SmartPasswords: Increasing Password Managers' Usability by Generating Compliant Passwords

João Miguel Pereira Campos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Fernando Peixoto Ferreira
Prof. Alexandra Sofia Ferreira Mendes

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. Ricardo Jorge Fernandes Chaves

November 2021

Acknowledgments

I would like to thank my parents and brother for their encouragement, love and support over all these years, for always being there and always believing in me.

I would also like to acknowledge my dissertation supervisors Prof. João F. Ferreira and Prof. Alexandra Mendes for their insights and knowledge sharing, as well as the colleagues from Passcert: it was great working with you.

Last but not least, a special thanks to my friends, for all your support and friendship. This journey would have been much harder without you in it.

To each and every one of you — Thank you!

This work was partially funded by the PassCert project, a CMU Portugal Exploratory Project funded by Fundação para a Ciência e Tecnologia (FCT), with reference CMU/TIC/0006/2019.

Abstract

Passwords are still the go-to method to provide efficient user authentication in web applications, despite research showing that users usually choose weak passwords and reuse them across different services. Security experts advocate the usage of password managers. These tools can improve account security by enabling the utilization of unique and robust passwords, simultaneously improving the usability and convenience of text password authentication.

However, these tools are not prepared to deal with overly restrictive password composition policies, which many websites employ. These policies pose challenges to password managers and may impact their usage: users become frustrated when generated passwords do not comply with such policies.

We aim to solve this problem by 1) combining a language capable of describing password rules and a widely used password manager — Bitwarden —, and 2) expanding said language to express policies suggested by experts, which combine security and usability.

We generated compliant passwords for every policy tested with our prototype, and Bitwarden accepted our solution to incorporate in their final product. These results are encouraging and suggest that password managers benefit from this ability to interpret password policies, which is a further step to increase the adoption of password managers.

Keywords

Passwords; Password Managers; Usability; Password Policies;

Resumo

As palavras-chave são o método mais utilizado de autenticação em aplicações na internet, ainda que estudos mostrem que, normalmente, os utilizadores escolhem palavras-chave fracas e reutilizam-nas em vários sites. Os especialistas em segurança recomendam a utilização de gerenciadores de palavras-chave. Estas ferramentas podem melhorar a segurança das contas dos utilizadores através da utilização de palavras-chave que são únicas e robustas, enquanto melhoram ainda a usabilidade e conveniência da autenticação com palavras-chave.Todavia, estas aplicações não estão preparadas para lidar com políticas de palavras-chave demasiado restritivas, utilizadas por diversos sites.

Este tipo de políticas impõem desafios acrescidos e podem impactar a sua utilização: os utilizadores dos gerenciadores ficam frustrados quando palavras-chave geradas aleatoriamente não cumprem os requisitos de tais políticas.Pretendemos resolver este problema através da 1) combinação de uma linguagem capaz de descrever políticas de passwords com um gerenciador de palavras-chave bastante utilizado — Bitwarden —, e da 2) expansão desta mesma linguagem para expressar políticas sugeridas pelos especialistas, que combinam segurança e usabilidade.

Para cada política testada com o nosso protótipo, conseguimos gerar palavras-chave que cumpriam essas políticas e a empresa Bitwarden aceitou a nossa solução para ser incorporada no produto final. Estes resultados são encorajadores e sugerem que os gerenciadores de palavras-chave beneficiam desta habilidade de interpretar as regras de composição de palavras-chave, sendo mais um passo para aumentar a adopção destes.

Palavras Chave

Palavras-Chave; Gestor de Palavras-Chave; Usabilidade; Políticas de Palavras-Chave;

Contents

List of Figures

Acronyms

DSL	Domain System Language
HTML	HyperText Markup Language
npm	Node Package Manager
API	Application Program Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
URL	Uniform Resource Locator
SSL	Secure Sockets Layer
SMS	Short Message Service
XSS	Cross-Site Scripting
SRP	Secure Remote Password
HMAC	Hash-Based Message Authentication Code
SHA-1	Secure Hash Algorithm 1
HIBP	Have I Been Pwned
OS's	Operating Systems
DOM	Domain Object Model
RNG	Random Number Generator
CLI	Command Line Interface

1

Introduction

Contents

Throughout the years, and still, to this day, passwords have been seen as a double-edged sword: on the one hand, they were — and still are — the go-to method to provide efficient authentication in web applications, not only due to its simplicity in implementation, or the low cost in maintenance but also because users have been using them for quite a while, making password-based login forms almost second nature to the general user.

On the other hand, users tend to choose weak passwords that are easy to crack [4, 5]. Most of the time, they have an incomplete mental model of how password-based security works, or worse, do not have one at all. This lack of knowledge leads users to commit to erroneous behaviors, like choosing easily guessable passwords or patterns (e.g., *qwerty* or *1qaz2wsx*) and, since they consider them strong passwords, they eventually reuse them in multiple accounts, as demonstrated by various studies [4–8].

Password managers are recommended [6, 9] to safely manage user credentials, but they still have some obstacles that prevent mass adoption by the users. Most users face a common problem when using password managers: the generated passwords are often not compliant [10, 11] with the password composition policies stipulated by the websites they use [12]. This leads to frustrated users and therefore possible cease of use of password managers. However, users are not to blame. As Stajano et al. [2] identified, this problem arises due to very restrictive password composition policies that services usually have [13]. These policies present a greater challenge to password managers since randomly generated passwords have a higher chance of being non-compliant with more restrictive policies (e.g., "*at least eight characters, at least one digit, and one symbol*"): there is a smaller subset of accepted characters, yet the password manager does not take that into account.

1.1 Work Objectives

Our main goal is to explore methods that enable password managers to generate compliant passwords according to each service's password requirements.

There are two possible solutions to this problem:

1. Let the user configure the password generator's parameters. This option may appear to be the most trivial but it puts the burden back on the user, negating one of the advantages of a password manager: remove the password generation burden. Another problem with this approach may present itself when websites do not explicitly express their password policies. The user will become frustrated and probably resort to password reuse.

2. Provide a Domain System Language (DSL) that services can use to specify their required password composition policies and password managers use it to interpret the policies expressed and generate compliant passwords. This method allows for seamless and transparent integration with

password managers, fulfilling their purpose. This approach has minor impacts in user's efforts but great impact on password manager's usability.

The first option is the *status quo*, being present in all password managers nowadays. In our work we delve into the second option because it has greater potential to improve password manager's usability since it makes the whole process transparent to the user.

Various academic studies already emphasize this approach: Stajano et al. proposed the creation of HyperText Markup Language (HTML) semantic labels [2] and Horsch et al. proposed the Password Policy Markup Language [14]. Oesch and Ruoti [15] recently reinforced this idea, suggesting that this type of annotations could help users adopting password managers, as well as increase the accuracy of the password generator.

While investigating a way to achieve this with modern password managers, we found that Apple has also developed a DSL to express Password Autofill Rules [16]. The idea is to add a specification to the HTML code, in the form of annotations. Google has also done something of this nature [17], in the form of an Application Program Interface (API) which could be called by a password manager when generating a password for a given web domain.

Having both Apple and Google — two tech giants — trying to solve this problem reinforces the importance of the problem and the solution. It shows that even the tech industry is striving to help users adopt password managers by making them more usable.

In this project we propose to:

- Review the state-of-the-art and any relevant solutions to this problem, studying the extent to which the recommendations made by researchers are being incorporated into today's password managers and *password-rules-expressing* DSL's.
- Explore whether existing DSL's are capable of effectively describing a website's password policy in a readable format for any password manager, rendering the generation of random, compliant passwords seamless and efficient.
- Extend existing DSL's with constructs that might overcome current limitations. For example, Apple's DSL mentioned above seems to be increasingly adopted, but it is still unable to express policies recommended by the academic literature, such as the policies recommended in Shay et al.'s or Tan et al.'s [18, 19] studies.
- Implement software packages that facilitate the adoption and integration of our proposals.

1.2 Contributions

In summary, our contributions are:

- A survey on existing state-of-the art languages that can be used to express password composition policies and that can be used as annotations for (online) password generators.
- **SmartPasswords**, a new feature in the popular password manager Bitwarden [20][1] that integrates Apple's Password Autofill rules, allowing Bitwarden to only generate compliant passwords. This feature has already been approved by the Bitwarden team and it will be adopted by Bitwarden (after going through their code review process).
- Integration of the SmartPasswords feature into the prototype password manager being developed in the PassCert project[2], which uses a formally verified password generator different from Bitwarden's password generator. This demonstrates that our proposal can be integrated with various products.
- Extension of Apple's DSL with three new features:
  - The `minclasses` rule, that allows to set a minimum number of character classes present in the password.
  - The `blocklist` rule, that allows to check the password against a list of previously breached passwords.
  - The **character range** feature, that allows to specify a minimum and maximum for a given character or character class.
- Creation of a Node Package Manager (npm) package that contains our extension of Apple's DSL, allowing other researchers and developers to use and integrate our extended DSL into their products.

This project is part of the PassCert [21] research project, a CMU-Portugal exploratory project that aims to build an open-source, proof-of-concept password manager that through the use of formal verification, is guaranteed to satisfy properties on data storage and password generation. Our main contribution is on improving the usability of PassCert's password manager.

Research Paper. Parts of the work presented in this thesis were used in the following research paper [22]:

- *Miguel Grilo, **João Campos**, João F. Ferreira, José Bacelar Almeida, and Alexandra Mendes. Verified Password Generation from Password Composition Policies. Submitted for publication.*

[1]Bitwarden is an Open-Source password manager, "*used by millions of individuals and businesses*" - `https://bitwarden.com/help/article/security-faqs/` - Point #4 of question "Why should I trust Bitwarden with my passwords?"

[2]PassCert is a CMU Portugal Exploratory Project funded by Fundação para a Ciência e Tecnologia (FCT), with reference CMU/TIC/0006/2019

1.3 Document Structure

This work is structured in the following way: Chapter 2 collects and briefly describes relevant studies about passwords, passwords managers and how this authentication method can be improved, not only in terms of security but also usability; Chapter 3 describes our work regarding extending Apple's Password Autofill Rules; Chapter 4 details the incorporation of Apple's DSL with Bitwarden's browser extension; Chapter 5 and Chapter 6 go through the evaluation process and the conclusion of our work, respectively.

2 Background Work

Contents

In this chapter, we explore previous work done on password managers and how they respond to password-based authentication security and usability problems. We also discuss studies involving users and their behaviour regarding password usage. Given our goals, we focus on aspects related to password generation and password autofill.

2.1 Password-Based Authentication

Passwords are still the most common method of authentication in web applications. They are simple to implement, have low maintenance and users are accustomed to them.

Even though the possible substitutes for this authentication mechanism appear promising and better security-wise, passwords seem to come up ahead when considering deployability. In 2012, Bonneau et al. [23] evaluated and compared passwords to other types of web authentication, like hardware tokens or biometric authentication. This evaluation was based on Usability, Deployability, and Security benefits. They concluded that all of the studied methods are far from perfect and that none of the alternatives was able to surpass passwords, i.e., to be better on one or more benefits and be as good as passwords on all the others. This means that despite most of the options do better on some criteria they are all worse in some other. The authors also make the case that for high-value accounts, this trade-off might be worth the cost. For instance, a company that deals with high risk and sensitive data may find that using a hardware token — that is preferred to passwords regarding security benefits — is worth the extra cost and may bring more benefits than traditional passwords. For websites, it seems that passwords are enough since most of these sites view the accounts as lower value accounts and the ones that do not, like banking accounts, force users to use some form of Multi-Factor Authentication. As the authors state, "*Thus, the current state of the world is a Pareto equilibrium. Replacing passwords with any of the schemes examined is not a question of giving up an inferior technology for something unarguably better, but of giving up one set of compromises and trade-offs in exchange for another*" [23].

This equilibrium seems to be too strong to disrupt, whether because it would imply that service providers adapt their services to alternative authentication methods, changing all their architecture and infrastructures or simply because users probably would not take the change lightly, since they are too accustomed to passwords and the cost per user is usually low — a user just needs to reuse a password or modify it slightly. Consequently, it appears to be a fair statement that passwords are here to stay.

