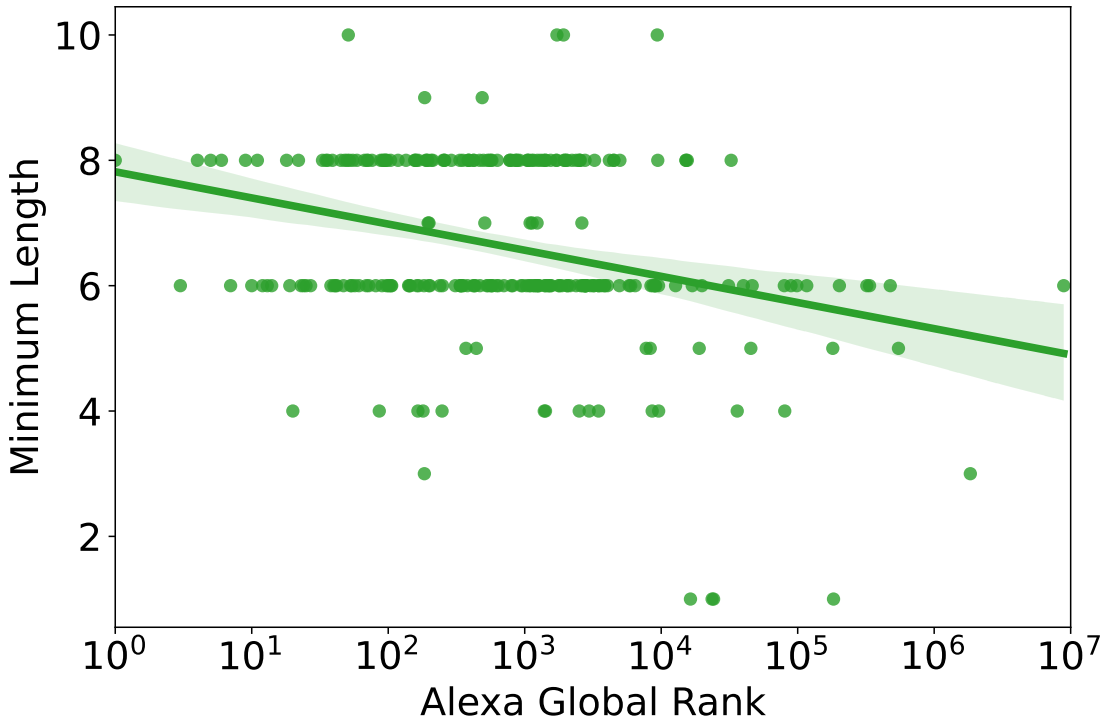
We find statistically significant difference for minimum length by country (one-way ANOVA—$F(10, 259) = 2.74$, $p < 0.01$), global rank (Pearson's-r—$r = -0.30$, $p < 0.001$), and use case (one-way ANOVA—$F(6, 263) = 3.57$, $p < 0.01$). Within these categories, high-ranked websites are much more likely to allow passwords shorter than six characters (see Figure 3.4b). Similarly, "streaming" websites have lower minimum length requirements (see Figure 3.4a), with the difference being statistically significant for "Ecommerce" ($p < 0.01$) and "Other" ($p < 0.05$).

We did not find statistically significant differences in maximum length by country (one-way ANOVA—$F(10, 259) = 1.05$, $p = 0.40$), global rank (Pearson's-r—$r = 0.01$, $p = 0.88$), or use case (one-way ANOVA—$F(6, 263) = 1.40$, $p = 0.21$). We did not see any meaningful difference for other restrictions, though we did not test for statistical significance.

Figures showing differences for minimum and maximum length based on country, global rank, and use case can be found in Appendix E. We also tested whether (i) ads, (ii) public usernames, (iii) or data breach history impacted PCP minimum and maximum length, finding no statistically significant differences.




(a) By use case




(b) By Alexa global rank

Figure 3.4: PCP minimum lengths

### 3.6.3 Website Analysis

We also examined the following items for each website: (a) whether account creation and login required HTTPS, (b) which SSO providers, if any, were supported, and (c) whether a password strength meter is shown to users.

For most websites (255; 94%) HTTPS was required to view the account creation and login pages. Still, there were fifteen (15; 6%) websites where we could access the account creation or login interface over HTTP.[4]

A third of websites (92; 34%) support at least one single sign-on (SSO) provider for account creation and authentication. The most popular SSO providers are Facebook (82/92; 89%), Google (65/92; 71%), Twitter (21/92; 23%), VK (10/92; 11%), and mail.ru (6/92; 7%), with the remaining 20 SSO providers being represented on fewer than five websites.

We find that just over a tenth (35; 13%) of websites show users a strength meter when they are creating passwords. We also find that just under a tenth (22; 8%) use a strength checker as part of their password policy—i.e., passwords must be a certain strength to be accepted.

#### Websites by Category

For websites whose account creation or login pages can be accessed over HTTP, the majority were in China: China (8/15; 53%), Russia (2/15; 13%), and one each (1/15; 7%) for India, Iran, Nigeria, Brazil, and the US. It is unclear why China is so different, but we find this correlation troubling. These types of websites are most likely to occur in less popular websites.[4]

Within certain countries we see much higher rates of adoption of SSO: Russian (11/19; 58%), Nigeria (6/13; 46%), Brazil (6/14; 43%), Australia (5/13; 38%), UK (3/8; 38%), India (3/9; 33%), Global (21/65; 32%), US (21/72; 29%), China (6/28; 21%), Iran (2/12; 17%). We also see a trend that the less popular sites are more likely to adopt 2FA: Top 10 (1/8; 13%), Top 50 (7/24; 29%), Top 100 (4/25; 16%), Top 500 (17/59; 29%), Top 1000 (6/25; 24%), Top 5000 (37/79; 47%), 5000+ (20/50; 40%). For categories, SSO is more evenly dispersed, though news (35/72; 49%) sites have higher support for SSO.

[4]The list of websites is given in Appendix C.

We do not find any meaningful effect from the categories on strength meters or internal strength checks for passwords.

## 3.7 Discussion

In this section, we discuss observations from our research.

### 3.7.1 PCP Recommendations

Of all the PCPs encountered in our analysis, we were most interested in the multi-rule PCPs, which allowed users to choose between short but complex or long but simple passwords. This ensures that passwords will resist offline attacks without causing unnecessary usability burdens. Moreover, this approach returns the locus of control to users—i.e., while PCPs are often viewed as restrictive, and therefore less usable [90, 155, 156], multi-rule PCPs give users a choice of which PCP is most appropriate for them. We hypothesize that by giving this control back to users, not only will they be more satisfied with the PCP, but they will also create stronger passwords. Future work could validate this hypothesis and try to determine what the ideal multi-rule construction is. For example, would more rules be even better, providing even more fine-grained control of the types of passwords users can select?

Another observation from our analysis is the importance of PCP design for ensuring the security of passwords not generated entirely at random. Whereas PCP requirements reduce the strength of passwords generated entirely at random (by shrinking the search space), they increase the strength of passwords generated with preferences to a given character class. Thus there is an interesting interplay between PCPs and passwords based on how they are generated. More specifically, we note that increasing length is the easiest way to improve strength, regardless of generation strategy. Similarly, we find that it is likely advantageous to limit users from having too much of their password be composed of digits (or symbols), as this significantly weakens those passwords and may lead to passwords vulnerable to online guessing attacks. As such, we recommend that administrators use a multi-rule approach that allows users to choose between long but simple passwords or short but complex passwords.

This allows machine-generated passwords to be short but ensures that human-generated passwords are strong enough to resist attack.

### 3.7.2 NIST Guidelines

NIST provides PCP guidelines (i.e., non-compulsory recommendations) for US companies and organization [62]. While our dataset includes a wealth of PCPs for global and non-US websites, we still think it is interesting to see which of these PCPs conform to the NIST guidelines.

We find that less than half of PCPs (106; 39%) meet NIST's recommended minimum length of eight characters. Similarly, we find that most (195; 72%) implement unnecessary maximum length requirements.

In line with NIST recommendations, most PCPs (177; 66%) do not have any composition requirements (this would be more positive if they met the minimum length requirements). Similarly, only a small fraction (8; 3%) reject specific symbols, which can be an indication of improper password hashing.

## 3.8 Comparison with related works

This section discusses comparison of our work with other related works on password generation, PCP languages, analysis of Web PCPs, and PCP usability. The comparison of our PCP with others in literature is shown in Table 3.3.

### 3.8.1 PCP Languages

Examining our data, none of the previous PCP languages(see §2.7.1) can encode all the PCPs in our dataset. However, these proposals could be extended to support the features identified in our research. During our PCP language generation process (see §3.2), our team built and tested several versions of our PCP language that were HTML- and XML-based. Ultimately, we rejected these approaches because our team felt that encoding policies in these languages was cumbersome and that the resulting policies were difficult to read. Still, the results of

Table 3.3: Comparison between PCP languages

| PCP Features | This paper | Daniel Bates [19] | Isiah Meadows [109] | Horsch et al. [74] |
|---|---|---|---|---|
| Define character sets | ✓ | ✓ | ✓ | ✓ |
| Multiple rule sets | ✓ | | | |
| `min_length` | ✓ | ✓ | ✓ | ✓ |
| `max_length` | ✓ | ✓ | ✓ | ✓ |
| `max_consecutive` | ✓ | ✓ | | ✓ |
| `prohibited_substrings` | ✓ | | ✓ | |
| `required` | ✓ | ✓ | ✓ | ✓ |
| `require_subset` | ✓ | | ✓ | |
| `charset_requirements` | | | | |
| `.min_required` | ✓ | | | ✓ |
| `.max_allowed` | ✓ | | | ✓ |
| `.max_consecutive` | ✓ | | | |
| `.required_locations` | ✓ | | | ✓ |
| `.prohibited_locations` | ✓ | | | ✓ |
| reverse indexing | ✓ | | | |

our user study show that there is significant room for improving our proposed language, and future work could explore integrating paradigms from these prior proposals with our language or testing whether, contrary to our team's perceptions, HTML- or XML-based would be better received than our JSON-based approach by developers. In this regard, the main contribution of our paper is the identification of features that must be included in such PCP languages.

### 3.8.2 Web PCP Analysis

For the most part, our results are similar to past findings of Florêncio and Herley [56] and Mayer et al. [107]. Overall, PCP strength (for random generation) is similar in all studies. However, as our improved strength calculation results in lower estimates of PCP strength, the similarity of our results suggests that PCPs have continued to get stronger over time, though that progress is slow and the delta is not that meaningful. When using PCP strength estimates based on random generation (as the prior work does), we find that PCP strength has become more bimodal, with a clear contrast between websites that require passwords to be offline-resilient and those that only require online-resilience. While this may only be an artifact of our increased precision in plotting PCP strength (the prior worked binned strength into large ranges), we do not believe so and think it is an area that could be explored more in future research. Like the prior work, we find that most PCPs reside within the online-offline chasm identified by Florêncio and Herley [57].

Like prior work, we find no statistically significant correlations when comparing PCP strength based on country, use case, public usernames, and past breaches. However, unlike the prior work, we find a correlation between a website's popularity and the strength of its PCPs. This difference is most likely explained by (a) our larger data set, (b) the increased fidelity of our strength estimates, and (c) the use of log adjusted strength and global ranks. Also, whereas prior work found a negative correlation between whether a website served ads and its PCP strength, we find no statistically significant correlation.

### 3.8.3 PCP Usability

Our research finds that length has the greatest impact on PCP strength for both passwords generated at random and using an alphabetic-first approach. As such, we echo prior recommendations for PCPs to focus on length as opposed to complexity. For those that want the best of both worlds, multi-rule PCPs can be used that allow short but complex or long but simple passwords, giving users the locus of control for this decision and thereby increasing usability. Similarly, due to the weaknesses of digit-first generated passwords, PCPs should likely restrict the usage of too many digits in a password.

## 3.9 Conclusion and Future Work

In this work, we developed a PCP language that websites and password managers can use to support the generation of compliant passwords. We hope that our work will signal to both communities that adopting a PCP language has tangible benefits. For websites, it allows them to unify their PCP specification and checking, allowing changes to the PCP file to automatically update how checking happens on both the client and server. For password managers, it not only improves the usability and utility of password management but also supports opinionated generation algorithms (e.g., mobile-aware generation [63], security-focused generation [124]), which would otherwise frequently generate non-compliant passwords.

While we are encouraged by the positive results of our user study, they also indicated that there is room for improvements. Future work could expand our PCP language by identifying and adding support for rarely used PCP features, such as restricting sequences of characters (e.g., "abcde") or keyboard patterns (e.g., "qwerty"). Similarly, our language could be enhanced to allow Unicode characters. Future research could also examine how to allow our PCP language to handle dynamic strings (e.g., usernames). One potential solution is to use placeholders in the `prohibited_substrings` requirement, providing appropriate values to the library at password validation. Finally, research could explore automatically identifying PCPs, both in whitebox scenarios, helping web developers identify their website's PCP, and blackbox scenarios, helping password managers identify PCPs for websites that do not publish

it, with care taken to avoid flooding servers with passwords guesses (approximating a DoS attack).

# Chapter 4

# Secure Browser Credential Entry Channel[1]

Password managers seek to improve the security of passwords by improving the usability of password generation, storage, and entry [123, 158], with the hope that doing so will encourage users to generate random passwords and avoid password reuse. Additionally, password managers allow users to audit the security of their stored passwords, detecting weak, reused, or compromised passwords. However, recent research has shown that users avoid using these features, [52, 139, 104, 125], limiting the security benefits of adopting a password manager.

Arising from this state of affairs, we ask the question, *is there anything that password managers can do to increase the security of passwords that does not rely on users to change their behavior?* In this chapter, we start to answer this question by investigating whether password managers can protect the passwords they autofill from theft. This includes preventing theft from web trackers [153], cross-site scripting (XSS) attacks [150], malicious browser extensions [85, 137], or compromised JavaScript libraries [129, 49] (i.e., a supply chain attack).

We start our investigation by exploring the design space for theft-resilient password entry. We identify five possible designs, evaluating each based on their security and usability. For security, we consider their ability to prevent theft by phishing, honest-but-curious scripts,

[1]This chapter is based on my work that is in process of being published. Prepublication paper of this chapter available at: https://doi.org/10.48550/arXiv.2402.06159

malicious scripts (e.g., XSS), and malicious browser extensions. For deployability, we consider whether these designs require changes to websites, extension APIs, or the browser itself.

Based on our analysis, we identify the following approach as providing the best mixture of security and deployability: When autofilling a password, the password manager instead autofills a fake password. It then provides the browser with (a) the fake password, (b) the real password, and (c) the origin bound to the password. The browser will then examine outgoing web requests for that web page, looking for the fake password in the web request body. If found, it replaces the fake password with the real password if and only if the password is being sent to the appropriate origin (as specified by the password manager). Critically, at no point is the real password ever contained in the DOM—preventing theft by client-side scripts—or within the web request bodies as seen by `webRequest` API—preventing theft by extensions.[2] Additionally, this design does not require modifying user behavior or websites.

To demonstrate the feasibility of this design, we forked and modified the Firefox browser to implement the above functionality. We also modified BitWarden, an open-source password manager, to work with our modified Firefox implementation. Importantly, our proof-of-concept implementation does not require modifying user behavior or websites.

We empirically evaluated the security of our design, demonstrating that it stops password exfiltration, both by DOM- and extension-based adversaries. Moreover, we describe how to prevent attacks by adversaries who are aware of this defense and may try to subvert it. We also evaluate our tool on the 573 sites with login forms from the Alexa Top 1000, showing that it is compatible with 97% of those sites. For the remaining 3% of websites, it is easy for the password manager to revert to the existing behavior, preventing any functionality regression.

We conclude the paper by discussing lessons learned from our design exploration and implementations. We also describe potential avenues for future research that, similar to our work, could modify the browser to add first-class support for authentication.

In summary, the contributions of our paper are as follows:

[2]This design is inspired by the work of Stock and Johns [160]. A comparison to their work is described later in this paper (§4.1.2).

1. **Threat model identification and design space exploration.** We identify a threat model for theft-resilient password entry. Using this model we identify five different approaches, evaluating each based on security and deployability. Several of these designs are inspired by work from Stock and Johns [160], though we expand on this work, demonstrating limitations in the original proposal and then showing how those limitations can be addressed in different ways to create three of the five designs.

2. **Proof-of-concept implementation.** We create a proof-of-concept implementation of the design we believe has the best combination of security and deployability. This includes both a modified version of the Firefox browser and the BitWarden password manager. While the final code diff is rather straightforward, creating it was not. Doing so required hundreds of hours of engineering effort and many discussions with Firefox developers, many of whom initially believed that what we were proposing would require too much of changes all across the codebase.

3. **Real-world evaluation.** We conducted empirical evaluations to demonstrate that our tool did not interrupt normal authentication flows, including authentication to websites we created and 554 out of 573 websites pulled from the Alexa top 1000 list. Additionally, we implement proof-of-concept password-theft attacks for malicious client-side scripts and browser extensions, demonstrating that these attacks worked without our proof-of-concept implementation but were stopped by our implementation.

## 4.1 Background

First, we describe the password entry workflow that is being secured in our work. Next, we compare our work to Stock and John's work [160] as our work is inspired by theirs. We then conclude with background on how browsers function, including form submission, the `webRequest` API, and browser extension permissions.

### 4.1.1 Password Entry Workflow

The password entry workflow can be split into the following steps:

1. The user visits a web page that has a `form` with an `input` element to enter their password.

2. The user enters their password by (a) manually typing it, (b) autofilling it from their password manager, or (c) copying and pasting it from their password manager. At this point, the password is stored within the web page's DOM.

3. The user submits the form. This causes the browser to process the form and send a web request to the server that contains the entered password.

4. The password is transferred over the network (preferably using a TLS connection).

5. The server receives and processes the password as it deems appropriate.

Note this flow also applies to account creation, which has the same general process for entering and transmitting passwords.

### 4.1.2 Relation to Stock and John's Work

Our desire to investigate how password managers could be used to strengthen password entry (as opposed to generation and storage) was motivated by the excellent work of Stock and Johns [160]. In their paper, Stock and Johns propose having the password manager inject a random value in place of the password, that will only be replaced with the real password during network transmission. Design #4, as described in §4.3 is based directly on Stock and Johns' proposal, with only a few minor tweaks. As such, it is natural to ask what scientific contributions this paper makes compared to that paper. Below we summarize the key ways we extend Stock and Johns' work:

- We identify and describe the threat model for securing autofilled passwords (§4.2).

- We perform a design space search for solutions to securing autofilled passwords (§4.3). In this process, we include Stock and John's proposal, two other approaches we create inspired by Stock and Johns' proposal, and two unrelated approaches. We evaluate and compare each design's strengths and weaknesses, demonstrating why our proposed

Design #5 is more secure and functional than other designs, including that of Stock and Johns.

- Our Design #5 (§4.3.3) protects against malicious extensions, something not possible for the design proposed by Stock and Johns. We also evaluate browser extensions found in Chrome Web Store to demonstrate the feasibility of this threat (§4.1.3). Finally, we empirically demonstrate that our implementation protects against this threat (§4.5.1).

- Our security evaluation considers how attackers would try to circumvent the proposed design using a reflection attack and discuss how password managers could mitigate these risks (§4.5.1). Stock and Johns' work did not consider how an adversary aware of the defense could circumvent it.

- Our solution is functional in modern browsers (§4.5.2). In contrast, Stock and Johns' proposed solution relies on functionality intentionally removed by browser makers. While our paper might make it seem like the idea to move this functionality deeper into the browser is obvious, the lack of any such proposals suggests that it may not be so. Moreover, as we found in our efforts, modifying the browser is challenging for researchers and practitioners with many potential pitfalls. Thus our implementation, including the code we created, is beneficial both for other researchers to build upon and for browser makers to adopt.

### 4.1.3 Browser Background

Below, we describe how the browser handles the webRequest API and extension permissions. These are each important parts of our design space exploration and proof-of-concept implementations.

#### `webRequest` API

The `webRequest` API is a browser extension-only API that lets extensions read, modify, or cancel web requests and responses. To access this API, extensions register event listeners for one or more stages of the web request processing lifecycle (see Figure 4.1).

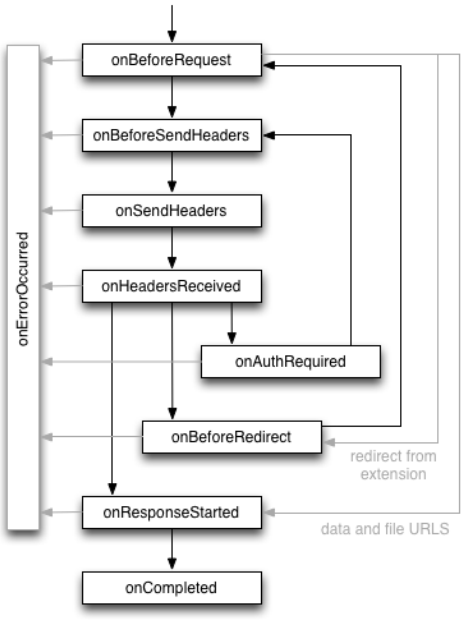



Figure 4.1: Web Request API flow [27]

In this lifecycle, the `onBeforeRequest`, `onBeforeSendHeaders`, and `onSendHeaders` events happen before the request is sent and provide access to the request body, allowing passwords included there to be exfiltrated. The `onBeforeRequest` and `onBeforeSendHeaders` allow extensions to redirect or cancel the request, while the latter also allows modification of request headers. The remaining stages are focused on what happens after the request has been submitted and lacking access to the request body is less important in the context of secure password entry.

Critically, none of these stages allow the request body to be modified. By the time `onBeforeRequest` is reached, the form submission process has already created the `FormSubmission` object, and the extensions are only provided with a read-only copy of the data in this object (not the object itself). The reason behind not allowing the request body to be changed is unknown, but requests to add this functionality have gone unfilled over the last decade and there is little evidence that it will be added [28, 114]. Interestingly, functionality allowing the modification of request bodies existed in older browser versions (e.g., Firefox, Safari, Internet Explorer) but was removed when these browsers migrated the Chromium model for browser extensions.

While the request body cannot be modified, the response body can be modified in the `onResponseStarted` stage. This allows extensions to arbitrarily inject client-side scripts into the web page. However, this ability to modify response bodies requires additional permissions (`declarativeNetRequest`) on top of those required to read the body.

### Extension Permissions

There are a couple of limitations on what a malicious browser extension can do. Most importantly, all extensions are given a unique origin, preventing one extension from accessing the data or scripts of another extension. As such, the browser's same origin policy prevents malicious extensions from directly accessing each others' data or scripts. The only way for a malicious extension to gain access to the passwords stored by a password manager is for that manager to copy those passwords to an origin where the malicious extension has access (i.e., the webpage).

The second limitation is that for a malicious extension to exfiltrate passwords, they need to request certain permissions in their manifest file [34]:

- The `scripting` permission allows the injection of client-side scripts on any web page [31].
- The `activeTab` permission allows the injection of client-side scripts on the root web page of the actively focused tab [29].
- The `content script` attribute injects a specified client-side script into webpages with a matching origin (wildcards are allowed) [33].
- The `declarativeNetRequest` permission allows the modification of web response bodies, which can be used to inject client-side scripts [30].
- The `webRequest` permission allows reading web request bodies (which can contain passwords) [32].

For an extension to be granted the permissions listed in its manifest, the user will need to approve these permissions during installation or when they are first used. However, research has shown that users struggle to understand these permissions and are likely to grant them without much consideration [91]. Still, permissions are useful. If an extension is compromised through a supply chain attack, the attacker will not be able to change the manifest and will be constrained by the extension's existing permissions.

To understand the prevalence of these permissions in existing extensions, between August 28, 2022, to September 13, 2022, we collected 101,414 browser extensions from the Chrome webstore. We then extracted and analyzed their manifest files. We also estimate user counts based on data fetched from CRXcavator[3] (pulled May 03, 2023).

We identified 12,576 extensions that can inject client-side scripts on any web page, including 55 with at least 500,000 users. We also identified 4,169 extensions that can read any web request's body, including 19 with at least 500,000 users. Of these extensions, 1,410 can read any web request's body but not inject client-side scripts, including 6 with at least 500,000 users.

[3] `https://crxcavator.io/`

## 4.2 Threat Model

Our threat model focuses on password managers and the exfiltration of users' passwords during the password entry workflow. This exfiltration can occur (i) after the password has been entered and is stored in the web page's DOM, (ii) as the authentication web request is processed in the browser, (iii) when the password is transmitted over the network, and (iv) when the password is received by the server. Since we focus on securing password entry for password managers, we do not consider the case where the user manually types the password or how the server handles the password after the server receives it (e.g., whether they store passwords salted and hashed).

In our threat model, we are concerned with the following adversaries:

1. **Honest-but-curious entities** (e.g., web trackers [153]). This entity can read the contents of any DOM element, including the password after it has been entered into the web page. Unlike the other adversaries, as an honest party, this adversary will not circumvent defensive measures designed to protect the password.

2. **DOM attacker.** This adversary has full control of the web page's DOM, able to read and modify the DOM at will. They can steal the password if it is ever included in the web page's DOM. They can also attempt to trick password managers into autofilling passwords outside the legitimate login page [157, 160, 123]. However, this adversary's capabilities are limited by security primitives built into the browser—they cannot violate the same origin policy (cannot directly access the password manager's data) and do not have access to web requests or responses that the attacker did not generate.

3. **Extension attacker.** This attacker can read the headers and body of all web requests and responses, allowing them to steal any passwords in web request bodies (as visible through the `webRequest` API). While a malicious extension can also inject a client-side script, this capability is already covered by the DOM attacker, so when talking about the extension attacker, we are focused on their ability to read web requests and responses.

4. **Man-in-the-middle (MitM) attacker.** This is a network attacker who sits between the user's device and the server. They can eavesdrop and modify network traffic as it is

transmitted between the two. As we evaluate the security of password entry defenses, we consider two variants of this adversary: one that cannot break TLS-encrypted communication and one that can (e.g., an attacker who leverages a substitute certificate attack [130]).

5. **Phisher.** This adversary has full control of the web page visited by the user and the server where the password will be sent, being able to steal the password if it is entered into the web page or transmitted to the server. While this attacker must convince a user to visit the phishing web page, we assume this is feasible [3]. For simplicity, we combine **compromised websites** with phishers as the two attackers have the same capabilities.

In addition to these adversaries, we are also concerned with a **supply chain attacker**. This attacker compromises a library used by a server, a client-side web page, or a browser extension. At this point, the attacker has the same capabilities as a phisher, DOM attacker, and MitM attacker, respectively. Thus, we don't evaluate this adversary separately from those adversaries. However, we mention them here to emphasize the feasibility of the other adversaries, as supply chain attacks can put users' passwords at risk, even if the user exercises extreme caution (e.g., vetting all links, disabling client-side scripts, or vetting all extensions).

In our threat model, we intentionally consider a wide range of adversaries to better evaluate the design space for password autofill defenses. However, we consider several threats as out of scope. First, we consider compromised browsers and operating systems as out of scope. If these items are compromised, there is no need to steal credentials during the autofill process, as they can simply be stolen from memory after the password manager decrypts them.

