Password-manager friendly (PMF): Semantic annotations to improve the effectiveness of password managers

Frank Stajano, Max Spencer, Graeme Jenkinson
{max.spencer, frank.stajano, graeme.jenkinson}@cl.cam.ac.uk

University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge, CB3 0FD, United Kingdom

Abstract. Subtle and sometimes baffling variations in the implementation of password-based authentication are widespread on the web. Despite being imperceptible to end users, such variations often require that password managers implement complex heuristics in order to act on the user's behalf. These heuristics are inherently brittle. As a result, password managers are unnecessarily complex and yet they still occasionally fail to work properly on some websites. In this paper we propose PMF, a simple set of semantic labels for password-related web forms. These semantic labels allow a software agent such as a password manager to extract meaning, such as which site the login form is for and what field in the form corresponds to the username. They also allow it to generate a strong password on the user's behalf. PMF reduces a password manager's dependency on complex heuristics, making its operation more effective and dependable and bringing usability and security advantages to users and website operators.

1 Introduction

We don't have to explain to this audience that, on the web, we are asked to remember way too many passwords. One reasonable way of coping with this burden is with a password manager—a piece of software that remembers passwords on the user's behalf and submits them automatically when required. All modern browsers such as Chrome, Firefox and Internet Explorer provide an integrated password manager. Because websites frequently have slight differences in the way they handle asking the user to type a password (or to define a new one), every password manager must implement complex heuristics in order to parse, auto-fill and submit the password-requesting web pages. Such code is inherently fragile and requires continuous maintenance as login web pages evolve and become fancier. As a result, some websites don't work seamlessly with password managers.

Password managers would be simpler and more dependable if websites adopted a set of semantic labels for HTML forms that allowed unambiguous registration and submission of passwords by programs acting on the user's behalf. In this

Pre-proceedings version. Do not circulate.
Final version to appear in *Proceedings of Passwords 2014*, Springer LNCS.

paper we offer two main contributions. First, we document the many ways in which websites ask for passwords and the many subtle ways in which the heuristics commonly employed by password managers can break, demonstrating how such code requires extensive maintenance to be reliable. Second, and most important, we propose PMF, a practical set of semantic labels that websites may immediately adopt. We also very briefly discuss incentives and benefits for the various parties involved.

2 Inconsistencies in password-based login on the web

Ignoring issues of style and presentation, password-based authentication on the web presents a fairly consistent interface to the user. To log in, users first find the login form, enter their username and password for that site into the appropriate boxes, and then press return or click the submit button. And, to a first approximation, the behaviour of the browser and the website is consistent across sites as well: the username and password entered into the form are sent to the server in an HTTPS `POST` request and a session cookie is returned. However, when we look in more detail, we notice a huge range of variations, some subtle and some baffling. Whilst these variations are imperceptible to the user, they present difficulties for a software agent parsing or automatically submitting the login form. This is because sub-tasks like entering the right username and password "into the appropriate boxes" are non-trivial and must rely on heuristics.

What are, then, the variations commonly exhibited by popular websites? A significant one is the modification of the static HTML login form by JavaScript. Some sites, such as Pinterest [1], use JavaScript to dynamically insert the login form into the page. Besides being annoying for users who browse with JavaScript disabled for security reasons, this practice also complicates the task of a password manager. Instead of parsing the HTML document just once at load-time to find any login forms, it must also monitor all changes made to the Document Object Model (DOM) by JavaScript thereafter.

The sins of JavaScript don't end there, though. Many sites use JavaScript to actually submit the form, thereby confusing utilities such as password managers that commonly intercept submission of the form to save the credentials. Forgoing a simple "submit" input adds little benefit and obfuscates the login process. Furthermore, to mitigate Cross-Site Request Forgery (CSRF) attacks [2], JavaScript is sometimes used to automatically insert values (nonces/challenges associated with the session) into hidden fields within the login form. For example, the following is added to the form on the Vimeo login page (`https://vimeo.com/log_in`) using JavaScript:

```
<input type="hidden" name="token" value="8113..." />
```

Attempts to log in to Vimeo with JavaScript disabled fail. Programs that parse or submit a login form must be compatible with such approaches, without being explicitly aware of what they achieve or even that they are being used.