However, this method is not without its flaws. Password reuse is evidently a major predicament regarding password security, which can be aggravated by Cross-Site Scripting (XSS) attacks. These attacks leverage weaknesses in websites that allow attackers to inject malicious code and grant the attacker the ability to steal sensitive data, like login credentials, a session token or cookies. This data can later be leaked to the internet.

We can imagine the possible perils that await a user that reuses a password on a vulnerable website. Whenever an attacker gets access to the user's credentials, a domino effect [24] takes place: all the sites where the user has this revealed password are no longer protected — it is only a matter of time to guess the user's username. Ives et al. [24] are very clear: "*Users who reuse passwords often fail to realize their most well-defended account is no more secure than the most poorly defended account for which they use that same password.*".

2.2 Password Reuse

The number of services and applications in the internet that require a user to authenticate has grown at an incredible pace [25]. To cope with this increasing number of accounts, users tend to reuse passwords across multiple websites. This behaviour allows for greater flexibility and less effort memorizing passwords, since users can cling to a set of passwords and constantly reutilize them across multiple accounts, introducing little to no variations to these passwords. Like a key that opens multiple doors facilitates the task of opening doors but is dangerous if lost, the same principle applies to reused passwords when a malicious user gets hold of these passwords — the attacker can now unlock multiple accounts with the same password.

In 2007, Florêncio et al. [4] found that the average user has 6.9 passwords and each of which is shared across 3.9 different sites; the big majority of these users choose passwords that contain only lower case letters. These users maintained about 25 accounts each, and entered 8.11 passwords on a daily basis.

Stobert and Biddle [8], in 2014, found that the participants in their study had between 2 and 20 unique passwords, with the median being 5 unique passwords. From the total of 27 participants, 26 reported reusing passwords between accounts, with 88% of these users claiming they reused more than one password.

Das et al. [6], also in 2014, observed that a substantial portion of their user universe, 43%, directly reuse passwords between sites and use small modifications to these passwords to make them distinct across the different webpages. The authors demonstrated that many of these users also share the same algorithm for introducing these modifications, which can certainly be a problem: if an attacker is aware of these common algorithms, which are quite simple, it is fairly easy to take advantage of them and therefore get a more efficient attack with improved guessing capabilities.

In 2016, Wash et al. [26] performed a study with 134 participants over the course of six weeks, and found that people tend to reuse their passwords on 1.7-3.4 different websites and visit an average of 118 pages per day with an average of 3.2 passwords entered per day. The subjects of the study used a median of 12 distinct passwords, which is not a big number, given how frequently subjects need to enter a password into a webpage. About 85% of the participants had fewer unique passwords than they did websites that they entered passwords into. The authors found that a median user had 6 unique passwords and inserted each of these passwords in a median of 3 websites. Each subject's most used reused password was used on an average of 9 different websites.

In 2017, Pearman et al. [7] conducted a study with 154 participants and found that most of them (122 users, 79.2%) adopted hybrid strategies incorporating both exact and partial reuse in order to manage their passwords. They also found that passwords that contain digits and symbols are more likely to be reused, possibly due to the greater probability of compliance with the various policies enforced by different websites.

In a more recent study with 30 participants, including users who use no password-specific tools at all, those who use password managers built into browsers or operating systems, and those who use separately installed password managers, Pearman et al. [9] found that users of built-in password managers may be driven more by convenience, while users of separately installed tools appear more driven by security. This helps explain why past findings conclude that there are higher levels of password reuse among users of built-in password managers. The authors also identify new obstacles for password manager adoption, such as confusion about the source of password prompts or the meaning of ”remember me” options.

Users are regularly regarded as lazy and unmotivated on security questions, especially regarding passwords. Herley [27] argues that this is both unfair and untrue: users view security guidance from a different perspective than security researchers — an economical one. Users consider adopting these pieces of advice and usually end up discarding them. This occurs because there is no clear advantage in doing otherwise: the benefits of adopting these measures do not outweigh the cost that comes as a consequence — the time that a user will spend following and adapting to comply with the advice. This antagonistic view occurs because users only care about the average or actual harm of an attack; nevertheless, security researchers frequently present guidelines with a worst-case scenario in mind. Herley formulates a rough draft about the cost of the user's time, commonly assumed to have no cost at all: $2.6 billion. With this number, we treat the “*user as a professional who bills at $2.6 billion an hour*” [27], which allows for a better understanding of why users ignore security policies.

Other authors [26] also view password reuse as a possible good practice, from a cost/benefit perspective, claiming that if strong passwords — like the ones enforced by companies — are reused it might be a benefit security-wise. However, this behaviour can also be dangerous if these stronger passwords are reused in lower-security websites, making the password vulnerable and therefore, the company.

2.3 Password Managers

Since passwords continue to be the most used security mechanism in terms of web-authentication, and in order to mitigate these vulnerabilities that are intrinsic to them, password managers present themselves as the missing piece of this intricate puzzle. They allow the generation of randomly strong passwords, and they relieve users from the burden of remembering the credentials to the multitude of web applications that an average user has accounts in. Notwithstanding, password managers also have some vulnerabilities. While it is true that they can randomly generate strong passwords, there is a big overhead for users to manually change passwords for all their accounts, which is seen as a big cost from the user's perspective. Ultimately, this translates to users not using this software or just storing their original passwords in password managers — as is hypothesised by Pearman's and Wash's work [7, 26] —, which may provide a false sense of security to a less security-savvy user. This is a big usability obstacle against wide-spread usage of password managers.

Security vulnerabilities also shadow password managers, ranging from occasionally generating easily guessable random passwords, to storing private information in clear-text and auto-filling information into possible endangered websites [15]. Despite all these setbacks, password managers are still recommended by security researchers [6, 9] as the best companion to brave through the abundance of applications that require password-based authentication.

During the last 16 years, there were multiple proposals to mitigate most of the risks associated with password-based authentication.

In 2005, Dhamija and Tygar [28] suggest a new way for a remote web server to authenticate itself, easier for humans to verify and hard for an attacker to emulate: Dynamic Security Skins. Aside from being based in the Secure Remote Password (SRP) protocol [29], meaning that the password is never sent to the server, this interesting browser extension provides a secure and trusted window in the browser that is dedicated to input sensitive data from the user, namely, the username and the password. To prevent eventual spoofing of this window by an attacker, a photographic image is used. This scheme also allows the server to generate unique abstract images for each user and each transaction, creating a skin that automatically customizes the browser window or the user-data inputs on a given website. This images permit the user to independently compute the image that he expects to receive from the server.

Web Wallet, created by Wu et al. [30], prevents phishing by displaying a sidebar and instantly blocking any input from the user to a sensitive form, forcing the user to use the app to fill the form. It also does some checks to verify if the site the user wants to access is a legitimate one or if it was somehow spoofed or tampered with. This is done by asking the user to indicate, from a list of websites, what website he is trying to access. If the site the user is trying to access is different than the one he wants to access, then a warning is issued and *Web Wallet* provides a legitimate and safe link for the website.

oPass, developed by Sun et al. [31] emerges with the premise that it is impossible to thwart password reuse attacks from any scheme where the users have to remember something. Thus, *oPass* mitigates this by relieving users from having to remember or type any passwords into conventional computers for authentication, using a cellphone, which is needed to generate one-time passwords, as a method to achieve this user authentication, transmitting the information over a different communication channel, Short Message Service (SMS). This is identical to Token Based Authentication, where the token is the SMS received and is used nowadays as an extra layer of protection for most applications.

LoginInspector [32] leverages the security advantages of password hashing techniques but it does not inherit their usability disadvantages, e.g., requiring users to migrate their original passwords to hashed passwords. When a user types its login credentials, *LoginInspector* intercepts this information. Then, it will inspect whether there is a corresponding successful login record for this website account in the database. If there is a corresponding login record, *LoginInspector* will submit the intercepted information to the server. If no record is found or the login credentials inserted do not match the record found, the user will see a warning mentioning that there is no record saved or that the record saved does not match the intercepted information, respectively. It is now up to the user whether or not to submit the intercepted login data to the server or cancel the submission.

Each successful login record is uniquely identified by a $recordHmac$ value. Each record also has a $domainHmac$ that identifies the domain for which the record is valid. These two values are calculated using formulas 2.1 and 2.2, respectively:

$$recordHmac = HMAC(key, d||u||p) \tag{2.1}$$

$$domainHmac = HMAC(key, d) \tag{2.2}$$

where key is a secret key either randomly generated by the extension or directly specified by a user when *LoginInspector* is installed; $HMAC$ is the Hash-Based Message Authentication Code (HMAC) mechanism together with the SHA-256 cryptographic hash function; d, u, and p represent the domain name, username and password, respectively; "||" is the string concatenation operator. The secret key is securely stored in the password manager of a browser.

This extension also allows reports to be sent to administrators, to assess overall security across an enterprise or a university campus.

McCarney et al. [1] evaluate the security of dual-possession authentication, that offers encrypted storage of passwords and theft-resistance without the use of a master password and furthermore, propose *Tapas*, a browser extension which takes advantage of this authentication method to provide a password manager that does not require server side changes, nor a master password whilst protecting all the stored data in the eventuality that the primary or secondary device is compromised. This dual-possession authentication involves two applications, a Manager and a Wallet, on different devices. The

Manager will initially generate a QR Code to establish a unidirectional authenticated and secret out-of-band (AS-OOB) with the Wallet. There are three main protocols: Pair (Figure 2.1), Store (Figure 2.2), and Retrieve (Figure 2.3).

Protocol 1: Pairing Manager and Wallet

User action: Upon a user choosing to set-up a new Wallet, the following protocol is initiated by the Manager.

Communication channel: A one-way authenticated and secret out-of-band (AS-OOB) channel from the Manager to the Wallet.

1. The Manager generates an authentication key pair for itself $\langle pk_m, sk_m \rangle$ and sends its public key pk_m to the Wallet.
2. The Manager generates an authentication key pair for the Wallet $\langle pk_w, sk_w \rangle$ and sends the pair to the Wallet.
3. The Manager generates a secret key k for a symmetric key authenticated encryption scheme $\mathsf{Enc}_k()$.

Output: The Manager stores $\langle pk_m, pk_w, sk_m, k \rangle$ and erases sk_w. The Wallet stores $\langle pk_m, pk_w, sk_w \rangle$.

Figure 2.1: Protocol 1 - Pairing Manager and Wallet. From McCarney et al.'s [1] work.

Protocol 2: Storing a Password

User action: Upon a user choosing to save a password p_i, the following protocol is initiated by the Manager.

Communication channel: A mutually-authenticated secure channel with perfect forward secrecy between the Manager and the Wallet. The participants, respectively, identify themselves with pk_m and pk_w.

1. The Manager takes user password p_i (entered by user) and site information s_i and computes $c_i = \mathsf{Enc}_k(p_i \| s_i)$.
2. The Manager sends $\langle c_i, s_i \rangle$ to the Wallet.
3. The Wallet prompts the user to create a tag t_i for referencing the site, using s_i to suggest a value for the tag.

Output: The Manager erases $\langle p_i, s_i, c_i \rangle$. The Wallet stores $\langle t_i, c_i \rangle$ and erases s_i.

Figure 2.2: Protocol 2 - Storing a Password. From McCarney et al.'s [1] work.

Protocol 3: Retrieving a Password

User action: Upon a user choosing a password for retrieval, the following protocol is initiated by the Wallet.

Communication channel: A mutually-authenticated secure channel with perfect forward secrecy between the Manager and the Wallet. The participants, respectively, identify themselves with pk_m and pk_w.

1. The Wallet retrieves the c_i value associated with the tapped t_i, and sends c_i to the Manager.
2. The Manager decrypts and authenticates c_i to retrieve s_i and p_i.
3. The Manager checks that s_i matches the site information for the current site that the browser is visiting.
4. The Manager transfers the user password p_i to the site.

Output: The Manager erases $\langle p_i, s_i, c_i \rangle$.

Figure 2.3: Protocol 3 - Retrieving a Password. From McCarney et al.'s [1] work.