Second, we do not consider session hijacking attacks. Poor cookie hygiene (e.g., failing to use `HTTPOnly` cookies) can allow session theft by any of our identified adversaries. However, existing defenses can prevent this attack for any adversary, up to and including a malicious extension (through token-bound cookies [142]).

Even if we exclude malicious extensions that can exfiltrate session cookies (using the `cookie` permission), there are still 11,452 extensions that can inject client-side scripts, 3,568

that can read requests' bodies, and 1,294 that can read web requests' bodies, but not inject client-side scripts. This indicates that even with this carveout there are a substantial number of extensions that satisfy our threat model.

Finally, we note that while malicious extensions may appear similar to a compromised browser, this is far from the case. First, most extensions have limited permissions, limiting the damage they can do when compromised. Second, even after granting a malicious extension every possible permission, it is still sandboxed by the same origin policy, meaning it cannot steal data stored by other extensions (i.e., the password manager). For these reasons, we believe it is reasonable to consider malicious extensions in scope but compromised browsers out of scope.

## 4.3 Design Space Exploration

Based on our threat model, we explored the design space for implementing password entry defenses in password managers. A review of the password manager and password exfiltration literature guided this exploration. Repeated discussions within our research group also informed it. In total, we identify five high-level designs for securing password autofill.

We then evaluate these designs along two axes: security and deployability. For security, we consider whether these designs could survive attacks by the attackers identified in our threat model. For deployability, we consider whether these designs avoid changes to websites and the browser. Avoiding changes to websites is critical, as requiring all websites to adopt new technology is unlikely to succeed, as can be seen with the lack of adoption for many proposed authentication technologies [21]. Avoiding changes to the browser is also ideal, though it is easier to change browsers than all websites. Moreover, we also distinguish between changes to the browser that affect the UI processes (similar to user-space processes) and those that affect the core browser process (similar to the kernel process).

A summary of our evaluation is shown in Table 4.1. At the bottom of this table, we compare the identified designs against existing approaches for security authentication. For password manager autofill, we note that there is partial protection from phishing attacks as the manager should not autofill passwords on phishing websites [123, 122]. However, the user

Table 4.1: An evaluation of the five designs based on security and deployment. Also includes an evaluation of 2FA as a comparison point.

| | Security | | | | | Deploy | |
|---|---|---|---|---|---|---|---|
| Design | Protection from honest-but-curious entity | Protection from DOM attacker | Protection from extension attacker | Protection from MitM attacker | Protection from phisher | No changes to websites | No changes to browsers |
| 1. Zero-knowledge proof | ● | ● | ● | ● | ● | ○ | ● |
| 2. Modified form handling | ● | ○ | ○ | ○ | ◐ | ● | ◐ |
| 3. JS-based nonce injection | ◐ | ◐ | ○ | ◐ | ◐ | ● | ● |
| 4. API-based nonce injection | ● | ● | ○ | ◐ | ● | ● | ◐ |
| 5. Browser-based nonce injection | ● | ● | ● | ◐ | ● | ● | ○ |
| Current password manager autofill | ○ | ○ | ○ | ○ | ◐ | ● | ● |
| Two-factor authentication (2FA) | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| Phishing-resistant 2FA | ◐ | ◐ | ◐ | ◐ | ◐ | ○ | ○ |

● Fully achieves the property

◐ Achieves the property with some limitations

○ Fails to achieve the property

can still copy and paste passwords into password into the phishing website. We also compare against 2FA (discussed at the end of this section) to highlight that moving to 2FA does not solve this issue as some might assume.

### 4.3.1 Design #1: Zero-Knowledge Proof

The most secure approach for password authentication is using zero-knowledge proofs (ZKPs), where the user does not reveal their password to the server. Examples of this approach are augmented password-authenticated key exchange (PAKE) protocols [172, 81].

In this design, websites add an endpoint supporting authentication using a ZKP. This endpoint is not a web page, but rather a method by which the password manager could directly authenticate on the user's behalf using the password stored in the manager. Once authentication succeeds, the password manager sets a session cookie for the domain to create an authenticated session. Notably, the password is never placed in the DOM.

**Security evaluation.** As the password is never present in the DOM, it cannot be stolen by attackers that rely on DOM-based exfiltration (honest-but-curious entity, DOM attacker). Similarly, the zero-knowledge nature of the authentication protocol means that no information sent over the network can be used to derive the password, preventing this avenue of exfiltration (extension attacker, MitM attacker). Finally, even if a ZKP is performed with a phisher, they will gain no knowledge of the password, protecting against phishing attacks.

**Deployability evaluation.** This design requires all websites to add an endpoint for conducting authentication using a ZKP, which is likely a non-starter. Note, PAKE-based ZKP for passwords have existed for decades but have never seen widespread adoption [68]. On a more positive note, this design could be implemented with existing browser functionality.

### 4.3.2 Design #2: No-Script Form Attribute

To stop DOM-based exfiltration of passwords, the browser could add support for a new attribute on forms or input elements (e.g., `noscript`) that prevents scripts from accessing the values stored in that form or input element. The password manager would set this attribute

on password fields before autofilling the password. At a high level, this approach is similar to the shadow DOM [115] that shows one view of the DOM to the user and another to scripts on the page. From the user's, web page's, and password manager's perspective, everything would work as it always has, except with an extra layer of password protection.

**Security evaluation.** This approach would protect against an honest-but-curious entity, as they would not have access to the protected value and would not try to circumvent this defense. However, this approach provides little to no security for the other attacks. First, a DOM attacker could simply create look-alike forms, display those over the actual form (either by removing it or using a higher z-index), and then steal passwords from the unprotected form. Second, the password is still sent over the wire allowing access by the extension attacker, the MitM attacker, and the phisher (assuming the user copies and pastes the password).

**Deployability evaluation.** Adding a form attribute will require modifying the browser's DOM code (UI process) but not code that executes in the browser's core process. As the manager is responsible for updating forms with this new attribute to form elements, no changes are needed to websites. However, if the web page relies on scripts to access and submit the authentication request, as opposed to letting the form do so itself, the web page's functionality could break.

### 4.3.3 Design #3–5: Nonce Injection

Nearly a decade ago, Stock and Johns [160] evaluated the security of password manager autofill, finding that managers autofilled passwords into malicious websites under many conditions. At the end of their paper, they proposed a possible solution to this problem:

1. When an autofill operation is triggered, a random placeholder for the password (*a password nonce*) is generated.

2. The password nonce is autofilled into the web page.

3. The manager will scan outgoing web requests looking for the password nonce.

4. If the password nonce is detected while being transmitted to an origin that matches the origin for the real password, the manager will replace the nonce with the password.

While the implementation approach proposed by Stock and Johns is not possible (browser extensions cannot modify web request bodies), the core idea remains sound. Inspired by Stock and Johns' proposal, we identified three designs that autofill password nonces to secure password entry. Each of these approaches has different security and deployability trade-offs, with Stock and Johns' proposal aligning with Design #4.

**Design #3: JavaScript-Based Nonce Injection**

In this design, the password manager will inject a script into the web page when autofilling the password nonce into the web page. This script has the following responsibilities:

1. If the password form already has an `onsubmit` method, the script will store this method.
2. Store any existing submit event listeners, then remove all these event listeners from the form.
3. The script will set the `onsubmit` method for the form. The new `onsubmit()` method will perform the following operations:
   (a) Call the stored `onsubmit` method, if any.
   (b) Call any stored submit event listeners.
   (c) Replace the password nonce with the actual password as long as the form data will be submitted to an appropriate origin.
4. Modify the DOM so that it is not possible to replace the new `onsubmit` method. If another script attempts to do so, the `onsubmit` method stored by this script will be replaced instead.
5. Modify the DOM so that event listeners can neither be added nor removed from the form, with attempts to do so simply updating the list of event listeners stored by this script.

Items 1–2 and 4–5 are necessary to ensure that no other scripts run after the replacement of the actual password occurs.

**Security evaluation.** This design protects against DOM-based exfiltration in so much as it can ensure that no other scripts can access the DOM after the submission occurs. While we describe how this could be done, there is no guarantee that this approach will always work. For example, the browser could change the form submission process, adding new avenues for scripts to run after the password replacement occurs. In this case, the password might even leak to an honest-but-curious entity. Alternatively, malicious scripts could look for ways to prevent the password manager's script from preventing modifications to the `onsubmit` method or event listeners (which run after `onsubmit` by default). While the manager could then update the script it injects to handle these, this sort of cat-and-mouse situation is never ideal. As such, we rate this design as providing limited protection against honest-but-curious entities and DOM-based attackers.

The real password is still sent in a web request, allowing exfiltration by an extension attacker (see Figure 4.2). Additionally, a MitM attacker can also steal the password during transmission. While password managers could only replace passwords if they are going to be sent over a valid TLS connection, the connection is often not properly implemented [123] and would not protect against a MitM attacker that can attack TLS connections.

Against a phisher, this design has the same properties as existing password entry using a password manager—the password will not be autofilled on a phishing website. However, if the user copies and pastes it, the phisher will still get the password. It is not possible to copy a password nonce, as the manager will have no way of knowing on which form to register the replacement of that nonce (though it could try to guess with varying success).

**Deployability evaluation.** This approach requires no modification of anything but the password manager. From this perspective, it is the most deployable of all the designs we identified. Still, like Design #2, this approach could break websites that rely on scripts to submit the password instead of the form.

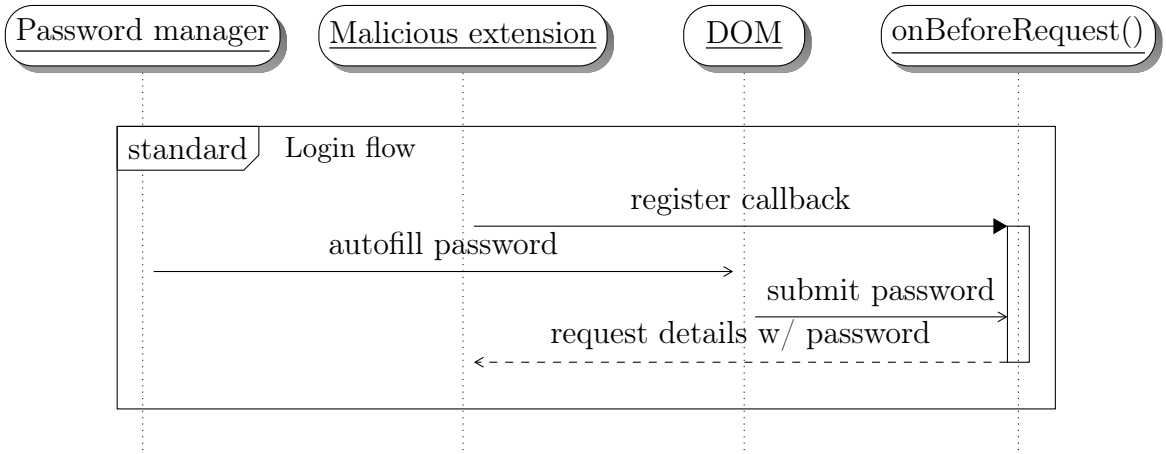



Figure 4.2: Diagram illustrating how an attacker can use an `onBeforeRequest` callback to exfiltrate passwords.

### Design #4: API-Based Nonce Injection

Design #3 can be improved by changing where nonce replacement happens from the beginning of the form submission process (i.e., in the webpage) to the browser's internal form submission process.This approach has several key benefits. First, as it happens outside the web page, no DOM attacker-controlled scripts can interfere with the process. Second, it will catch all web requests, not just those created through form submission, providing better support if the website submits the values using a client-side script.

To implement this functionality, the `webRequest` API is used. While this API does not support web request body modification and likely never will (see §4.1.3), it is possible to achieve the desired functionality at the `onBeforeRequest` stage. At this stage, the web request's body has not yet been created. As such, the browser can be instructed to modify its internal copy of the form data (not DOM-accessible), replacing the password nonce with the real password (if the origin is appropriate). Finally, the generated web request body will contain the user's password. This design most closely aligns with the design proposed by Stock and Johns [160].

This design effectively adds the ability to modify web request bodies. Allowing such modifications could introduce new security and performance issues [28, 114]. Thus, the browser should control the replacement rather than the password manager (i.e., the extension). Ideally, the extension would list the following items when requesting a replacement: *(i)* the password nonce, *(ii)* the replacement password, *(iii)* the origin (scheme, host, port) associated with the password, *(iv)* the name of the field that was autofilled with the password nonce.

The browser will search through the key-value pairs in the `FormSubmission` object, making the requested substitution if and only if *(a)* the indicated field's value matches the password nonce exactly, *(b)* the web request will be sent to the indicated origin. Requiring (a) may lead to some compatibility issues if a script creates a web request using a different field name, though we didn't encounter this in our testing (see §4.5). If necessary, this requirement could be dropped, though it might have unexpected consequences.[4] Additionally, because the replacement happens within the `FormSubmission` object, proper sanitization and encoding

[4]We could not identify any security issues that couldn't already be caused by malicious client-side scripts.

of the key-value pairs will occur, preventing the substitution of one value from being later treated as a different value or multiple values in the request body.

Finally, we note that unlike Design #3, this design allows copying and pasting of password nonces from the password manager. As the web request API examines all outgoing web requests, replacement of nonces is still possible. Still, there will need to be functionality allowing users to copy real passwords if they are to be entered outside of the browser (though this can be made harder to activate, incentivizing the use of nonces where possible).

**Security evaluation.** As the replacement of the password nonce happens after any client-side scripts are allowed to execute and the actual password is never included in the DOM, neither the honest-but-curious entity nor the DOM attacker can exfiltrate the password from the DOM. The DOM attacker could change the destination to which form data will be sent, but this will result in an origin not associated with the password, preventing replacement.

As with Design #3, the actual password is still visible in the web request body, allowing exfiltration by an extension attacker (see Figure 4.2). Also, a MitM attacker can steal the password during transmission. Unlike Design #3, this design provides strong protection against a phisher. Not only will autofill be prevented but even if a password nonce is copied into the phishing website, it will not be replaced as it will not have the appropriate origin.

**Deployability evaluation.** This design does not require any changes to websites. It does require changes to the extension API and the form submission code (both executed in the UI processes), but not to code that runs in the browser's core process. It requires changes to the browser's content code but not its core code.

#### Design #5: Browser-Based Nonce Injection

To protect against malicious extensions that can read web request bodies, the replacement code must execute after web requests are last allowed to see the web request body. Our investigation determined that this is best achieved by moving the replacement code into the browser's networking code. While this has significant implementational challenges (the

network code is in a different security domain from where extensions operate), it protects against malicious extensions.

In this design, password managers register autofilled nonces with the browser. Then, when the browser's networking code is about to send the web request to the operating system for transmission over the network, it first scans for any registered nonces. If it finds them, it asks the managers for replacement credentials. At this point, replacement happens as it did in Design #4, with the manager listing the actual password, the origin, and the field, and the browser only making the changes if all these values match as expected. Note, even though this replacement happens after the web request body has been formed, modifications still happen through an object, ensuring proper sanitization and encoding occur.

**Security evaluation.** As with Design #4, and for the same reasons, Design #5 is impervious from DOM-based password exfiltration (honest-but-curious entity, DOM attacker). As the password nonce replacement happens after extensions can view the web request body, this design protects against malicious extensions. It performs the same as Design #4 and for the same reasons regarding MitM attacks and phishers.

**Deployability evaluation.** This design does not require any changes to websites. It does require changes to the code that executes in the browser's core process but not to any code executed in the UI processes. Most standard browsers are built on standard HTML specifications using WebIDL [169], so deployment into other browsers would be very simple.

### 4.3.4 Discussion

Examining the five designs, we see that all five designs improve upon the security of the current password entry process used by password managers. Of these, Design #1 (zero-knowledge proofs) has the best security. However, its reliance on websites supporting this technique likely means it is a non-starter.

Looking at the remaining designs, Design #3 stands out as requiring no changes to websites or the browser. While it has limited security benefits, it still does better than current practices and is certainly something password managers could explore.

However, we find Design #5 to be the most compelling. While it does require modifying the browser, it comes the closest to zero-knowledge proofs in terms of security, theoretically preventing credential exfiltration in all cases except against a MitM attacker who can compromise the security of TLS (a high bar). As such, this is the design we chose to implement and discuss for the remainder of the paper.

Lastly, we compare these designs against two-factor authentication (2FA) since some may consider these a solution to the problem of password exfiltration. Two-factor authentication does not do anything to prevent password exfiltration, only limiting the impact of that exfiltration [21]. Research has also shown that many 2FA schemes are vulnerable to the exfiltration of the codes created by the something-you-have factor [48, 161]. While there are approaches to secure 2FA against phishing of the secondary factor, this still does not address the issue of password theft. While it does lessen the impact of a stolen password, it does not remove it, as the stolen password (or a close variant) may be used on other sites. From a deployability standpoint, 2FA requires more changes than any of our proposed solutions, suggesting they could see faster and more widespread adoption than 2FA.

## 4.4 Implementation

We implemented Designs #3–#5 to demonstrate their feasibility and to perform security evaluations. In this section, we provide the implementation details of Design #5, which we found to be the most effective solution.

To implement Design #5, we modified Mozilla Firefox 107.0 and the Bitwarden password manager. We added a write-only `onRequestCredentials` API to the browser that enables the password manager to detect the submission of an inserted nonce to the server, replacing it with the appropriate credential just before the request is sent over the wire and after other extensions could read the modified request body. A diagram of the modified flow is given in Figure 4.3.

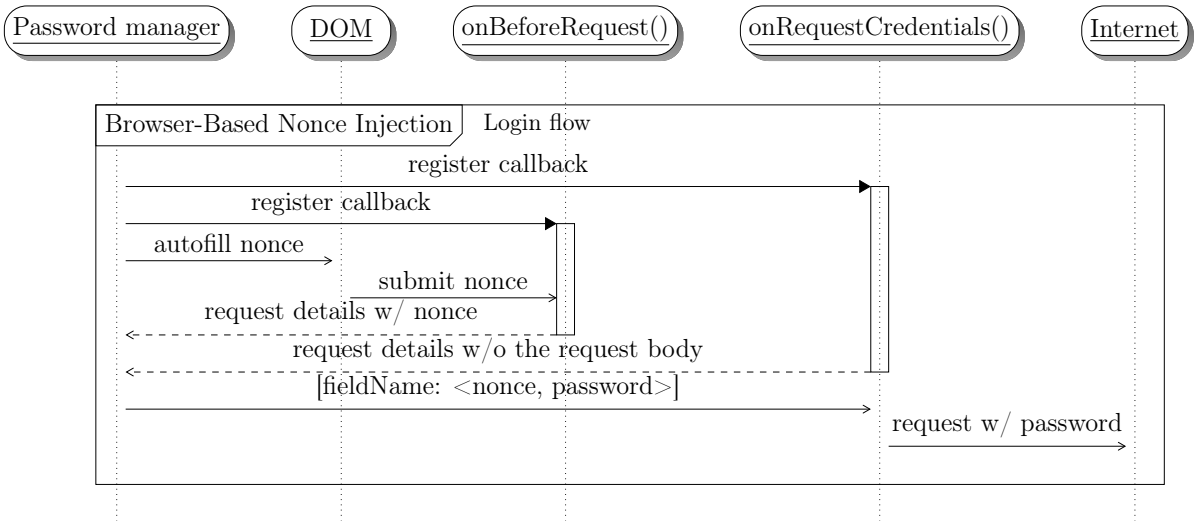



Figure 4.3: This diagram gives the flow for autofilling and replacing nonces as implemented by Design #5.

### 4.4.1 Getting Setup

First, the password manager will register two callbacks, one for `onBeforeRequest` and one for the new `onRequestCredential`. While we could have collapsed the functionality of both of these callbacks into `onRequestCredential`, doing so would have required a larger change for how callbacks are handled in the `webRequest` API. As this could lead to unintended regressions, we decided on the approach that changed fewer lines of code in the browser, even if it led to a few more lines of code in the password manager implementations.

Next, as needed, the password manager will autofill a nonce in place of a password. After doing so, it will internally store an association between the web page and (i) the nonce, and (ii) the name of the field storing the nonce, and (iii) the password manager entry that is being autofilled

### 4.4.2 `onBeforeRequest`

Eventually, the page will be submitted and a `webRequest` will be created by the browser, with this `webRequest` containing the autofilled nonce. This will cause the callback registered with `onBeforeRequest` to be triggered. Within this callback, the password manager will be able to see the details of the `webRequest`, including the destination URL, the HTTP method used, and the request body. Using this information, the password manager determines if (a) there is a nonce associated with the web page that generated the `webRequest`, (b) the nonce is in the request body, and (c) replacing the nonce would be safe. If all these checks pass, the password manager will internally store an association between the `webRequest` and (i) the nonce, and (ii) the name of the field storing the nonce, and (iii) the password manager entry that is being autofilled

Based on recommendations from the research literature [160, 102, 157, 123, 122] and to prevent a reflection attack discussed in §4.5.1, we recommend that password managers make the following safety checks:

1. Check that the web page is not displayed in an `iFrame`.
2. Check that the submission channel does not use HTTP or an insecure HTTPS connection.

3. Check that the origin (protocol, domain, port) matches the origin identified by the password manager entry being autofilled. Even better, the password manager can store the exact URL where credentials should be submitted, only submitting passwords to that URL.[5]

4. Check that the nonce is not in the GET parameters.

5. Check that the name of the field storing the nonce remains unchanged since the nonce was autofilled there (which itself should have been checked before autofilling [123, 122]).

### 4.4.3 `onRequestCredential`

Immediately before sending the `webRequest` over the wire, the callback registered with `onRequestCredential` will be triggered. This callback once again receives request details as input, though in this case the request body has been stripped out. This is necessary to prevent extensions from reading any nonce substitutions that might have been made by other `onRequestCredential` callbacks. In this step, if the password manager has a nonce associated with the current `webRequest`, it will simply return the associated nonce, password, field storing the nonce, and the URL that the password manager expects the password to be submitted to.

The password manager will then take this information and use it to locate and replace the nonce in the request body. To prevent unintended (or malicious) consequences from replacing the nonce, the browser will only make this substitution if,

i) The field name storing the nonce exactly matches the field name specified by the password manager.

ii) The field value exactly matches the nonce with no additional characters.

iii) The submission URL matches the origin (protocol, domain, port) of the URL provided by the password manager.

[5]Determining the exact URL could be done by having password managers store associations for popular websites, storing prior successful submission locations, or crowdsourcing the creation of associations.

While the password manager should have already checked these items before submitting the password to be replaced, we have the browser also check these items to ensure that they will be checked. This is an important form of defense in depth [122]. If any of these checks fail, no substitution is made, and the request will be sent still containing the nonce.

## 4.5 Evaluations

We evaluated our implementations of Designs #3, #4, and #5 in terms of security. For Design #5, we also implemented it in terms of functionality and overhead.

### 4.5.1 Security Evaluation

To evaluate the security of our implementations, we first created a basic PHP login page that takes in a username and password as form input and logs the credentials in plaintext for verification purposes. To test this web page, we would autofill with the password and then submit the web page.

To simulate a DOM-based attack, we injected JavaScript on the test web page that scrapes the password field before page submission. For this attacker, all three designs prevented the script from accessing the password. However, Design #3 only succeeded at stopping this attack because we simulated an attacker who was unaware of the defense. By simulating an attacker aware of Defense #3, it is trivial to steal the password.

To simulate a malicious extension, we built and installed a browser extension that logs all outgoing requests using the onBeforeRequest web extension API. For both Designs #3 and #4, the password was successfully stolen. On the other hand, Design #5 was successful in preventing password exfiltration, with only the nonces being exfiltrated. This is true even if we allowed the attack to register a `onReplaceCredential` callback (which has no access to the request body).

To further verify our results, we conducted these same tests on the login pages for the top 10 Alexa websites. In each case, our results were the same as our initial test.

#### Reflection Attack

An attacker who is aware of Defense #5 could attempt to conduct a *reflection attack*, wherein they attempt to trick the browser into submitting the real password to a web page that displays those values to the user. For example, the attacker could change the field name for the password to "username" for web pages where an error is displayed when the username doesn't exist (e.g., "${username} doesn't exist"). Alternatively, the attacker can also modify the submission URL to point to a page where it is more likely that request body values will be reflected on the web page.

In each of these cases, the attacker is relying on the password manager and the browser to approve the replacement of the nonce even though the password will be sent to an unsafe location. To address this attack, our implementation requires the password manager to (i) specify the origin associated with the password and (ii) identify the field storing the nonce. If either of these values does not match, the nonce will not be replaced with the password.

For even better security, the password manager can store and check the exact URL and field name that should be used to submit the password. This completely foils this reflection attack. As several password managers are already checking these items before autofilling passwords [123], we believe it would be reasonable for them to implement these checks.

### 4.5.2 Functional Evaluation

To evaluate the functionality of our browser-based nonce injection implementation, we conducted tests on real websites to ensure that our implementation does not break their authentication flows.

Using the Alexa top 1000 sites from May 1, 2022, we ran a Selenium script that started at the root page of each domain and traversed all links on the page up to a maximum depth of three, search for login pages. In total, we identified login pages on 623 sites. After filtering for the subdomains of the same website, like Google and Amazon for different countries, we were left with 573 unique login pages.

For all 573 login pages, we used a Selenium script to submit credentials from both an unmodified Firefox browser and from a modified version of Firefox implementing Design

#5. As illustrated in Figure 4.4, we set up a proxy server to record all the outgoing web requests from the browsers and save the request body. We then compared the credentials in the authentication requests sent from both browsers.

In 554 of 573 cases (97%), there was no difference in the credentials submitted to the websites. 11 sites (2%) generated an integrity check value based on the autofilled nonce. In these cases, the nonce would still be replaced by the real password, but the integrity check value would no longer match. While we did not confirm that this would cause the login to fail (we didn't have actual accounts on these sites), we still consider these as failed cases. Finally, 7 websites (1%) either hashed or base-64 encoded the nonce, which prevented the password manager from replacing it with the real password.

While 100% compatibility would be ideal, *being able to improve the security of 97% is still a significant step forward.* In practice, password managers could keep a list of websites that don't support secure autofill and not use it for those websites, preventing any functionality regressions. Moreover, as our testing shows, identifying non-compliant websites can be fully automated, making the creation of such a list highly reasonable. Additionally, if our solution was adopted, we would hope websites would abandon their ad hoc attempts at secure password entry in favor of the more robust security offered by our secure password autofill system.

### 4.5.3 Overhead Evaluation

To measure the overhead incurred by our implementation, we examined the logs for HttpBaseChannel [116] generated during our functional evaluation of our modified browser. These logs give exact measurements for each stage of the `webReuqest` lifecycle, allowing us to pinpoint the processing time used by our code.

On average, the `webRequest` lifetime was 4.5222 seconds, with our code accounting for 0.443 seconds, representing 10.6% of the total request duration. This overhead can entirely be explained by the time needed to destroy and recreate the request stream's body when replacing the nonce with the passwords. In other words, there is no meaningful overhead when nonce replacement is not needed. As such, when a password has not been autofilled (the vast majority of cases) there is no meaningful difference in the time taken to submit a `webRequest`.

