SmartPasswords: Increasing Password Managers' Usability by Generating Compliant Passwords

João Miguel Pereira Campos
joao.miguel.campos@tecnico.ulisboa.pt
Instituto Superior Técnico
Lisboa, Portugal

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.

The imposition of overly restrictive password policies poses 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

1 INTRODUCTION

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 [13, 15]. 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 [11, 13, 15, 29, 35].

Password managers are recommended [11, 30] to safely manage user credentials, but they still have some obstacles that prevent mass adoption by the users, e.g., the generated passwords are often not compliant [7, 39] with the password composition policies stipulated by the websites they use [10]. This leads to frustrated users and therefore possible cease of use of password managers. However, users are not to blame. As Stajano et al. [34] identified, this problem arises due to very restrictive password composition policies that services usually have [12].

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 HTML semantic labels [34] and Horsch et al. proposed the Password Policy Markup Language [19]. Oesch and Ruoti [25] 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 [3]. The idea is to add a specification to the HTML code, in the form of annotations. Google has also done something of this nature [16], in the form of an 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 [32, 38] 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 [8][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 [26] 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 [17]:

- *Miguel Grilo, **João Campos**, João F. Ferreira, José Bacelar Almeida, and Alexandra Mendes. Verified Password Generation from Password Composition Policies. Submitted to publication. 2021*

2 BACKGROUND WORK

In this section, 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. [9] 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. 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*" [9].

This equilibrium seems to be too strong to disrupt, whether because it would imply that service providers adapt their services to alternative authentication methods or simply because users probably would not take the change lightly, since they are too accustomed to passwords. Consequently, it appears to be a fair statement that passwords are here to stay.

Password reuse is evidently a major predicament regarding password security, which can be aggravated by XSS attacks, using weaknesses in websites to inject malicious code and grant the attacker the ability to steal sensitive data. Whenever an attacker gets access to the user's credentials, a domino effect [20] 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. [20] 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.*".

[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

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 [40]. To cope with this increasing number of accounts, users tend to reuse passwords across multiple websites. 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 a 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. [30] 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 [18] 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 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*" [18], which allows for a better understanding of why users ignore security policies.

2.3 Password Managers

Password managers are recommended by security researchers [11, 30]. 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. 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. 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 [25].

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

oPass, developed by Sun et al. [37] relieves 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, 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.

McCarney et al. [23] 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.

Recently, Oesch and Ruoti [25] revisited previous work [14, 22, 33, 36] done on password managers' security and usability. 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 HTTPS certificate or a significant amount of unencrypted metadata being stored, like a page 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 past work developed on the subject.

2.3.1 Password Generation. Password generation 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 [25] 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 containing letters, digits, and symbols, e.g., *d@rKn3s5* or *Tz5a5a5a*.

Ross et al. [31] 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.

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 [25] discovered that the vulnerabilities exposed by Gasti and Rasmussen [14], 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[3] 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 [33, 36].

With their work, Oesch and Ruoti [25] 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.

Both Stajano et al.'s [34] and Horch et al.'s [19] 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.

2.3.4 Password Rules Annotations. Stajano et al. [34] propose adding HTML semantic labels, or annotations, to facilitate and normalize the work done by password managers, by allowing them to read password policies declared by websites.

Horch et al. [19] put forward a *Password Policy Markup Language*, that allows websites to describe their policies regarding passwords. This language is exposed as a service.

2.3.5 Google's Password Requirements API. Google has also implemented a solution regarding this problem [16]. 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) [4]. As of February of 2021, this API includes password requirements for 237 websites[5].

2.3.6 Apple's Password Autofill Rules. Apple created the Password Autofill Rules [3]. These rules are described using an HTML annotation — `passwordrules` — that lets the webadmin 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.

3 EXTENDING APPLE'S PASSWORD AUTOFILL RULES

Initially we had planned to create a new DSL to accommodate suggestions made by Stajano et al.'s work [34] and reinforced by Oesch and Ruoti's research [25]. However, during the development phase of this project, we found that Apple had already made efforts in this direction [3], 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 [3] 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;
```

3.2 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

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

[4] Protobuf - https://github.com/protocolbuffers/protobuf/releases/tag/v3.19.0

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

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;
```

This rule will allow passwords like abcdefghi, aBCDEFGHI, a12345 67 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.