These protocols are built in a way that if data from either the Manager or the Wallet is stolen, the attacker cannot determine the stored password for any given account with any greater success than attacking the account directly. This is achieved by encrypting each password with a key held by the Manager and storing the resulting ciphertext on the Wallet. Thus, by stealing the Manager, the adversary obtains the decryption key but not the ciphertext to decrypt, and by stealing the Wallet, the adversary only has a set of ciphertexts resistant to offline attacks.

Mayer et al. [33] present an interesting feature that some password managers already employ [1]: automatic password change. This extension allows the users to record the actions taken to change their passwords on a given website, store them as a *blueprint* and then share it with other users. For this, a user would first need to go to a website, use the extension's option to record a blueprint, and then add the form fields necessary to change the password. This blueprint would also ask the user for the password policy of the website if the website was not using a standardized way of expressing it like the ones suggested by [2, 14]. There is a possibility for multiple trust authorities, but the simplest would be the developer of the password manager. The authors propose that the trust authorities should digitally sign all crowdsourced information to ensure its authenticity.

Recently, Oesch and Ruoti [15] revisited previous work done on password managers' security and usability. All the work related to security issues of autofill and data storage [34–37] is more than five years old. Even though some of the vulnerabilities exposed have been fixed, some still remain to this day, such as autofill in a website with an invalid Hypertext Transfer Protocol Secure (HTTPS) certificate or a significant amount of unencrypted metadata being stored, like a page Uniform Resource Locator (URL), a user's username or information about the creation and last access of a given account.

Oesch and Ruoti study eleven browser-based password managers and two desktop password managers, and, according to them, their work is the first to consider all three stages of a password manager lifecycle — password generation, password storage, and password autofill. The next subsections concern each one of these stages, the vulnerabilities that the authors found, and the past work developed on the subject.

2.3.1 Password Generation

Password generation is the first stage of a password manager lifecycle, and it concerns the generation of strong, random and unique passwords, such that these generated passwords are very difficult to be cracked by guessing attacks and are nearly impossible for a regular user to memorize them.

Oesch and Ruoti's [15] work discovered that not all studied password managers include the same character set, which can be misleading for a user when generating an allegedly *unique and random* password. It also concludes that passwords containing 12 or more characters generated by the analyzed password managers are, generally, resilient against online and offline guessing attacks. Still, there were some discrepancies in the strength of generated passwords, specially with length 8, which significantly impacted the percentage of passwords that were secure against offline guessing attacks — almost every password was secure against online guessing attacks. These differences in strength can be justified by the different sets of characters used to generate a password. There is also a problem related to the randomness of these generated passwords: they can be randomly weak passwords, even when

[1] `https://www.dashlane.com/features/password-changer`

containing letters, digits, and symbols, e.g., *d@rKn3s5* or *Tz5a5a5a*.

Ross et al. [38] suggest *PwdHash*, a browser extension that creates a different password for each site, which defends against password phishing and improves general security. These passwords are generated using cryptographic hash functions, in combination with the actual plaintext password, some basic information of the website, and an optional private salt stored in the client machine. It is fairly simple to understand that, if an attacker gets access to the password of a given site, he just got the hashed value of the password, and not the password itself, leaving other user login information that shares the same password protected.

Halderman et al. developed *Password Multiplier* [39], which relies only on a strengthened cryptographic hash function that computes secure passwords for an arbitrary number of accounts. This browser extension runs only in the client-side, and requires no server-side adaptation. The authors claim that previous work [38] is primarily intended to defend against phishing and spoofing attacks, whilst their solution includes protection against offline brute force attacks, by using the key stretching technique presented by Kelsey et al. [40]. *Password Multiplier* avoids password reuse entirely while requiring the user to memorize only a single master password, offering greater convenience, since the account passwords are generated deterministically, allowing the users to run the software on different machines to access their information data.

Also related to password hashing, Yee and Sitaker [41] present a browser extension, *Passpet*, with the premise that a user should only have to memorize one secret, instead of many, and still have a unique password at each site, in order to reduce user's vulnerability. Passpet allows users to create labels for each website, making it easier for the user to recognize the website. The extensions associates the site label entered by the user with a *site identifier* consisting of *(root_key, field_name, field_value)*. For Secure Sockets Layer (SSL) sites, *root_key* is the fingerprint of the root certificate authority's public key. If the site's SSL certificate has an Organization Name field, then *field_name* is "O" and *field_value* contains the Organization Name. Otherwise, *field_name* is "CN" and *field_value* contains the certificate's Common Name instead. If the user assigns the label *"my bank"* to a bank's SSL site, another site can only cause the site label *"my bank"* to appear if it can obtain a certificate with a certificate chain ultimately signed by the same root authority and which yields the same identifier. Thus, *Passpet* guarantees that the user is on the desired website, rather than a fake website. This works like a nickname, or a petname, hence the extension's name.

2.3.2 Password Storage

The second stage of the lifecycle is password storage. It concerns the safe storage of the generated passwords and all the user details that help a password manager identify the website in question.

With their analysis, Oesch and Ruoti [15] discovered that the vulnerabilities exposed by Gasti and

Rasmussen [34], which allowed an attacker to either gain *read access* or *read and write access* to the password manager's database, were mostly mitigated, resulting in better storage of the password manager's metadata. Even so, there are still some password managers that store relevant metadata in plaintext, either by default or as a last resource. This metadata can be the manager's settings or information that allows to identify a user like website URL, website icons or the username of the user in a particular website.

A study conducted in 2019[2] also found that most of the password managers were not encrypting passwords written in memory, making it relatively easy for an attacker to extract passwords from the password vault even when not in use.

2.3.3 Password Autofill

Autofill is the third and last step of the lifecycle. Autofill is the ability that a password manager has to fill login forms automatically. It is a useful tool for users since they can skip the trouble of having to type passwords or skim through their list of credentials stored in the manager, but it is not without security concerns. For most applications, autofill is still done automatically, not requiring user interaction. However, as pointed out by some authors, this is dangerous [36, 37].

With their work, Oesch and Ruoti [15] found that, of the studied browser-based password managers, only Safari's would require user interaction always. Firefox's manager defaults to autofill without any user interaction, even if there is an available option to revert this. Chrome always autofills user credentials and does not have an option to change this setting. However, Chrome does not autofill for sites with a bad HTTPS certificate, whilst Firefox maintains its regular behaviour of autofilling, endangering the user.

Regarding iframes, autofill is very dangerous, whether user interaction is required or not [36, 37]. A user is vulnerable, mainly, to two attacks: clickjacking — which consists in tricking users into providing the necessary user interaction to autofill their passwords for vulnerable websites loaded in an iframe (same-origin or cross-origin) — and harvesting attack — where the attacker can programatically harvest the user's credentials for all vulnerable websites where the user has an account, by loading these compromised websites into iframes (cross-origin only). For these attacks to work, the user must first visit a malicious website, that launches the attacks [15]. The authors found that Chrome would require user interaction before autofilling passwords into a cross-origin iframe but not into a same-origin iframe — making them vulnerable to clickjacking attacks on both scenarios. Firefox defaults to its regular behaviour: not requiring user interaction to autofill passwords, leaving users vulnerable to the harvesting attack.

To verify that the form that will be filled is not compromised, password managers employ some checks regarding the security and integrity of that same form. Oesch and Ruoti [15] expose that if the password

[2] https://www.ise.io/casestudies/password-manager-hacking/ - Last access: 30 October 2021

was saved on a form served over HTTPS, Chrome will refuse to fill the credentials in a form served with a bad HTTPS certificate or served over Hypertext Transfer Protocol (HTTP). Firefox will also refuse to fill it if the form is served over HTTP, but ignores the bad certificate scenario. Upon the page first load, if there is a contradiction in the form's *action* property — the URL to which the form will be submitted to — Firefox will display a warning and will refuse to autofill the form. However if this URL is changed after the first load, i.e., dynamically, Firefox will display a warning to the user, but will still autofill the form.

Gonzalez et al. [42] developed *Lupin*, an extension that can steal all the credentials that a user has stored in a browser-based password manager. It does so by taking advantage of the autofill policy of the password manager: if no user interaction is required, then most likely the attack will succeed, since this attack's success depends on the ability to deceive the password manager to autofill the user's credentials into a web page that has been tampered by an attacker.

Stock and Johns [37] present a solid approach to mitigate most XSS attacks to browser-based password managers. They make a case that most password managers automatically fill out the forms with the passwords in clear-text, making these passwords accessible to client-side code. The authors come up with a solution: fill the password fields with a placeholder value which will be converted to the real passwords only when the request is sent to the server. The authors claim that their implementation "*effectively hinders an attacker who utilizes XSS attacks against victims.*". However, man-in-the-middle attacks can still be successful if the password data is transmitted in clear-text to the server. The extension developed has strict integrity checks related to the password's form context and only exchanges the placeholder for the real password if the origins and names of both URL and saved password match.

Both Stajano et al.'s [2] and Horch et al.'s [14] work proposes similar solutions to the problem of password managers not being able to know the password composition rules that a given website has in place. We take a closer look in subsection 2.3.3.A.

2.3.3.A Password Rules Annotations

Stajano et al. [2] propose adding HTML semantic labels, or annotations, to facilitate and normalize the work done by password managers. These labels allow for a standardized way for password managers to extract the semantics from the HTML form, to better understand the type of form that is being addressed — a login form, a password change form or a regular form, which, in this case, would not have such labels. This minimizes the percentage of false positives — the password manager offers to save a wrong password, either from a wrong form or a wrong field in the form — and false negatives — the password manager does not offer to save the correct password, inserted in the correct password field — that users may face with password managers. So, in a register form, the password manager could offer to save the password but not the username, leaving this entry incomplete and putting the effort on the user to remember the username.

```
<form action="/change" method="POST" class="pmf-change-password">
    <input type="hidden" class="pmf-username" value="jimbojones"/>
    <p>
        Current password:
        <input type="password" name="current" class="pmf-password"/>
    </p>
    <p>
        New password:
        <input type="password" name="new" class="pmf-new-password"/>
    </p>
    <p>
        Confirm new password:
        <input type="password" name="confirm" class="pmf-new-password"/>
    </p>
    <p>
        <input type="submit" value="Change password"/>
    </p>
</form>
```

Figure 2.4: An example of the HTML annotations that allow a password manager to infer the different types of forms. Example based on the work of Stajano et al. [2] and extracted from the project's git repository [3].

Figure 2.4 demonstrates an example of a form to change an account's password using these annotations.

The authors also suggest that most password policies are too restrictive, which can be a sign of passwords not being hashed [43], and argue that passwords, if salted and hashed, the hash to be stored will have fixed length, no matter the length of the password itself. Thus, they advocate the use of a simple heuristic:

```
# we are sure it is not a human-generated pw
    if (pw_length >= t):
        accept_password();
```

Even though this seems like a good policy, there is a trade-off: it is troublesome to retype such a big password, but the password should not be so short that transcription becomes the preferred *modus operandi*. The authors suggest t=50 base-64 characters, ensuring that the machine-generated passwords will be strong, random and too long to be memorized or transcribed. If a user decides that a certain password needs to be shorter than that, he can change the length, being subjected to the website's password policies.

Horch et al. [14] put forward a *Password Policy Markup Language*, that allows websites to describe their policies regarding passwords. This permits the password manager to better help the user and automate the whole password lifecycle: create, login, change, and reset passwords. For this, it is only necessary that the web developers expose a service detailing password policies and a URL for the respective actions.

2.3.3.B Google's Password Requirements API

Google has also implemented a solution regarding this problem [17]. They implemented an API which could be called by a password manager when generating a password for a given web domain. This way, the generator can learn the password requirements of that particular website — if the API has any information regarding it. The data is returned as Procotol Buffers (protobuf) [3]. As of February of 2021, this API includes password requirements for 237 websites[4].

2.3.3.C Apple's Password Autofill Rules

Apple created the Password Autofill Rules [16]. These rules are described using an HTML annotation — `passwordrules` — that lets the web admin define the rules for creating a valid password. These rules can later be parsed by any password manager to generate compliant password.

We found that Apple's approach is more straightforward: it is easier to use, from the webadmin's perspective, the code is open-source, and there is more support for developers. Plus, it is closely related to previous research suggestions. Thus, we will base our solution on these annotations. We explain further details in Chapter 3.