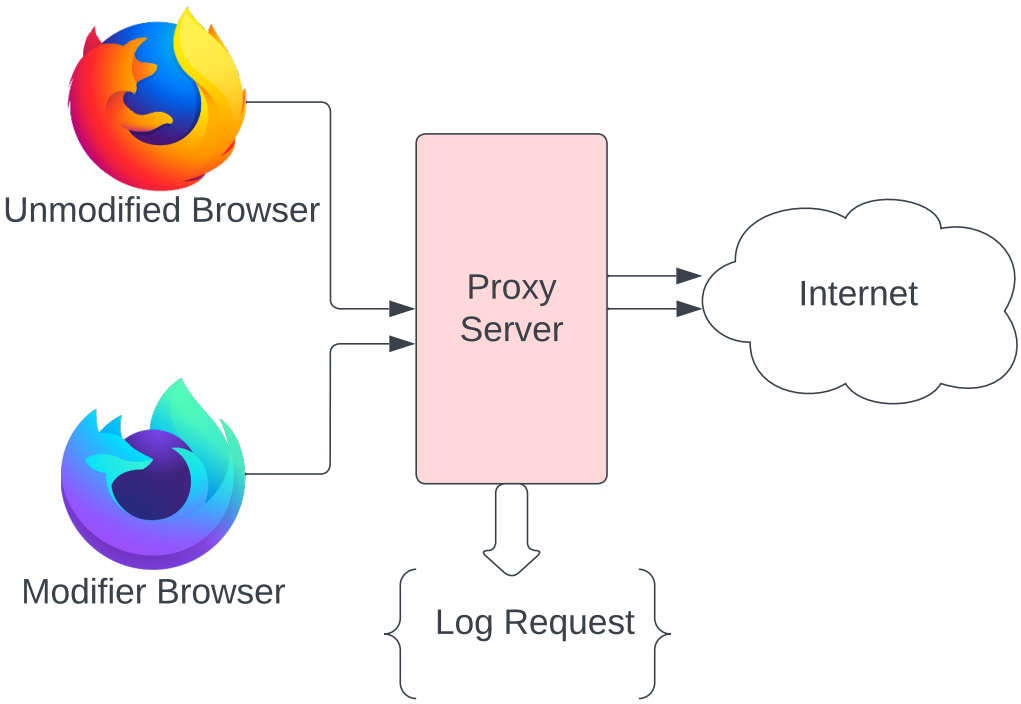



Figure 4.4: Functional Evaluation Architecture

## 4.6 Discussion

Below we discuss implications and limitations related to our design and implementation.

### 4.6.1 Deployment and Adoption

We worked with Mozilla Firefox developers for implementation details on our design ideas. With the future goal of merging the feature into Mozilla Firefox, we have made careful consideration for changes made in the code to be within acceptable standards for the Firefox repository, getting it reviewed with the developers wherever necessary. We hope that this will speed deployment of our design.

It is also important to consider that users often reject security advice if they perceive the effort required to implement it outweighs the extra security gained [73]. Our proposed defenses for improving the credential entry workflow do not require any changes to the login user experience, which may increase the likelihood of adopting these protocols. Additionally, our design only requires small changes to password managers and web browsers, which could be easier to deploy given that both entities have an interest in increasing users' security.

### 4.6.2 Securing Manual Password Entry

Protecting manual password entry is challenging, particularly as an attacker can inject client-side scripts (honest-but-curious entity, DOM attacker) that listen and record keystrokes (i.e., a key logger). One possible way to secure manual password entry would be to allow the browser to enter a special password entry mode using a conditioned-safe ceremony [87]. In this mode, the browser would record the user's keystrokes, replacing each stroke with one or more random password nonce characters. The browser would also prevent any scripts or extensions from recording keystrokes while the password entry mode is active. After the user finishes entering the password, they would leave the password entry mode.

While this approach could protect manually entered passwords, research would be needed to make sure it works in practice. First, care will need to be taken in selecting the conditioned safe ceremony to ensure that users always enable it when needed. Second, it will be necessary to design the password entry mode such that it is clear when it is activated [45]. Third,

research would be needed to explore how users could be made aware of and encouraged to use this functionality. Lastly, user studies would be needed to ensure that it has sufficiently high usability to encourage users to make use of this functionality.

### 4.6.3 Denial of Service for Nonce Injection

An attacker could detect nonces entered onto a web page's DOM and modify them. This would cause the authentication flow to fail, as the original nonce would no longer exist and the real password would never be substituted. After repeated failed authentication attempts, a user may assume that there is something wrong with their password manager and manually enter their password, opening their password up to exfiltration. It is important to note that this is not additional behavior due to our solution as extensions already have the capability to alter DOM input. Future work could explore this issue to measure user behavior when encountering a DoS attack and explore potential solutions to mitigate its impact.

### 4.6.4 User Confusion

User confusion is a potential issue with nonce injection defenses. If users view the autofilled password, they may realize it is not their password, especially if they don't use a generated password. This could lead to confusion. Although this issue may not be that prevalent because users are less likely to investigate passwords inserted through a password manager, it is still prudent to study the behavior of users when they encounter such confusion.

While the shadow DOM is not sufficiently secure to be used to show users their actual password [148], the browser could add a similar utility to securely display the real password to the user, without allowing that password to be accessed by malicious client-side scripts or extensions. Future research could also investigate this potential solution.

## 4.7 Conclusion

There are many avenues for adversaries to exfiltrate passwords: through client-side scripts, the `webRequest` API, during network transmission, and through phishing. In this paper, we

explore how the password manager can take an active role in protecting passwords from theft. To this end, we identify a strong threat model for password exfiltration. We recommend that future work on this topic also consider a threat model at least this strong.

Based on this threat model we conduct a design space exploration, and identify five potential methods that password managers can use to secure password entry. Of these, the most secure design involves having password managers authenticate directly with websites using a zero-knowledge proof. While we do not pursue this implementation further in this paper, we still think it is compelling, and recommend that password managers work with websites to further explore this design.

Instead, we settle on a design based on password nonce injection. This design does not require modifying websites but still provides most of the security benefits of using a zero-knowledge proof. Critically, this design requires no change in user behavior or awareness. We believe that both of these properties are necessary to promote the possibility of adoption.

To verify that this design is feasible, we implemented it in Firefox and BitWarden. We then conduct security and functionality evaluations of our prototypes, demonstrating that they can stop credential exfiltration by malicious client-side scripts and browser extensions. We also demonstrate that it works with 97% of the Alexa top 1000 websites.

While not perfect, our implementation is working, publicly available code that can already improve the security of autofill on the vast majority of websites. For the remaining websites, it is easy to automatically detect compatibility issues and not use our secure autofill API, preventing any functionality regressions. As such, our work not only pushes forward our scientific understanding of this area but makes an important practical contribution.

# Chapter 5

# Securing FIDO2 Credential Entry[1]

It is well established that passwords are a weak form of authentication, as users typically choose weak passwords, reuse passwords across multiple accounts, and fall victim to phishing attacks. To address these issues, the concept of multi-factor authentication (MFA) has been proposed as a more secure addition to password-only authentication. These MFA systems typically require users to provide two or more forms of authentication, such as something they know (e.g., a password), something they have (e.g., a smartphone), and something they are (e.g., a fingerprint). To address the issues of password security, two-factor authentication (2FA) and passwordless authentication have been proposed as more secure alternatives to passwords.

The state-of-the-art for two-factor and passwordless authentication on the Web is FIDO2, the FIDO (Fast Identity Online) Alliance and the W3C's newest set of specifications supporting two-factor authentication (2FA), multi-factor authentication (MFA), and passwordless authentication. FIDO2 provides a standard web services API (WebAuthn) on client machines to authenticate users using public-key cryptography. The API is available in all popular browsers and is experiencing increased adoption by major service providers, including Facebook, GitHub, and Gmail.

The FIDO2 design thwarts remote attackers. It assumes a trusted client (browser and OS) and defends against account compromise due to stolen passwords and phishing attacks.

[1]This chapter is part of a broader research project conducted in collaboration with researchers at Brigham Young University. Content here only includes aspects of the project in which I was directly involved.

Prior research has shown the FIDO2 protocol is vulnerable to local attacks, such as an authenticator rebinding attack, synchronized login, and protocol downgrade attack [64, 77, 173]. Vulnerabilities to local attacks are unsurprising given the FIDO2 threat model assumes a trusted client and TLS for network communication. The protocol does not provide confidentiality and integrity. The data is not encrypted and can be read and modified by the client and a network eavesdropper.

Local attacks against FIDO2 can originate from (1) cross-site scripting (XSS), (2) a malicious browser extension, (3) a compromised browser, and (4) a compromised operating system. In this chapter, we focus on the first two attack vectors, as they have a lower barrier to entry compared to compromising a browser or operating system for root access. Our defense, *secure browser channel FIDO2* (*sbc-FIDO2*), fortifies against such attackers with a modest deployment cost that requires changes solely within existing browsers, plus minimal changes to web servers, consisting of just 2-3 lines of code. We implement a proof-of-concept prototype for *sbc-FIDO2* in Mozilla Firefox, showing the ease the defense can be implemented in a real-world browser. We discuss the security and deployability properties of the design, showing that it is effective against local attacks and has a low deployment cost. We then discuss how our defense can be extended to encompass additional browser APIs that facilitate the transmission of sensitive data to servers, such as the Clipboard API.

## 5.1 Secure Browser Channel - *sbc-FIDO2*

In this section, we first describe an adversary model and the security properties necessary to defend against this adversary. Next, we present the design and analysis of a defense against the adversary. Finally, we describe our implementation of the defense.

### 5.1.1 Adversary model: $\mathcal{A}$

Adversary $\mathcal{A}$ represents a malicious browser extension or malicious JavaScript injected into the victim's browser through XSS.

The adversary has two primary goals:

*Goal 1:* To execute a short-term attack on the victim's account, leading to long-term access from the attacker's device. The adversary wants to impersonate Bob from their device over an extended period without detection after executing a short-duration one-time attack from Bob's device or another device that Bob uses just once. By relying on a single malicious code execution from Bob's device, $\mathcal{A}$'s continued dependence on an extension's malicious code to access Bob's accounts decreases, thus lowering the chances of detection. Many malicious extensions are removed from browsers after researchers detect them. A short-lived attack may remain undetected and persist after the extension is removed. The adversary cannot achieve its goal by registering OAuth tokens, stealing session cookies, or monitoring all user communication. Many websites do not use OAuth access tokens (e.g., banking), and the tokens last only for several hours to a couple of weeks. Furthermore, cookie-based login sessions expire as soon as a user logs out. Therefore, if the attacker wants long-term access, they have to steal cookies frequently, which is infeasible if the victim only logs in once from a vulnerable browser.

*Goal 2:* To log into a victim's account even when the victim does not log into that account. These two goals cover all local attacks described by [173] except the attack on the clone detection algorithm.

We do not consider adversaries that try to trick the user into submitting credentials to them using social engineering attacks.

**How $\mathcal{A}$ can execute attacks on FIDO2?** Within the browser, a malicious extension can leverage web extension APIs, granting it the capability to read from and write to the current page as well as network requests. This capability encompasses two specific functionalities: (1) overriding browser-provided methods with malicious alternatives and (2) accessing and modifying the request/response header and body. When it comes to intercepting and modifying FIDO2 requests or responses, a browser extension utilizes one of the following techniques:

*JS injection:* A browser extension can insert malicious JavaScript code in the target webpage to replace the FIDO2 request/response or redirect it to an attacker-controlled authenticator. One way to achieve this is by overriding the built-in *credentials.create*

and *credential.get* APIs. Whenever the webpage registers or authenticates, it calls the attacker's functions instead. The attacker can set up a virtual authenticator to register their authenticator instead of the user's [64]. Malicious attackers can also execute this attack by leveraging XSS without a browser extension.

*Network Request/Response Interception:* A browser extension can use the WebRequest API [112] to intercept network requests or responses. The WebRequest API allows browser extensions to intercept and read the HTTP request at various stages of an HTTP request life cycle.

**Feasibility** To demonstrate the feasibility of Adversary $\mathcal{A}$ compromising a webAuthn client, we built a prototype of a malicious Chrome extension to read and modify FIDO2 requests and responses using the above techniques. In our Chrome extension, *content_script* allowed us to obtain details and change the web page's DOM a user visits. We replace Chrome's web API function *navigator.credentials.create* with our custom handler on every webpage. Our custom handler modifies/replaces the original FIDO2 request or response with a malicious one.

There are many instances of malicious browser extensions used by millions of users on official Chrome/Firefox extension stores. A few recent examples are: (1) in 2020, 500 Chrome browser extensions were discovered secretly uploading private browsing data to attacker-controlled servers and redirecting victims to malware-laced websites[131], (2) Awake discovered 111 malicious Chrome extensions that were downloaded over 32 million times [15]. These extensions can take screenshots, read the clipboard, harvest credential tokens stored in cookies or parameters, grab user keystrokes (like passwords), etc., and (3) in 2021, Cato identified 87 out of 551 unique Chrome extensions on customer networks as malicious [24].

Similar to Kaprevelos et at. [85], we look into chrome extensions in the wild to show the feasibility of malicious extensions. We analyzed the permissions requested for 163,699 Chrome extensions, extracted by CRXcavator [50] from the Chrome web store on Jan 21st, 2021. We found that 35,211 Chrome extensions have one of these permissions that allow them to run on all websites: $< all\_urls >$, $https://*/*$, or $*://*/*$. These extensions can use *content_script* and interact with any webpage's DOM, which allows them to execute a MITM

attack by overriding webAuthn client APIs. 18,943 extensions have *webRequestBlocking* permission, which is enough to intercept and read FIDO2 request or repsonse. Table 5.1 shows the distribution of users among these extensions. There are hundreds of extensions with more than millions of users. A malicious actor only has to compromise one of these extensions to be in a position to launch an attack on over a million users.

**Security properties to mitigate local attacks:** Based on the adversary's goals and local attacks described in previous research we present three properties and how the lack of these properties leads to various local attacks. We aim to achieve these security properties for our defenses to defend against $\mathcal{A}$.

1. Confidentiality: $\mathcal{A}$ can compromise a user's privacy by intercepting and reading the FIDO2 request/response, thereby gaining access to sensitive information such as the counter value [64] and FIDO2 extension details.

2. Integrity: $\mathcal{A}$ can manipulate the FIDO2 registration request/response to launch an authenticator rebinding attack or an algorithm downgrade attack [77, 64, 173]. The latter can be particularly concerning in the post-quantum computing era, as many authenticators may not support post-quantum cryptography. Consequently, servers would need to support conventional legacy cryptography. Adversaries can downgrade the signing algorithm to a weak one, potentially exploiting it at a later stage.

3. Session Accountability: The authenticator signs the challenge with the private key associated with a user's account during authentication. However, when a user taps the hardware security key ($HSK$) (or enters PIN on a PIN based $HSK$), users cannot verify for which account they are authenticating. As a result, an adversary can clandestinely authenticate to one of the victim's accounts in the background without their knowledge. This attack is known as a synchronized login attack [173].

Table 4.1 shows the status of the security properties in the current FIDO2 implementation. $\mathcal{A}$ poses a threat to both confidentiality and integrity within the context of the FIDO2 protocol. However, session accountability remains intact to some extent as the browser

Table 5.1: Browser Extensions by Number of Users

| # Users | # Extensions | Permissions | |
|---|---|---|---|
| | | webRequestBlocking | All URLs |
| 1-10000 | 163699 | 18943 | 35211 |
| 10000-50000 | 5506 | 1017 | 1480 |
| 50000-100000 | 2379 | 543 | 821 |
| 100000-500000 | 1177 | 354 | 497 |
| 500000-1000000 | 631 | 144 | 290 |
| >1000000 | 254 | 127 | 153 |

ensures transparency by displaying the domain name on a popup before the user's approval on the *HSK*, as shown in Fig 5.1. In addition, the browser controls the webAuthn client, preventing $\mathcal{A}$ from accessing it directly. Consequently, authentication requests are channeled through the browser, and $\mathcal{A}$ cannot manipulate the popup displayed to users. However, since browsers do not display a username, this allows $\mathcal{A}$ to log in to another account without the user's knowledge if an *HSK* has multiple accounts for the same *RP*.

### 5.1.2 Design: *sbc-FIDO2*

This section presents the design of a secure in-browser communication channel, *sbc-FIDO2*, that defends against $\mathcal{A}$.

*Session Accountability:* To ensure session accountability for each unique combination of username and authentication request, browsers should add the username to the popup notification shown in Figure 5.1 to inform users of the account they are authenticating to. Displaying the username during authentication may help users detect an unauthorized login request.

*Confidentiality & Integrity:* This design addresses the security concern related to the potential interception and modification of FIDO2 requests and responses by malicious browser extensions. This design requires an *RP* to send a dummy FIDO2 registration/authentication request within the payload while the original request is transmitted in the header $webauthn\_req$. Additionally, the *RP* sends a header $URL\_resp$ containing the complete URL to which the FIDO2 response will be directed. The browser intercepts the genuine FIDO2 request at the earliest possible stage within its networking module, stores it in secure storage, and removes the FIDO2 header. Meanwhile, the dummy request within the payload proceeds through the standard flow where browser extensions can access it. Simultaneously, the genuine FIDO2 request remains confidential and intact in secure storage until it is retrieved through the webAuthn API at the end of the flow. Consequently, this design effectively safeguards the FIDO2 request from exposure against the DOM.

Similarly, the FIDO2 response from the authenticator is intercepted within the browser's webAuthn API, placed in secure storage, and substituted with a dummy response. Finally,

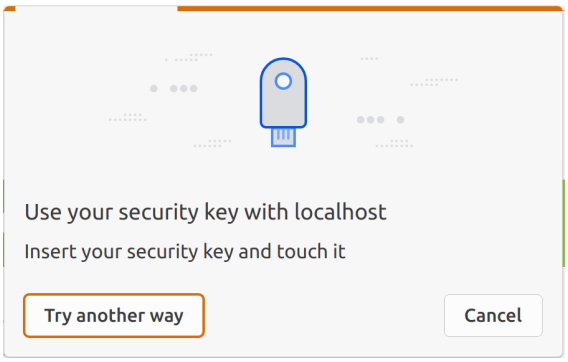



Figure 5.1: Chrome browser popup for FIDO2 authentication initiated on localhost.

the browser's networking code inserts the original FIDO2 response in the *webauthn_resp* header in the reply before transmitting it over the network.

When the browser receives *webauthn_req* and *URL_resp*, it starts monitoring all the outgoing web requests' URLs for 120 sec, which is the maximum recommended timeout for which the caller is willing to wait for the call to complete according to the FIDO2 specification. If the browser detects an outgoing request to *URL_resp*, it appends the *webauthn_resp* header to the original FIDO2 response.

To accomplish the secure storage, we utilize the singleton design pattern, which facilitates the secure storage and redirection of FIDO2 requests and responses away from browser extensions. The flow of request/response through the secure storage can be seen in Figure 5.2. The singleton pattern restricts the instantiation of a class to a single instance within a given process. In our design, the singleton class acts as a container for requests and responses, which are stored during the early stage of the flow and later injected back into the flow at the final stage. This design simplifies access to the stored variables, as the singleton file is readily available anytime during the flow.

The use of the singleton pattern has been subject to debate among software developers due to its potential as a global class. However, in our specific implementation, a global class is necessary to establish a direct and efficient path for securely transferring requests and responses from the beginning to the end of a flow. Moreover, the singleton file serves as a global object, eliminating the need to pass object references between different objects and reducing the risk of coding errors.

The *RP* employs dummy FIDO requests or responses to identify any malicious JavaScript or browser extensions that attempt to tamper with FIDO2 communication or leakage of sensitive data. If the *RP* receives a FIDO2 response corresponding to the dummy FIDO2 request in the payload, the *RP* will consider it an attack and can warn users. If a FIDO2 extension communicates sensitive data and the *RP* receives the dummy sensitive data in the future, the *RP* will identify it as an attack and inform the user. Furthermore, the *RP*s can notify browsers when they detect an attack to allow browsers to capture the current state of the Document Object Model (DOM) and browser extensions for profiling purposes.

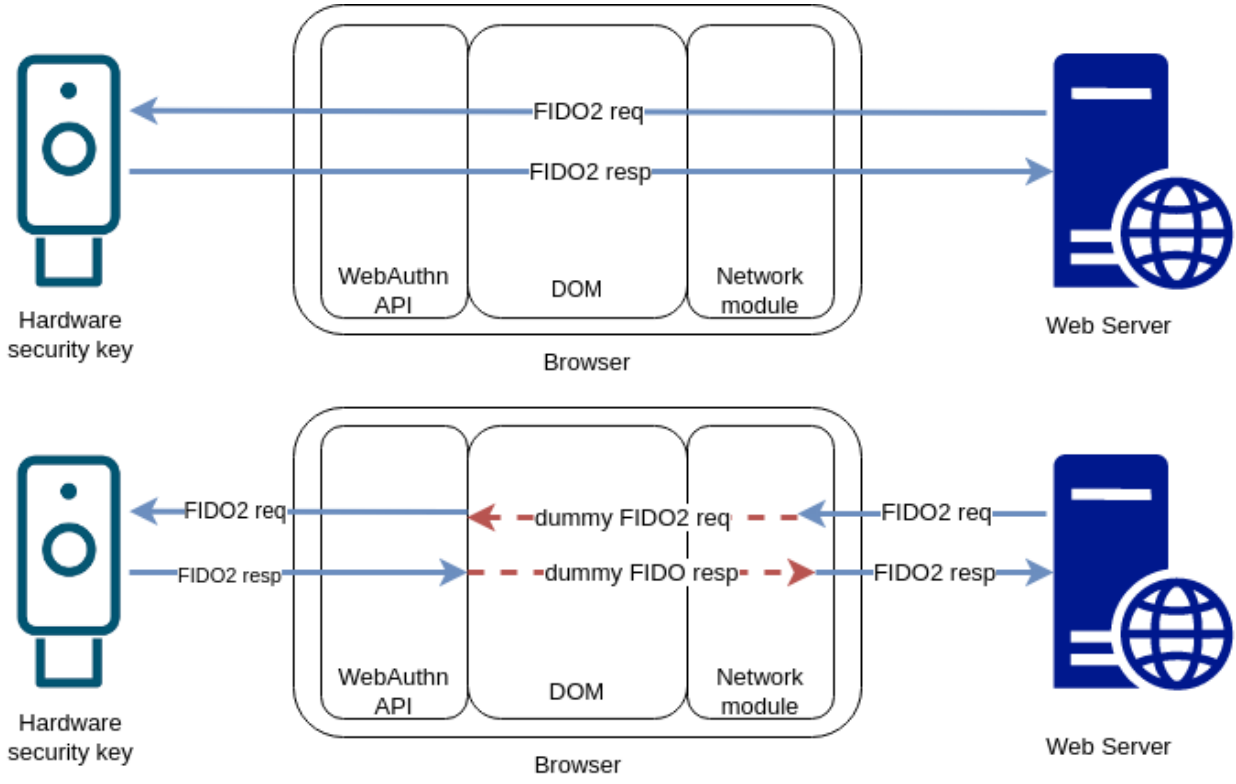



Figure 5.2: *sbc-FIDO2*

There are two ways to generate dummy requests or responses, each serving different objectives. First, create a valid FIDO2 request with randomly generated fields. This method prioritizes maximum confidentiality by ensuring no correlation between the dummy and original requests/responses. However, this approach signals potential attackers that the request is not genuine, thus dissuading them from executing an attack that risks detection by attack profiling.

Second, duplicate the original request and substitute all parameters with random values that conform to the specific constraints outlined in the FIDO2 specification. These random values must also match the length of the original values. Furthermore, any values already present in the Document Object Model (DOM), such as the username in the context of two-factor authentication (2FA), should retain their original values within the dummy request. This strategy may deceive potential attackers into believing the dummy request is a genuine FIDO2 request and replacing it with their attack request. This approach increases the likelihood of successful attack profiling, allowing for more comprehensive analysis and notifying the user regarding the threat.

### 5.1.3 Security and Deployability Analysis

In *sbc-FIDO2*, the flow of the original FIDO2 request or response bypasses the domain of the Document Object Model (DOM) and any accessible parts of the networking module that malicious browser extensions could exploit. Furthermore, the secure channel established through the singleton class operates exclusively within the privileged environment of the browser, with no exposed API accessible to browser extensions. This design prevents any unauthorized access or tampering by malicious browser extensions.

*Deployability: sbc-FIDO2* introduces modifications to the browser architecture to establish a secure channel within the browser environment. Our *sbc-FIDO2* design necessitates the inclusion of an additional header in the FIDO2 web request and response, which could be updated by only changing one line of code as described in Section 5.1.4. The transmission of FIDO2 requests and responses through headers allows the browser to remove the FIDO2 request earlier and add the FIDO2 response later in the flow compared to the HTTP body. Future research can explore alternative approaches that involve the browser scanning incoming

requests for designated keywords associated with FIDO2 requests and triggering the secure channel process accordingly. However, for long-term implementation, we recommend adopting the headers approach due to its minimal overhead and the flexibility it offers servers, granting them the ability to permit extensions to intercept requests if required for specific purposes. This option to allow interception may prove particularly advantageous for applications beyond the scope of FIDO2, into other applications supported by the web browser such as Clipboad, File System, and Geolocation APIs.

### 5.1.4 Implementation

We modified the Firefox browser nightly version 104.0a1 to demonstrate the feasibility of *sbc-FIDO2*. The modified Firefox implementation was tested on Windows 10 and 11.

To safeguard FIDO2 requests/responses from interference by browser extensions, it is imperative to safeguard the FIDO2 request before the *onHeaderReceived* event and the FIDO2 response before the *onBeforeRequest* event. Implementing these measures guarantees that the FIDO2 requests/responses remain inaccessible to DOM and browser extension's Web request interception capabilities.