Another problem facing a software agent such as a password manager is extracting the meaning (semantics) from the HTML login form. In the first instance, we'd like to determine what site the login form is for. The continued prevalence of phishing attacks demonstrates that reliably determining the website of a login page is too difficult for many humans. Software agents should have an advantage here. Indexing the username and password by the site's URL ensures that, provided HTTPS is used, the username and password are only submitted to correct site. Unfortunately, things are rarely this simple. Some websites have login forms on multiple pages—for example Facebook has one on its main landing page (`www.facebook.com`) and one on a dedicated login page (`www.facebook.com/login.php`). Should these login forms be considered as being for the same service? In the case of Facebook, both URLs are in the same second level domain `facebook.com`, so the answer is probably yes. But what about in a corporate intranet, where diverse services such as for submitting expenses and time sheets are all likely to be under the same second level domain?

It might be argued that services should only be considered the same if they have exactly the same URL. But what about the query string? Does that have to match as well? What about the order of the query parameters? What about dynamic URLs that provide alternative but equivalent encodings of the URL's query component? Any heuristic trying to shed light on this morass is likely to get things wrong (at least some of the time). Should users really have to accept that the computer doesn't even know what service is being logged in to?

Whilst it's obvious to a human whether a login to Facebook or Gmail succeeded, it's actually pretty hard for a software agent to know what happened. Whether or not the submitted username and password were correct, a HTTP 200 OK response is returned, indicating that a page was successfully served in response to the request (possibly after a sequence of redirects—in the case of Gmail rather a lot of redirects). Password managers can not reliably differentiate between these outcomes and, as a result, they often ask users to save passwords that are mistyped or just plain wrong. This seems needlessly annoying.

3 Incentives

Our proposal offers obvious advantages to users in terms of usability (you don't have to remember or type the passwords any more) and security[1] (you can use strong, distinct passwords). The advantages for password manager writers are even clearer (without guesswork, code becomes simpler, more reliable and much easier to maintain). Let's thus spend a few words on the incentives for website operators.

We believe it is in the best interests of website operators to support password managers: the website users will gain in usability and security. If users, thanks to

[1] Users of password managers are still exposed to malware; we are not claiming that the security offered by password managers is absolute (see section 5). Besides, our proposal implicitly also supports higher-security password managers running on dedicated hardware.

password managers, adopted strong unique random passwords, website operators would have much less to worry about confidentiality compromises of their hashed password file.

We understand that website operators don't want to allow bots to register thousands of accounts and we support this goal. Any techniques the websites may wish to use to ensure the presence of a human registrant (from CAPTCHAs to telephone callbacks and so forth) will continue to be available. We are only concerned with helping the human registrant store the password in a password manager instead of having to remember it in their brain. Only websites with delusions of grandeur may still believe that, regardless of all other demands on the user's memory and patience, *their* password is so important that it must be uncrackably strong *and* different from any others *and* never written down. They should study the Compliance Budget model [3], manage risks more maturely and cure their superiority complex.

4 The PMF semantic markup

4.1 Overview

We propose adding "password-manager friendly" (PMF) semantic markup to forms related to creating, accessing and managing user accounts, to simplify the following tasks:

- Finding forms and determining their purpose (login, registration, etc.).
- Finding the important inputs within the forms.
- Parsing password policies and generating valid new passwords.
- Detecting errors.