[3]Protobuf - https://github.com/protocolbuffers/protobuf/releases/tag/v3.19.0
[4]https://github.com/apple/password-manager-resources/issues/427

3

Extending Apple's Password Autofill Rules

Contents

In this chapter we present our work to extend Apple's DSL to make it capable of expressing password policies suggested by researchers.

Initially we had planned to create a new DSL to accommodate suggestions made by Stajano et al.'s work [2] and reinforced by Oesch and Ruoti's research [15]. However, during the development phase of this project, we found that Apple had already made efforts in this direction [16], and that their browser, Safari, and most of the applications in macOS and iOS take advantage of this. Thus, in order to maximize the possible impact of our work, we decided that we could start our work building off of these Password Autofill Rules.

3.1 Apple's Password Autofill Rules

Apple's Password Autofill Rules [16] are a DSL that can be used to express password composition policies. The goal is to provide a standardized way for applications to generate strong passwords that comply with a specified policy.

Apple's DSL is based on five properties — *required, allowed, max-consecutive, minlength, and maxlength* — and some identifiers that describe character classes — *upper, lower, digit, special, ascii-printable, and unicode*. These are the elements that allow the description of the password rules. It is also possible to specify a custom set of characters by surrounding it with square brackets (e.g., *[abcd]* denotes the lowercase letters from *a* to *d*). For example, to require a password with at least eight characters consisting of a mix of uppercase letters, lowercase letters, and numbers, the following rules can be used:

```
required: upper; required: lower; required: digit; minlength: 8;
```

A more formal description of the grammar is shown in Figure 3.1.

3.1.1 Properties description

The `required` property is used when the restrictions must be followed by all generated passwords. The `allowed` property is used to specify a subset of allowed characters, i.e., it is used when a password is permitted to have a given character class, but it is not mandatory.

If `allowed` is not included in the rule, all the `required` characters are permitted. If both properties are specified, the subspace of all `required` and `allowed` is permitted. For example, to have a password that contains at least one lowercase letter, minimum size of 8 and can have uppercase and digits, these rules can be used:

```
minlength: 8; required: lower; allowed: upper, digit;
```

```
<rule> ::= (<required> | <allowed> | <length_reqs> | <max_consecutive>)*
<required> ::= "required: " <list_ids_classes> "; "
<allowed> ::= "allowed: " <list_ids_classes> "; "
<length_reqs> ::= "minlength: " <non_negative_integer> "; "
                | "maxlength: " <non_negative_integer> "; "
<max_consecutive> ::= "max-consecutive: " <non_negative_integer> "; "
<id_class> ::= (<identifier> | <character_class>)
<list_ids_classes> ::= <id_class> | <id_class> ", " <list_ids_classes>
<identifier> ::= "lower" | "upper" | "digit" | "special"
              | "ascii-printable" | "unicode"
<character_class> ::= "[" (<upper> | <lower> | <special> | <digit>)+ "]"
<digit> ::= <non_negative_integer>
<non_negative_integer> ::= [0-9]+
<lower> ::= [a-z]
<upper> ::= [A-Z]
<special> ::= "-" | "~" | "!" | "@" | "#" | "$" | "%"
            | "^" | "&" | "*" | "_" | "+" | "=" | "`" | "|" | "("
            | ")" | "{" | "}" | "[" | ":" | ";" | """ | "'" | "<"
            | ">" | "," | "." | "?" | " " | "]"
```

Figure 3.1: Grammar used by Apple's Password Autofill Rules. The * means repeated application of 0 or more rules. The + means repeated application of 1 or more rules.

This rule will allow passwords like `abcdefghi`, `aBCDEFGHI`, `a1234567` or `aBC12345`. If neither `required` nor `allowed` is specified, every ASCII character is permitted.

The `max-consecutive` property represents the maximum length of a run of consecutive identical characters that can be present in the generated password, e.g., the sequence `aah` would be possible with `max-consecutive: 2`, but `aaah` would not. If multiple `max-consecutive` properties are specified, the value considered will be the minimum of them all.

The `minlength` and `maxlength` properties denote the minimum and maximum number of characters, respectively, that a password can have to be accepted. Both numbers need to be greater than 0 and `minlength` has to be at most `maxlength`; otherwise, the default length of the password manager will be used.

3.1.2 Identifiers

Next to the `allowed` or `required` properties, we can use any of the default ***identifiers***, which describe *conventional* character classes. The identifier `upper` describes the character class that includes all uppercase letters, i.e., *[A-Z]*; the identifier `lower` describes the character class that includes all lowercase letters, i.e., *[a-z]*; the `digit` identifier describes the character class that includes all digits, i.e., *[0-9]*; and the `special` identifier describes the character class that includes -˜!@#$%ˆ&*_+=‘|(){}[:;”’<>,.?] and ␣.

The identifiers `ascii-printable` and `unicode` describe the character classes that include all ASCII printable characters and all the unicode characters, respectively.

Additionally, users of the DSL can choose to describe their custom character classes by surrounding the characters with squared brackets — `[]` — e.g., to require a password to have at least one lowercase vowel, minimum length of 8, and to allow digits and uppercase letters, the following rule can be used:

```
minlength: 8; required: [aeiou]; allowed: upper, digit;
```

The rule `required: [aeiou];` requires that at least one of the characters in this custom set *must be* present in the password.

The default password rule when no rule is defined is `allowed: ascii-printable;`.

3.1.3 Weaknesses

Apple's DSL is an effort from password manager's developers to augment usability and reduce users' frustration when a generated password fails to comply with a website's password policy [12]. With it, it is possible to achieve a great set of password policies with all these rules. Apple has even provided a website that allows a web admin to test password policies and view passwords that comply with such policies [44]. However, it appears there are some incoherences, either with the official documentation or with the password generator itself.

For example, in the official documentation [16], one can read "*To require at least one digit or one special character, but not both, add this to your markup*":

```
required: upper; required: lower; required: digit, [-().&\@?'#,";+];
max-consecutive: 2; minlength: 8;
```

From our understanding, these rules would accept passwords like `ABcd56eF` or like `ABcd-#eF`, but not like `ABcd56-#`. That is to say that these rules restrict the required characters: the password must have `upper`, `lower`, and either `digit` ***or*** `[-().&\@?'#,";+]`, but not both. Still, this is not the case in the official generator [44], which generates passwords like `&z,#Iu5(` and `id3LYk+H` for the same rules: they both have digits and special characters.

Another shortcoming of this DSL is the fact that some password policies studied and suggested by recent research literature on password composition policies are not possible to describe (e.g. the policies used by Tan et al.'s and Shay et al.'s work [18, 19], which are displayed in Table A.2 and Table A.1, respectively.). In particular, it is not possible to express the blocklist constraints or the restraints on the minimum number of classes that a password should have. It is also impossible to restrict the frequency of a character — the `required` rule only guarantees that the character will appear once. Such was the motivation that led us to extend Apple's DSL, as described in the following section.

3.2 Extending Apple's DSL

In order to improve Apple's DSL to be compatible with password policies suggested in academic research [18, 19, 45], we added three new functionalities:

1. The `blocklist` rule;
2. The `minclasses` rule;
3. The possibility to add a range for any given character class.

The new grammar can be seen in Figure 3.2 and a comparison can be seen in Table A.3. Notice the new rules `<blocklist>` and `<minclasses>` and the addition of the ranged rule in both `<identifier>` and `<character_class>`.

```
<rule> ::= (<required> | <allowed> | <length_reqs> | <max_consecutive>
              | <blocklist> | <minclasses> )*
<required> ::= "required: " <list_ids_classes> "; "
<allowed> ::= "allowed: " <list_ids_classes> "; "
<length_reqs> ::= "minlength: " <non_negative_integer> "; "
                | "maxlength: " <non_negative_integer> "; "
<max_consecutive> ::= "max-consecutive: " <non_negative_integer> "; "
 <blocklist>  ::= "blocklist: " <blocklist_values> "; "
 <minclasses>  ::= "minclasses: " <non_negative_integer> "; "
<id_class> ::= (<identifier> | <character_class>)
<list_ids_classes> ::= <id_class> | <id_class> ", " <list_ids_classes>
<identifier> ::= ( <ident_range>  | <ident_no_range>)
 <ident_range>   ::= ("lower" | "upper" | "digit" | "special"
                            | "ascii-printable" | "unicode")
                            "(" <non_negative_integer> ", "
                            <non_negative_integer> ")"
<ident_no_range> ::= "lower" | "upper" | "digit" | "special"
                            | "ascii-printable" | "unicode"
<character_class> ::= ( <cc_range>  | <cc_no_range>)
 <cc_range>  ::= "[" (<upper> | <lower> | <special> | <digit>)+ "]"
                            "(" <non_negative_integer> ", "
                            <non_negative_integer> ")"
<cc_no_range> ::= "[" (<upper> | <lower> | <special> | <digit>)+ "]"
<blocklist_values> ::= "hibp" | "default"
<digit> ::= <non_negative_integer>
<non_negative_integer> ::= [0-9]+
<lower> ::= [a-z]
<upper> ::= [A-Z]
<special> ::= "-" | "~" | "!" | "@" | "#" | "$" | "%"
              | "^" | "&" | "*" | "_" | "+" | "=" | "`" | "|" | "("
              | ")" | "{" | "}" | "[" | ":" | ";" | "\"" | "'" | "<"
              | ">" | "," | "." | "?" | " " | "]"
```

Figure 3.2: Grammar of extended Apple's DSL. The rules `blocklist` and `minclasses` were added, as well as the possibility to add character ranges (see highlighted items).

Blocklist Rule The `blocklist` rule was added to inform the password manager that the website performs a hash verification of the password against a blocklist, i.e., a list containing the most commonly used passwords. This is reinforced by Tan et al's work "*(...) minimumlength or blocklist requirements, can strengthen passwords with less negative impact on usability (...)*" [19].

Thus, to generate a compliant password, the password manager needs to perform a check against a blocklist. Usually, these lists are built by compiling data breaches that end up being public. A great resource that collects and compiles these data breaches is 'Have I Been Pwned (HIBP)'[1], where anyone can check if a given password matches any of all the 613 584 246 exposed passwords[2] present in the HIBP list. HIBP offers an API that lets developers perform these kinds of verifications.

This rule takes one of two values:

- `blocklist: hibp;`
- `blocklist: default;`

The value `hibp` indicates that the website will be checking the password with HIBP and the password manager should also check the generated password with this tool by using the available API. All passwords within HIBP's collection are hashed using Secure Hash Algorithm 1 (SHA-1). Thus, to utilize this tool, the password must be hashed, using SHA-1, and then only the first 5 characters of the resulting hash are passed to the API, since the service implements a `k-Anonimity` model, which enables a password to be searched for by a partial hash, never revealing the complete hash of the password. When a password hash that has the same first 5 characters is found, the API will respond with the suffix of all the passwords that start with the specified prefix, as well as the number of times each appears in the data set. The check will be complete when the developer verifies all the found suffixes, returned by the API, and searches to find the suffix of the generated password: if it is not present, the password has not been found in any data breaches (yet!).

As an example, suppose we want to search the HIBP data set for the password `hello world`. For this, we would calculate the SHA-1 of the password, and get the hash:

```
2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
```

The first 5 digits are `2aae6`, which are the ones we send to the API, and we will get the results shown on Figure 3.3. The last record matches the SHA-1 hash suffix of our test password, which has been found 136 times in over 500 million passwords, i.e., in all the leaks that this library includes, the password `hello world` was found 136 times.

The HIBP tool also offers the possibility of padding the answer with a random number — between 800 and 1000 — of entries, which means that every request will have answers with different sizes,

[1] https://haveibeenpwned.com/Passwords
[2] At the time of writting.

```
GET https://api.pwnedpasswords.com/range/2aae6

0095F96A5C2C3F33C7ED4B9E33000629B0B:2
00E20AB3F9624537DAE8B62E292A70D2799:18
0196171953B6E5C30E0EBF35C0062F62A8D:2
...
C35C94FCFB415DBE95F408B9CE91EE846ED:136
```

Figure 3.3: HIBP's API call for the test password `hello world`. The last result is the correspondent suffix of the SHA-1 hash.

even if the query is the same. This is relevant to protect the user from an attacker that may be sniffing the network[3]. With this padding, there is a *remote* possibility that one of the random hashes used for padding the answer will match with the actual hash of the password.