Figure 5.3 illustrates the standard and modified Firefox flows for FIDO2 requests and responses. When the relying party ($RP$) sends an original FIDO2 request, it is transmitted through an HTTP header field named *webauthn_req* along with *URL_resp*. In the modified Firefox, the browser examines every incoming server response in the `nsHttpChannel`, which is the earliest point of entry for HTTP packets in Firefox. Upon detecting a header containing the *webauthn_req* field, the browser retrieves the values of *webauthn_req* and *URL_resp*, passes them to the singleton file, and subsequently removes them from the header. The dummy request in the payload undergoes the standard flow, where browser extensions can interact with it. The *WebAuthnTransactionParent* retrieves the original FIDO2 request from the singleton file, creates a `WebAuthnMakeCredentialInfo` object, and transmits it to the authenticator through the `WebAuthnManager`'s register function. The user interacts with the security token, and the authenticator returns the FIDO2 response. The original FIDO2 response is stored in the singleton file by the `WebAuthnTransactionParent`, while a dummy FIDO2 response is provided to the DOM. When the `nsHttpChannel` detects an outgoing web

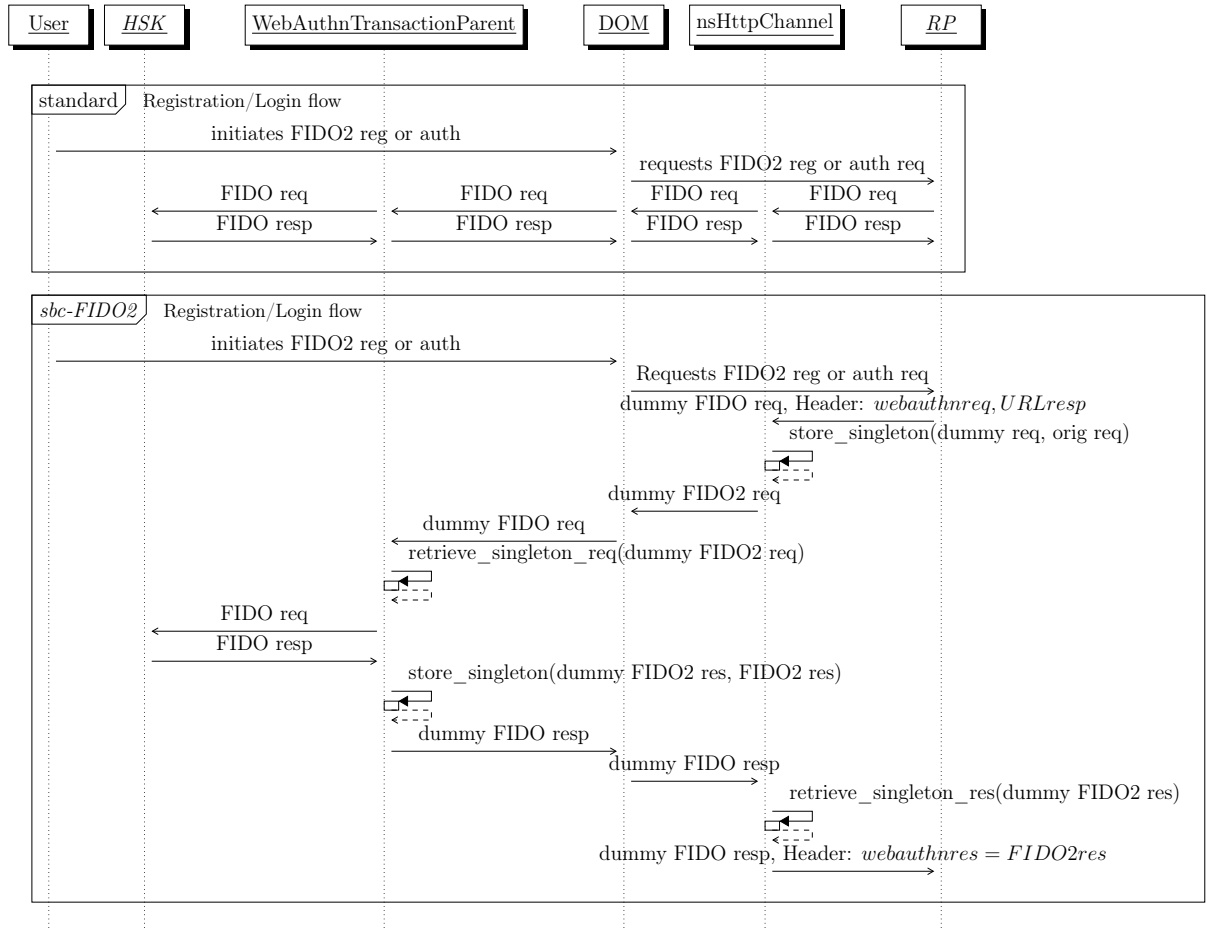



Figure 5.3: *sbc-FIDO2* sequence diagram

request to *URL_ resp*, it retrieves the original FIDO response and adds it to the outgoing request's header.

To validate our implementation, we employed a Java-based WebAuthn Demo Server [176]. To enable our server we modified the server by replacing the line `Response.ok(json).build();` with `Response.ok(json_dummy).header("webauthn_req", json)` `.header("URL_resp", url).build();`. To assess the effectiveness of our implementation and the feasibility of the *sbc-FIDO2* design, we installed the malicious browser extension developed as described in Section 5.1.1. Subsequently, we performed registration and authentication processes, wherein the malicious browser extension failed to read or modify FIDO2 requests or responses, thereby confirming the effectiveness of our implementation.

## 5.2 Discussion

Although these mitigations address attacks outside the initial threat model of FIDO2, they may become more consequential with the move to passwordless authentication. Instead of attackers needing to compromise a password and a second factor, attackers can now compromise accounts by compromising a single, stronger authentication factor. High-value targets may be susceptible to motivated attackers that pursue attack vectors outside the scope of the initial FIDO2 design.

### 5.2.1 Effectiveness of Defenses

XSS and malicious browser extensions ($\mathcal{A}$) pose substantial threats in the present landscape, demanding the implementation of robust defenses to counteract their potential harm. The masses can adopt our *sbc-FIDO2* defense to protect against adversary $\mathcal{A}$ to secure themselves with relatively less overhead. It is important to highlight that the defenses we propose only safeguard against attacks directed at FIDO2 and other additional precautions and mitigations should be taken to protect against other potential threats. For instance, users along with our *sbc-FIDO2* should refrain from granting cookie permissions to untrusted extensions, and *RP*s should utilize HttpOnly cookies to prevent malicious JavaScript from accessing cookies.

### 5.2.2 Deployment

The *sbc-FIDO2* significantly enhances FIDO2 security against XSS and malicious browser extensions with minimal effort. This design maintains the same user experience while introducing a slight increase in cognitive load through the addition of the username in the browser's popup before authentication on the *HSK*. Deploying this defense is relatively straightforward, involving major changes in the browser and minor modifications, as little as two lines of code, for *RP* entities. Updating third-party FIDO2 libraries used by most *RP* entities can often suffice for deployment. However, it is crucial for these libraries to provide *RP* entities with the flexibility to enable/disable this feature. Overall, this minimal impact on usability and easier deployability enables widespread adoption of this defense mechanism.

Other designs utilizing TEEs can be utilized by high-value targets to protect against more sophisticated attacks, such as compromised browser, operating system, and hardware components as in the case of malware and other malicious agents. But the compromise is that these designs require significant changes to the browser, *RP*, and *HSK*, making them less deployable.

## 5.3 Conclusion

In this chapter, we presented a defence against local attacks on FIDO2 that utilizes the Secure Browser Channel (SBC) concept. We identified the threat model for local attacks on FIDO2 and explored the design space for SBC-based defences. We implemented a proof of concept of the SBC-based defence in the Firefox browser and evaluated its proper functioning. Even though the local attacks on FIDO2 are not as prevalent as remote attacks, they are still a significant threat and should be addressed before they become more common.

# Chapter 6

# Detecting and Auditing Password Theft

Even with the assistance of password managers, vulnerabilities persist, leaving user credentials exposed to threats such as Cross-Site Scripting (XSS) attacks [150] and malicious browser extensions [85, 137]. XSS attacks are particularly dangerous due to their capacity to compromise large groups of users, while malicious extensions pose a threat through their ability to conduct extended and stealthy information theft. The frequent and evolving nature of these security threats highlights their importance in today's cybersecurity landscape.

OWASP recognizes XSS attacks as one of the top 10 most exploited vulnerabilities on the web [132]. The realm of cross-site scripting attacks and their mitigation is a subject of extensive research [147, 159, 160, 96, 10, 67, 89]. Past studies have identified malicious extensions as a significant threat to credential security [173]. Even if users only install reputable web extensions, compromised libraries, such as those in supply chain attacks, can still turn an extension into a malicious tool [129, 49]. Additionally, there have been other instances where benign extensions turned malicious either due to the creators, by compromise of code, or by sale to malicious agents [72, 22, 11, 36].

These attacks can lead to various types of damage, ranging from ransomware installation and user surveillance to credential theft. The implications in the case of credential theft are especially grave, often resulting in the loss of accounts. This can lead to financial harm when financial accounts are compromised and substantial privacy violations for other types of accounts. The risk becomes particularly critical in situations where account privacy is

paramount, such as for journalists operating in authoritarian states, where the repercussions of such breaches can be extremely severe.

Effective detection of these attacks is crucial in preventing the loss of user accounts. For administrators, the mass leakage of user credentials results in considerable time and financial burdens in addressing such security breaches. Timely detection can substantially lessen the impact on both users and server administrators by either preventing or mitigating the damage. From an attacker's standpoint, the likelihood of their attacks being detected, and the potential exposure of their methods, act as significant deterrents. This not only disrupts the immediate attack but also contributes to the identification and resolution of the vulnerabilities being exploited. Consequently, effective detection serves to discourage future attacks by elevating the perceived risk for potential attackers.

In this work, we implement a system able to detect and audit attacks on user credentials in the browser. Our work builds upon the research of Gautam et al. [59], who proposed mechanisms for securing users against the threats of malicious scripts and extensions. We extend their methodology to detect ongoing attacks and audit the mechanism of attack, primarily focusing on malicious scripts and extensions. This knowledge enables users and administrators to proactively address and mitigate the effects of such security incidents, thereby reducing their overall impact. Utilizing the information about the method of attack, an administrator can priotize mitigations to certain groups of users. Also, the threat of detection alone provides a deterrent to potential attackers, thereby reducing the likelihood of future attacks.

We utilize the secure nonce inserted in [59] to find out the existence of a credential breach attack and then perform audit of how the attack happened by cooperation between the users' browser and the server. We utilize the browser audit logs to find the script responsible for accessing potentially sensitive information (password fields in this case). We implement and evaluate attack audit in the Firefox browser and verify that it is able to identify potential source of the attack for a variety of methods. With the performance evaluation, we show that our system is able to keep up with the generated logs in real-time, without the requirement of a lot of storage space. We discuss the potential future use cases of audit-log based such provenance engine in the browser, privacy implications with this method and future works.

To summarize, our contributions are as follows:

1. We utilize the nonce-based password replacement mechanism to detect local attacks on the passwords.

2. We implement a mechanism for attack audit by utilizing the browser audit logs to find the scripts responsible for accessing sensitive information.

3. We implement a proof of concept server and evaluate the performance of both attack discovery and browser provenance method to show that the system is able to work in real-time.

4. We discuss methods to identify path of attacks and privacy implications of the system in the server.

## 6.1 Background

### 6.1.1 General and targeted XSS attacks

Cross-site scripting(XSS) attacks can be used to target specific group of individuals. A specific example of using targeted attacks is using watering hole attacks targeted towards a certain organization. Watering hole attacks are a type of cyberattack where attackers specifically target certain organizations or groups. This is done by compromising websites that are frequently visited by these groups. Attackers determine which websites are popular with their targets and then infect these websites with malware. When individuals from the targeted group access these compromised websites, their computer systems are at risk of being infected. These attacks can be tailored to specific users, often based on their IP addresses, making detection and analysis more difficult.

### 6.1.2 Browser Developer Tools

Modern web browsers like Chrome and Firefox include Browser Developer Tools [61, 113], which help developers understand and fix web pages directly in the browser. These tools

allow for checking the page's HTML, CSS, and JavaScript, and tracking how fast the page loads. They are essential for quick editing and problem-solving in web development, making it easier to build and improve websites.

Chromium based browsers utilize Chrome DevTools Protocol [60] to provide developer tools with their ability to debug websites. Firefox, too, supports a subset of these protocols These protocols enable scripts to access functionalities akin to developer tools, such as inspecting and modifying HTML and CSS in real-time, monitoring and controlling network requests, debugging JavaScript through breakpoints and stack traces, and analyzing website performance and optimization metrics.

### 6.1.3 Credential Swapping Mechanism

In the work by Gautam et al. [59], they implement a credential swapping mechanism in the browser designed to protect against malicious entities accessing a user's actual credentials. The mechanism operates as follows:

1. The user commands the password manager to fill the credentials on the page.
2. The password manager inserts dummy credentials, or a nonce, into the page's DOM and registers the actual credentials with the browser.
3. The user initiates the form submission on the page.
4. The browser replaces the dummy credentials with the actual credentials on the webpage before submission.
5. The form, with the real credentials, is submitted to the server.

This mechanism ensures that any extensions or scripts accessing the DOM do not obtain the actual credentials. The swap of credentials occurs after these potential threats have lost access. However, these entities can access the dummy credentials input by the password manager.

For clarity in this paper, we define the following terms:

- **Credentials:** The real user credentials saved by the password manager and swapped in by the browser during submission.
- **Nonce:** The dummy credential filled by the password manager into the DOM, which may be accessed by adversaries.

## 6.2 Threat model

Our threat model primarily addresses adversaries who are capable of extracting credentials from the DOM. Building upon the framework set by Gautam et al. [59], we consider the following types of adversaries in our model: i) Honest-but-curious entities, ii) DOM attackers, and iii) Extension attackers.

Our approach builds on Gautam et al.'s evaluation, which confirms the effectiveness of their mechanism against identified adversaries. We introduce a strategic layer by leveraging attackers' potential oversight—misidentifying nonce credentials as genuine. Our model operates under the assumption that these adversaries are restricted to accessing only system-inserted nonces, creating a deceptive layer of security.

We assume that the server and the user's local system are secure against direct attacks, instead focusing on browser threats. Compromising the local system or the server would require far more resources and have more severe consequences than compromising the browser. We consider man-in-the-middle attacks to be out of scope for a similar reason, as the securing of SSL is a well-established practice.

A typical adversary in our model can retrieve and subsequently use the nonces from our system, submitting them to the server at a later time. They operate under the assumption that these retrieved values are genuine credentials. Similar to the extension attacker described by Gautam et al., our adversary has the capability to remain in the system indefinitely, monitoring all requests and responses from any web page visited in the browser. Extension attackers are deemed particularly threatening due to their ability to persist within the system and continuously exfiltrate user information.

An interesting aspect of the threat posed by malicious extensions is their level of access, equivalent to that of password managers, since both are types of web extensions. This scenario

introduces challenges within the browser's security model; password managers are trusted entities, yet malicious extensions could potentially access the same sensitive information. Therefore, it's crucial to carefully manage the information that password managers read from and write into the browser.

### 6.2.1 Motivating Example

To illustrate the relevance of our work, we present an attack scenario drawing examples on real-world incidents:

Alice oversees the security and operational integrity of example.com, a website utilizing a password-based authentication system. Her role is pivotal in implementing security measures against potential cyber threats. Bob, a regular user, accesses the site via a web browser, such as Firefox, using his username and password. Trudy, an attacker, aims to compromise Bob's account by stealing his credentials.

In this scenario, Trudy sends Bob a sophisticated phishing message containing a link to a legitimate but compromised site. The message might seem to originate from a trusted source, as seen in 2019 Yahoo XSS vulnerability [92] where email clients were compromised to send malicious emails.

This XSS threat could stem from various sources, ranging from individual malicious actors employing XSS attacks — evidenced by numerous CVEs in 2023 such as CVE-2023-6217, CVE-2023-5758, etc — to sophisticated nation-state actors like APT35 [98, 120]. These actors often compromise widely-used websites persistently. Notable examples include CVE-2023-5758, affecting Firefox for iOS, and CVE-2019-18426, targeting WhatsApp users. Additionally, nation-state actors might deploy malware like malicious browser extensions, tricking users into installing them from seemingly genuine sources.

Once such a script or extension is active in Bob's browser, it captures his credentials by monitoring DOM password entries. As a result, Bob's account is either infiltrated, leading to continuous data theft, or completely hijacked, denying him account access. For Alice, the repercussions are broader, as Trudy might exploit Bob's account to target other users or employ similar methods to compromise more accounts, potentially leading to a widespread security crisis.

Awareness is key in such situations. If Bob recognizes the attack and understands how he was compromised, he can be more vigilant in the future. Similarly, if Alice has knowledge of the attack method, it enables her to take proactive measures such as fixing website vulnerabilities, alerting affected users, or informing all users about the potential breach. As such, our system aims to provide both Bob and Alice with the necessary tools to detect and understand such attacks, thereby mitigating their impact.

## 6.3 System Design

In this section, we delve into the theoretical design of our honeypot system, that enables us to detect an attack has happened and pinpoint how the attack happened. The overall design of the system and the interactions between different actors is illustrated in Figure 6.1.

To provide a clear understanding of the system, the section is organized into several parts:

- Section 6.3.1 identifies the assumptions for the system design
- Section 6.3.2 outlines all the key actors involved in our system and describes their respective capabilities within the design framework.
- Section 6.3.3 details the operational flow of our system utilizing the actors and services defined.
- Section 6.3.4 elaborates on the ideal services and components that our system utilizes to achieve its objectives.

### 6.3.1 Basis of the final system

Our defense system is fundamentally built on the credential swapping mechanism, as we have detailed in section 6.1.3. This mechanism revolves around the concept of a nonce - a dummy credential representing access to the user's Document Object Model (DOM) or outgoing request. In our model, there are three distinct scenarios where nonce access occurs:

1. A curious user inspects the content inserted into the page.

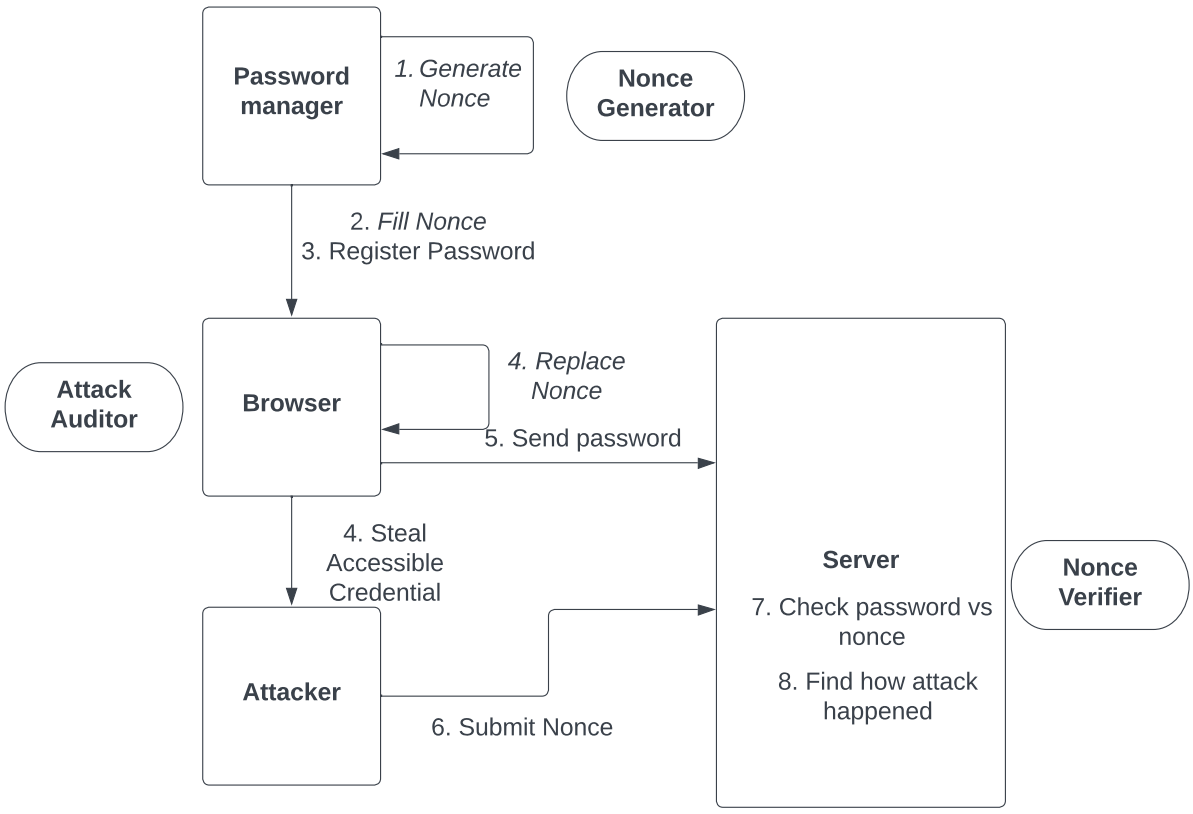



Figure 6.1: Overall diagram of the honeypot system

2. An attacker, exploiting vulnerabilities, runs scripts on the current page that gain access to the DOM.

3. A malicious browser extension with DOM access, possibly used for prolonged and stealthy information theft.

While the first scenario is harmless, involving just the user's curiosity, the latter two are serious security concerns, indicating potential breaches. The presence of a nonce submitted to the server becomes a pivotal indicator here, signaling unauthorized DOM access. In such cases, it is presumed that attackers, mistaking the nonce for genuine credentials, attempt to use it for unauthorized access, thereby revealing their presence.

There are however some genuine reasons for a script to access the credential field, such as a password manager or a legitimate script utilized by the website. Care must be given to differentiate these from the malicious scripts and extensions.

We also operate under the assumption that a legitimate user can successfully log into their account using their browser. Therefore, in collaboration with the server and the authenticated user, we aim to deduce the timing and method of any attack.

#### System goals

With the foundational assumptions in consideration, there are three apparent intermediate goals of the system: (1) Generate proper nonces (2) Identify that an attack has occured using the nonce submitted, and (3) Locate the attack with collaboration between legitimate user and the server after an attack is detected.

In order to attain these primary system goals, our system introduces the necessity for information transfer between the user and the server. This potentially raises privacy concerns for the user with the server. Consequently, a secondary but crucial goal of our system is to minimize the potential for such privacy infringements.

### 6.3.2 Actors

This subsection introduces the key actors, or components, of our system, each with a distinct role: the password manager, the browser, the website server, and the attacker.

**Password Manager:** The password manager primarily manages users' credentials, focusing on storage and autofill functionalities. It incorporates a *Nonce Generator* service, creating nonces to replace actual credentials on webpages, tailored for our detection mechanism. The password manager also interfaces with the browser's API for credential registration. In scenarios without a password manager, the *Nonce Generator* is integrated into the browser itself.

**Web Browser:** The browser's function is to render webpages and provide an interface for webpage interaction. In our system, the browser is modified to offer an API that allows password managers to register credentials for specific form fields. When a form field has a registered credential, the browser automatically replaces any input in that field with the actual registered credential. If the server identifies an attack, the browser utilizes an *Attack auditor* service to determine the nature of the attack. If the server identifies an attack, the browser utilizes an *Attack auditor* service to determine the nature of the attack.

**Website Server:** The website server's responsibility is to deliver the website to the browser and verify login credentials. Upon receiving valid credentials, the server authenticates the user. Invalid credentials are passed to a *Nonce Verifier* service, which determines if they are mere typos or nonces generated by the *Nonce Generator* for detecting credential theft.

**Attacker:** The attacker is a malicious entity aiming to steal user credentials from the client-side. They may employ XSS attacks or malicious extensions to gain access to the client browser. The attacker's goal is to capture credentials using their chosen method of attack.

**Client:** The design involves different interactions with the server which can be done from both the browser and the password manager, choice depending on performance and privacy issues involved with the design. So, for the rest of the paper, we consider the client as a combination of the browser and the password manager.

### 6.3.3 Process

In this section, we outline the operational flow of our system, detailing the steps involved in the process. The operational flow involves interactions between the actors specified in Section 6.3.2 and ideal services, special modules that have specific operational requirements, defined in Section 6.3.4. Figure 6.1 illustrates the ideal operational flow involving the main actors, which proceeds as follows:

1. The browser requests a webpage from the server and displays it to the user.
2. The user prompts the password manager to autofill credentials into the webpage.
3. The password manager requests a dummy nonce from the *Nonce Generator*.
4. The password manager fills the webpage with the nonce.
5. The password manager uses the browser's Autofill API to register the actual user credentials with the browser.
6. The user instructs the browser to submit the form.
7. The browser substitutes the nonce with the actual user credentials.
8. The web server receives the form submission and verifies the credentials. Correct credentials lead to user authentication.
9. The browser sends the nonces to the server to be utilized by the *Nonce Checker*.

When an attacker is involved, the system's operation consists of three phases: the credential theft phase, the attack detection phase, and the attack audit phase. In theses phases, the system performs the following actions:

1. **Credential Theft**: Adversaries attempt to extract users' credentials but instead obtain the nonces, which they mistakenly assume to be the actual credentials. The system is gathering information about it's execution to identify attacks in the future.
2. **Attack Detection**: The system identifies that an attack has occurred when the stolen nonces are submitted.

3. **Attack Audit**: The system collaborates between the user's browser and the server to gather specific details about the attack.

The detection phase occurs when the server receives a nonce submitted by the attacker. In the detection phase:

1. The server recognizes that the submitted credential does not match the user's actual credentials.

2. The server consults the *Nonce Verifier* service to confirm that the submitted credential is a security nonce.

3. The server informs the user's browser that an attack has occurred.

The audit phase occurs directly after the detection phase. In this, the server requests the user to investigate the attack.

1. The server informs the user's browser that an attack has occurred and provides with additional information required for detection.

2. The server then utilizes the *Attack auditor* service to determine how the attack was executed. This service can be in the server or the user's browser.

3. The attack information is then shared between the user's browser and the server for further mitigation steps.

4. Based on this knowledge, the server takes appropriate measures to secure the user's account and address the exploited attack vector with minimal effort.

### 6.3.4 Ideal Services

This section focuses on various ideal services, envisioned as system blocks that provide specific functionalities in an ideal manner. Each ideal service aligns directly with a specific goal of our system, making them essential components within the overall framework. In the design phase, these services are assumed to function optimally, flawlessly executing their designated roles.

We assume that each service performs without errors or limitations, providing an ideal environment to base our design decisions on. This approach allows us to first conceptualize the optimal scenario and then work through a design search to validate and justify our choices. As we progress through the design process, we will critically evaluate each service, considering practical limitations and challenges to refine our initial ideal assumptions into feasible, effective solutions.

### Nonce Generator

The Nonce Generator is responsible for creating nonces that are inserted into the Document Object Model (DOM).

The nonces generated have specific characteristics:

- **Indistinguishability:** Nonces are designed to be indistinguishable from real passwords. This ensures that any entity accessing only the nonce cannot discern whether it is a security nonce or an actual user's password.
- **Password Privacy:** The nonces reveal nothing about the user's actual password.

Given these properties, there are several design options for generating nonces:

1. Creating PCP (Password Composition Policy)-compliant random passwords. [58]
2. Producing human-like passwords.
3. Utilizing a deterministic seed to generate random passwords.
4. Mutating the user's existing password.

These mechanisms and assumptions underlying nonce generators are typically seen in honeyword systems aimed at identifying password database thefts, as noted in existing literature [84, 8, 47, 166]. Our Nonce Generator, however, differs from these systems. In our approach, adversaries do not access the user's actual credentials, preventing them from comparing different strings one of which is user's actual password. Unlike typical honeyword systems, where adversaries might analyze a set of credentials to guess the actual password,

our system does not allow access to genuine credentials or a variety of passwords for different users. Therefore, our nonces do not require password privacy attribute and need not resemble the users' actual passwords. Accordingly, our design avoids mutating user passwords to prevent adversaries from learning about the real credentials.

A password manager, while not essential, is a frequently used component in our system Users employing password managers often generate passwords [125, 121], leaving adversaries uncertain whether the current users' passwords are generated or manually created. This ambiguity deters adversaries from dismissing credentials that appear random but might be genuine user passwords Thus, as generated random passwords are equally probable, our system doesn't rely on the indistinguishability attribute, . Accordingly, our design uses PCP-compliant random passwords as they're the most secure and does not undermine the characteristics required for the system.

### Nonce Verifier

As previously established, our second system goal is to identify that an attack has occured using the nonce submitted. Nonce verifier is the ideal service that allows us to achieve this. The primary goal of the nonce verifier is that it verifies if a string submitted is a nonce or not. The way it is meant to be used is that a user or administrator is able to submit a string and verify if it is a nonce. For any login attempt, the service will be able to determine either it is or is not a nonce.