We adopt a simple and pragmatic approach used in other HTML microformats, of using semantic class names. A `class` attribute value can be specified for any HTML element [4] and the use of semantic class names is supported by the W3C [5]. We use the `pmf` prefix as a poor man's namespace to avoid clashes with programmer-defined class names. For example[2], a login form is marked with the `pmf-login` class:

```
<form action="/login" method="POST" class="pmf-login">
```

Although form inputs have other attributes such as `name` and `type` which may *often* give sufficient semantic information, standardised class values can be used to remove *any* ambiguity. For example, not all inputs with `type="password"` are for long-term passwords: some are for one-time codes generated by hardware tokens. Furthermore, as `name` attribute values are sometimes automatically generated by web frameworks or are specified by other standards such as OAuth [6], use of these attributes could cause conflicts. In contrast, any HTML element may have multiple classes [4], so our use of semantic class names ensures interoperability.

[2] In these examples, underlined text denotes PMF-related additions.

4.2 Forms

Being able to reliably determine the type or purpose of a given form enables a software agent like a password manager to offer a richer and/or more consistent user experience. `form` elements should be marked with the semantic classes specified in Table 1.

Table 1. Semantic classes for forms.

Form type	Semantic class name
Login	`pmf-login`
Registration	`pmf-registration`
Change password	`pmf-change-password`
Password reset	`pmf-reset-password`

4.3 Inputs

Username Login and registration forms typically contain an `input` element of type `text` or `email` for entering a username (which is often the user's email address). These inputs should be marked with the `pmf-username` class:

```
Username or email address:
<input type="text" name="user" class="pmf-username"/>
```

Password resets and changes are tricky for a password manager because the software cannot tell—in the case where a user may have multiple accounts with the same website—which password is being changed. For example, a simple experiment using Firefox's built-in password manager and two Google accounts reveals that, in some cases, the password manager must prompt the user to ask which account they are updating the password for, even though they are already logged in.

We propose that site authors should include a `hidden`-type field in these forms, marked with the `pmf-username` semantic class and with its value set to the username of the relevant account:

```
<form action="/reset" method="POST" class="pmf-reset-password">
  <input type="hidden" class="pmf-username" value="jimbojones"/>
  ...
</form>
```

Passwords Inputs for passwords typically appear in all four of the above form types. Some password inputs, such as those in registration forms, are for new passwords, while others are for existing passwords. These sub-types are unambiguously distinguished by the `pmf-new-password` and `pmf-password` semantic classes respectively. It is useful to distinguish them because they appear together

in "change password" forms. These typically contains three `password`-type inputs, one for the user's current password and two for their desired new password (one to confirm the other). All three will have a different `name` attribute values but, using semantic class names, the purpose of each input is made clear:

```
<form action="/change" method="POST"
    class="pmf-change-password">
  <input type="password" name="current" class="pmf-password"/>
  <input type="password" name="new" class="pmf-new-password"/>
  <input type="password" name="confirm"
     class="pmf-new-password"/>
</form>
```

Stay signed in Many login forms include a "stay signed in" check box which allows the user to control whether their session with a website should persist across multiple browser sessions. If present, this input should be marked with the `pmf-stay-signed-in` class:

```
Stay signed in?
<input type="checkbox" name="persist"
    class="pmf-stay-signed-in"/>
```

Annotating the "stay signed in" check box allows a software agent to apply a global policy on staying signed in for the user, across all websites. Many websites tick the "stay signed in" box by default and users accept this. But, if their password manager could apply a "never stay signed in" policy for them, they may be happy for it to do so and thereby gain a valuable security (and privacy) boost by not being permanently signed-in to their online accounts.

Another scenario in which this feature might be useful is the cybercafé: for the benefit of the patrons, the web browsers installed on the public cybercafé machines might be configured to disable the "stay signed in" feature by default.

Hidden inputs Forms often contain `hidden`-type input elements which are not visible when the HTML is rendered[3]. As human users are normally unaware of and cannot interact with these inputs, it is not useful for a software agent acting on the user's behalf to be able to interact with them either and we don't propose any additional markup.