The value `default` indicates that the website will have a local list of passwords to be avoided and the password manager should also check the generated password against a reasonable list of popular passwords. Whilst the list of the website may follow a logic that is unknown to any password manager, it is common sense that such a list should contain some very popular password choices, like `123456` and `password`. There is also the possibility to configure the list at will, allowing developers using our extension to create and use their own list.

Minclasses Rule The `minclasses` rule was added to bring more flexibility to the generator. With Apple's DSL it is not possible to specify a policy like "*Password must contain only characters from 3 different character classes*". Because in this case there are 4 character classes — uppercase, lowercase, digit and special —, there are $C_3(4) = 4$ possibilities[4] of combinations. On the other hand, if we had the policy "*Password must contain only characters from 2 different character classes*", we would have $C_2(4) = 6$ possibilities. We could write a rule like:

```
required: upper; required: lower;
```

However, this rule would always force the password to contain both uppercase and lowercase letters. Consequently, the creation of passwords would not consider digits and special characters — a total of 41 characters combined —, limiting the randomness of the password; therefore, its strength. There is also an inability to express academically recommended policies in Apple's DSL — for example, it is not possible to express the policy `3c8`, which requires a minimum of 8 characters and at least 3 character classes, nor `2c8`, which requires a minimum of 8 characters and at least 2 character classes [18,19,45]. This was the motivation to create this new rule.

[3] `https://github.com/lakiw/pwnedpasswords_padding`

[4] $C_k(n) = \binom{n}{k} = \frac{n!}{k!(n-k)!}$ - A combination of a k-th class of n elements is an unordered k-element group formed from a set of n elements. The elements are not repeated, and it does not matter the order of the group's elements.

In spite of Tan et al.'s experimental results, "*Our experimental results provide the first concrete evidence that character-class requirements should be avoided not only because users tend to find them annoying, but also because they don't provide substantial benefit against attackers using state-of-the-art password-cracking tools: an expert attacker can guess 1c8, 3c8, and 4c8 passwords with equal success rates.*" [19], we still think it is beneficial to include this feature in the language, because this DSL should be as flexible as possible, adjusted to the users' needs. If there are a lot of websites and services using this type of policy (and there are!), then the password manager should be able to provide compliant passwords, every time.

The range of values for `minclasses` is equivalent to the minimum and maximum number of classes: 1 and 4, respectively. So, to represent two policies suggested in the literature, `3c8` and `2c8`, we could write, respectively:

- `minlength: 8; allowed: ascii-printable; minclasses: 3;`
- `minlength: 8; allowed: ascii-printable; minclasses: 2;`

Character Range Aside from greater control over passwords, there is a compatibility aspect to this feature. As we mentioned before, Google has an API that also provides the password policies for a list of websites [17]. In an attempt to synchronize the policies gathered by Google with Apple's list [46], there was a suggestion to import Google's list [5]. Yet, Google's API allows for the manipulation of minimum and maximum values for any given character class. This reinforces the importance of our extension.

The range property for both named and custom character classes is defined by using two integers inside parenthesis, e.g., `<character_class>(minimum, maximum);`. The first number represents the minimum number of occurrences for that character class and the second number represents the maximum number of occurrences for that character class. To represent a policy such as "*Password must contain 3 'a', at least 3 uppercase letters, and no more than 6 lowercase letters.*", one could write:

```
minlength: 10; maxlength: 20; required:[a](3,3);
  required: upper(3,20); required: lower(1,6);
```

There are two named character classes which render this property useless, `ascii-printable` and `unicode`, because every character is permitted. With these two character classes, the only range restrictions should be `minlength` and `maxlength`, i.e., the only range that these two classes will respect are established by the `minlength` and `maxlength` rules. It is possible to mix the `ascii-printable` or `unicode` values and character classes with range, though. The policy "*Password must contain at least 8 characters and contain exactly 3 lowercase vowels*" can be expressed by:

```
minlength: 8; required: [aeiou](3,3); allowed: ascii-printable;
```

[5]Suggestion here: `https://github.com/apple/password-manager-resources/issues/427`

These are the core changes of our extension to Apple's Password Autofill Rules. We believe they provide extra flexibility to describe even more password policies. In doing so, we minimize the generation of non-compliant passwords using password managers. Hence, we can increase usability and, hopefully, user adoption of password managers.

3.3 The npm package

In order to implement our extension to Apple's DSL, we based our code on their parser[6], which is written in plain JavaScript. This parser receives an input that contains the password rules and parses them into an array of rules. As an example, for the rules

```
required: upper; allowed: upper; allowed: lower; minlength: 12; maxlength: 73;
```

the parser will return an array containing 4 rules, each with a correspondent name and value, as we can see in Figure 3.4.

[6] Apple's JavaScript parser. https://github.com/apple/password-manager-resources/blob/main/tools/PasswordRulesParser.js

```
[
  {
    "_name": "required",
    "value": [
      {
        "_name": "upper"
      }
    ]
  },
  {
    "_name": "allowed",
    "value": [
      {
        "_name": "upper"
      },
      {
        "_name": "lower"
      }
    ]
  },
  {
    "_name": "minlength",
    "value": 12
  },
  {
    "_name": "maxlength",
    "value": 73
  }
]
```

Figure 3.4: Return value of the parser when given the rules `required: upper; allowed: upper; allowed: lower; minlength: 12; maxlength: 73;` as an input.

We wanted to make the process of using this parser as simple as possible, i.e., one simple install command and we could use its inherent functionalities. So, we opted to create an npm package. According to their website, npm is "*a public collection of packages of open-source code for Node.js, front-end web apps, mobile apps, robots, routers, and countless other needs of the JavaScript community*".

To create this package, we migrated Apple's parser code into Typescript[7] and implemented our extension to the parser.

Blocklist Rule When the input rules contain a `blocklist` rule, there are two possibilities:

- The rule value is `hibp`. In this case, the value is returned without changes, to let the password manager know that a check against HIBP's API must be done.
- The rule value is `default`. This value will make the parser return the list of the 100 000 most commonly used passwords [47]. Now the password manager should verify that the generated password does not contain any of these passwords.

The password blocklist, in our extension, is a *Singleton*. This means that there is only one point of access to it, and there is only one instance of this blocklist. Thus, it is possible to substitute the blocklist by another at will, by assigning a new list to it, e.g.:

```
let blist = PasswordBlocklist.getInstance();
blist.blocklist = ['123', 'password'];
// now, the blocklist only contains the passwords '123' and 'password'
```

There is also a method in our extension, `appendToTheBlocklist(newPasswords: string[])`, that grants the possibility to expand the blocklist.

Minclasses Rule The `minclasses` rule is always present, even when it is unused. Its default value is 1, i.e., every password must contain at least one character class. For instance, a policy such as "*Password must contain at least 8 characters and at least 1 uppercase letter*" can be described as:

```
minlength: 8; required: upper; allowed: ascii-printable;
```

Upon giving this set of rules to the parser, the result will include a `minclasses` rule, with its default value, as we can see in Figure 3.5.

The possible values for the `minclasses` rule are the integers 1 through 4. If the rule has a value that is lower than 1 or greater than 4, the parser will set it to 1 and 4, respectively.

[7]TypeScript is a strongly typed programming language which builds on JavaScript and provides better tooling at any scale. https://www.typescriptlang.org/

```
[
  {
    "_name": "required",
    "value": [
      {
        "_name": "upper"
      }
    ]
  },
  {
    "_name": "allowed",
    "value": [
      {
        "_name": "ascii-printable"
      },
    ]
  },
  {
    "_name": "minlength",
    "value": 8
  },
  {
    "_name": "minclasses",
    "value": 1
  }
]
```

Figure 3.5: Return value of the parser when given the rules `minlength: 8; required: upper; allowed: ascii-printable;` as an input. The `minclasses` rule is always included, with its default value, 1.

Character Range The character range is an enhancement to the definition of character classes. It is possible to mix character classes containing range restrictions with character classes that do not contain these restrictions. These are the restrictions to using this functionality:

- The `minimum` and `maximum` values should be greater than or equal to 0.
- The `minimum` value will be converted to 1 if the value is 0 and is specified in a `required` rule.
- The `minimum` value will be converted to 0 if the value is greater than 1 and is specified in an `allowed` rule.
- The `minimum` value should be less than or equal to the maximum.
  - The `minimum` and `maximum` values can the same — this means that the character class **should have exactly** that number of occurrences.
- This functionality must be combined with, at least, the `minlength` rule.

To maintain coherence and the correct functionality of this extension, there are some edge cases where the ranges will be discarded. These are those cases:

- The `minlength` rule is not present.
- The sum of all `required` rules' `maximum` values is less than the `minlength` value.
  - If a `required` rule does not have a range, its `maximum` value will be considered as an integer greater than 100, forcing this sum to be greater than the `minlength` value. A requirement for a password to be at least 100 characters long is practically unreal.
- The sum of all `required` rules' `minimum` values is greater than the `maxlength` value — if `maxlength` is specified.
- The `minimum` and `maximum` values are both 0.
- The range is used with values `ascii-printable` or `unicode`.

Figure 3.6 shows the results of parsing the following policy:

```
minlength: 10; maxlength: 20; required:[a](3,3);
  required: upper(3,20); required: lower(1,6);
```

Our package has been published in the npm official repository, under the name `pwrules-annotations` [48]. We intend to propose our changes to Apple and hopefully have our work integrated with the official repository. At the time of writing, we have not yet issued a pull request.

3.4 Chapter Overview

In this chapter, we analyzed our extension to Apple's DSL, i.e., Passcert's DSL. With this extension, we created two new rules, `blocklist` and `minclasses`, as well as a new option for character classes — character range. The `blocklist` rule is used to ensure that once the password is generated, it will be checked against a list of forbidden passwords: if a substring of the generated password is found in this list, then the password should be regenerated. The `minclasses` rule specifies the minimum number of character classes that must be present in the password. The character range allows for greater flexibility to specify the minimum and maximum number of times a character or character class should appear in the password. We also delved deeper into the details of Passcert's npm package.

```
[
  {
    "_name": "required",
    "value": [
      {
        "_characters": ["a"],
        "minChars": "3",
        "maxChars": "3"
      }
    ]
  },
  {
    "_name": "required",
    "value": [
      {
        "_name": "upper",
        "minChars": "3",
        "maxChars": "20"
      }
    ]
  },
  {
    "_name": "required",
    "value": [
      {
        "_name": "lower",
        "minChars": "1",
        "maxChars": "6"
      }
    ]
  },
  {
    "_name": "allowed",
    "value": [
      {
        "_name": "upper"
      },
      {
        "_name": "lower"
      }
    ]
  },
  ...
]
// omitting the minlength, maxlength and minclasses rules, because they are
    trivial and we have seen them previously.
```

Figure 3.6: Return value of the parser when given the rules `minlength: 10; maxlength: 20; required:[a](3,3); required: upper(3,20); required: lower(1,6);` as an input.

4

SmartPasswords: Integrating Apple's Password Autofill Rules with Bitwarden

Contents

In this chapter, we describe a prototype that incorporates our npm package and thus takes advantage of the extension to Apple's DSL to describe password policies. This feature offers great potential in terms of usability. Some websites implement strict password policies (e.g., Fnac Portugal has a policy of "*Minimum of 8 characters; At least one lowercase letter, one uppercase letter, one digit and one special* ") , which is not the best practice as Tan et al.'s work suggests [19] — "*Although prior work has repeatedly found that requiring more character classes decreases guessability, researchers have shown that character-class requirements lead to frustration and difficulty for users. Since other requirements, e.g., minimum length or blocklist requirements, can strengthen passwords with less negative impact on usability research has advocated retiring character-class requirements. These recommendations have been standardized in recent NIST password-policy guidance.*". For users that already take the extra effort of using a password manager and use its randomly generated passwords, such policies may cause discontent and frustration towards password managers because a generated password may not comply with them [12]. This hinders usability, and there is a considerable chance that the user will resort to password reuse or create an easily guessed password to overcome this obstacle, both undesirable behaviors.