Most obvious case might be that everything that is not the users' genuine credential can be flagged as a nonce. But there are genuine cases in which users either submit typos of their existing credentials, enter passwords for a wrong website, or are trying different permutations of passwords to remember which permutation [162] they used as their actual credential. Therefore, the service cannot flag everything that is not the users' genuine credential as a nonce. The ability of the Nonce Verifier to accurately identify nonces ensures that genuine user errors are not mistaken for security incidents, thereby maintaining the integrity and reliability of our system's security measures. To not open up channels of security and privacy attacks, we only allow checking of the nonces in the server.

A simple method for the verifier to work is by a mechanism that allows to store nonces either in the server or the client. If storing in the server, for every failed login attempt, the server checks if the submitted credential is a registered nonce. This would require a mechanism for the client to register the nonces with the server. And every time a genuine user logs in, the server can notify the client that an attack has happened. To prevent an adversary from misusing the nonce registration process, the server can limit only authenticated users to register nonces and have a throttling mechanism to prevent overloading the server with nonces.

On the other hand, we can opt to store nonces in the client side. This would then require the server to query the client with failed notifications to verify if the submitted credential is a registered nonce. So, every time a user logs in, the server sends failed logins sent to the server since the last check to the client. As the client can already verify that it's a genuine server due to SSL certificate, there's no issue of an adversary acting as the server to input false positives into the system. In either case, the communication between the client and the server can be done to identify an attack has happened. After an identification of an attack has occured, the server and the client can collaborate to let each other know and identify the source of the attack.

The malicious extension adversary having the same capabilities as the password manager poses a challenge to send nonces to the server as any method we introduce for the password manager to send nonces to the server can be utilized by the adversary. As the adversary can also read outgoing requests from other extensions, the adversary can also read the nonces sent to the server. And as the user needs to authenticated to the server, utilizing users's authenticated tokens from the password manager or the browser exposes the authentication token to other security risks. Therefore, we opt to store the nonces in the client side and have the server query the client for the nonces. An advantage of this method is that the client can then decide if they want to let the server know about the attack and any associated details.

**TOTP based approach** TOTP [142] is a mechanism to generate(or share) common strings, usually 6 digits numbers, between two different entities. This sounds very similar to our problem to share generated nonce between the client and the server. So, it might seem that a

TOTP [142] like creation and checking mechanism for nonces, by simultaneously generating it on both client and server side, is a good solution. This method would be simple and efficient, as it would only require a pre-shared secret key and a synchronized clock in 30 seconds block. However, this mechanism is not suitable for our system as the adversary might utilize the nonces at any time in the future. So, we'll not have a timestamp to regenerate the nonces in the server to check. Therefore, this method is impractical for nonce verification in our system.

**Process** With the discussion of mechanism of nonce verifier, we can now discuss the process of nonce verifier. The process of nonce verifier, as identified in figure 6.2, is as follows:

1. The client side generates a nonce and other required metadata.
2. The client securely saves the credentials for future use.
3. The user logs in with the actual credential <U, P> to the server.
4. The server authenticates the user.
5. If the authentication is successful, the server looks up any failed logins that the user has had since the last login.
6. The server sends the list of failed logins to the client.
7. The client checks if any of the failed logins is a registered nonce. If there's a match, an attack has taken place.
8. The client performs further operation to identify the source of the attack.
9. The client informs the server about the attack, deciding the amount of data to send to the server based on their privacy preference.

### Attack auditor

The final goal of our system is to locate how the attack occured after an attack is detected. Attack auditor is the ideal service that is responsible to locate the mechanism of attack after

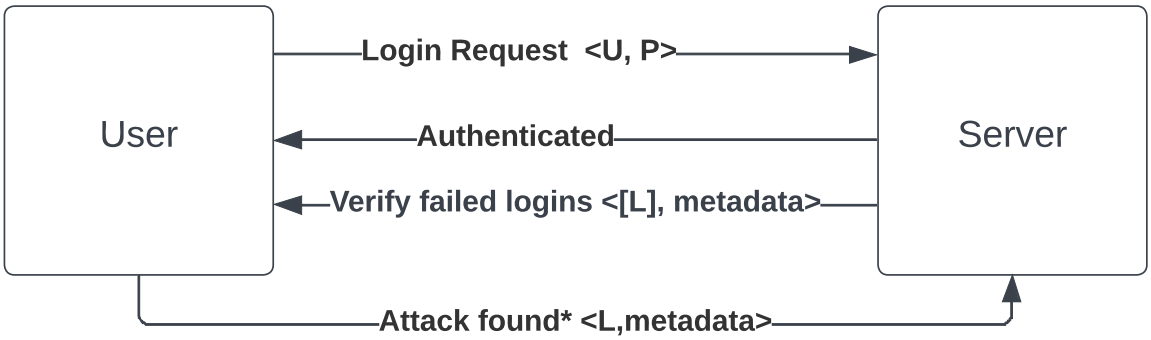



Figure 6.2: Mechanism to register and detect nonces

an attack has been detected. After the server identifies that a nonce has been submitted, and there's a possibility of an attack, the server communicates with the browser, after a user logs in, and attempts to identify the source of the attack.

The identification of attack can happen in the users' side or the server side. For attack identification to happen in the server side, the user needs to send all the saved information to the server. We discuss the advantages and issues with this method in § 6.3.4. Overall, due to privacy issues of sending all the information to the server, we decide to investigate the attack in the users' side.

We draw similarities from attack reconstruction and audit systems, which are widely used in post-mortem attack analysis [99, 168, 46]. Typically, the process includes recording events and agents to create a provenance graph, which is then used to trace the preceding or succeeding nodes and edges, starting from the specific agent or event in question. This way the system is able to identify the agents and events directly responsible for a given incident, and pinpoint events that directly cause a specific event or agent. So, the attack auditor is able to identify the source of the attack by tracing the preceding or succeeding nodes and edges, starting from the specific agent or event in question. This graph-based approach is deterministic and provides a clear path to identify the source of the attack. The process involves the following steps: (1) Recording events and agents during normal execution of the system, recording events occuring during the attack, (2) Analyzing the recorded events to identify the agents and events directly responsible for the attack.

**Recording attack information** The attack auditor records information about the system during normal execution. The information includes events, actors, and their connections. Usually, the information is stored in a graph structure, where the nodes are actors and events, and the edges are the connections between them. The graph structure allows the system to trace the preceding or succeeding nodes and edges, starting from the specific agent or event in question. The graph structure is deterministic and provides a clear path to identify the source of the attack, even though it might have some false positives.

To acquire the information, existing systems utilize logs of different events that happen, such as system calls, network connections, and file accesses in case of operating systems [168].

The logs are acquired by instrumenting the system and utilizing the exising audit logs of the software. Utilizing logs has the advantage of being less compute heavy, and most systems already record logs for a certain period of time. Web browser being run in a less powerful machine, we decide to utilize audit logs to record information about the system.

Systems involving linux kernels identify sensitive information sources such as /etc/passwd, /etc/shadow, and /etc/sudoers, and record the access to these files, among other accesses. In terms of credentials in the web browser, we identify that any form input field can contain sensitive information. So, we record the access to these fields.

**Possible attacks to detect** The most common attack to exfiltrate the credentials is through script execution in the browser that accesses the credential field. We identify two main ways these can occur: (1) malicious scripts that directly access the credential field, and (2) malicious extensions that inject scripts into the current page. Malicious scripts typically get loaded with the page in cases of cross site scripting attacks.

Another way malicious extensions can access the credential field is by utilizing the webRequest webextension API to read the outgoing request data. This threat vector is not as common as access through the DOM, but is unique to browser extensions.

Additionally, there are ways in which scripts don't access the credential field directly, but can infer the credential field information indirectly [71]. One such example is that the script triggers CSS attr styling based on the value of the credential field and reads the styling to infer the value of the credential field.

So, the attack auditor needs to be able connect credential field access to these adversaries. Identifying the adversaries and the methods they utilize to access the credential, we finalize the events that need to be recorded in the browser:

- DOM field loaded
- Script inserted by an extension
- Script inserted during page load.
- Access to the credential field via DOM

- Access to the outgoing request data

Figure 6.3 shows the graph of all the nodes and events that are recorded in the browser.

Uniquely identifying each individual script and extension is crucial for the attack auditor to identify the source of the attack. For extensions, we can utilize the extension id to identify them. However, scripts do not have a unique identifier. So, we utilize a hash of the script and url to identify them. To identify the session, we generate a unique identifier for each session, and include it in the graph. And then to identify input fields, we utilize the element id if present otherwise we generate unique id for the field. In order to preserve maximum information, we can include complete scripts and html for long term storage.

Also, the records can be stored for a long time before an attack is detected. During that time, a lot of scripts and extensions will have accessed the credential. To temporally divide the time sections, we introduce the concept of profiling indicator. Profiling indicator is a unique identifier string that identifies a period of time when the nonce was used. The browser creates unique profiling indicators for different periods of time, registers it along with the nonce during the *Nonce Verifier* mechanism, and utilizes it during the attack detection. The server provides the client with the profiling indicator for a more fine-grained analysis and to identify smaller set of possible attacks. The profiling indicator is included in the graph.

**Analysis of data** For analysis intrusion detection systems utlilize two different kinds of methods, mostly depending on the richness of recorded data, methods utilizing provenance by building a data flow graph or inference mechanisms such as machine learning and statistical reasonings. Inference methods are better in case the recorded information is not rich enough and there needs to be some intuitive inference about the data. Modern statistical and machine learning methods work pretty well for these, but still have lower accuracy. These methods are also slower. They also need to trained, and keep batch training during the execution to work which requires a powerful machine able to run the compute-heavy algorithms. Graph-based methods, however, deterministically create a graph of data flow path and analyze the graph. It's more accurate but requires rich data that can be converted into a graph.

Identifying our need for the mechanism to be globally implementable in less powerful machines, we decide to utilize graph-based method that requires minimal compute overhead.

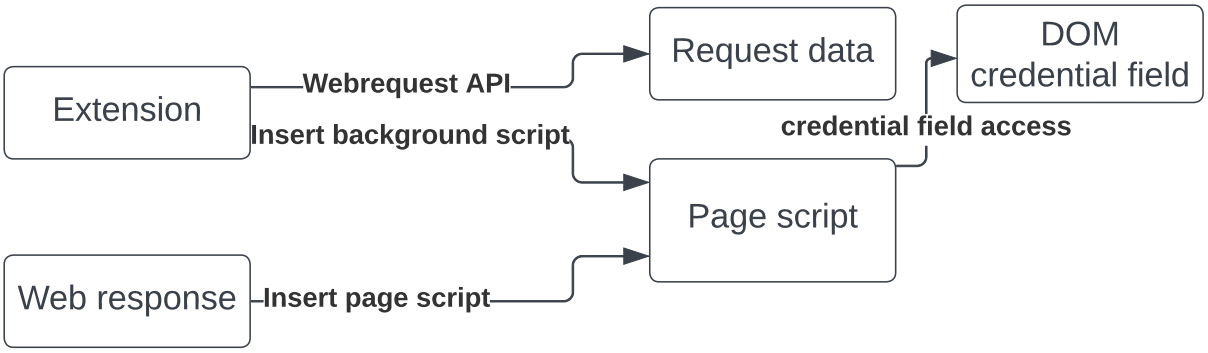



Figure 6.3: All nodes and events for attack audit

The processes stored(figure 6.3) are meant to provide a causality from either script or extension to credential access. In the graph, edges are events occuring in the browser. The nodes are different actors in these, such as browser extension, script, and credential field. Each individual browser extension, scripts, and credential fields have unique identifier associated with them. As we can see in figure 6.3, we can follow the graph back from each of these access to the script or extension that caused the access.

**Process** With the design of the attack auditor identified individually, the complete process breaks down to:

1. The browser saves the log information during it's normal run. The information include profiling indicator, nonce, events, and actors.
2. The profiling indicator is included during registration of nonce in the noce verifier.
3. When the server detects a possible attack has occured.
4. When a user is logged in, the server lets the user know that an attack has been detected, including a profiling indicator.
5. The browser uses given profiling indicator to identify the possible avenues of attack.
6. The browser identifies candidate attackers, and send this to the server.
7. The server uses the information identify the attacker (and utilize the script hash to remove genuine server scripts from candidate list)
8. The server sends back the refined list of attackers to the browser.
9. With the information of how the attack occured, the user and the server can do the response to the attack.

Any adversary listening to the requests and response can see the failed logins sent by the server to the client. If they keep listening to the overall communication, they might be able to find that a string was a nonce and it was detected. But at that point, the adversary has already been detected and the user and the server can take appropriate measures to prevent

further damage. The adversary will be able to know they've submitted a nonce, but only after they have already submitted a nonce and the attack has been detected.

#### Attack audit in the server

In the previous sections, we explored leveraging client-side computation, specifically the user's browser, for attack audit purposes. However, the server can act as an alternative for conducting the audit, should it receive all necessary information from the client. Given that browsers operate across a spectrum of devices with varying processing capabilities, the server undertaking the audit analysis emerges as a beneficial strategy. This entails the transfer of all collected information from the user's system to the server, allowing for a comprehensive audit.

The server-based attack audit's primary advantage is its ability to compile a holistic view of the attack by accessing data from all users. Moreover, the server can scale its computational and memory resources more effectively than individual user systems, providing an efficient means to store and analyze extensive data. Nonetheless, this approach is not without its drawbacks. The potential for computational overload becomes apparent with a large user base, as the server must manage and process significant volumes of data. More critically, this method poses substantial privacy risks, granting the server access to detailed information on the user's system, including scripts and extensions unrelated to any attack. Such access could inadvertently expose user-specific software, like ad blockers and VPNs, to misuse.

Further, the centralized nature of this approach introduces a significant security vulnerability, as any successful attack on the server could compromise the privacy of all users' systems en masse. After carefully considering these factors, especially the profound privacy concerns and security risks against the marginal benefits, we have decided against adopting this server-based audit method.

### 6.3.5 Utilizing a trusted third-party server

All previously discussed approaches rely on direct client-server communication for nonce detection. An alternative method involves employing a trusted third-party(TTP) server

to coordinate the detection and analysis of nonces. This server serves as an intermediary, informing both the client and the server of ongoing attacks. It offers a centralized repository for storing nonces and profiling indicators, and it notifies the main server of detected attacks. This arrangement allows the main server to consult the third-party server, which is more accessible than the user's browser, thereby reducing storage burdens on both the client and server sides.

Integrating a trusted third-party server for nonce detection presents distinct benefits. It acts as an intermediary, streamlining communication between the client and server regarding attack notifications. This server centralizes nonce storage and profiling indicators, alleviating storage demands on the client and server while facilitating broader attack analysis. Its consistent availability enhances the system's ability to detect and respond to threats efficiently.

However, this approach also introduces notable challenges, primarily concerning privacy. The first point of concern is establishing trust with the third party server. The server's access to extensive user data for all participating web servers poses risks of misuse, especially for advertising or tracking. Even if the users and servers trust the data usage by third-party, establishing trust is complex, with risks of impersonation by malicious entities. With the third-party server acting as a central repository for nonces and profiling indicators, it becomes a single point of failure, potentially compromising system security. Moreover, adopting a third-party server adds financial costs and system complexity, requiring careful oversight.

**Dealing with privacy issues in the third-party server** In the section below we discuss potential solutions to issues about trust of user data with the third party server. The privacy issues that come up with the third-party server are:

- TTP can identify usernames and credentials(as failed logins also contain typos of actual credentials) for the user.
- TTP can connect the user with the website server.
- If a malicious entity gains access to the TTP, it can access all the information about the user.

To prevent the TTP from identifying usernames and valid credentials, we utilize truncated hash based k-anonymity [100]. The basic idea for this is that the TTP only has access to truncated bytes of hash (If we assume a hashing algorithm that returns 8 characters as ABCDEFGH for the hash of msg, kn(msg) would return some truncated portion of the hash, i.e. ABCDE, instead of the whole hash). So, even if TTP can find a string that matches the hash, it can't be said with certainty that it is the actual credential because some information is lost in the truncation. So, using this method, we can prevent the TTP from identifying usernames and valid credentials for the user. As the TTP at no point has identifying information about logins or nonces, we also prevent malicious entities from inferring user's credentials or nonces even if they gain access to the TTP. As a downside, we introduce some possibility of collision and false positive.

The same can be done for server identification. The server and client both only submit truncated hashes of server identifier. However, this method introduces a new problem. Malicious entities can act like the server and send the credential thtey have to check if it's a nonce. To prevent this, we utilize signed messages using the server's private key. But, the downside of this is TTP can identify the server by the signature and connect user with the server. We relax this privacy requirement to improve security by not allowing malicious entities to check nonces.

In Figure 6.4, the following are the meaning of the symbols:

- U: username
- N: nonce
- L: Actual user credential
- PI: Profiling Indicator
- Y/N: The decision if an attack has been detected. Yes/No?
- Sign(_): Signed message
- kn(_): Truncated hash value.

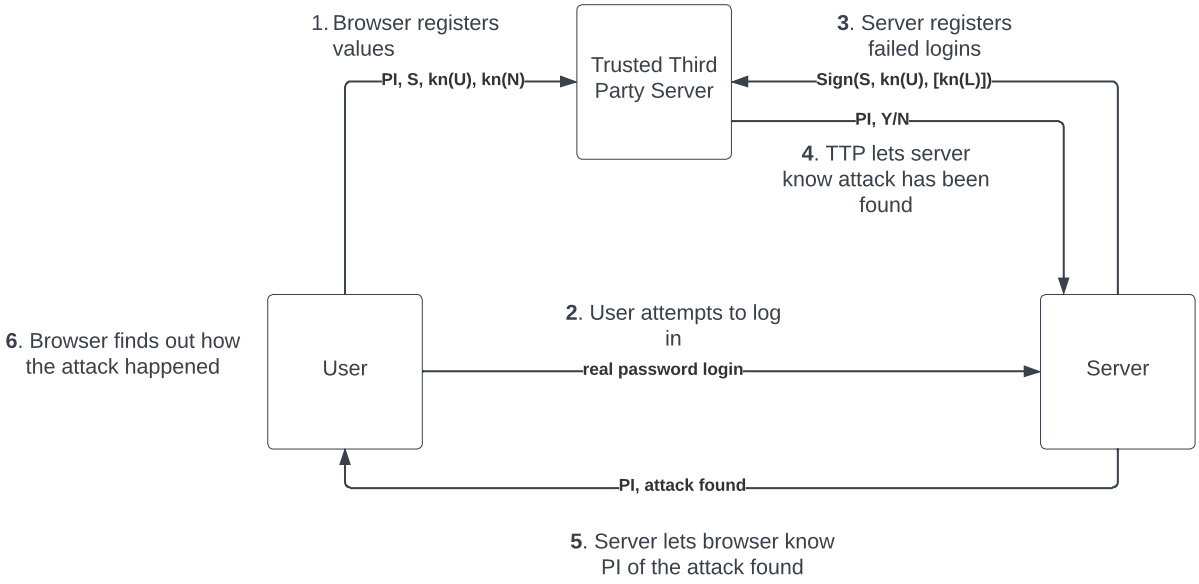



Figure 6.4: Mechanism to verify nonces using a trusted third-party server

Figure 6.4 shows communication between user (user browser), Trusted-Third Party Server, and the website server. Trusted-third party system is shown here to graphically separate out different functions happening in the system. However, direct communication between user and server is also possible.

The overall process that lets us verify a nonce, i.e. to detect an attack occured, is as follows:

1. The browser generates a nonce(N) and an accompanying profiling indicator(PI). This profiling indicator will help in further processes down the line.

2. For every generated nonce, the browser registers <PI,S, kn(U), kn(N)> to the trusted third party server.

3. The browser sends the actual credential to the server to log in.

4. The server saves all saved logins and registers it at the server by sending a signed message <S, kn(U), kn(L), Sign(S, kn(U), kn(L))> to the TTS.

5. The TTP performs the required process to figure out if an attack has occured. If an attack has occured, TTS lets the server know by including the PI.

6. Server lets the browser know an attack has happened with the specific PI.

7. The browser conducts processes to find out how the attack occured.

## 6.4 Implementation

In this section, we detail the implementation details of our system. For the server side, we implemented a custom website in flask that implements all the functionalities of the system. This includes endpoints to authenticate user, register nonces, and detect attacks. As the system requires new endpoints on the server side unlike Gautam et al. [59], we did not test our system on existing websites. On the browser side, we implemented:

- A browser extension that acts as a password manager to generate nonces, utilize the secure API[59] to insert actual credentials, and register the credentials with the server.

- Audit logs on Firefox 107.0 that provides information about script access to the DOM elements and all related events.
- A service that listens to the browser logs, identifies attacks and lets the server know about the attack.
- Malicious scripts that listen to password fields and exfiltrate the credentials.
- Malicious extension that inserts scripts into the page to exfiltrate credentials, as well as listens to outgoing requests using webRequest API.

In section 6.3 we went through different alternatives for the design of the system. There were multiple subjective design decisions that we chose as they were optimal for our design goals. But as goals of the system change, the design decisions might also change. For example, if the system is to be deployed in a corporate environment, the system might need to be deployed in a way that it does not impact the performance of the browser. Browser code is monolithic and change in code requires a lot of time to merge. So, design changes in the future will require changes in the browser itself, which is a difficult task. More changes in the browser might lead to vulnerabilities and performance impacts. For those reasons, we chose to limit the amount of changes required in the browser and decouple the subjective decisions from the browser. Therefore, we opted to implement a separate service that listens to triggers from the browser to identify attacks like in Mnenosyne [7, 99].

We implemented an auditor service in Python. The auditor service keeps listening to the browser logs and act on the triggers based on the audit logs. The audit logs are tagged as 'SecureBrowserChannel::' to identify just the required logs. The auditor service then parses logs one at a time to get to the goals of the system. As they're audit logs, all the logs are timestamped.

### 6.4.1 Generation of nonce

Password generation being trivial, we utilize the password manager to generate random passwords that comply with the existing system.

### 6.4.2 Verification of nonce

We opt to save credentials in the client side. The password manager inserts the credentials into the DOM and then registers the nonce with the browser. In the secure browser channel mechanism [59], the password manager sends the credentials to the browser only if it detects that the nonce is not tampered with. When the actual credential replacement take place, the browser triggers a log event 'RegisterNonce' in nsHTTPChannel containing the nonce and the url to be registered. The auditor service reads this and then records the nonce in a sqlite server for future reference.

At a later time, the user logs in to the server. The server checks to see if there's any failed logins between the current time and the last time there was a nonce verification event. If there is, the server sends an HTTP response to the browser with the header 'X-Nonce-Verification' and the body containing all the hashes of failed login credentials and timestamps, and related metadata. In the browser, we check if the header contains 'X-Nonce-Verification' when the channel is being created i.e. HTTPBaseChannel and if it does, we trigger a log event 'NonceVerification' containing the request body i.e. the hashes of the failed login credentials. The auditor then reads this log event and checks if any of the failed credentials match with the nonce. If there's a match, we found that an attack is detected. This triggers the attack audit phase of the system where we investigate the method of attack. For this proof of concept, we implemented the endpoint \sbc_attack at the website to let the website know that an attack has been detected for a specific failed login. To prevent any other agent to read the failed login credentials request, we remove the header 'X-Nonce-Verification' from the request.

### 6.4.3 Attack auditor

During normal functioninng of the browser, we utilize the browser logs to create a provenance graph that shows connection between script execution and credential field access. When an attack is detected, attack audit phase is triggered. The service then tries to collect all the possible agents that have accessed the credential field during the time of the attack. The service then tries to find the source of the attack by analyzing the provenance graph.

### Audit log data collection

We try to utilize existing browser audit logs to gather information about potential attack information. As we found that most existing logs weren't enough to trace back the source of the attack, we decided to insert our own logs to gather more information. We insert different log events at different points in the code. All logs are tagged as SecureBrowserChannel:: to identify the logs.

**Script load:** When a script is loaded by the browser in ScriptLoader, we log the whole script, script hash, url, and channel id in the log as 'ScriptLoad'. When the script is loaded, we trace the source of the script and log the source of the script as well. The source is a url that belongs to a remote domain or url belonging to a web extension.

**Element access:** We overload the value getter method of HTMLInputElement(all <input> fields) to log the raw html, url, and channel id in the log as 'ElementAccess'. When element access occurs, we also trigger a log event 'CallStack' to log the call stack of the script that accessed the element.

**Call stack:** When an element is accessed, some script will have called it from somewhere. Previous works log individual code block calls, and then try to trace the source of the call back in the provenance graph. This is very expensive as we're logging every single code block call. Also, malicious code can trigger benign code to access the element with multiple levels of redirection. So, actually tracing it back to that element is non-trivial, causing a lot of false positives.

In contrast to previous approaches, we query and then log javascript call stack traces when the element is accessed. This is more efficient way of tracking the source of access, as logging is only triggered when there's a possibility of malicious behavior. As the value of the input element is present in a single point, and any script requiring access to the data needs to access the getter, we can be confident that the call stack is the source of the access. This stack trace is logged in the log as 'CallStack'.

For that, we utilize the current JSContext of the channel. We follow frame by frame of the JSContext, printing the script, column, and line number of the frame. As we save the whole script already, this allows for any future analysis to trace back the source of the access.

For extension scripts, there's an additional step. The scripts have url for internal UUID of the extension like `0fb1d4a6-43e8-4a42-9098-d8bb80e69114` which is not very useful. We convert this to the extension id of the extension like `b3001491e9ad5b72fa596ac2c38fb2b6ba1991c6@temporary-addon`, by querying the internal uuid in the browser.

**Request access:** When a request is made by the browser, we log the url and the channel id in the log as 'RequestAccess'. When the browser emits the 'http-on-modify-request' event, we log all the extensions that listen on the request from 'WebRequest' API.

### Finding candidate agents

When the auditor service consumes the logs, we utilize the logs to find the candidate agents while the log is being parsed. For the script access, we walk through the call stack log to find all the scripts that access the element. An example call stack is shown in figure 6.5. In this example, the remote script 'https://example.com/login/pw_listener.js' is the source of the access and puts in a listener to the field. The listener is triggered due to web browser password manager inserting the credentials into the password field. It also contains all the script triggers from within the browser. So, for candidate agents we filter all the browser internal scripts identified by 'resource://gre/' and 'self-hosted' scripts.

The auditor service filters the browser interal scripts and then keeps a script of the remote scripts that access the element. The overall process of filtering can be found in Figure 6.6. As there can be indirect access to the element, instead of having just the top script, we keep all the scripts that access the element.

In case of remote script, we get store the url and the hash of the script. Scripts inserted by browser extensions have the browser id in their urls. So, these scripts can be easily identified and stored in the database. Also, we store the browser extension id from the WEbRequest API in the database.