4.4 Password policies

Large-scale password leaks have shown that many users optimise for memorability and convenience rather than security, choosing trivially-guessable passwords

[3] The values of these hidden inputs are usually populated by the web server when it generates the HTML of the page and then not changed on the client side. For example web frameworks, such as Django [7], use them to implement Cross Site Request Forgery protection.

like `123456`, `qwerty` or `password`. Password composition policies ("between 8 and 16 characters, of which at least one uppercase, one digit and one symbol") are an attempt to enforce selection of passwords that will be harder to guess. Although we may not agree with the password policies that websites impose,[4] we believe that their rules should be made available in machine-readable form to allow password managers to generate strong compliant passwords.

In this section we therefore define a simple specification for a machine-readable (JSON) description of a password composition policy. Our goals have been to make it easy for the website developer to write their intended policy and for the spec to be sufficiently expressive that most commonly observed policies can be represented with it.[5]

A machine-readable password composition policy is included in an HTML document as the value of a `hidden`-type input element[6]. This hidden input should be marked with the `pmf-policy` semantic class and appear within a form with the `pmf-register` semantic class. A library routine, written once and for all as part of the standard, can then generate the corresponding human-readable version and localize it to any language[7]:

```
<form action="/register" method="POST" class="pmf-register">
  ...
  New password:
  <input type="password" name="new" class="pmf-new-password"/>
  <input type="hidden" class="pmf-policy" value='
      Machine-readable policy as a multiline string
      according to the syntax described below.
  '/>
  Human-readable policy
  ...
```

[4] The debate on whether they are effective would sidetrack us into a different discussion; what we note here is that such policies may reject some otherwise very strong passwords such as those that a software agent might generate. For example because they exceed the maximum length, or because they fail to include a character from one of the classes, or because they include a disallowed character, maybe outside the ASCII range. Bonneau and Xu's study [8] of non-ASCII passwords, or more accurately of passwords from people whose native language doesn't fit into ASCII, is instructive.

[5] We have strongly resisted the temptation to cover absolutely all cases and become Turing-complete. Useful rules that our notation cannot express include blacklists of known weak passwords and stateful checks such as "you can't use your username or a variation of it". But such violations are very unlikely to occur if the password is generated randomly.

[6] Note that, to allow double quotes in the policy as required by the JSON syntax, the whole policy must be enclosed in single quotes. The policy itself won't normally contain single quotes but, should it need to, those must be escaped.

[7] The routine could even be embedded as JavaScript in the page itself. Alternatively, the translation to human-readable form could be performed offline and the result statically embedded in the page, as in this example.

```
</form>
```

The policy optionally specifies a minimum and a (bleah)[8] maximum length as non-negative integer fields. It then specifies what classes of characters are allowed and which classes the password must contain.

In the spirit of the 2014 Stanford password policy [9], we allow the rules to be more strict for short passwords but more relaxed for long ones: this is achieved in practice by defining multiple sub-policies that apply depending on the length of the password.

Character class notation Several common character classes are predefined in Table 2. Although many current password systems only work with ASCII characters, for future-proofing we allow the definition of password policies that allow arbitrary subsets of Unicode characters. Arbitrary character classes can thus be defined by enumerating the Unicode characters they contain, in any order, in a JSON list. For example, the `symbol` class could also be written as

```
["!", "@", "#", "$", "%", "&", "*", "-", "+", "/", "="]
```

or, using Unicode escape sequences,[9] as

```
[
  "\u0021", "\u0040", "\u0023", "\u0024", "\u0025",
  "\u0026", "\u002a", "\u002d", "\u002b", "\u002f",
  "\u003d"
]
```

As a shorthand, contiguous ranges of Unicode characters in the list can be specified by listing the first and last character with the string `"..."` between them. For example, the `digit` class could be written as

```
["\u0030", "...", "\u0039"]
```

Table 2. Predefined character classes

String constant	Class contents
`"lower"`	the 26 lowercase ASCII letters
`"upper"`	the 26 uppercase ASCII letters
`"digit"`	the 10 ASCII digits
`"symbol"`	the following 11 symbols[10]: `! @ # $ % & * - + / =`
`"base64"`	the 64 ASCII characters defined by the "base 64" encoding
`"ascii"`	the 95 printable ASCII characters from 32 (space) to 126 (tilde)

[8] We believe setting a maximum password length is a dumb idea but here we are trying to allow websites to express their policy in machine-readable format rather than compelling them to switch to a sensible policy.