4.1 Password Manager Choice

Nowadays, there are multiple password managers, all of them with great compatibility between Operating Systems (OS's) and devices. To the best of our knowledge, 1Password is the only password manager that does what we aim to achieve [49]. They use Apple's DSL and the website quirks[1] found in Apple's repository [50] containing the password policies of websites specified in Apple's DSL. These quirks are crowd-sourced since everyone can contribute to them.

There were two main candidates to become our prototype, Google Chrome's built-in password manager and Bitwarden [20]. Although Chrome is the browser with the most users, according to a recent study[2], due to being a browser extension, and it presented more clarity in its code, and the fact that this project was developed in Passcert's context — which adopted Bitwarden as its base password manager — Bitwarden felt more suitable to develop our work.

4.2 The Prototype

We started by investigating Bitwarden's browser extension. Browser extensions communicate using messages between content scripts and the rest of the extension. Content scripts are files that run in the

[1]"Quirk" is a term from web browser development that refers to a website-specific, hard-coded behavior to work around an issue with a website that can not be fixed in a principled, universal way.

[2]https://gs.statcounter.com/browser-market-share

context of web pages. These scripts use the Domain Object Model (DOM)[3] so they are able to read and change details of webpages. They can also pass information to their parent extension.

Taking advantage of this, we created a content script that accesses the DOM and searches for the `passwordrules` annotation, which Apple's DSL uses to start describing the password policy rules. Our extension also uses this annotation. Upon finding the annotation, the content script sends a message to its parent extension containing the password rules. If no annotation is found, a message is also sent, with a special value — `no-rules`. This can be seen in Figure 4.1. We assume that in a given web page there is only one password policy described: having multiple password policies would be strange from a usability point of view and could result in a possible crash of this feature.

```
function searchDomForPasswordRules(): void {
    const policiesFound = document.querySelectorAll('[passwordrules]');

    if (policiesFound.length > 0) {
        chrome.runtime.sendMessage({
            command: 'bgWebsitePasswordRules',
            rulesValue: policiesFound[0].attributes
                            .getNamedItem('passwordrules').value,
            sender: 'websitePasswordRules',
        });
    }
    else if (policiesFound.length === 0) {
        chrome.runtime.sendMessage({
            command: 'bgWebsitePasswordRules',
            rulesValue: 'no-rules',
            sender: 'websitePasswordRules',
        });
    }
}
```

Figure 4.1: The content script that searches the DOM for the `passwordrules` annotation.

Having received a message with the rules, the extension now utilizes our npm package to parse them. The core of the extension will then use these parsed rules to convert them to a format that Bitwarden recognizes, allowing it to generate a compliant password.

The general workflow of the generation of passwords can be seen in Figure 4.2.

Because we believe our work to be important in fixing a usability problem and because our solution, whilst not innovative in contents, brings value to a real-world application, we submitted our changes to Bitwarden [51]. Bitwarden has internally approved our features and will be going through the code review process to get them ready to be merged into the product. This is a major accomplishment since there will be *millions* of users benefiting from our improvements.

[3]https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

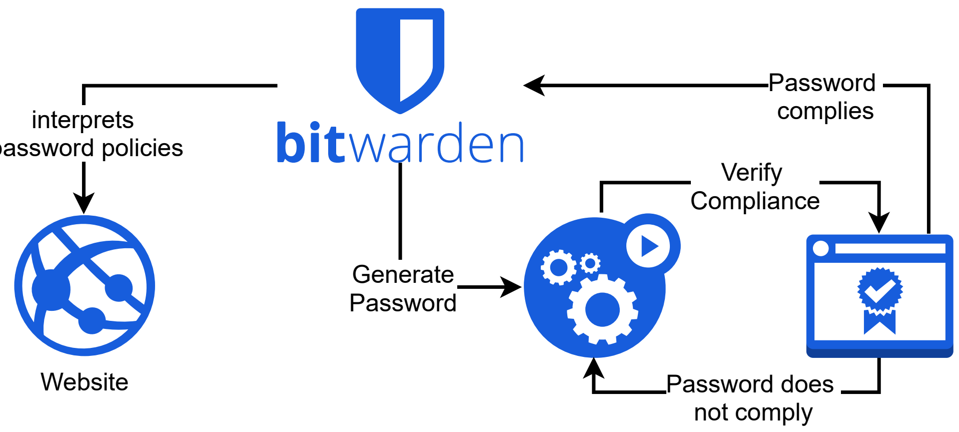

Figure 4.2: The general workflow of the generation of passwords in Bitwarden, using our npm package.

4.2.1 Generation and Compliance

Aside from the checks that were already being made by Bitwarden's extension — password length, permitted characters, etc. — we created three new compliance checks, each one correlated with each new feature introduced in our npm package. Thus, after a password is generated, we verify if it is compliant with the `blocklist` rule, the `minclasses` rule and the character range requirement.

Blocklist To verify the compliance of the password with the `blocklist` rule, we verify if it contains any word inside the blocklist. So if we have a blocklist containing 3 leaked passwords, e.g., `password`, `1234`, and `helloworld`, we check for the occurrence of each one of these words in any part of the generated password. For example, the password `9PHeyEGBg.*aP3` does not contain any of the words in the blocklist, but the password `9PHeyEGBg.*aP31234` does. Consequently, another password would have to be generated and evaluated accordingly.

Minclasses To ensure compliance with the `minclasses` rule, we separate each letter of the password into its list, according to the character set it belongs to — uppercase, lowercase, digit, or special. Once the last character of the password is analyzed, we count, out of the 4 lists we created, how many have elements. If this number is less than the `minclasses` value, the password is not compliant and must be regenerated. To exemplify, imagine the `minclasses` rule value of 3 the password `9pheyegbg12ap3`: it has only lowercase letters and digits. As such, the password is invalid and will be regenerated.

Character Range Much like the `minclasses` verification, to ensure compliance with the character's range, we ascertain that the password contains at least the minimum required characters, according to their range. For instance, a character range such as `required: upper(3, 6);`, and omitting the other rules, would make the password `9PHeyEGBG.*aP3` invalid.

Our verification for password compliance incurs in a possible non-termination error, i.e., there is a possibility that the password generated will always be non-compliant with the policies. However, this risk is low, because the generator uses a Random Number Generator (RNG).

4.3 Bitwarden and Passcert's Generator

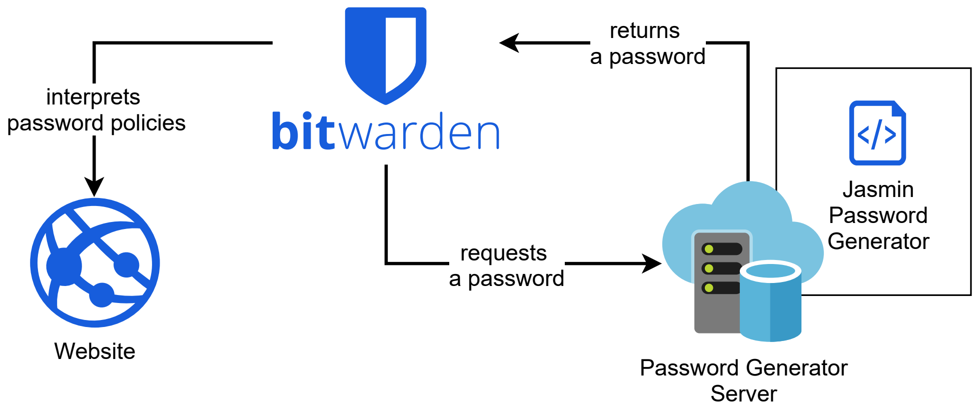

Figure 4.3: Overview of the prototype combining Bitwarden and Passcert's password generator.

In the context of Passcert's project, a formally verified password generator is being developed. To contribute to the goals of PassCert and to demonstrate that our development can be integrated with different products, we extended Passcert's password manager with our SmartPasswords feature. Passcert's password manager is an extension of Bitwarden and Passcert's generator is written in Jasmin [52] and follows an algorithm such that, when given a password composition policy, it will generate a random password. This generator has two verified properties:

- **Functional Correctness**: Given any password composition policy, the generated password will always satisfy the policy.
- **Security**: Given any password composition policy, the password is generated according to a uniform distribution. This means that every possible password that satisfies the given policy has the same probability of being generated.

An overview of our integration of Passcert's password generator with the Bitwarden browser extension is shown in Figure 4.3. Bitwarden scans the DOM for the policies and uses our npm package to parse them. We replaced Bitwarden's default password generator with Passcert's Jasmin password generator. Since in the context of the browser extension it is not possible to directly run local processes, we exposed our password generator as a RESTful service: the extension sends a POST request, with the body of the request containing the required password policy. The server then sends a response with the generated password. The connection between the browser extension and the server uses HTTPS.

4.4 Overview

In this chapter, we presented the two main candidates to become Passcert's password manager prototype, Google Chrome's built-in password manager and Bitwarden, which was the chosen one. We also explained the process that allows Bitwarden's browser extension to read the password policies from a website and then, using our npm package, parse these policies and generate a password that is compliant with them. There are three new constraint checks added to Bitwarden's generator to accomodate the rules created by Passcert's DSL — `blocklist`, `minclasses` and character range. Lastly, we detail Passcert's password manager prototype, which uses a verified generator to generate compliant passwords. This generator is exposed as a service.

5 Evaluation

Contents

In this chapter, we detail the evaluation methods used on our solution. Since we want to obtain passwords that satisfy certain rules, specified by each website, using both Apple's DSL and our extension, we propose to measure how many passwords fail to comply with such rules when using the password managers extended with SmartPasswords.

To ascertain the functional correctness — given any password policy, the generated password will always satisfy the policy — of every generated password, we created a script[1] to automate this process — `policy_compliance_check.py`. This script takes as input the path to the test folder as well as a policy, to which all the passwords will be compared. This policy should be specified using numbers only, exactly with the order that follows:

```
length minimumLowercase maximumLowercase minimumUppercase maximumUppercase
minimumNumbers maximumNumbers minimumSpecial maximumSpecial
```

There is also the possibility to activate the flags for the `blocklist` and `minclasses` rules:

```
--minclasses <minclassesValue> --blocklist
```

This script verifies that the password contains only characters pertaining to required or allowed character classes, as well as length constraints. It also has the ability to verify that the `blocklist` rule, the `minclasses`, and the character range constraints are being satisfied by the password.

The tool was first built to test our integration with Passcert's password generator, which generates a password with a given length, and not a *minimum length*. Because of this, the tool only ensures that the password has said length value. The same goes for the maximum value for each character class: it is always, at most, the same value as the length of the password, i.e., can take values between 0 and the length of the password. The usage of the tool is exemplified in Figure 5.1. So, to test the compliance of a list of passwords against a policy that *requires* at least one character from each character class and minimum length of 14, we would write `14 1 14 1 14 1 14 1 14`. To test against a policy that *requires* length of 14, a minimum of three character classes and the verification against a blocklist, we would write `14 0 14 0 14 0 14 0 14 --minclasses 3 --blocklist`.

This is the methodology we followed:

1. Choose a policy, preferably one that generates conflict with Bitwarden's default settings, since it is where the usability problem occurs.

2. Generate a total of 10000 passwords and distribute them across 10 files for greater detail: we can derive results from the whole lot of generated passwords or from a specific file. This generation is done using Bitwarden's current solution via their Command Line Interface (CLI) application , i.e., not including the SmartPasswords feature. We have a script that facilitates this process, called `generate_bw_passwords.py`.

[1] GitHub repository: https://github.com/passcert-project/pw_generator_server

```
# <the_policy_to_check_against> should take this format:

# length minimumLowercase maximumLowercase minimumUppercase maximumUppercase
    minimumNumbers maximumNumbers minimumSpecial maximumSpecial

# the maximum of each character class can be, at most, the same as the length

# the <value> should be an int between 1 and 4

./policy_compliance_check.py <path_to_the_folder_with_test_data>
    <the_policy_to_check_against> --minclasses <value> --blocklist
```

Figure 5.1: The usage of the compliance checker tool.

3. Run the policy compliance script we wrote, called `policy_compliance_check.py`, to find the number of non-compliant passwords.