In our mechanism, we are logging every single access to the element. This opens up to the attack of the system being overwhelmed by the logs. To prevent this, the auditor service only records unique candidates for a single session. The session can be differentiated with the

```
2024-04-07 17:36:00.654305 UTC - [Socket 1916219:Main Thread]: E/nsHttp
    SecureBrowserChannel::CallStack
0monitorPasswordChanges/<() ["https://example.com/pw_listener.js":12:6]
1_fillForm() ["resource://gre/modules/LoginManagerChild.jsm":2861:24]
2loginsFound() ["resource://gre/modules/LoginManagerChild.jsm":1345:9]
3loginsFound() ["self-hosted":1230:26]
```

Figure 6.5: Example call stack triggered by browser password manager and listener script

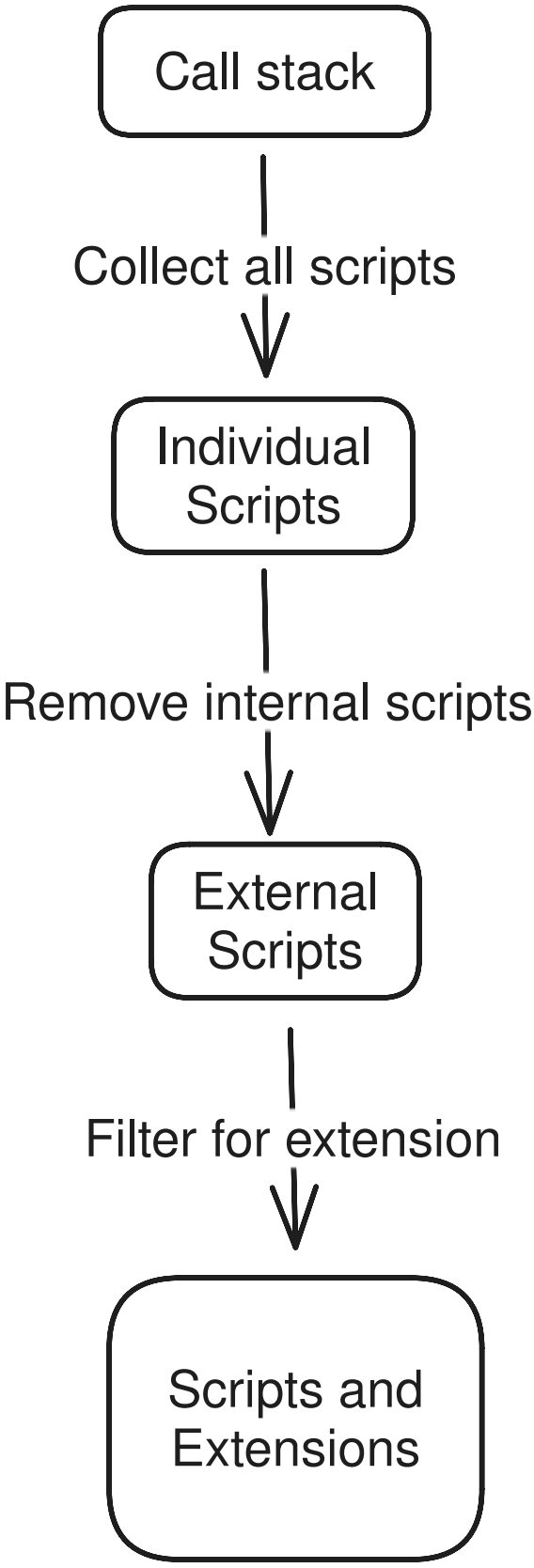



Figure 6.6: Filter call stack to find candidate agents

channel id. This way, the maximum number of elements that is being stored is the unique number of scripts and extensions working in the page.

### 6.4.4 Data from multiple users

From our mechanism, we can only find the candidate agents that have accessed the element. Due to the nature of the browser, finding candidate agents might be the only thing that can be found out from the logs. But the server has multiple users. So, the server can combine the logs of multiple users to find the actual source of the attack. The server can combine candidate agents of multiple users to find the actual source of the attack. This can be done by finding the intersection of the candidate agents of multiple users in the server.

**Script attacks** Some of the candidate agents common among multiple users are genuine scripts that are used by the website. The website utilizes the hash included with the script url to distinguish genuine script from malicious xss scripts. The common scripts among multiple users that are not in the website's whitelist are considered as the attacker scripts.

**Extension attacker** In case of an extension attacker, the extension id will be seen as a candidate attacker. When sent to the server, the server can see a pattern of same extension attacker in multiple users. A benign candidate is a password manager. But as extension identifiers are the same across different users, the server can differentiate between the password manager and the extension attacker.

## 6.5 Evaluation

In this section, we are evaluating the system for correctness and performance. For correctness, we run the system with different types of attacks to verify that the system can detect those attacks as discussed in the design section. In terms of nonce verification, the system should be able to register and verify that a failed login from the server is a nonce. Along with that, For the attack auditor, the system should be able to detect the scripts that are accessing the password field and the source of the script. The system would then be able to make a list of

unique scripts accessing the field and store them for future lookup. As all these operation affect the browser, we test that the added functionality in the browser would not overwhelm the existing performance of the browser. Furthermore, we test that the auditor can keep up with the logs generated by the browser so that the system is viable of running in real-time.

For evaluation, we implement a concept website that contains a user login page. We implemented detection and audit features independently to evaluate their correct working and performance. We implemented the auditor service in Python, which runs in the same machine as the browser and continuously parses the logs as they are being generated. Both the auditor and the browser are running in an XUbuntu virtual machine with a single core of Intel i9-9900 CPU and 6GB of RAM. As our service runs in parallel to the browser, most of the time overhead for the browser is limited to the generation of the logs. All the timing overhead in the browser is measured using `chrono::high_resolution_clock` in C++, by adding measuring code snippets around the code that is being measured. For the auditor, `datetime.datetime` is used in Python. Using our proof of concept website, we evaluate the performance of the system.

### 6.5.1 Nonce verifier

We ran the complete nonce verification process i.e. registration of nonce from the browser, and checking for the nonce in the failed logins.

**Registration of nonce** The password manager extension inserts the credentials into the login form during the submission phase. A single line of additional log is generated during the registration of the nonce that is parsed by the auditor service. The generation of singular log during the submission phase takes 763ns. Comparing with the whole process of submitting the form, which takes 2 seconds, this is an increase of $38*10^{-6}$%. A very negligible timing overhead. The time taken by the auditor service to parse the line of log and then insert into the sqlite database if the nonce is not present is 20.41ms, shorter than the time taken to submit the form. This short amount of time makes it possible that the auditor can keep up with the browser logs in real-time.

Storage space might be a scarce resource on the client side where the browser is running, so we evaluate the additional storage space required for our system. We assume that 1 million failed logins occur in the website between the time of previous nonce check and the current nonce check. As the server will have different throttling mechanisms to control the number of failed logins that can be input for an account and the number of failed logins will be limited. 1 million failed logins correspond to 2 months of failed logins at the rate of one failed login request every 5 seconds. In the server, the 1M failed logins take up 65.2MB space without compression. This isn't a huge storage space for the server. And servers usually always store logs for a certain amount of time for debugging purposes. The number of nonces will always be less than that as there is only so many times a user logs into the system. If a user logs into the page once a day, 1000 nonces is a good estimate for 3 years time. Storage of 1000 nonces in the sqlite database by the auditor service takes up 136.72KB space. This small amount makes it possible that even devices with limited storage space can run the this system without running into memory issues.

**Checking for the nonce** For checking of the nonce, our server sends a static list of 1M failed logins to the browser with 'X-Nonce-Verification' header. This is a separate request to the browser which opens up it's own HTTPChannel and does not affect the performance of when the user is in the website. The auditor takes up 0.89 seconds(897.35 ms) time to check if any of the 1M failed logins in the list is present in the nonce database. This list of failed logins doesn't need to be stored locally as it is only used for verification. Due to this very short amount of time required for checking the credentials, the auditor can keep up with the browser logs in real-time and we can detect the nonce in real-time.

### 6.5.2 Attack auditor

The attack auditor is different than the nonce verifier as there can be different methods of attack.

**Methods of attacks** We identify different methods by which an adversary can try to access the field, to verify that the auditor can detect the scripts that are accessing the field.

We first identify the methods by which scripts could be added to the current page, with our threat model in mind. We identify the following methods: (1) developer console, (2) inline script in the HTML served, (3) script file attached to the HTML served, and (4) content script injected by a malicious extension. The only method to read the value of the field is by using the value attribute of the field[111]. We also consider the possibility of scripts accessing the credential in an indirect way, i.e. by using a listener on the field to fetch the value on different events or by using a timer to periodically check the value of the field.

Combining the method by which scripts could be added to the page and the method by which the scripts could access the field, we identify all the possible methods using a cartesian product. For each of the method above, we were successfully able to identify the scripts ran, and the source (devtools console, inline script, separate script, malicious extension) of the script. In the figure 6.7a, we show an example where the same field is accessed by two different scripts. The auditor successfully keeps track of the candidates and stores it in the sqlite database as can be seen in 6.7b.

**Performance** To identify if the auditor is able to run in real-time, we try to verify how long does it take to parse the logs generated in an hour. For this, we're trying to have a scenario which is the upper limit of the logs that would be generated for a login page. We utilized the same proof of concept website containing the login form for this evaluation. We ran 100 scripts in the website that accessed the password field and 2 scripts that did not. The scripts had a timer that ran every 5 seconds to check the value of the password field, triggering an audit log. We ran the whole system for 1 hour for performance verification. This is a very unlikely scenario as the number of scripts that access the password field will be limited to the number of scripts that are running in the website and will be far lower than 100.

As we discussed in the implementation section 6.4, even though there could be multiple times that a specific script accesses the input field, we only need to store unique agents that access it. We do store the raw log for future use, but for this research the raw logs are not used and can be deleted after a certain time. The auditor checks if the agent is already

```
2024-04-07 18:43:00.913254 UTC - [Socket 1916219:Main Thread]: E/nsHttp
    SecureBrowserChannel::ElementAccess <input type="password"
    placeholder="Enter Password" name="psw" id="psw" required="">
2024-04-07 18:43:00.919102 UTC - [Socket 1916219:Main Thread]: E/nsHttp
    SecureBrowserChannel::CallStack
0monitorPasswordChanges/<() [
    "https://web.eecs.utk.edu/~agautam1/login/pw_listener1.js":12:6]
1_fillForm() ["resource://gre/modules/LoginManagerChild.jsm":2861:24]
2loginsFound() ["resource://gre/modules/LoginManagerChild.jsm":1345:9]
3loginsFound() ["self-hosted":1230:26]

2024-04-07 18:43:01.291747 UTC - [Socket 1916219:Main Thread]: E/nsHttp
    SecureBrowserChannel::ElementAccess <input type="password"
    placeholder="Enter Password" name="psw" id="psw" required="">
2024-04-07 18:43:01.292232 UTC - [Socket 1916219:Main Thread]: E/nsHttp
    SecureBrowserChannel::CallStack
0monitorPasswordChanges/<() [
    "https://web.eecs.utk.edu/~agautam1/login/pw_listener2.js":12:6]
1_fillForm() ["resource://gre/modules/LoginManagerChild.jsm":2861:24]
2loginsFound() ["resource://gre/modules/LoginManagerChild.jsm":1345:9]
3loginsFound() ["self-hosted":1230:26]
```

(a) Multiple call stacks for the same field access

```
timestamp | url | field_id | agent_url | hash
2024-04-07 18:43:00.919102 UTC| https://example.com/login/ | psw |
    https://example.com/login/pw_listener1.js | d9f2d16bb5f0
2024-04-07 18:43:01.292232 UTC| https://example.com/login/ | psw |
    https://example.com/login/pw_listener2.js | d9f2d16bb5f0
```

(b) The database entry created by the auditor. (The hash of the script is truncated for demonstration purposes.)

Figure 6.7: The logs generated by the browser and the database entry created by the auditor for multiple call stacks

present for this url and field, and if it is, it doesn't store the log. So, this reduces the number of logs to unique agents acting in the page (i.e. scripts, extensions, etc).

With 100 scripts running in the page and accessing the password field, the auditor requires 1.34 MB space to store the logs and 28.19 seconds time to parse the complete log. The time taken is vastly sandbagged by the time taken to insert individual agents into the sqlite database, which can be optimized by using bulk insert methods. The unlikely scenario of batch processing of all the logs for the hour, the time taken is vastly reduced to 0.35 seconds (351.21 ms). As 3600 seconds of execution logs can be parsed in 28.19 seconds(a ratio of 127:1), the auditor can keep up with the logs generated by the browser in real-time. This is important as the browser logs aren't permanent and should be parsed in real-time to avoid losing the logs in case of memory issues or browser crashes.

## 6.6 Discussion

**Widespread deployment** The implementation here, in contrast to the secure browser channel [59], requires changes to the websites. Future work can look into how to help websites to adopt this mechanism without requiring much effort. This could be a framework or a microservice that could be deployed in the website's server, which can be easily deployed alongside the main website through containerization. We also explore a trusted third-party architecture, which minimizes the functions of the website, but still requires some changes. The trusted thrid-party architecture could be a good starting point for the widespread deployment, even though this introduces aspects of trust that need to researched further.

**Extension of audit capabilities** This work only looks into sensitive fields and their access in the browser. There are other mechanisms such as downloading and deploying malware, etc that can be a mechanism of attack in the browser. So, recording most actions that happen in the browser can be a good mechanism to keep track of the provenance of the browser. This extended provenance can be used in the future to keep track of attacks that are unknown right now but are discovered in the future. So, future work can look into extending the audit capabilities of this system with more adversaries and provenance information.

**Analyzing full scripts** When we were analyzing data during finding the cause of attack, we only used the identifier of the scripts and where they originated from. This successfully identifies the script that is responsible for the access, possibly indicating that this script is malicious. However, there is much more information that can be gathered from the analysis of whole scripts that are saved. There are challenges in this approach, as this requires navigating through different obfuscation, minification, and other techniques that scripts could use to hide their true purpose. Future work can look into analyzing these full scripts to get more fine-grained information about the attack.

## 6.7 Conclusion

In this work, we proposed a mechanism that utilizes nonce-based password replacement mechanism to detect and audit password theft. We utilized the nonce-based password replacement method, and the intuition that the usage of nonces can help identify an attack. We also utilized the browser audit logs to identify the scripts and extensions responsible for accessing sensitive information. We identified different design decisions that need to be made to implement the system, and reasoned about the trade-offs in the design decisions. We implemented proof of concept system in the Firefox browser and then evaluated the performance impact in the system. Through our evaluation, we show that the system does not have a significant performance impact in terms of time and storage overhead making it possible to deploy in real-world scenarios even in memory and computationally constrained devices. We also show that the system is able to consume the logs in real-time without getting overwhelmed.

# Chapter 7

# Conclusion and Future Works

In this dissertation, we utilized the concept that the password manager could be supported better by standardizing interactions for different authentication scenarios. We identified scenarios in which password managers could be supported better by standardizing interactions for different authentication scenarios. By making different parts of web authentication aware of the password manager and introducing standardized interfaces, we were able to improve the usability and security of password-based authentication.

In Chapter 3, we developed a PCP language that websites and password managers can use to support the generation of compliant passwords. With websites better supporting the generation feature of the password managers using the PCP language, we introduced automatic generation of PCP-compliant passwords. We hope that our work will signal to both communities that adopting a PCP language has tangible benefits. For websites, it allows them to unify their PCP specification and checking, allowing changes to the PCP file to automatically update how checking happens on both the client and server. For password managers, it not only improves the usability and utility of password management but also supports opinionated generation algorithms (e.g., mobile-aware generation [63], security-focused generation [124]), which would otherwise frequently generate non-compliant passwords. Utilizing a user study, we demonstrated that our PCP language is easy to author and understand, even for complex policies.

Moving from usability, in Chapter 4, we explored how password managers can help improve security of password entry in the browser. We identified a strong threat model for password

exfiltration and explored five different designs to secure password entry. We implemented the most secure design, which involves injecting a fake password into the web page and then replacing it with the real password in the browser. We demonstrated that this design can stop credential exfiltration by malicious client-side scripts and browser extensions. On evaluation, we find that our design works with 97% of the Alexa top 1000 websites improving the security of credential entry on the vast majority of websites. For rest of the websites, it is easy to automatically detect compatibility issues and not use our secure autofill API, preventing any functionality regressions.

To show that our secure browser channel is not just limited to passwords, in Chapter 5, we extended our work to FIDO2 authentication. We show that similar to passwords, FIDO2 authentication can also be secured against local adversaries. We implemented the secure browser channel for FIDO2 authentication and demonstrated that it can stop credential exfiltration by malicious client-side scripts and browser extensions. Even though these attacks are not common currently, it is important to proactively tackle local attacks due to the significant threat posed by XSS, malicious extensions, and malware. Delaying the response until after such attacks occur puts users at risk unnecessarily.

Finally, we show that the secure browser channel can be used for more than just securing password entry. In Chapter 6, we demonstrate how the secure browser channel can be used to detect and audit attacks. Utilizing the credential swap mechanism introduced in Chapter 4 and Chapter 5, we were able to implement a server architecture capable of detecting credential theft and then triggering audit mechanisms to identify the root cause of the attack. Utilizing the browser audit logs and sensitive input elements, we were able to find the agents that could be responsible for the attack. We show that we can do this without much performance overhead in the browser.

With these works, we show that password manager aware authentication can improve existing security and usability issues in authentication systems. We hope that our work will inspire further research in this area and that the secure browser channel will be adopted by password managers and websites to improve the security and usability of web authentication.

## 7.1 Lessons Learned

Below, we discuss key lessons learned from our research over the course of this dissertation.

### 7.1.1 Password managers as an opportunity

The major takeaway of the whole dissertation is that password managers are a great opportunity for improving the security and usability of web authentication. Currently, password managers are used for generation, storage, and autofill of passwords, but they can do more than that. Even in the current condition, password managers are central in authentication and has been shown to secure users from various threats by providing them with strong, unique passwords. By making the browser and the server aware of the password manager, we can further improve the security and usability of web authentication. By leveraging the use of password managers, we can improve the security and usability of web authentication without requiring users to change their behavior. We see that password managers can be used to generate secure compliant passwords, secure the entry of passwords more than manually entering them, and detect and audit attacks in the browser. With more research, password managers can see more use cases, and act as a central point of secure authentication for users.

### 7.1.2 No standard PCP

In chapter 3, we found that there is no standard PCP that websites follow. With the dataset we collected, that expands across geographical regions and huge range of ranks, we found that there is no standard PCP that websites follow. Even though NIST has guidelines on how to create a PCP, websites do not follow them. In terms of length requirements, which we found the most important for password strength, websites have a wide range of requirements. Some websites require a minimum of 6 characters, while others require a minimum of 20 characters. There is a vast distribution for character and composition requirements of password policies. There isn't even a standard definition for the "symbols" character set. Some websites consider only a few characters as symbols, while others consider a wide range of characters as symbols. This, no standard PCP, is a huge source of user frustration. Deployment of our PCP should

help websites to standardize their PCP and make it easier for users to generate compliant passwords.

### 7.1.3 Humans generate passwords differently than machines

In chapter 3, with the insights of previous work about password datasets, we saw that humans from different geographical regions have different preferences for passwords. PCPs are usually enforced by websites to force humans to select stronger passwords. We found that using the PCPs enforced by websites, half of randomly generated passwords are good enough against offline attacks. But for randomly generated passwords, PCPs actually make the password search space smaller, making the passwords weaker. Taking human preferences into account, almost all the passwords aren't good enough against offline attacks. So, there's a paradoxical situation where PCPs are enforced to make passwords stronger, but they make randomly generated passwords weaker and don't make human-generated passwords strong enough. The multi-rule PCP that we found to be supported by some websites is a good compromise to force humans to make stronger passwords and not make randomly generated passwords weaker.

### 7.1.4 Misdirected security concerns

In chapter 4, we found that some websites are concerned about the security of submitted forms. So, they introduce different mechanisms such as encryption, hashing, and encoding to secure the form data before being submitted to the server. This shows that websites are concerned about the security of the data being submitted to the server. Previous research has shown that TLS is pretty secure against eavesdropping [93, 5, 94, 95], so data in tranmission isn't as big of a concern. However, as the actual process of manipulating data before submission is done by scripts, the process of the manipulation in itself is more vulnerable to attacks. This points towards a misdirected concern for security, that websites are more concerned about the data being submitted to the server than the data being manipulated by scripts before submission.

### 7.1.5 Need to secure against local attacks

Throughout the work, we see that these local attacks within the browser such as XSS attacks and malicious extensions are very much feasible. XSS attacks are in the OWASP top 10 list of web application security risks [132], with huge amounts of CVEs being reported every year. In chapters 4 and 5, we show that malicious extensions are also a huge threat to the security of the browser. There are a lot of extensions currently in the Chrome Web Store, that can exfiltrate credentials from the browser. Research has shown that users do not pay much attention to the permissions that they give to the extensions [86]. This shows that these attacks are very much feasible and need to be secured against. It is difficult to measure how much these attacks are being executed currently, but it is important to proactively secure against these attacks before they become rampant.

### 7.1.6 Need for provenance system in the browser

In chapter 6, we implement a system that is able to detect and audit attacks on password credentials in the browser. We show that it is feasible to store this data in the local system and analyze it to find the root cause of the attack. However, this system currently works against identified attacks. If there's a provenance system in the browser that can compactly record the actions in the browser, it could be easier to identify other attack vectors that are not currently identified.

## 7.2 Future Works

With the research conducted in this dissertation, we have laid the groundwork for improving the security and usability of password-based authentication. There are several avenues for future work that can build on the work presented in this dissertation.

### 7.2.1 Further improving password generation

In our work with the PCP language in Chapter 3, we have shown that it is possible to standardize and enhance password generation processes. The PCP language provides a good

and easy way to describe password policies and generate compliant passwords. In our current application, we propose websites using the PCP language to generate compliant passwords. We looked at restrictions on the PCP due to website policies and dictated by other policies.

PCPs can also be dictated by user requirements rather than website requirements too. Users have to input passwords in many different kinds of devices with various input modalities. And research has shown that having to input passwords into different devices impacts users decision on using a password generator. For example, users' passwords for streaming services are often entered on smart TVs, which have different input modalities than a computer. Input of passwords in different devices is a major pain point for the users. Future work should explore the different types of devices that users input their passwords in. This information helps guide the devices requiring immediate attention for password generation. Furthermore, user studies can be conducted to understand the pain points and character set preferences for users in different input modalities. With this information at hand, password generation can be tailored to the user's input modality, making it easier for the user to generate and input passwords.

On the other hand, users have other preferences at hand. Such as users preferring passwords to not contain characters that can easily be confused with each other (e.g., '1' and 'l'). Future work should explore these human driver requirements and how to incorporate these preferences into the PCP langauge and password generation.

Even though we have shown that the PCP language is easy to author and understand, websites maintainers still have to write the PCP language. This is the biggest bottleneck in the process of automatic password generation support. Future work can explore how to automatically extract PCPs from websites, utilizing descriptions in the websites, the javascript used to validate the passwords, and other black box techniques. This way, the creation of PCP can be automated and websites can easily support automatic password generation.

### 7.2.2 Securing manual entry of passwords

In Chapter 4, we explored how password managers can help improve security of password entry in the browser. Future work can explore securing password entry when users manually

insert them. The browser can provide a secure credential entry mode, such as a special entry interface, where users can securely insert their credentials.

Research needs to be done on multiple ends for this to be successful. Firstly, as extensions are able to mimic most of the browser's behavior, users need to be able to know that this entry interface is safe. For this, research needs to be done on how to make users aware of this functionality and how to encourage them to use it. Secondly, research needs to be done on how to design the password entry mode such that it is clear when it is activated.

### 7.2.3 Enhanced Application Support for Secure Browser Channels

The introduced defense strategy is applicable to additional APIs that handle sensitive data without depending on JavaScript within the browser. By creating a secure communication channel for the exchange of request and response parameters, the integrity and confidentiality of data transmitted via these APIs can be safeguarded against harmful browser extensions.

Web servers can signal their desire to employ *sbc-FIDO2* through standardized headers designed to block unauthorized browser extensions from accessing sensitive data. Alternatively, web servers might choose not to use these headers if they prefer to allow extensions access to the data. This approach grants web servers the flexibility to tailor their use of the mechanism to fit various requirements.

This defense approach could enhance the security of numerous existing APIs that manage sensitive data external to the browser. For example, the Clipboard API enables web applications to handle clipboard commands and interact seamlessly with the system clipboard. The File System Access API offers functionalities for reading, writing, and managing files, whereas the File API provides access to file contents. The File and Directory Entries API allows web applications to simulate a local file system, making it possible to easily navigate and manage files. Moreover, the Geolocation API permits users to share their location with web applications, and the Media Capture and Streams API supports the smooth capture and streaming of audio and video media.

In addition, several experimental browser APIs can be secured against browser extensions without direct JavaScript interaction. These include Web Bluetooth (for connecting to Bluetooth Low Energy devices), Barcode Detection (for recognizing barcodes in images),

WebOTP (for verifying phone numbers using one-time passwords), Web NFC (for NFC data exchange), HID (for interfacing with human input/output devices), WebUSB (for accessing services of non-standard USB-compatible devices), MediaStream Image Capture (for capturing images and videos), and Contact Picker (for selecting and sharing limited contact information).

Further APIs could also benefit from this protective measure. The main point is that by implementing this proposed defense strategy, web servers can significantly improve the security of data transfers and defend against unauthorized manipulation or access by browser extensions, thus ensuring the integrity and confidentiality of sensitive data.

### 7.2.4 Usability study of security indicators

Not much is known about how users perceive signals about security. Systems like Fidelius [51] utilize physical lights as security indicator, and systems like Dynamic Security Skins [45] visual software indicators to indicate to the user that the security feature is enabled. Similar indicator can be used to indicate to the user that the secure browser channel is enabled. But users are known to ignore these indicators(like the padlock icon in the browser), and there isn't a good understanding of how to design these indicators to make them more noticeable. Future work can explore how to design these indicators to make them more noticeable and how to make users more aware of these indicators. User studies can be conducted to understand how users perceive different security indicators and how to design them to make them more noticeable. UX elements such as color, size, and position can be explored to make these indicators more noticeable in the browser, the password manager, and/or the operating system. Similar research can be done for supplementary hardware indicators, such as lights on the keyboard or 2fa devices.

### 7.2.5 Stronger threats for credentials

In chapters 4 and 5, we explored the threat model for password exfiltration. The threats we specifically looked at were client-side scripts, the `webRequest` API, and during network transmission. However, there are stronger threats that are able to exfiltrate credentials.

Threats such as a completely compromised browser, malware, and software and harware keyloggers are able to exfiltrate credentials. These threats are strictly stronger than the threats that we consider, but we do not consider them due to the difficulty of executing these attacks. But nonetheless, these threats are still possible and should be considered in future work. It shouldn't be left for when these threats become viable, but rather proactively tackle these threats before they become viable. For completely compromised systems, future work can explore utilizing secure enclaves to secure the password entry process. An operating system based secure credential entry channel, where the operating system in a secure enclave handles authentication centrally, can be a good solution against a stronger threat mode.

### 7.2.6 Trusted extensions in the browser

In Chapter 4, one of the biggest concern to improve security was because the password manager had the same abilities as any malicious extensions. Users already put in a lot of trust in the password managers that they use to store their credentials and other sensitive data. Also, there are a limited number of password managers in use currently. Future work can explore a trust hierarchy for extensions in the browser, so that password managers are trusted more and have more(or exclusive) capabilities than other extensions. This way, the password manager can have more capabilities to secure the password entry process and other sensitive data in the browser.

### 7.2.7 Stronger audit mechanisms

In Chapter 6, we explored how to detect and audit attacks. On the client side, we were able to detect candidate scripts and extensions that could be responsible for the attack. In this research, we limited our scope to password based attacks. As we proposed, the secure browser channel can be extended to other applications. Future work can explore how to extend the secure browser channel to other applications and how to detect and audit attacks in these applications. Future research can explore further mechanisms to accurately identify attack mechanisms. Firstly, if there's much obfuscation in the script, it is hard to accurately identify the scope of it. Future research can explore how to accurately identify the attack

even if the script is obfuscated. A complete provenance mechanism in the browser handling most events, in addition to a consolidated browser secure channel for different APIs, could provide a comprehensive audit mechanism. Secondly, future research can look into static analysis methods to identify attacks while the attack is happening than afterwards. Also, future research can explore how to accurately predict the type and scope of an attack with the data from multiple users, while still maintaining the privacy of the users.