3.3 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 *-~!@#$%^&*_+=`|(){}[:;"'<>,.?]* 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;.

3.4 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 [10]. 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 [6]. However, it appears there are some incoherences, either with the official documentation or with the password generator itself.

For example, in the official documentation [3], 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 [6], 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 [32, 38]). 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.

3.5 The npm package

In order to implement our extension to Apple's DSL, we based our code on their parser[6], which 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.

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 [24]. 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.

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

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

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.

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.

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 the `minlength` rule, at least.

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`.

Our package has been published in the npm official repository, under the name `pwrules-annotations` [28]. 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.

4 SMARTPASSWORDS: INTEGRATING APPLE'S DSL WITH BITWARDEN

In this section, 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 [38] — "*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., minimumlength 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 [10]. 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 OS's and devices. To the best of our knowledge, 1Password is the only password manager that does what we aim to achieve [1]. They use Apple's DSL and the website quirks[8] found in Apple's repository [4] 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 [8]. Although Chrome is the browser with the most users, according to a recent study[9], 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 context of web pages. These scripts use the DOM[10] 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`. We assume that in a given web page there is only one password policy described: having

[8] "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.
[9] https://gs.statcounter.com/browser-market-share
[10] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

multiple password policies would be strange from a usability point of view and could result in a possible crash of this feature.

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.

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 [27]. 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.

4.3 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 RNG.

4.4 Bitwarden and Passcert's 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 [2] 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.

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.

5 EVALUATION

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[11] to automate this process — `policy_compliance_check.py`. It verifies if each password satisfies a given policy, by checking if each 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 policy that is used to verify the compliance of any password is specified by using nine numbers in a row, separated by a space. Each number represents, respectively, *the password length, the minimumLower, maximumLower, minimumUpper, maximumUpper, minimumNumbers, maximumNumbers, minimumSpecial, maximumSpecial*. 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`.

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

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 CLI application , i.e., not including the SmartPasswords feature. We have a script that facilitates this process, called `generate_bw_passwords.py`.
(3) Run the policy compliance script we wrote to find the number of non-compliant passwords, called `policy_compliance_check.py`.
(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[12].