4. Generate a total of 1000 passwords using our SmartPasswords feature, i.e., Bitwarden's generator and the ability to read Apple's DSL. This step is done manually, because, at the time of writting, we were not able to include our SmartPassword feature in Bitwarden's CLI app. Thus, we need to open our version of Bitwarden's browser extension, where our feature is implemented, and manually copy each SmartPassword generated, which is time-consuming. Hence the lower number of generated passwords.

5. Run, again, our policy compliance script, regarding the same policy as before.

6. Compare both results of the compliance check to take conclusions of how SmartPasswords compare with regular Bitwarden's passwords.

All these scripts and test data can be found in one of our GitHub repositories[2].

5.1 Evaluating Apple's DSL Integration with Bitwarden

To test the need for our solution and the impact it can have, we generated 10 test files, each containing 1000 randomly generated passwords using Bitwarden's generator default settings — 14-character password with lowercase, uppercase, and numbers. We used the following policy, which is used by British government services, according to Apple's quirks [46]:

`minlength: 10; required: lower; required: upper; required: digit; required: special;`

We checked if the passwords generated by Bitwarden satisfy this policy, using a policy that includes at least one character of each character class, `14 1 14 1 14 1 14 1 14`. All passwords failed this test

[2]See footnote 1.

since Bitwarden's default settings do not include symbols. Granted, to solve this, a simple tick in a checkbox on the User Interface is enough to include special characters in the password generation. But even with special characters included, there are some problems. According to the same source [46], Virgin Mobile's[3] website has this policy:

```
minlength: 8; required: lower; required: upper; required: digit; required: [!#$@];
```

We generated again a total of 10000 passwords, distributed by 10 files, using Bitwarden's generator and, this time, including symbols.

Since our compliance verification tool checks the `special` characters by comparing them with Bitwarden's special characters set — `[!@#$%^&*]` —, we had to change this set on our tool, so that the tool would check only the website's required special symbols — `[!#$@]`. In other words, after this change in the `special` character set, a password is compliant if and only if it contains one of the four symbols required (and, of course, follows the other rules as well!).

The results obtained confirm our suspicions that this policy would present challenges to Bitwarden: 2671 passwords — 26,71% — failed. This means that, roughly, one in every four passwords generated by Bitwarden would not be accepted by this website. We tested again with `14 1 14 1 14 1 14 1 14`.

This is an instance of the problem discussed above, regarding users' frustration with the generation of non-compliant passwords and it can easily be solved using our solution.

Having confirmed the problem, we generated 1000 passwords, distributed across 10 files, using our SmartPasswords and we got the expected result: 100% compliance with both the policies seen previously. We only generated 1000 passwords because this is a manual, time-consuming process. In a close future, we hope to make this generation easier, by using an adequate generation script.

5.2 Evaluating Passcert's DSL Integration with Bitwarden

Having justified the need of our SmartPassword solution with a couple of tests, we now aim to analyse what kind of policies our extension to Apple's DSL supports. The main objective is to ensure that all passwords generated are compliant with the specified policy, i.e., how effective our solution is.

We created a new, different version of Bitwarden's browser extension that supports SmartPasswords, but with our DSL instead of Apple's. This version is able to interpret the `blocklist` and `minclasses` rules, as well as the character range feature.

We followed a similar methodology as before, with a few changes:

1. Choose a policy that tests our solution.

[3]https://virginmobile.ca

2. Generate a total of 100 passwords using our SmartPasswords feature, i.e., Bitwarden's generator and the ability to read Passcert's DSL. This step is done manually, because, at the time of writting, we were not able to include our SmartPassword feature in Bitwarden's CLI app. Thus, we need to open our version of Bitwarden's browser extension, where our feature is implemented, and manually copy each SmartPassword generated, which is time-consuming. Hence the lower number of generated passwords.

3. Run our policy compliance script, regarding the same policy as before.

4. Assess the results and verify the level of effectiveness of our solution.

We tested four policies, as they appeared to be a fair representation of the new features we introduced. The policies chosen were:

1. `minlength: 8; allowed: ascii-printable; minclasses: 3; blocklist:default;`

2. `minlength: 10; required: upper(4, 10); required: lower(4, 6); required: digit(4, 8); required: special (4, 10);`

3. `minlength: 14; required: lower(5, 10); required: digit(5, 10); allowed: upper(0, 4), special; minclasses: 3;`

4. `minlength: 14; required: lower(5, 10); required: digit(5, 10); allowed: upper(0, 4), special; minclasses: 3; blocklist:default;`

5. `minlength: 14; required: lower(5, 10), digit(5, 10); allowed: upper, special;`

Policy 1 This is the description of a classic password rule throughout academic studies [18, 19], `3c8`, "*a password that must be at least 8 characters long and must contain at least 3 character classes*".

We used our tool to verify if all 100 passwords were compliant against the policy `8 0 8 0 8 0 8 0 8 --minclasses 3` and only 94 were, leaving 6 generated passwords to be non-compliant. This happens because Bitwarden has a default value for password length, 14. If the `minlength` is lower than 14, it will be changed to 14. Thus, the generator was working with 14 as the maximum capacity for each character class. All 6 non-compliant passwords failed because they had more than 8 characters of one class. When we tested again, but using the policy `14 0 14 0 14 0 14 0 14 --minclasses 3`, all passwords were compliant.

Policy 2 This policy requires that the password has at least 4 characters of each character class. It lets us test the range property we introduced. We tested this batch of passwords with `16 4 6 4 10 4 8 4 10`, and we got 100% compliance. The `minlength` of this password will always be 16 in Bitwarden's generator due to the restrictions for each character class. Thus, we check that the password has to have 16 characters.

Policy 3 In this case, we use ranges for the `allowed` rule as well. This means that the password can have special characters and, at most, 4 uppercase characters. It also must have at least 5 lowercase characters and at least 5 digits. At least 3 character classes must be present in the password. We checked these rules against `14 5 10 0 4 5 10 0 14 --minclasses 3` and got, as expected, 100% compliance. However, we also tested against the same policy, adding the `--blocklist` verification, and we got 91% compliance. This means that 9 passwords had some substring that was found in a previous password leak. These substrings were all composed of 4 digits together, e.g., `2yfp0t31d8995G` failed due to the substring `8995` since 4 digits together are usually a PIN number, and PIN numbers are weak passwords.

Policy 4 With this policy, we aim to test ranges, `minclasses` and the `blocklist`. This policy is built from **Policy 3** with the addition of the `blocklist`. We tested against the same two policies as **Policy 3** and got 100% compliance in both of them.

Policy 5 This policy allows to test disjunctive rules. So, these constraints force the password to contain either 5 lowercase characters or 5 digits, at least, but not both. The password can also have special characters and uppercase characters. To test this, we had to do three tests: (1) test if the passwords had both digits and lowercase letters; (2) test if passwords were containing just lowercase letters and not digits; (3) test if passwords were containing only digits and no lowercase letters. Thus we tested (1) with `14 5 10 0 14 5 10 0 14` and got 0% compliance as expected: no password contains both digits and lowercase letters. After, we tested (2) with `14 5 10 0 14 0 0 0 14` and got 51% compliance: 51 passwords contain lowercase letters and no digits. Lastly, we tested (3) with `14 0 0 0 14 5 10 0 14` and got 49% of compliance, as expected: the rest of the passwords do not contain lowercase letters and contain digits.

Testing non-termination. As a last test to address the possible non-termination of the password generation, mentioned in Chapter 4, we generated, manually, 1000 passwords with the following policy:

```
minlength: 16; blocklist: default; minclasses: 1;
```

This policy allows every `ascii-printable` character and restricts the password to, at least, have 16 characters and checks its substrings against a blocklist. This is suggested as a good behaviour by Tan et al.: "*Since other requirements, e.g., minimumlength or blocklist requirements, can strengthen passwords with less negative impact on usability research has advocated retiring character-class requirements.*".

Our results, 100% compliance with the policy `16 0 16 0 16 0 16 0 16 --blocklist`, help to demonstrate that the non-termination scenario is indeed rare, as we had suggested. The chances of getting this non-termination problem increase when greater restrictions are inserted in the policy: more restrictions like character ranges or removing permitted characters imply less margin for randomness, and, as we have seen, this is not advisable.

5.3 Overview

In this chapter, we explained the details of the testing phase of our project. We began by explaining the somewhat complex nature of the input for the script that verifies the compliance for each generated password. We also went through the details of the testing methodology used. We proceeded with the evaluation of Apple's DSL integration with Bitwarden which yielded very positive results and followed with the evaluation of Passcert's DSL and its integration with Bitwarden. Lastly, we reinforced that the non-termination issue that can arise with our adapted algorithm is rare, although more restricting policies can augment this probability.

6

Conclusion

Our work aimed to review the current state-of-the-art regarding password managers and password composition policies and verify if researcher's recommendations for passwords are being incorporated in them. Through our analysis, we found multiple languages capable of expressing password policies and we found that one of these, Apple's DSL, was more suited for our investigation.

Based on our experiments integrating Apple's DSL with Bitwarden's browser extension, there is a great benefit in websites using this language to express their password policies to password managers: we achieved great results in generating compliant passwords. Our solution was **accepted internally by Bitwarden** and is now going through their code review process in order to be merged into the final product. This will impact *millions* of users.

To accommodate recent researcher's insight [18, 19, 45] on password policies, we expanded Apple's DSL with 3 new features — `minclasses`, `blocklist` and character ranges — and created a prototype with Bitwarden, which also yielded great success rates.

Lastly, our work allowed the creation of a prototype of a password manager with a formally verified password generator, which is Passcert's main goal.

The most difficult part of the project was integrating our new rules into Bitwarden's password generator. In fact, it is still incomplete, since there is the non-termination problem present.

Future Work

While our work is a concrete solution to a common problem, it can still be enriched. Apple's DSL can be further expanded with rules like `max-frequency` or `exclude`.

The `max-frequency` is currently an open issue in Apple's Github[1] and would restrict the frequency of any character to this value, e.g., for `max-frequency: 3;`, the password `HelloWorld` would be compliant, but the password `HellloWorlld` would not — notice that the frequency of the `l` character is greater than 3. This suggestion may be resolved with the character range that we proposed in Chapter 3. To achieve this, we could specify a rule like `allowed: [l](3,3);`. In spite of this, such a rule may be relevant: to restrict the frequency of every character with our character range would be a great burden for the administrator of the website.

The `exclude` would exclude a set of custom characters, e.g. `exclude:[aeiouAEIOU]` would exclude all vowels from the password.

The tool that verifies password compliance is not equipped to test custom character classes, e.g., `[aeiou]`. This would allow us to test even more combinations and assert if the passwords generated are effectively compliant with the policies restraining them.

There is also room for improvement regarding the password generation in Bitwarden, when Smart-

[1] Apple's Github Issue: https://github.com/apple/password-manager-resources/issues/387

Passwords are live in production, to completely eliminate the chance of non-termination of the generation algorithm.

Lastly, our feature needs to be able to read from Apple's quirks [46] to be effective. However, our improvements must be met halfway by webadmins, who should strive to include password composition policies in their websites.

Bibliography

[1] D. McCarney, D. Barrera, J. Clark, S. Chiasson, and P. C. Van Oorschot, “Tapas: design, implementation, and usability evaluation of a password manager,” in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 89–98.

[2] F. Stajano, M. Spencer, G. Jenkinson, and Q. Stafford-Fraser, “Password-manager friendly (pmf): Semantic annotations to improve the effectiveness of password managers,” in *International Conference on Passwords*. Springer, 2014, pp. 61–73.

[3] PMFriendly, “PMFriendly’s Github,” https://github.com/pmfriendly/pmf-specification/blob/master/specification.md, 2015, [Online; accessed 07-January-2021].

[4] D. Florencio and C. Herley, “A large-scale study of web password habits,” in *Proceedings of the 16th international conference on World Wide Web*, 2007, pp. 657–666.

[5] S. Gaw and E. W. Felten, “Password management strategies for online accounts,” in *Proceedings of the second symposium on Usable privacy and security*, 2006, pp. 44–55.

[6] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, “The tangled web of password reuse.” in *NDSS*, vol. 14, no. 2014, 2014, pp. 23–26.