# Bibliography

[1] (2013). Krypton | let's make two-factor easy & secure. `https://krypt.co/`. 23

[2] Adams, A. and Sasse, M. A. (1999). Users are not the enemy. *Communications of the ACM*, 42(12):40–46. 16

[3] Akerlof, G. A. and Shiller, R. J. (2015). Phishing for phools. In *Phishing for Phools*. Princeton University Press. 17, 68

[4] Aldiabat, K. M. and Le Navenec, C.-L. (2018). Data saturation: The mysterious step in grounded theory methodology. *The Qualitative Report*, 23(1):245–261. 29

[5] AlFardan, N., Bernstein, D. J., Paterson, K. G., Poettering, B., and Schuldt, J. C. (2013). On the security of {RC4} in {TLS}. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 305–320. 148

[6] Alkaldi, N. and Renaud, K. (2016). Why do people adopt, or reject, smartphone password managers? *EuroUSEC*. 1

[7] Allen, J., Yang, Z., Landen, M., Bhat, R., Grover, H., Chang, A., Ji, Y., Perdisci, R., and Lee, W. (2020). Mnemosyne: An effective and efficient postmortem watering hole attack investigation system. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 787–802. 24, 132

[8] Almeshekah, M. H., Gutierrez, C. N., Atallah, M. J., and Spafford, E. H. (2015). Ersatzpasswords: Ending password cracking and detecting password leakage. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 311–320. 117

[9] Alqubaisi, F., Wazan, A. S., Ahmad, L., and Chadwick, D. W. (2020). Should we rush to implement password-less single factor FIDO2 based authentication? In *2020 12th Annual Undergraduate Research Conference on Applied Computing (URC)*, pages 1–6. IEEE. 22

[10] Alsaffar, M., Aljaloud, S., Mohammed, B. A., Al-Mekhlafi, Z. G., Almurayziq, T. S., Alshammari, G., and Alshammari, A. (2022). Detection of web cross-site scripting (xss) attacks. *Electronics*, 11(14):2212. 105

[11] Amadeo, R. (2014). Adware vendors buy chrome extensions to send ad- and malware-filled updates. 105

[12] Anand, M. K., Bowers, S., and Ludäscher, B. (2010). Provenance browser: Displaying and querying scientific workflow provenance graphs. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 1201–1204. IEEE. 24

[13] Angelogianni, A. (2018). Analysis and implementation of the FIDO protocol in a trusted environment. Master's thesis, University of Piraeus. 23

[14] Aurigemma, S., Mattson, T., and Leonard, L. (2017). So much promise, so little use: What is stopping home end-users from using password manager applications? 1

[15] Awake Security (2022). Discovery of a massive, criminal surveillance campaign. `https://awakesecurity.com/blog/the-internets-new-arms-dealers\-malicious-domain-registrars/`. 19, 93

[16] Bakry, T. H. and Mysk, T. (2020). Precise location information leaking through system pasteboard. `https://www.mysk.blog/2020/02/24/precise-location-information-leaking-through-system-pasteboard/`. Accessed: 2020-06-13. 17, 20

[17] Bangor, A., Kortum, P. T., and Miller, J. T. (2008). An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction*, 24(6):574–594. 41

[18] Bates, A., Tian, D. J., Butler, K. R., and Moyer, T. (2015). Trustworthy {Whole-System} provenance for the linux kernel. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 319–334. 24

[19] Bates, D. (2021). Proposal: Html passwordrules attribute. `https://github.com/whatwg/html/issues/3518`. 15, 55

[20] Bhargav-Spantzel, A. (2014). Trusted execution environment for privacy preserving biometric authentication. *Intel Technology Journal*, 18(4). 23

[21] Bonneau, J., Herley, C., van Oorschot, P. C., and Stajano, F. (2012). The quest to replace passwords: a framework for comparative evaluation of web authentication schemes. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 553–567. IEEE, IEEE. 1, 26, 69, 79

[22] Brinkmann, M. (2019). Hoverzoom's malware controversy and imagus alternative - ghacks tech news. 105

[23] Brooke, J. (1996). Sus: a "quick and dirty'usability. *Usability evaluation in industry*, 189(3). 38

[24] Cato Networks (2022). Threat intelligence feeds and endpoint protection systems fail to detect 24 malicious chrome extensions. `https://www.catonetworks.com/blog/threat-intelligence-feeds-and-endpoint-\protection-systems-fail-to-detect-24\-malicious-chrome-extensions/`. 19, 93

[25] Chakraborty, D. and Bugiel, S. (2019). simFIDO: FIDO2 user authentication with simTPM. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2569–2571. 23

[26] Chang, D., Mishra, S., Sanadhya, S. K., and Singh, A. P. (2017). On making U2F protocol leakage-resilient via re-keying. *IACR Cryptol. ePrint Arch.*, 2017:721. 22

[27] Chrome, G. (2023). Chrome.webrequest. `https://developer.chrome.com/docs/extensions/reference/webRequest/#event-onBeforeRequest`. Accessed: 2023-05-03. xiv, 64

[28] Chromium (2011). 91191 - chromium - webrequest: Access to post data in 'onbeforerequest'. `https://bugs.chromium.org/p/chromium/issues/detail?id=91191`. Accessed: 2023-05-03. 65, 76

[29] Chromium (2023a). The activetab permission. https://developer.chrome.com/docs/extensions/mv3/manifest/activeTab/. Accessed: 2023-05-03. 66

[30] Chromium (2023b). chrome.declarativenetrequest. https://developer.chrome.com/docs/extensions/reference/declarativeNetRequest/. Accessed: 2023-05-03. 66

[31] Chromium (2023c). chrome.scripting. https://developer.chrome.com/docs/extensions/reference/scripting/. Accessed: 2023-05-03. 66

[32] Chromium (2023d). chrome.webrequest. https://developer.chrome.com/docs/extensions/reference/webRequest/. Accessed: 2023-05-03. 66

[33] Chromium (2023e). Content scripts. https://developer.chrome.com/docs/extensions/mv3/content_scripts/. Accessed: 2023-05-03. 66

[34] Chromium (2023f). Manifest file format. https://developer.chrome.com/docs/extensions/mv3/manifest/. Accessed: 2023-05-03. 66

[35] Ciampa, M. (2013). A comparison of user preferences for browser password managers. *Journal of Applied Security Research*, 8(4):455–466. 21

[36] Cimpanu, C. (2018). Mega.nz chrome extension caught stealing passwords, cryptocurrency private keys. 105

[37] Ciolino, S., Parkin, S., and Dunphy, P. (2019). Of two minds about two-factor: Understanding everyday FIDO U2F usability through device comparison and experience sampling. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*. 22

[38] Condé, R. C., Maziero, C. A., and Will, N. C. (2018). Using Intel SGX to protect authentication credentials in an untrusted operating system. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00158–00163. IEEE. 23

[39] CVEdetails (2024). Security vulnerabilities, cves, xss, cross site scripting published in january 2024. 18

[40] Dambra, S., Sanchez-Rola, I., Bilge, L., and Balzarotti, D. (2022). When sally met trackers: Web tracking from the users' perspective. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2189–2206. 17

[41] Das, A., Bonneau, J., Caesar, M., Borisov, N., and Wang, X. (2014). The tangled web of password reuse. In *Proceedings of the 22nd Network and Distributed System Security Symposium*, volume 14, pages 23–26. Internet Society. 26

[42] Das, S., Dingman, A., and Camp, L. J. (2018). Why johnny doesn't use two factor a two-phase usability study of the FIDO U2F security key. In *International Conference on Financial Cryptography and Data Security*, pages 160–179. Springer. 22

[43] Dauterman, E., Corrigan-Gibbs, H., Mazières, D., Boneh, D., and Rizzo, D. (2019). True2f: Backdoor-resistant authentication tokens. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 398–416. IEEE. 22

[44] Dell'Amico, M., Michiardi, P., and Roudier, Y. (2010). Password strength: An empirical analysis. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE. 26

[45] Dhamija, R. and Tygar, J. D. (2005). The battle against phishing: Dynamic security skins. In *Proceedings of the 2005 symposium on Usable privacy and security*, pages 77–88. 87, 152

[46] Ding, H., Zhai, J., Deng, D., and Ma, S. (2023). The case for learned provenance graph storage systems. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3277–3294. 122

[47] Dionysiou, A. and Athanasopoulos, E. (2022). Lethe: Practical data breach detection with zero persistent secret state. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 223–235. IEEE. 23, 117

[48] Dmitrienko, A., Liebchen, C., Rossow, C., and Sadeghi, A.-R. (2014). On the (in) security of mobile two-factor authentication. In *Financial Cryptography and Data Security: 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers 18*, pages 365–383. Springer. 79

[49] Duan, R., Alrawi, O., Kasturi, R. P., Elder, R., Saltaformaggio, B., and Lee, W. (2020). Towards measuring supply chain attacks on package managers for interpreted languages. *arXiv preprint arXiv:2002.01139*. 18, 59, 105

[50] Duo and Cisco (2023). Crxcavator chrome extension permissions. "`https://crxcavator.io/`". 93

[51] Eskandarian, S., Cogan, J., Birnbaum, S., Brandon, P. C. W., Franke, D., Fraser, F., Garcia, G., Gong, E., Nguyen, H. T., Sethi, T. K., et al. (2019). Fidelius: Protecting user secrets from compromised browsers. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 264–280. IEEE. 23, 152

[52] Fagan, M., Albayram, Y., Khan, M. M. H., and Buck, R. (2017). An investigation into users' considerations towards using password managers. *Human-centric computing and information sciences*, 7(1):1–20. 59

[53] Fahl, S., Harbach, M., Oltrogge, M., Muders, T., and Smith, M. (2013). Hey, you, get off of my clipboard. In *International Conference on Financial Cryptography and Data Security*, pages 144–161. Springer. 17, 20

[54] Fass, A., Somé, D. F., Backes, M., and Stock, B. (2021). Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1804. 24

[55] Florencio, D. and Herley, C. (2007). A large-scale study of web password habits. In *Proceedings of the 16th International Conference on World Wide Web*, pages 657–666. ACM, ACM Press. 16, 26, 43

[56] Florêncio, D. and Herley, C. (2010). Where do security policies come from? In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, pages 1–14. 16, 27, 28, 42, 43, 56

[57] Florêncio, D., Herley, C., and Van Oorschot, P. C. (2014). An administrator's guide to internet password research. In *28th Large Installation System Administration Conference (LISA14)*, pages 44–61. 35, 43, 45, 56

[58] Gautam, A., Lalani, S., and Ruoti, S. (2022). Improving password generation through the design of a password composition policy description language. In *Proceedings of the 18th Symposium on Usable Privacy and Security*. USENIX. 117

[59] Gautam, A., Yadav, T. K., Seamons, K., and Ruoti, S. (2024). Passwords are meant to be secret: A practical secure password entry channel for web browsers. 106, 108, 109, 131, 133, 143

[60] Google (2024a). `https://chromedevtools.github.io/devtools-protocol/`. 108

[61] Google (2024b). Chrome devtools : chrome for developers. `https://developer.chrome.com/docs/devtools`. 107

[62] Grassi, P. A., Fenton, J. L., Newton, E. M., Perlner, R. A., Regenscheid, A. R., Burr, W. E., Richer, J. P., Lefkovitz, N. B., Danker, J. M., Choong, Y., et al. (2016). Nist special publication 800-63b: Digital identity guidelines. *National Institute of Standards and Technology (NIST)*, 27. 17, 54

[63] Greene, K. K., Kelsey, J. M., Franklin, J. M., et al. (2016). *Measuring the usability and security of permuted passwords on mobile platforms*. US Department of Commerce, National Institute of Standards and Technology. 57, 145

[64] Guan, J., Li, H., Ye, H., and Zhao, Z. (2022). A formal analysis of the fido2 protocols. In *Computer Security–ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*, pages 3–21. Springer. 91, 93, 94

[65] Guirat, I. B. and Halpin, H. (2018). Formal verification of the W3C web authentication protocol. In *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*, pages 1–10. 21

[66] Han, X., Pasquier, T., Bates, A., Mickens, J., and Seltzer, M. (2020). Unicorn: Runtime provenance-based detector for advanced persistent threats. *arXiv preprint arXiv:2001.01525*. 24

[67] Hannousse, A., Yahiouche, S., and Nait-Hamoud, M. C. (2022). Twenty-two years since revealing cross-site scripting attacks: a systematic mapping and a comprehensive survey. *arXiv preprint arXiv:2205.08425*. 105

[68] Hao, F. and van Oorschot, P. C. (2022). Sok: Password-authenticated key exchange–theory, practice, standardization and real-world lessons. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 697–711. 71

[69] Hassan, W. U., Guo, S., Li, D., Chen, Z., Jee, K., Li, Z., and Bates, A. (2019). Nodoze: Combatting threat alert fatigue with automated provenance triage. In *network and distributed systems security symposium*. 24

[70] Hassan, W. U., Noureddine, M. A., Datta, P., and Bates, A. (2020). Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. In *Network and distributed system security symposium*. 24

[71] Heiderich, M., Niemietz, M., Schuster, F., Holz, T., and Schwenk, J. (2012). Scriptless attacks: stealing the pie without touching the sill. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 760–771. 123

[72] Helms, K. (2020). Google pulls 49 cryptocurrency wallet browser extensions found stealing private keys – security bitcoin news. 105

[73] Herley, C. (2009). So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proceedings of the 2009 workshop on New security paradigms workshop*, pages 133–144. ACM. 87

[74] Horsch, M., Schlipf, M., Braun, J., and Buchmann, J. (2016). Password requirements markup language. In *Australasian Conference on Information Security and Privacy*, pages 426–439. Springer. 15, 28, 55

[75] Hossain, M. N., Milajerdi, S. M., Wang, J., Eshete, B., Gjomemo, R., Sekar, R., Stoller, S., and Venkatakrishnan, V. (2017). {SLEUTH}: Real-time attack scenario reconstruction from {COTS} audit data. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 487–504. 24

[76] House, F. (2021). Freedom house (fh) freedom of the press report. https://freedomhouse.org/reports/publication-archives. Accessed: 2021-05-01. 27

[77] Hu, K. and Zhang, Z. (2016). Security analysis of an attractive online authentication standard: Fido uaf protocol. *China Communications*, 13(12):189–198. 21, 91, 94

[78] Huaman, N., Amft, S., Oltrogge, M., Acar, Y., and Fahl, S. (2021). They would do better if they worked together: The case of interaction problems between password managers and websites. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 1626–1640, Los Alamitos, CA, USA. IEEE Computer Society. 20, 26

[79] Jacomme, C. and Kremer, S. (2018). An extensive formal analysis of multi-factor authentication protocols. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 1–15. IEEE. 22

[80] Jakkal, V. (2024). The passwordless future with microsoft. https://www.microsoft.com/en-us/security/blog/2021/09/15/the-passwordless-future-is-here-for-your-microsoft-account/. 11

[81] Jarecki, S., Krawczyk, H., and Xu, J. (2018). Opaque: an asymmetric pake protocol secure against pre-computation attacks. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part III 37*, pages 456–486. Springer. 71

[82] Ji, Y., Lee, S., Downing, E., Wang, W., Fazzini, M., Kim, T., Orso, A., and Lee, W. (2017). Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 377–390. 24

[83] Jia, Z., Cui, X., Liu, Q., Wang, X., and Liu, C. (2018). Micro-honeypot: using browser fingerprinting to track attackers. In *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, pages 197–204. IEEE. 25

[84] Juels, A. and Rivest, R. L. (2013). Honeywords: Making password-cracking detectable. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 145–160. 23, 117

[85] Kapravelos, A., Grier, C., Chachra, N., Kruegel, C., Vigna, G., and Paxson, V. (2014). Hulk: Eliciting malicious behavior in browser extensions. In *23rd {USENIX} security symposium ({USENIX} Security 14)*, pages 641–654. 59, 93, 105

[86] Kariryaa, A., Savino, G.-L., Stellmacher, C., and Schöning, J. (2021). Understanding users' knowledge about the privacy and security of browser extensions. In *Proceedings of the Seventeenth Symposium on Usable Privacy and Security*. USENIX. 15, 149

[87] Karlof, C., Tygar, J. D., and Wagner, D. A. (2009). Conditioned-safe ceremonies and a user study of an application to web authentication. In *NDSS*. 87

[88] Karole, A., Saxena, N., and Christin, N. (2011). A comparative usability evaluation of traditional password managers. In *Information Security and Cryptology-ICISC 2010: 13th International Conference, Seoul, Korea, December 1-3, 2010, Revised Selected Papers 13*, pages 233–251. Springer. 21

[89] Kaur, J., Garg, U., and Bathla, G. (2023). Detection of cross-site scripting (xss) attacks using machine learning techniques: a review. *Artificial Intelligence Review*, pages 1–45. 105

[90] Komanduri, S., Shay, R., Kelley, P. G., Mazurek, M. L., Bauer, L., Christin, N., Cranor, L. F., and Egelman, S. (2011). Of passwords and people: measuring the effect of password-composition policies. In *Proceedings of the sigchi conference on human factors in computing systems*, pages 2595–2604. 16, 53

[91] Korir, M., Parkin, S., and Dunphy, P. (2022). An empirical study of a decentralized identity wallet: Usability, security, and perspectives on user control. In *Proceedings of the 18th Symposium on Usable Privacy and Security*, Boston, MA. USENIX. 66

[92] Kovacs, E. (2019). Researcher earns $10,000 for another xss flaw in yahoo mail. 110

[93] Krawczyk, H., Paterson, K. G., and Wee, H. (2013). On the security of the tls protocol: A systematic analysis. In *Annual Cryptology Conference*, pages 429–448. Springer. 148

[94] Kumari, N. and Mohapatra, A. (2022). A comprehensive and critical analysis of tls 1.3. *Journal of Information and Optimization Sciences*, 43(4):689–703. 148

[95] Lee, H., Kim, D., and Kwon, Y. (2021). Tls 1.3 in practice: How tls 1.3 contributes to the internet. In *Proceedings of the Web Conference 2021*, pages 70–79. 148

[96] Lee, S., Wi, S., and Son, S. (2022). Link: Black-box detection of cross-site scripting vulnerabilities using reinforcement learning. In *Proceedings of the ACM Web Conference 2022*, pages 743–754. 105

[97] Lemos, R. (2021). Dependency problems increase for open source components. 18

[98] Lemos, R. (2024). Microsoft: Iran's mint sandstorm apt blasts educators, researchers. 110

[99] Li, B., Vadrevu, P., Lee, K. H., Perdisci, R., Liu, J., Rahbarinia, B., Li, K., and Antonakakis, M. (2018). Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *NDSS*. 122, 132

[100] Li, L., Pal, B., Ali, J., Sullivan, N., Chatterjee, R., and Ristenpart, T. (2019). Protocols for checking compromised credentials. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1387–1403. 129

[101] Li, Z., Han, W., and Xu, W. (2014a). A large-scale empirical analysis of chinese web passwords. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 559–574, San Diego, CA. USENIX Association. 27, 43, 50, 179, 182

[102] Li, Z., He, W., Akhawe, D., and Song, D. (2014b). The emperor's new password manager: Security analysis of web-based password managers. In *USENIX Security Symposium*, pages 465–479. 20, 21, 81

[103] Liu, Y., Zhang, M., Li, D., Jee, K., Li, Z., Wu, Z., Rhee, J., and Mittal, P. (2018). Towards a timely causality analysis for enterprise security. In *NDSS*. 24

[104] Lyastani, S. G., Schilling, M., Fahl, S., Backes, M., and Bugiel, S. (2018). Better managed than memorized? studying the impact of managers on password strength and reuse. In *Proceedings of the 28th USENIX Security Symposium*, pages 203–220. USENIX. 20, 26, 59

[105] Lyastani, S. G., Schilling, M., Neumayr, M., Backes, M., and Bugiel, S. (2020). Is FIDO2 the kingslayer of user authentication? a comparative usability study of FIDO2 passwordless authentication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 268–285. IEEE. 22

[106] Margo, D. W. and Seltzer, M. I. (2009). The case for browser provenance. In *Workshop on the Theory and Practice of Provenance*. 24

[107] Mayer, P., Kirchner, J., and Volkamer, M. (2017). A second look at password composition policies in the wild: Comparing samples from 2010 and 2016. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 13–28, Santa Clara, CA. USENIX Association. 16, 27, 28, 42, 43, 56

[108] Mayer, P., Munyendo, C. W., Mazurek, M. L., and Aviv, A. J. (2022). Why users (don't) use password managers at a large educational institution. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1849–1866. 1

[109] Meadows, I. (2020). Add password restriction attributes. `https://discourse.wicg.io/t/add-password-restriction-attributes-to-input-type-password/4767`. 15, 55

[110] Milajerdi, S. M., Gjomemo, R., Eshete, B., Sekar, R., and Venkatakrishnan, V. (2019). Holmes: real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1137–1152. IEEE. 24

[111] MozDevNet (2024a). Htmldataelement: Value property web apis: Mdn. `https://developer.mozilla.org/en-US/docs/Web/API/HTMLDataElement/value`. 141

[112] MozDevNet (2024b). Webrequest - mozilla: Mdn. `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest`. 93

[113] MozDevNet, M. (2024c). What are browser developer tools? - learn web development: Mdn. 107

[114] Mozilla (2017). 1376155 - webrequest: Support modifying request bodies (e.g. via requestbody blockingresponse). https://bugzilla.mozilla.org/show_bug.cgi?id=1376155. Accessed: 2023-05-03. 65, 76

[115] Mozilla (2023). Using shadow dom. https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM. Accessed: 2023-05-04. 72

[116] Mozilla (2024). Httpbasechannel.cpp - mozsearch. https://searchfox.org/mozilla-central/source/netwerk/protocol/http/HttpBaseChannel.cpp. 85

[117] Mozilla Developer Network (2024). Browser extensions - mdn web docs. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions. Accessed: 2024-03-30. 14

[118] Naiakshina, A., Danilova, A., Gerlitz, E., and Smith, M. (2020). On conducting security developer studies with cs students: Examining a password-storage study with cs students, freelancers, and company developers. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13. 39

[119] Naiakshina, A., Danilova, A., Gerlitz, E., Von Zezschwitz, E., and Smith, M. (2019). " if you want, i can store the encrypted password" a password-storage field study with freelance developers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12. 39

[120] Nelson, N. (2023). Iran apt targets the mediterranean with watering-hole attacks. 110

[121] Oesch, S., Abu-Salma, R., Diallo, O., Krämer, J., Simmons, J., Wu, J., and Ruoti, S. (2020). Understanding user perceptions of security and privacy for group chat: a survey of users in the US and UK. In *Proceedings of the 36th Annual Computer Security Applications Conference*. ACM. 118

[122] Oesch, S., Gautam, A., and Ruoti, S. (2021). The emperor's new autofill framework: a security analysis of autofill on iOS and Android. In *Proceedings of the 37th Annual Computer Security Applications Conference*. ACM. 19, 20, 21, 69, 81, 82, 83

[123] Oesch, S. and Ruoti, S. (2020a). That was then, this is now: A security evaluation of password generation, storage, and autofill in browser-based password managers. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA. USENIX Association. 19, 20, 21, 59, 67, 69, 74, 81, 82, 84

[124] Oesch, S. and Ruoti, S. (2020b). That was then, this is now: A security evaluation of password generation, storage, and autofill in browser-based password managers. In *USENIX Security Symposium*. 36, 48, 57, 145

[125] Oesch, S., Ruoti, S., Simmons, J., and Gautam, A. (2022). "it basically started using me:" An observational study of password manager usage. In *Proceedings of the 40th ACM CHI Conference on Human Factors in Computing Systems*. ACM. 20, 21, 59, 118

[126] Oesch, T. (2021). *An Analysis of Modern Password Manager Security and Usage on Desktop and Mobile Devices*. PhD thesis, The University of Tennessee. 26

[127] O'Flynn, C. (2019). MIN()imum failure:EMFI attacks against USB stacks. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. 22

[128] O'Gorman, L. (2003). Comparing passwords, tokens, and biometrics for user authentication. *Proceedings of the IEEE*, 91(12):2021–2040. 6

[129] Ohm, M., Plate, H., Sykosch, A., and Meier, M. (2020). Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pages 23–43. Springer. 18, 59, 105

[130] O'Neill, M., Ruoti, S., Seamons, K., and Zappala, D. (2016). Tls proxies: Friend or foe? In *Proceedings of the 2016 Internet Measurement Conference*, pages 551–557. ACM, ACM. 17, 68

[131] OSITCOM (2021). Google removes 500 plus malicious chrome extensions. https://www.ositcom.com/61. Accessed: 2023-05-03. 19, 93

[132] OWASP (2021). Password special characters. https://owasp.org/www-community/password-special-characters. Accessed: 2021-05-01. 14, 29, 105, 149

[133] OWASP (2022). Cross site scripting (xss). https://owasp.org/www-community/attacks/xss/. Accessed: 2023-05-03. 18

[134] Paccagnella, R., Datta, P., Hassan, W. U., Bates, A., Fletcher, C., Miller, A., and Tian, D. (2020). Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *Network and distributed system security symposium*. 24

[135] Paladi, N. and Karlsson, L. (2017). Safeguarding vnf credentials with Intel SGX. In *Proceedings of the SIGCOMM Posters and Demos*, pages 144–146. 23

[136] Panos, C., Malliaros, S., Ntantogian, C., Panou, A., and Xenakis, C. (2017). A security evaluation of FIDO's UAF protocol in mobile and embedded devices. In *International Tyrrhenian Workshop on Digital Communication*, pages 127–142. Springer. 21

[137] Pantelaios, N., Nikiforakis, N., and Kapravelos, A. (2020). You've changed: Detecting malicious browser extensions through their update deltas. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 477–491. 59, 105

[138] Pearman, S., Thomas, J., Naeini, P. E., Habib, H., Bauer, L., Christin, N., Cranor, L. F., Egelman, S., and Forget, A. (2017). Let's go in for a closer look: Observing passwords in their natural habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 295–310. ACM. 26

[139] Pearman, S., Zhang, S. A., Bauer, L., Christin, N., and Cranor, L. F. (2019a). Why people don't use password managers effectively. In *Proceedings of the 15th Symposium On Usable Privacy and Security*. USENIX. 1, 59

[140] Pearman, S., Zhang, S. A., Bauer, L., Christin, N., and Cranor, L. F. (2019b). Why people (don't) use password managers effectively. In *Fifteenth Symposium On Usable*

*Privacy and Security (SOUPS 2019). USENIX Association, Santa Clara, CA*, pages 319–338. 26

[141] Pereira, O., Rochet, F., and Wiedling, C. (2017). Formal analysis of the FIDO 1. x protocol. In *International Symposium on Foundations and Practice of Security*, pages 68–82. Springer. 21

[142] Popov, A., Nystroem, M., Balfanz, D., and Hodges, J. (2018). The token binding protocol version 1.0. RFC 8471, RFC Editor. 68, 119, 120