[9] `"\u"` followed by 4 hexadecimal digits (`http://json.org/`)

Simple policies Most of the policies observed in the wild are "simple" policies that apply the same character class requirements to passwords of all lengths. Simple policies are represented by a list of length one, containing a single sub-policy object:

```
[
  {
    minLen: 8,
    mustHave: ["upper", "lower", "digit"],
    mayHave: ["base64", "␣"],
  }
]
```

In the above example, the password must have a length of at least 8 characters inclusive; it must contain at least one character from each of the three classes listed on the `mustHave` line[11]; and it may also contain any character in any of the two classes listed on the `mayHave` line.

For example the `amazon.com` policy, at the time of writing, simply requires the password to be between 6 and 128 characters and can therefore be rendered as follows.[12]

```
[
  {
    minLen: 6,
    maxLen: 128,
    mustHave: [],
    mayHave: ["ascii"]
  }
]
```

Complex policies A complex policy is one where, as in the Stanford policy [9], the password composition rules depend on the length of the password. It consists of a list of two or more sub-policies with non-overlapping length ranges. If a certain password length is not included in any of the ranges, then passwords of that length are not allowed. For example, Fig. 1 is a rendering of the Stanford policy as written up in their poster.

4.5 Errors

As mentioned previously, determining whether a login attempt (or other action) was successful or not is a difficult problem for software agents, because at the HTTP layer a "200 OK" status code is returned in both cases. The user is informed of any problems using prominent human-readable error messages within

[11] Our notation cannot express more elaborate rules such as "must include characters from at least 3 of these 5 classes".

[12] The `mayHave` line is just guesswork: we have not tested whether Amazon allows even more characters—perhaps even non-ascii ones.

```
[
  {
    minLen: 8,
    maxLen: 11,
    mustHave: ["upper", "lower", "digit", "\u0020",
        "...", "\u002f"],
    mayHave: "ascii";
  },
  {
    minLen: 12,
    maxLen: 15,
    mustHave: ["upper", "lower", "digit"],
    mayHave: "ascii";
  },
  {
    minLen: 16,
    maxLen: 19,
    mustHave: ["upper", "lower"],
    mayHave: "ascii";
  },
  {
    minLen: 20,
    mustHave: [],
    mayHave: "ascii";
  },
]
```

Fig. 1. Stanford password policy expressed in the PMF policy language.

the returned HTML page, but we would like these messages to be just as easy to find for machines.

We propose marking these error messages with the `pmf-error` semantic class name to make them trivial for software agents to find:

```
<p class="pmf-error">Incorrect username and/or password</p>
```

5 Related work

Bonneau and Preibusch's [10] comprehensive review of the authentication landscape on the web argues that some sites deploying passwords do so primarily for psychological rather than security reasons. For example, they speculate that password-protecting accounts serves as a justification for collecting marketing data and as a way to build trusted relationships with customers. Whatever the underlying reasons, it is apparent that the number of password-protected accounts an average user manages has increased markedly since the advent of the web. Florencio and Herley [11] report that the average user has 6.5 passwords, each of which is shared across 3.9 different sites. Furthermore, each user has about 25 accounts that require passwords. Without the reported level of password reuse, managing 25 separate accounts with unique random passwords is barely imaginable for most users.

A password manager, either as a separate program such as PasswordSafe [12] or integrated with or in the browser, is now a well established solution for managing the increasing burden password-based authentication on the web. Given the increasing reliance on password managers[13], a recent thread of research has investigated their security properties.