[7] S. Pearman, J. Thomas, P. E. Naeini, H. Habib, L. Bauer, N. Christin, L. F. Cranor, S. Egelman, and A. Forget, “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*, 2017, pp. 295–310.

[8] E. Stobert and R. Biddle, “The password life cycle: user behaviour in managing passwords,” in *10th Symposium On Usable Privacy and Security ({SOUPS} 2014)*, 2014, pp. 243–255.

[9] S. Pearman, S. A. Zhang, L. Bauer, N. Christin, and L. F. Cranor, “Why people (don’t) use password managers effectively,” in *Fifteenth Symposium On Usable Privacy and Security (SOUPS 2019). USENIX Association, Santa Clara, CA*, 2019, pp. 319–338.

[10] Apple, “Web sites won’t accept Safari generated strong passwords due to dashes or other criteria,” https://discussions.apple.com/thread/251341081, 2021, [Online; accessed 26-October-2021].

[11] TechNet, “Can’t create local user ”Password does not meet password policy requirements” - but it does,” https://web.archive.org/web/20211026082725/https://social.technet.microsoft.com/Forums/en-US/12b06881-ea1a-403d-aafb-99bbe7d4d1b0/cant-create-local-user-quotpassword-does-not-meet-password-policy-requirementsquot-but-it?forum=win10itprosecurity, 2021, [Online; accessed 26-October-2021; archived 26-October-2021].

[12] S. Chiasson, P. C. van Oorschot, and R. Biddle, “A usability study and critique of two password managers.” in *USENIX Security Symposium*, vol. 15, 2006, pp. 1–16.

[13] EA, “Password Does Not Meet Requirements,” https://web.archive.org/web/20210817105229/https://answers.ea.com/t5/EA-General-Questions/quot-Password-Does-Not-Meet-Requirements-quot/td-p/5744758, 2021, [Online; accessed 26-October-2021; archived 26-October-2021].

[14] M. Horsch, M. Schlipf, S. Haas, J. Braun, and J. Buchmann, “Password policy markup language,” 2016.

[15] S. Oesch and S. Ruoti, “That was then, this is now: A security evaluation of password generation, storage, and autofill in browser-based password managers.” in *USENIX Security Symposium*, 2020.

[16] Apple, “Customizing Password AutoFill Rules,” https://developer.apple.com/documentation/security/password_autofill/customizing_password_autofill_rules, 2021, [Online; accessed 12-October-2021].

[17] Google, “Password Requirements Proto ,” https://chromium.googlesource.com/chromium/src/+/refs/heads/main/components/autofill/core/browser/proto/password_requirements.proto, 2021, [Online; accessed 08-October-2021].

[18] R. Shay, S. Komanduri, A. L. Durity, P. Huh, M. L. Mazurek, S. M. Segreti, B. Ur, L. Bauer, N. Christin, and L. F. Cranor, “Designing password policies for strength and usability,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 4, pp. 1–34, 2016.

[19] J. Tan, L. Bauer, N. Christin, and L. F. Cranor, “Practical recommendations for stronger, more usable passwords combining minimum-strength, minimum-length, and blocklist requirements,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1407–1426.

[20] Bitwarden, “Bitwarden Home Page,” https://bitwarden.com/, 2021, [Online; accessed 08-October-2021].

[21] Passcert, “Passcert Homepage,” https://passcert-project.github.io/, 2021, [Online; accessed 15-October-2021].

[22] M. Grilo, J. Campos, J. F. Ferreira, J. B. Almeida, and A. Mendes, “Verified password generation from password composition policies,” 2021, Submitted for publication. Draft available from authors.

[23] J. Bonneau, C. Herley, P. C. Van Oorschot, and F. Stajano, “The quest to replace passwords: A framework for comparative evaluation of web authentication schemes,” in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 553–567.

[24] B. Ives, K. R. Walsh, and H. Schneider, “The domino effect of password reuse,” *Communications of the ACM*, vol. 47, no. 4, pp. 75–78, 2004.

[25] TNW, “The Next Web - 2021 Digital Trends,” https://thenextweb.com/news/insights-global-state-of-digital-social-media-2021, 2021, [Online; accessed 10-October-2021].

[26] R. Wash, E. Rader, R. Berman, and Z. Wellmer, “Understanding password choices: How frequently entered passwords are re-used across websites,” in *Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016)*, 2016, pp. 175–188.

[27] C. Herley, “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*, 2009, pp. 133–144.

[28] R. Dhamija and J. D. Tygar, “The battle against phishing: Dynamic security skins,” in *Proceedings of the 2005 symposium on Usable privacy and security*, 2005, pp. 77–88.

[29] T. D. Wu *et al.*, “The secure remote password protocol.” in *NDSS*, vol. 98. Citeseer, 1998, pp. 97–111.

[30] M. Wu, R. C. Miller, and G. Little, “Web wallet: preventing phishing attacks by revealing user intentions,” in *Proceedings of the second symposium on Usable privacy and security*, 2006, pp. 102–113.

[31] H.-M. Sun, Y.-H. Chen, and Y.-H. Lin, “opass: A user authentication protocol resistant to password stealing and password reuse attacks,” *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 2, pp. 651–663, 2011.

[32] C. Yue, “Preventing the revealing of online passwords to inappropriate websites with logininspector,” in *Presented as part of the 26th Large Installation System Administration Conference ({LISA} 12)*, 2012, pp. 67–81.

[33] P. Mayer, H. Berket, and M. Volkamer, “Enabling automatic password change in password managers through crowdsourcing,” *Proc. PASSWORDS. Springer*, 2016.

[34] P. Gasti and K. B. Rasmussen, “On the security of password manager database formats,” in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 770–787.

[35] Z. Li, W. He, D. Akhawe, and D. Song, “The emperor’s new password manager: Security analysis of web-based password managers,” in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 465–479. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/li_zhiwei

[36] D. Silver, S. Jana, D. Boneh, E. Chen, and C. Jackson, “Password managers: Attacks and defenses,” in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 449–464.

[37] B. Stock and M. Johns, “Protecting users against xss-based password manager abuse,” in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 183–194.

[38] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell, “Stronger password authentication using browser extensions.” in *USENIX Security Symposium*. Baltimore, MD, USA, 2005, pp. 17–32.

[39] J. A. Halderman, B. Waters, and E. W. Felten, “A convenient method for securely managing passwords,” in *Proceedings of the 14th international conference on World Wide Web*, 2005, pp. 471–479.

[40] J. Kelsey, B. Schneier, C. Hall, and D. Wagner, “Secure applications of low-entropy keys,” in *International Workshop on Information Security*. Springer, 1997, pp. 121–134.

[41] K.-P. Yee and K. Sitaker, “Passpet: convenient password management and phishing protection,” in *Proceedings of the second symposium on Usable privacy and security*, 2006, pp. 32–43.

[42] R. Gonzalez, E. Y. Chen, and C. Jackson, “Automated password extraction attack on modern password managers,” *arXiv preprint arXiv:1309.1416*, 2013.

[43] J. Bonneau and S. Preibusch, “The password thicket: Technical and market failures in human authentication on the web.” in *WEIS*, 2010.

[44] Apple, “Password Rules Validation Tool,” https://developer.apple.com/password-rules/, 2021, [Online; accessed 12-October-2021].

[45] S. Johnson, J. F. Ferreira, A. Mendes, and J. Cordry, “Skeptic: Automatic, justified and privacy-preserving password composition policy selection,” in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 101–115.

[46] Apple, “Password Quirks,” https://github.com/apple/password-manager-resources/blob/main/quirks/password-rules.json, 2021, [Online; accessed 13-October-2021].

[47] D. Miessler, J. Haddix, and g0tmi1k, “SecLists,” https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-100000.txt, 2021, [Online; accessed 13-October-2021].

[48] Passcert, “pwrules-annotations,” https://www.npmjs.com/package/@passcert/pwrules-annotations, 2021, [Online; accessed 13-October-2021].

[49] 1Password, “1Password Smart Passwords,” https://blog.1password.com/a-smarter-password-generator/, 2021, [Online; accessed 15-October-2021].

[50] Apple, “Password Manager Resources,” https://github.com/apple/password-manager-resources, 2021, [Online; accessed 15-October-2021].

[51] Passcert, “Passcert’s Pull Request,” https://github.com/bitwarden/browser/pull/2047, 2021, [Online; accessed 17-October-2021].

[52] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, “Jasmin: High-assurance and high-speed cryptography,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1807–1823.

A

Auxiliary Tables

Policy Annotation	Policy Description
comp8	At least: 1 lowercase English letter, 1 uppercase English letter, 1 digit and 1 symbol - any character other than a digit or an English letter; together, the letters must not form a word on a dictionary
basic12	At least 12 characters
basic16	At least 16 characters
basic20	At least 20 characters
2class12	At least 2 character classes
3class12	At least 3 character classes
3class16	At least 3 character classes
2word12	At least 2 words (letter sequences separated by a nonletter sequence)
2word16	At least 2 words (letter sequences separated by a nonletter sequence)
2list12	Combines 2class12 with a **blacklist**: 123!, amazon, character, monkey, number, survey, this, turk; Any year between 1950 and 2049; The same character four or more times in a row; Any four consecutive characters from password; Any four sequential digits (e.g., 5678); Any four sequential letters in the alphabet (e.g., wxyz); Any four consecutive characters on the keyboard (e.g., wsxc)
2s-list12	Combines 2class12 with a **blacklist warning**: "*Do not include words commonly found in passwords (e.g. password), keyboard patterns (e.g., qazx), or other common patterns (e.g. 5678).*"
2pattern12	Combines 2class12 with the **pattern requirement**: "*Passwords should start and end with a lowercase letter*"
2list-patt12	Combines the 2class12 with the **pattern requirement and the blacklist**
2s-list-patt12	Combines the 2class12 with the **pattern requirements and the blacklist warning**

Table A.1: Policies described in Shay et al.'s work [18].

Policy Annotation	Policy Description
1c8	At least 1 character class. At least 8 characters
1c10	At least 1 character class. At least 10 characters
1c12	At least 1 character class. At least 12 characters
1c16	At least 1 character class. At least 16 character
3c8	At least 3 character classes. At least 8 characters
3c12	At least 3 character classes. At least 12 characters
4c8	4 character classes. At least 8 characters

Table A.2: Policies described in Tan et al.'s work [19]. The authors added extra constraints

Policy Annotation	Apple's DSL	Passcert's DSL
1c8	minlength: 8; allowed: ascii-printable;	minlength: 8; allowed: ascii-printable;
1c10	minlength: 10; allowed: ascii-printable;	minlength:10; allowed: ascii-printable;
1c12	minlength: 12; allowed: ascii-printable;	minlength: 12; allowed: ascii-printable;
1c16	minlength: 16; allowed: ascii-printable;	minlength: 16; allowed: ascii-printable;
3c8		minlength: 8; allowed: ascii-printable; minclasses: 3;
3c12		minlength:12; allowed: ascii-printable; minclasses: 3;
4c8	minlength: 8; required: ascii-printable;	minlength: 8; required: ascii-printable; minclasses: 4;
comp8	minlength: 8; required: upper; required: lower; required: digit; required: special;	minlength: 8; required: upper; required: lower; required: digit; required: special;
basic12	minlength: 12; allowed: ascii-printable;	
basic16	minlength: 16; allowed: ascii-printable;	
basic20	minlength: 20; allowed: ascii-printable;	
2class12		minlength: 12; allowed: ascii-printable; minclasses: 2;
3class12		minlength: 12; allowed: ascii-printable; minclasses: 3;
3class16		minlength: 16; allowed: ascii-printable; minclasses: 3;
2word12		
2word16		
2list12		minlength: 12; allowed: ascii-printable; minclasses: 2; blocklist: default;
2s-list12		minlength: 12; allowed: ascii-printable; minclasses: 2; blocklist: default;
2pattern12		
2list-patt12		
2s-list-patt12		

Table A.3: Comparison between Apple's DSL and Passcert's Extension to it, regarding the ability to express policies from A.1 and A.2. We assume that `comp8` is mostly fulfilled by the specified rules because the randomness of the generator will make it difficult to form any words. To be extra careful, it could be described with the extra `blocklist` rule. We also assume that the words from the default blocklist contain all forbidden words used in Shay et al.'s work [18].