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 [5]:

```
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 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 [5], Virgin Mobile's[13] website has this policy:

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

[12] See footnote 11.
[13] https://virginmobile.ca

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.
(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), digit(5, 10); allowed: upper, special;`

Policy 1. This is the description of a classic password rule throughout academic studies [32, 38], 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., `2yfpOt31d8995G` failed due to the substring `8995` since 4 digits together are usually a PIN number, and PIN numbers are weak passwords. This was resolved by generating passwords with the same policy and the addition of the `blocklist` rule.

Policy 4. 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, 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.

6 CONCLUSION

Our work aimed to review the current state-of-the-art regarding password managers, 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 [21, 32, 38] 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.

6.1 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[14] and would restrict the frequency of any character to this value.

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

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

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 SmartPasswords 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 [5] to be effective. However, our improvements must be met halfway by webadmins, who should strive to include password composition policies in their websites.

ACKNOWLEDGMENTS

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.

REFERENCES

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

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1807–1823.

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

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

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

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

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

[8] Bitwarden. 2021. Bitwarden Home Page. https://bitwarden.com/. [Online; accessed 08-October-2021].

[9] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. 2012. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 553–567.

[10] Sonia Chiasson, Paul C van Oorschot, and Robert Biddle. 2006. A Usability Study and Critique of Two Password Managers.. In *USENIX Security Symposium*, Vol. 15. 1–16.

[11] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. 2014. The tangled web of password reuse.. In *NDSS*, Vol. 14. 23–26.

[12] EA. 2021. 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. [Online; accessed 26-October-2021; archived 26-October-2021].

[13] Dinei Florencio and Cormac Herley. 2007. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web*. 657–666.

[14] Paolo Gasti and Kasper B Rasmussen. 2012. On the security of password manager database formats. In *European Symposium on Research in Computer Security*. Springer, 770–787.

[15] Shirley Gaw and Edward W Felten. 2006. Password management strategies for online accounts. In *Proceedings of the second symposium on Usable privacy and security*. 44–55.

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

[17] Miguel Grilo, João Campos, João F. Ferreira, José Bacelar Almeida, and Alexandra Mendes. 2021. Verified Password Generation from Password Composition Policies. Submitted for publication. Draft available from authors..

[18] Cormac Herley. 2009. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proceedings of the 2009 workshop on New security paradigms workshop*. 133–144.

[19] Moritz Horsch, Mario Schlipf, Stefan Haas, Johannes Braun, and Johannes Buchmann. 2016. Password Policy Markup Language. (2016).

[20] Blake Ives, Kenneth R Walsh, and Helmut Schneider. 2004. The domino effect of password reuse. *Commun. ACM* 47, 4 (2004), 75–78.

[21] Saul Johnson, João F Ferreira, Alexandra Mendes, and Julien Cordry. 2020. Skeptic: Automatic, justified and privacy-preserving password composition policy selection. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 101–115.

[22] Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. 2014. The Emperor's New Password Manager: Security Analysis of Web-based Password Managers. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 465–479. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/li_zhiwei

[23] Daniel McCarney, David Barrera, Jeremy Clark, Sonia Chiasson, and Paul C Van Oorschot. 2012. Tapas: design, implementation, and usability evaluation of a password manager. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 89–98.

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

[25] Sean Oesch and Scott Ruoti. 2020. That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Browser-Based Password Managers.. In *USENIX Security Symposium*.

[26] Passcert. 2021. Passcert Homepage. https://passcert-project.github.io/. [Online; accessed 15-October-2021].

[27] Passcert. 2021. Passcert's Pull Request. https://github.com/bitwarden/browser/pull/2047. [Online; accessed 17-October-2021].

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

[29] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. 2017. Let's go in for a closer look: Observing passwords in their natural habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 295–310.

[30] Sarah Pearman, Shikun Aerin Zhang, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. 2019. Why people (don't) use password managers effectively. In *Fifteenth Symposium On Usable Privacy and Security (SOUPS 2019). USENIX Association, Santa Clara, CA*. 319–338.

[31] Blake Ross, Collin Jackson, Nick Miyake, Dan Boneh, and John C Mitchell. 2005. Stronger Password Authentication Using Browser Extensions.. In *USENIX Security Symposium*. Baltimore, MD, USA, 17–32.

[32] Richard Shay, Saranga Komanduri, Adam L Durity, Phillip Huh, Michelle L Mazurek, Sean M Segreti, Blase Ur, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. 2016. Designing password policies for strength and usability. *ACM Transactions on Information and System Security (TISSEC)* 18, 4 (2016), 1–34.

[33] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. 2014. Password managers: Attacks and defenses. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 449–464.

[34] Frank Stajano, Max Spencer, Graeme Jenkinson, and Quentin Stafford-Fraser. 2014. Password-manager friendly (PMF): Semantic annotations to improve the effectiveness of password managers. In *International Conference on Passwords*. Springer, 61–73.

[35] Elizabeth Stobert and Robert Biddle. 2014. The password life cycle: user behaviour in managing passwords. In *10th Symposium On Usable Privacy and Security ({SOUPS} 2014)*. 243–255.

[36] Ben Stock and Martin Johns. 2014. Protecting users against XSS-based password manager abuse. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 183–194.

[37] Hung-Min Sun, Yao-Hsin Chen, and Yue-Hsun Lin. 2011. oPass: A user authentication protocol resistant to password stealing and password reuse attacks. *IEEE Transactions on Information Forensics and Security* 7, 2 (2011), 651–663.

[38] Joshua Tan, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. 2020. 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*. 1407–1426.

[39] TechNet. 2021. 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. [Online; accessed 26-October-2021; archived 26-October-2021].

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