Gasti and Rasmussen [13] investigate the security properties of the password database formats used in range of popular password managers. They define two new games to analyse the security of password manager databases: *indistinguishability of databases* (IND-CDBA) game and *chosen database* (MAL-CDBA) game; the *indistinguishability of databases* game models the capabilities of a realistic passive adversary, and the *chosen database* game models the capabilities of an active adversary able to both read and write the password database file. Google Chrome stores plaintext username/passwords in the user's profile directory. As a result, an attacker can trivially win both the IND-CDBA and MAL-CDBA games with Chrome as the Challenger. Firefox also fails both games; however, Firefox optionally allows users to encrypt the passwords stored in the password managers database under a user-supplied master key. This option provides at least some security benefits over Google Chrome's password manager, even if the full benefits of indistinguishability under the IND-CDBA and MAL-CDBA games aren't afforded. Gasti and Rasmussen's analysis concludes that, among the systems they studied, only PasswordSafe v3 [12] is invulnerable to attackers under the IND-CDBA and MAL-CDBA security models.

[13] As an example, 1Password alone is estimated to have a install base of 2 to 3 million users.

Silver *et al* [14] identify a class of vulnerabilities exploitable when using several popular passwords managers. The treat model they consider is a user connecting to a network controlled by the attacker, such as a rogue WiFi hot-spot. Under this model the attacker is able to inject, block and modify packets on the network. The attacker's goal is to extract passwords stored by the password manager without further action from the user. The attacks presented by Silver *et al* rely on exploiting the password manager auto-filling policies: for example, the password manager can be coerced into auto-filling forms in invisible iframes embedded within the WiFi hot-spot's landing page.[14]

Li *et al* [15] analysed the security of five popular integrated password managers (that is, password managers integrated with or in the web browser). Four key concerns with browser-based password managers were identified in this study: bookmarklet vulnerabilities, web vulnerabilities, authorisation vulnerabilities and user interface vulnerabilities. Bookmarklet[15] vulnerabilities, introduced by Adida *et al* [16], result from the bookmarklet's code running in a JavaScript environment potentially under the control of an attacker. Li *et al* show that such vulnerabilities are still widespread in popular password managers. The web vulnerabilities identified by Li *et al* consist of well know cross-site request forgery (CSRF) and cross-site scripting (XSS) attacks. The authorisation flaws identified by Li *et al* result from sloppy implementations. User interface vulnerabilities can be considered as *phishing* attacks against the password manager itself. In cases where the user is not authenticated to their password manager, a number of in-browser password managers automatically open the login form for the password manager in an iframe. Users have no means to differentiate between this behaviour and a phishing attack.

Password managers should be considered as tactical solutions, alleviating some of the gross security and usability failings of passwords. Pico [17] is a strategic solution seeking a more usable and secure replacement for passwords everywhere they are used (not just on the web). Recent work on Pico has attempted to provide a mechanism that can work alongside passwords [18]. The Pico bootstrapping technologies, whilst not being a password manager in the classic sense, are required to parse and automatically submit login forms on the user's behalf and would thus also benefit from our semantic annotations.

[14] Auto-filling of forms by the password manager improves usability and therefore, before mitigating this vulnerability by disabling the auto-filling, careful consideration is needed of the inherent trade off between security and usability. We shouldn't lose sight of the fact that normal users don't have threat models; therefore, simply asking them whether they want to enable or disable auto-filling is a bit of a cop out.

[15] A bookmarklet is a bookmark containing JavaScript that can be used to extend a web browser's capabilities. Bookmarklets have advantages over alternatives such as addons or extensions as they are cross browser and are managed by the user like bookmarks.

6 Conclusions

All password managers rely on fallible heuristics. Such code is complex, never fully accurate and it requires constant updates, besides wasteful replication of efforts by every password manager developer. We argue that all parties would benefit if websites offered a standard interface to password managers, enabling consistent and accurate agent-supported password creation, registration and login, without brittle programmatic guesswork.