[143] Proctor, R. W., Lien, M.-C., Vu, K.-P. L., Schultz, E. E., and Salvendy, G. (2002). Improving computer security for authentication of users: Influence of proactive password restrictions. *Behavior Research Methods, Instruments, & Computers*, 34(2):163–169. 16

[144] Ray, H., Wolf, F., Kuber, R., and Aviv, A. J. (2021). Why older adults (don't) use password managers. In *Proceedings of the 30th USENIX Security Symposium*. USENIX. 1

[145] Rijswijk-Deij, R. v. and Poll, E. (2013). Using trusted execution environments in two-factor authentication: comparing approaches. *Open Identity Summit 2013*. 23

[146] Riley, S. (2006). Password security: What users know and what they actually do. *Usability News*, 8(1):2833–2836. 26

[147] Rodríguez, G. E., Torres, J. G., Flores, P., and Benavides, D. E. (2020). Cross-site scripting (xss) attacks and mitigation: A survey. *Computer Networks*, 166:106960. 105

[148] Ruoti, S., Andersen, J., Monson, T., Zappala, D., and Seamons, K. (2016). Messageguard: A browser-based platform for usable, content-based encryption research. 88

[149] Sauro, J. and Lewis, J. R. (2016). *Quantifying the user experience: Practical statistics for user research*. Morgan Kaufmann. 37, 39, 41

[150] Security, H. N. (2021). Why xss is still an xxl issue in 2021. 59, 105

[151] Seiler-Hwang, S., Arias-Cabarcos, P., Marin, A., Almenares, F., Diaz-Sanchez, D., and Becker, C. (2019a). "i don't see why i would ever want to use it" analyzing the usability

of popular smartphone password managers. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1937–1953. 1

[152] Seiler-Hwang, S., Arias-Cabarcos, P., Marín, A., Almenares, F., Díaz-Sánchez, D., and Becker, C. (2019b). "I don't see why i would ever want to use it:" Analyzing the usability of popular smartphone password managers. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*. ACM. 20

[153] Senol, A., Acar, G., Humbert, M., and Borgesius, F. Z. (2022). Leaky forms: A study of email and password exfiltration before form submission. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1813–1830. 17, 59, 67

[154] Shah, Y., Choyi, V., and Subramanian, L. (2015). Multi-factor authentication as a service. In *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 144–150. IEEE. 23

[155] Shay, R., Komanduri, S., Durity, A. L., Huh, P., Mazurek, M. L., Segreti, S. M., Ur, B., Bauer, L., Christin, N., and Cranor, L. F. (2014). Can long passwords be secure and usable? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2927–2936. 16, 53

[156] Shay, R., Komanduri, S., Durity, A. L., Huh, P., Mazurek, M. L., Segreti, S. M., Ur, B., Bauer, L., Christin, N., and Cranor, L. F. (2016). Designing password policies for strength and usability. *ACM Transactions on Information and System Security (TISSEC)*, 18(4):1–34. 16, 53

[157] Silver, D., Jana, S., Boneh, D., Chen, E. Y., and Jackson, C. (2014). Password managers: Attacks and defenses. In *USENIX Security Symposium*, pages 449–464. 20, 21, 67, 81

[158] Simmons, J., Diallo, O., Oesch, S., and Ruoti, S. (2021). Systematization of password manager use cases and design paradigms. In *Proceedings of the 37th Annual Computer Security Applications Conference*. ACM. 20, 21, 59

[159] Steffens, M., Rossow, C., Johns, M., and Stock, B. (2019). Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. 105

[160] Stock, B. and Johns, M. (2014). Protecting users against xss-based password manager abuse. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 183–194. ACM. 3, 20, 21, 22, 60, 61, 62, 67, 72, 76, 81, 105

[161] Ulqinaku, E., Assal, H., AbdelRahman, A., Chiasson, S., and Capkun, S. (2021). Is real-time phishing eliminated with fido? social engineering downgrade attacks against fido protocols. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*, pages 3811–3828. USENIX Association. 79

[162] Ur, B., Noma, F., Bees, J., Segreti, S. M., Shay, R., Bauer, L., Christin, N., and Cranor, L. F. (2015). 'I added '!' at the end to make it secure': Observing Password Creation in the Lab. In *Proceedings of the Eleventh Symposium On Usable Privacy and Security*. 26, 118

[163] Vu, K.-P. L., Proctor, R. W., Bhargav-Spantzel, A., Tai, B.-L. B., Cook, J., and Schultz, E. E. (2007). Improving password security and memorability to protect personal and organizational information. *International Journal of Human-Computer Studies*, 65(8):744–757. 16

[164] Wang, D., Wang, P., He, D., and Tian, Y. (2019). Birthday, name and bifacial-security: understanding passwords of chinese web users. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1537–1555. 27, 43, 50, 182

[165] Wang, K. C. and Reiter, M. K. (2018). How to end password reuse on the web. *arXiv preprint arXiv:1805.00566*. 26

[166] Wang, K. C. and Reiter, M. K. (2021). Using amnesia to detect credential database breaches. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 839–855. 23, 24, 117

[167] Wang, Q., Hassan, W. U., Bates, A., and Gunter, C. (2018). Fear and logging in the internet of things. In *Network and Distributed Systems Symposium*. 24

[168] Wang, R., Peng, Y., Sun, Y., Zhang, X., Wan, H., and Zhao, X. (2023). Tesec: Accurate server-side attack investigation for web applications. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2799–2816. IEEE. 122

[169] WHATWG (2023). Web idl. `https://webidl.spec.whatwg.org/`. 78

[170] WhiteSource (2022). Remediating vulnerabilities in npm packages - whitesource. 18

[171] Will, N. C. and Maziero, C. A. (2020). Using a shared SGX enclave in the UNIX PAM authentication service. In *Proceedings of the 14th Annual International Systems Conference, Montreal, QC, Canadá. IEEE*. 23

[172] Wu, T. et al. (1998). The secure remote password protocol. In *Internet Society Symposium on Network and Distributed System Security*, volume 98, pages 97–111. Citeseer. 71

[173] Yadav, T. K. and Seamons, K. (2024). A security and usability analysis of local attacks against fido2. In *Network and Distributed Systems Security (NDSS) Symposium*, San Diego, CA, USA. 91, 92, 94, 105

[174] Yang, R., Ma, S., Xu, H., Zhang, X., and Chen, Y. (2020). Uiscope: Accurate, instrumentation-free, and visible attack investigation for gui applications. In *NDSS*. 24

[175] Yu, D., Chander, A., Islam, N., and Serikov, I. (2007). Javascript instrumentation for browser security. *ACM SIGPLAN Notices*, 42(1):237–249. 25

[176] Yubico (2024). Java webauthn server. `"https://github.com/Yubico/java-webauthn-server"`. 103

[177] Zhang, Y., Zhao, S., Qin, Y., Yang, B., and Feng, D. (2015). Trusttokenf: A generic security framework for mobile two-factor authentication using trustzone. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 41–48. IEEE. 23

[178] Zhao, J., Goble, C., Stevens, R., and Bechhofer, S. (2004). Semantically linking and browsing provenance logs for e-science. In *Semantics of a Networked World. Semantics for*

*Grid Databases: First International IFIP Conference, ICSNW 2004, Paris, France, June 17-19, 2004, Revised Selected Papers*, pages 158–176. Springer. 24

[179] Zipperle, M., Gottwalt, F., Chang, E., and Dillon, T. (2022). Provenance-based intrusion detection systems: A survey. *ACM Computing Surveys*, 55(7):1–36. 24

# Appendices

## A Study Instrument For Password Policy Authoring

**Setup** For this study, you will be using a python library we developed. Please install this library using pip: `python3 -m pip install -user -upgrade password-policy`. If for some reason, you don't have pip installed, you can install it using: `python3 -m ensurepip -user -upgrade`.

After installation is complete, check that everything is working correctly by copying and pasting the following command into your terminal. Enter the resulting output below: `python3 -c "import password_policy; print(password_policy.__version__)"`.

**Q1.** Enter version

**Demographics**
**Q2.1.** What is your class standing?

∘ *Junior* ∘ *Senior* ∘ *MS student* ∘ *PhD student*

**Q2.2.** What is your major?

∘ *Computer Science* ∘ *Computer Engineering* ∘ *Electrical Engineering* ∘ *other [Enter here]*

**Q2.3.** What is your sex?

∘ *Male* ∘ *Female* ∘ *Non-binary* ∘ *Prefer not to answer*

**Tasks** Different websites have different requirements for passwords. For example, some websites may require passwords to have a minimum length, include certain types of characters, and avoid using other characters. In our research group, we are studying a system for describing password policies using JSON. We are also studying libraries that can be used to construct these JSON policy descriptions and validate passwords based on these descriptions.

In this study, you will use this system and a python library to encode several password policies. Our goal is to understand how usable this system and library is.

To help you learn about this system and the python library you installed, please click [this link to view the relevant documentation]. You will be using the knowledge from this documentation for the rest of the study. Feel free to refer to it throughout the study. A link to this documentation will always be available on the pages describing your tasks for this study.

When you feel ready to use this system, click continue to be given your first task.

*The following questions were the same for each policy, except for the policy requirements. We give the full text for Policy 1's questions, and only the policy requirements for Policy 2–5.*

**Q3.1.1.** Using the python library, please write a policy description for the following password policy. When finished, encode it in JSON and enter it into the text field below. We will validate the entered policy description to make sure it is correct. You may also directly write the policy as JSON (not using the library) if desired.

Password policy:

- The password must be at least 8 characters long

[Documentation link]

**Q3.1.2.** Did you manually write the JSON policy description, or did you generate it using the python library?

∘ *Generated it using Python library* ∘ *Manually entered the JSON policy*

**Q3.1.3.** Based on your experience authoring the JSON policy description, indicate to what extent you agree with the following statements. Options:Strongly disagree-1..Strongly agree-7

∘ *Overall, I am satisfied with the ease of completing this task.* ∘ *Overall, I am satisfied with the amount of time it took to complete this task.* ∘ *Overall, I am satisfied with the support information (on-line help, messages, documentation) when completing this task.*

**Q3.2.1.** Password policy:

- The password must be at least 8 characters long
- The password must contain characters from at least two of the following: uppercase letters, lowercase letters, numbers, symbols

**Q3.3.1.** Password policy:

- The password must be at least 12 characters long
- The password must contain at least one letter and one number
- The password must NOT contain space

**Q3.4.1.** Password policy:

- The password must satisfy one OR the other of the following policies:
  - The password must be at at least 8 characters long
  - The password must contain at least one letter and one number
- OR
  - The password must be at least 15 characters long

**Q3.5.1.** Password policy:

- The password must be at least 8 characters long
- The password must contain at least two symbols
- The password must contain at least one uppercase letter and one lowercase letter
- The password must NOT contain space, the carrot symbol (^), quotes ('), double quotes ("), semicolons (;), slashes (/), or backslashes (\).
- The password must NOT contain the substring "mywebsite"

**Post-Study Questionnaire** That was the last policy you will need to write for this study. We will now ask you a few questions about your experience the password policy description system and python library.

**Q4.1.** Please answer the following question about your experience. Try to give your immediate reaction to each statement without pausing to think for a long time. Mark the middle column if you don't have a response to a particular statement.

Options: Strongly Disagree, Disagree, Neither Agree nor Disagree, Agree, Strong Agree

1. I think that I would like to use this system frequently
2. I found the system unnecessarily complex
3. I thought the system was easy to use
4. I think that I would need the support of a technical person to be able to use this system
5. I found the various functions in this system were well integrated
6. I thought there was too much inconsistency in this system
7. I would imagine that most people would learn to use this system very quickly
8. I found the system very cumbersome to use
9. I felt very confident using the system
10. I needed to learn a lot of things before I could get going with this system

**Q4.2.** What did you like the most about the system and library?

**Q4.3.** What did you like the least about the system and library?

**Q4.4.** Is there any other feedback you would like us to know about the system or library?

# B PCP Strength Calculations

We measure the strength of password composition policies (PCPs) by estimating how many passwords exist that (a) satisfy the PCP and (b) are of the shortest possible length. We then divide this number by two to estimate the average number of guesses an adversary needs to find a user's passwords. This approach gives an exact estimate of strength when passwords are generated entirely at random. To estimate strength for human-generated passwords, we allow our strength estimates to be parameterized by what character classes are preferred [101].

## B.1 Algorithm

**Step 1—Preprocessing** First, we filter the rules and only consider those that have the smallest `min_length` (there may be multiple). Next, we simplify `require_subset`, creating a new rule with `require` set for each possible combination of the listed `options` of length `count`. Lastly, we simplify the shortcut rules `require`, setting `min_require` for each charset listed in the requirement.

**Step 2—Enumerating Password Compositions** In this step, we enumerate all possible password compositions for the rules identified in Step 1. A password composition is simply a list specifying how many characters from each character class are used to make up a password. For example, for a PCP that (a) only allows lowercase letters and digits and (b) has a rule that sets `min_length` to 2 (but no other requirements), there are three password compositions: (1) two lowercase letters, (2) two digits, (3) one lowercase letter and one digit. Note, we only consider compositions where the sum of character counts equals `min_length`.

We take the following steps to derive the password compositions for a rule. First, we create a password composition with values set based on `min_required` for each charset. We also calculate `required_chars`, which tracks the total number of required characters (sum of the calculated password composition). Second, we create a list of length `min_length` − `required_chars`. At each index $i$ (one-indexed) of this new list, we include a list of which character classes could appear $i$ more times in the password composition without violating `max_allowed` for each charset (if set). Third, we calculate the full factorial combination of

items in this list of lists. For each such combination, we create a new password composition that takes the original password combination and adds the character classes in the combination. For each composition, we also store any restrictions related to that composition that may not yet have been handled (e.g., `max_consecutive`).

For example, consider a policy with `min_length` set to 3, which requires the `alphabet` character set to be used once and has at most one digit. Our initial password composition would be `[1,0,0]` representing 1 alphabet character, 0 digits, and 0 symbols; `required_characters` would be 1. Our list of lists would be `[[alphabet,digit,symbol], [alphabet,symbol]]`. In total, there are six $(3 * 2)$ possible combinations of this list, which after added to initial password composition result give the following password compositions: `[[3,0,0], [2,0,1], [2,1,0], [1,1,1], [2,0,1], [1,0,2]]`.

This method will not result in overlapping compositions within a given rule but can between rules. If this occurs, duplicate compositions are trimmed.

**Step 3—Calculating Combinations and Permutations** For each composition, we will calculate the number of passwords (i.e., size of the search space) represented by each composition that also satisfy the PCP. As a password only maps to a single composition, the sum of search space sizes for each composition is the size of the overall password search space. For each composition, we take the following steps to calculate its search space size:

We start by calculating the number of combinations of characters from the charsets that make up the composition:

$$\prod_i \texttt{charset\_size}_i^{\texttt{composition}_i} \tag{1}$$

We then multiply this value by the number of unique permutations in the composition:

$$\frac{(\sum_i composition_i)!}{\prod_i (composition_i!)} \tag{2}$$

If there are no additional requirements to be considered, this value is used as the composition's search space size. If there are additional requirements, we will reduce this calculated by value by the number of passwords removed by each requirement.

First, we consider the `required_locations` requirement. If used, we recalculate our baseline using the same calculations above, except that we reduce the permutation calculation to only consider character classes not at fixed positions due to `required_locations`.

For the remaining four requirements, we take an approach wherein we create one or more invalid compositions that violate the requirement, calculate the search space for the invalid composition, and subtract the invalid composition's search space size from the overall composition's search space size (calculated above). We continue doing so until there are no more requirements to handle. We generate these invalid compositions as follows:

- For `max_consecutive`, we identify all charsets which have enough occurrences in the composition to violate this rule. For each of these charsets, we create a new, invalid composition that removes (`max_consecutive` + 1) occurrences from matched charset and adds a single occurrence of a new charset of size equal to the matched charset (representing the repeated character).

- For `max_consecutive` in `charset_restrictions`, we do much the same as above, except that the size of the new charset in the invalid compositions will equal $\texttt{matched\_charset\_size}^{\texttt{max\_consecutive}+1}$, representing all possible combinations of the charset.

- For each substring in `prohibited_substrings`, we create a new, invalid composition that removes the appropriate charset for each character in the substring. We then append a charset of size 1 to the composition, representing the prohibited string.

- For each location in `prohibited_location`, we do not modify the current composition but instead calculate its search space as if the prohibited location were required.

## B.2 Estimating Human-Generation

Prior research has shown that when generating passwords, humans prefer characters from specific character classes, though this preference can differ based on country [101, 164]. Our PCP strength estimation can be parameterized based on what character classes users prefer to represent this behavior. For example, American users' preferences might be lowercase, uppercase, then digits [101]. For Chinese users, their preferences are more likely to be digits, lowercase, then uppercase [101, 164].

We handle these preferences in Step 2 of our calculations. We initially execute step two as described up through calculating the list of lists representing characters that can occur in the remaining spots of the initially calculated password composition. For each sublist of charsets, we check to see if any of those charsets appears in the list of preferred charsets. If one or more do, we replace the sublist with a new list with a single element matching the highest-ranked matching charsets. After this modification, calculations proceed as described.

Note, these preference-based calculations are Fermi approximations, underestimating character class diversity in user passwords and overestimating diversity of character selection within a character class, with the two errors hopefully canceling out. Even though these are not exact estimates for human-generated passwords, they are sufficient to help administrators and researchers estimate the overall strengths and weaknesses of a PCP.

## B.3 Limitations

For PCPs that do not use any of the final four requirements discussed in Step 3, our method precisely calculates the PCP's search space. Our calculation is also correct if only a single one of these requirements are used for a composition. Of the 270 PCPs in our dataset, 260 do not use any of the five requirements, and of the ten that do, each uses only a single requirements. This means that calculations used in our analysis are all precise, and it suggests that most PCPs will have their search space calculated precisely.

Still, more complicated PCPs that use multiple of the five requirements could have their search spaces underestimated. This occurs because these requirements have the possibility of removing the same passwords. To our knowledge, the only way to prevent this would be to

enumerate the password combinations and permutations–as we did with compositions—but this is not tractable for any meaningful length of passwords. However, as the reduction to the search space for each of these requirements will generally be small compared to the overall size of the composition's search space, we believe that the underestimates should be minimal. Additionally, in terms of strength estimates, underestimates are safer than overestimates.

# C Webpages Accessible with HTTP

The following is the list of website for which we were able to access the account creation or login page using HTTP:

| Website | Country | Popularity | Category |
|---|---|---|---|
| weibo.com | China | Top 50 | Social |
| babytree.com | China | Top 100 | Social |
| usatoday.com | US | Top 500 | News |
| yaplakal.com | Russia | Top 5000 | Social |
| ig.com.br | Brazil | Top 5000 | Social |
| wikidot.com | China | Top 5000 | Other |
| fb.ru | Russia | Top 5000 | News |
| javlibrary.com | China | 5000+ | Stream |
| dwnews.com | China | 5000+ | News |
| metacafe.com | India | 5000+ | Social |
| eskimi.com | Nigeria | 5000+ | Social |
| ci123.com | China | 5000+ | Stream |
| sinovision.net | China | 5000+ | News |
| sugardaddyforme.com | China | 5000+ | Social |
| mydiba.xyz | Iran | 5000+ | Stream |

Figure 1: List of websites accessible with HTTP

# D PCP Strength By Category

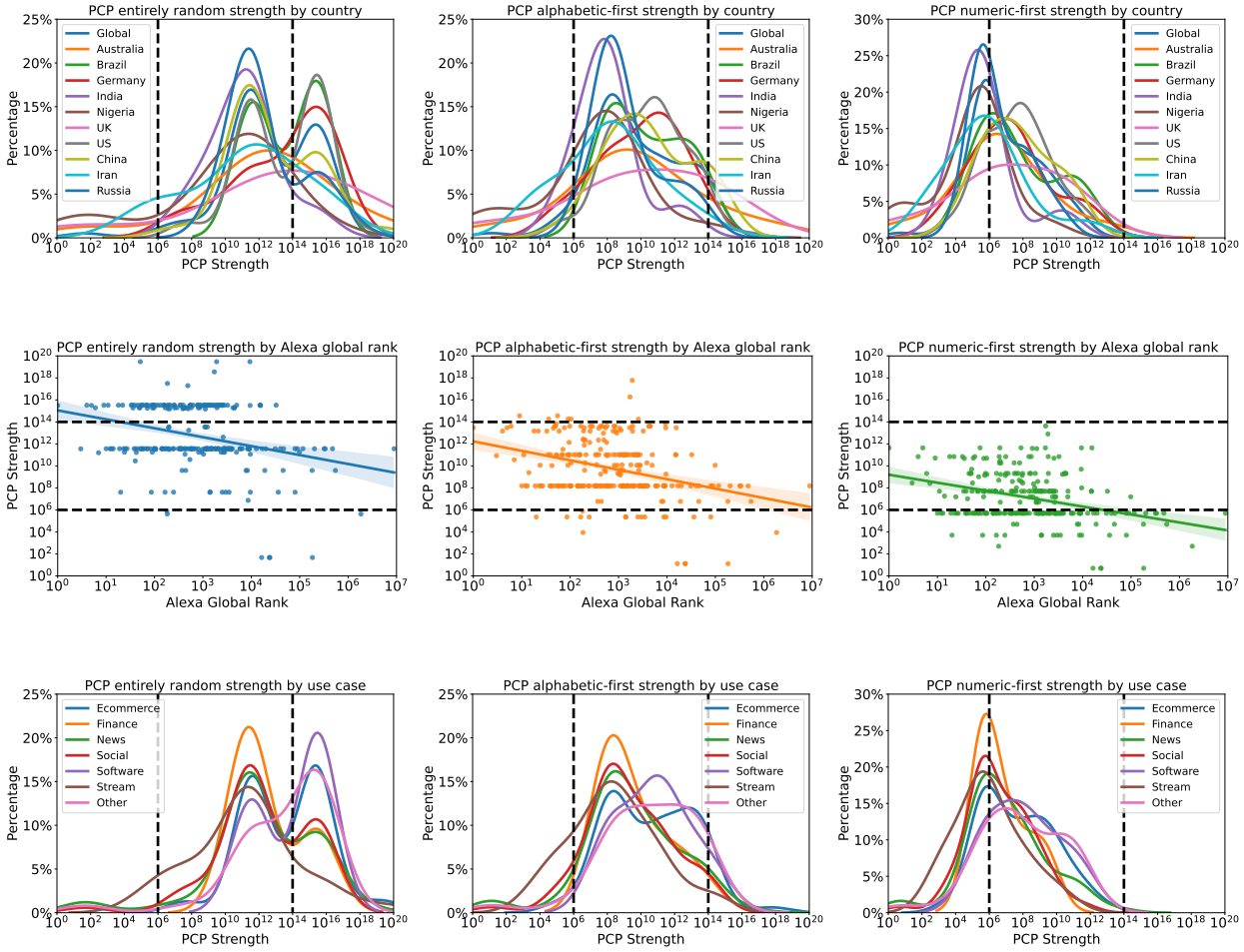



Figure 2: PCP strengths by for different character preference by category

# E PCP Features by Category

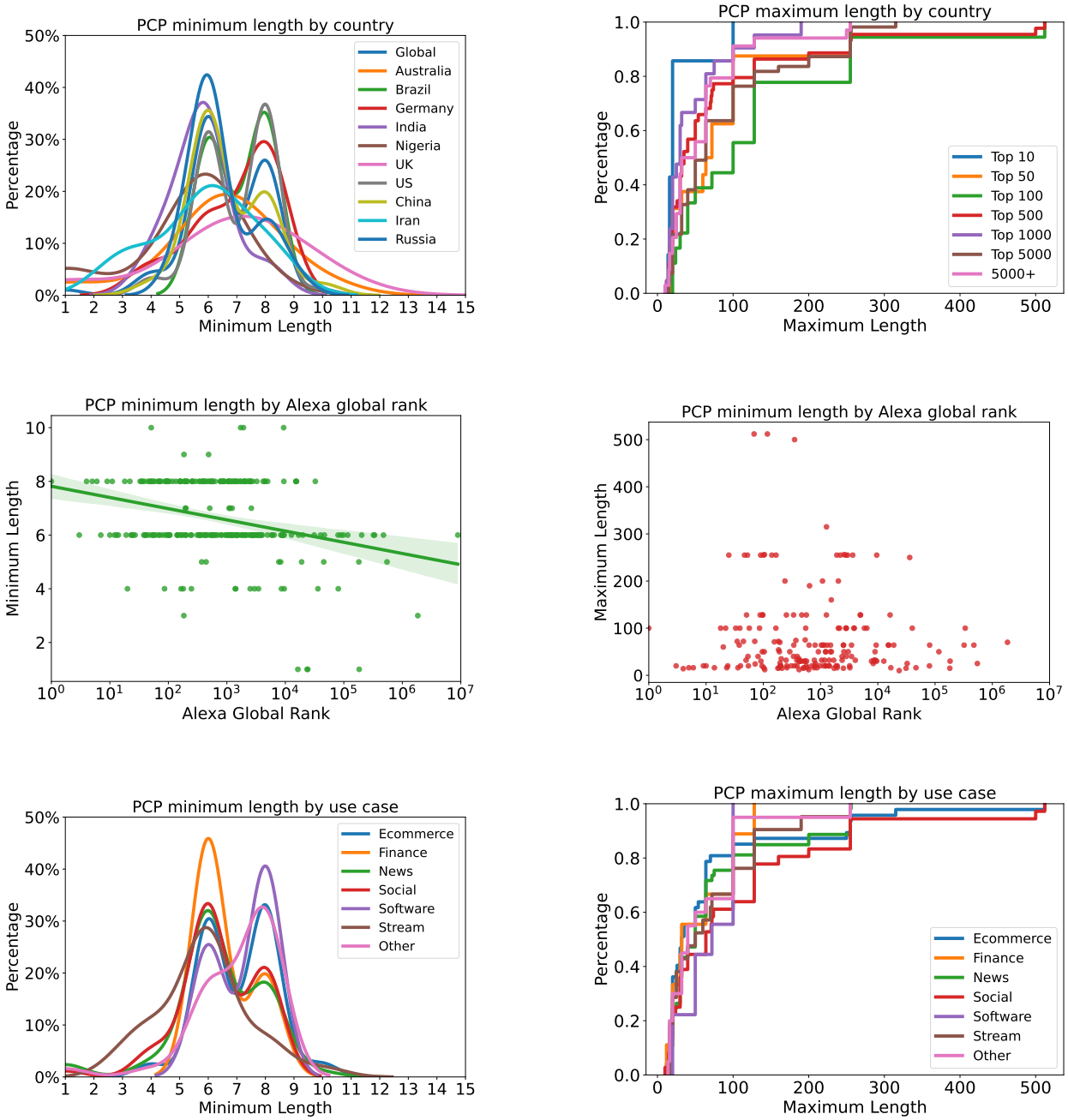



Figure 3: PCP features by for different character preference by category

# Vita

Born and raised in Kathmandu, Nepal, Anuj Gautam received his Bachelor's degree in Electronics and Communication Engineering from Tribhuvan University. He worked as a software engineer for various companies for three years before joining University of Tennessee at Knoxville. Anuj Gautam joined Unviersity of Tennessee at Knoxville in August 2019 for an MS in Computer Engineering, eventually switching to pursue a PhD in Compuer Science.