Our PMF proposal, of augmenting a website's password pages with simple and unambiguous machine-readable semantics, makes the operation of password managers much simpler and more reliable. Users benefit from reduced cognitive load and reduced typing burden. Reliable generation of strong random passwords increases security for both users and websites. A well-defined interface eliminates guesswork and makes the password manager code leaner and much easier to maintain. We feel PMF is beneficial for all parties involved: users, website operators, password manager developers. We will be pleased to work with developers of websites, browsers and password managers, as well as with standards bodies, to promote its widespread adoption.

7 Acknowledgements

References

1. Pinterest. `https://pinterest.com` Accessed: 2014-11-07.
2. OWASP: Cross-site request forgery (csrf) prevention cheat sheet. `https:///www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet` (August 2014) Accessed: 2014-11-06.
3. Beautement, A., Sasse, M.A., Wonham, M.: The compliance budget: Managing security behaviour in organisations. In: Proceedings of the 2008 Workshop on New Security Paradigms. NSPW '08, New York, NY, USA, ACM (2008) 47–58
4. Berjon, R., Faulkner, S., Leithead, T., Pfeiffer, S., O'Connor, E., Doyle Navara, E.: HTML5. Candidate recommendation, W3C (October 2014)
5. Stuven, Sybrel (W3C): Use class with semantics in mind. `http://www.w3.org/QA/Tips/goodclassnames` Accessed: 2014-11-07.
6. Hardt, D.: The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard) (October 2012)
7. Django documentation: Cross Site Request Forgery protection. `https://docs.djangoproject.com/en/1.7/ref/contrib/csrf/` Accessed: 2014-11-07.
8. Bonneau, J., Xu, R.: Of contraseñas, sysmawt, and mìmă: Character encoding issues for web passwords. In: Web 2.0 Security & Privacy. (May 2012)
9. Stanford University. `https://itservices.stanford.edu/service/accounts/passwords/quickguide` Accessed: 2014-11-07.
10. Bonneau, J., Preibusch, S.: The password thicket: technical and market failures in human authentication on the web. In: WEIS 2010. (2010)
11. Florencio, D., Herley, C.: A large-scale study of web password habits. In: Proceedings of the 16th International Conference on World Wide Web. WWW '07, New York, NY, USA, ACM (2007) 657–666

12. Schneier, B.: Password safe. `https://www.schneier.com/passsafe.html` Accessed: 2014-11-06.
13. Gasti, P., Rasmussen, K.B.: On the security of password manager database formats. In: ESORICS. (2012) 770–787
14. Silver, D., Jana, S., Boneh, D., Chen, E., Jackson, C.: Password managers: Attacks and defenses. In: 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USENIX Association (August 2014) 449–464
15. Li, Z., He, W., Akhawe, D., Song, D.: 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 (August 2014) 465–479
16. Adida, B., Barth, A., Jackson, C.: Rootkits for javascript environments. In: Proceedings of the 3rd USENIX Conference on Offensive Technologies. WOOT'09, Berkeley, CA, USA, USENIX Association (2009) 4–4
17. Stajano, F.: Pico: no more passwords! In: Proceedings of the 19th international conference on Security Protocols. SP'11, Berlin, Heidelberg, Springer-Verlag (2011) 49–81
18. Stajano, F., Jenkinson, G., Payne, J., Spencer, M., Stafford-Fraser, Q., Warrington, C.: Bootstrapping adoption of the pico password replacement system. In Christianson, B., Malcolm, J.A., Matyás, V., Svenda, P., Stajano, F., Anderson, J., eds.: Security Protocols XXII - 22nd International Workshop Cambridge, UK, March 19-21, 2014 Revised Selected Papers. Volume 8809 of Lecture Notes in Computer Science., Springer (2014) 172–186