Password Meters and Generators on the Web: From Large-Scale Empirical Study to Getting It Right

Steven Van Acker
iMinds-DistriNet, KU Leuven,
3001 Leuven, Belgium

Daniel Hausknecht
Chalmers University of
Technology, Sweden

Andrei Sabelfeld
Chalmers University of
Technology, Sweden

ABSTRACT

Web services heavily rely on passwords for user authentication. To help users chose stronger passwords, *password meter* and *password generator* facilities are becoming increasingly popular. Password meters estimate the strength of passwords provided by users. Password generators help users with generating stronger passwords.

This paper turns the spotlight on the state of the art of password meters and generators on the web. Orthogonal to the large body of work on password metrics, we focus on getting password meters and generators right in the web setting. We report on the state of affairs via a large-scale empirical study of web password meters and generators. Our findings reveal pervasive trust to third-party code to have access to the passwords. We uncover three cases when this trust is abused to leak the passwords to third parties. Furthermore, we discover that often the passwords are sent out to the network, invisibly to users, and sometimes in clear. To improve the state of the art, we propose SandPass, a general web framework that allows secure and modular porting of password meter and generation modules. We demonstrate the usefulness of the framework by a reference implementation and a case study with a password meter by the Swedish Post and Telecommunication Agency.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access

Keywords

web security; passwords; sandboxing

1. INTRODUCTION

The use of passwords is ubiquitous on the Internet. Although a variety of authentication mechanisms have been proposed [6], password-based authentication, i.e. matching the combination of username and password against credentials stored on the server, is still a widespread way of authenticating on the Internet. Databases with user credentials are often leaked after a website has been compromised [59]. Password storage best practices [40] prescribe organizations to store the passwords hashed with a cryptographically strong one-way hashing algorithm and a credential-specific salt.

Password cracking. Motivated attackers will nevertheless try to reverse the stored hashes into plaintext password by *cracking* the hashes with special tools such as John The Ripper [38]. To crack a password hash, password crackers generate hashes of candidate passwords and compare them to the original hash. If a match is found, the original password was recovered or at least a password that results in the same hash value.

For short enough passwords, it is possible to enumerate passwords of a given length and store all the hashes in a database. This database, known as a *rainbow table* [37], can be used to speedup the cracking of hashes of short-length passwords. To avoid this, passwords can be combined with a *salt* [34] before hashing. Adding a salt to a hash makes rainbow tables less practical because they would have to contain all the hashes of passwords combined with all salts.

With the knowledge that users often select passwords that are based on dictionary words [25], a good strategy for a password cracker is then to use a dictionary of words as the basis for input for the cracker. This practice is known as a *dictionary attack* [34] and is used by the popular CrackLib [10] library to verify the strength of passwords entered by users. Password hashes can often be cracked despite newest hashing algorithms, although it may require a significant amount of time and resources if the plaintext password is well chosen [8].

Password meters and generators. It is thus of vital importance that users pick "strong" passwords, i.e, passwords that are not easily guessable or crackable by cracking tools. However, picking a sufficiently strong password is a difficult task for a typical user [65]. To help users with this task, tools have emerged that both evaluate the strength of user-chosen passwords and generate strong passwords using heuristics. These tools are called *password meters* and *password generators*, respectively.

Although password meters and password generators can help to select stronger passwords [56], they bring a new breed of security problems if designed or implemented carelessly. In the web setting, they are an immediate subject to all the ailments of web applications.

Passwords meters and generators on the web. This paper turns the spotlight on the state of the art of password meters and generators on the web. Orthogonal to the large

CODASPY'15, March 2–4, 2015, San Antonio, Texas, USA.
Copyright © 2015 ACM 978-1-4503-3191-3/15/03 ...$15.00.
http://dx.doi.org/10.1145/2699026.2699118.

body of work on password metrics [8, 7, 44, 64, 23], we focus on getting password meters and generators right in the web setting.

Browser extensions, as BadPass [5], to indicate password strength, avoid some security problems by running separately from the code on web pages, but they have the obvious inconvenience of requiring users to install an extension. The abundance of web pages with password meters and generators (analyzed in Section 2) speaks for the popularity of these services in the form of web services, which justifies our focus.

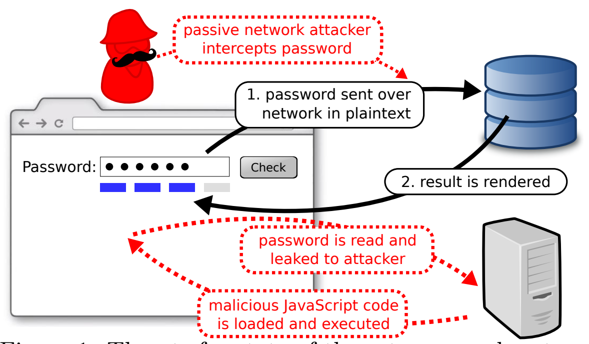
Threat model. First, we are interested in the *passive network attacker* [20] that sniffs the traffic on the network. This attacker might be able to get hold of passwords that are transmitted on the network in clear. Second, we are interested in the *web attacker* [2] that controls certain web sites. Of particular concern are *third-party web attackers* that might harvest passwords when a script from the attacker-controlled web site is included in a password meter or generator service. Also of concern are *second-party web attackers* that are in control of stand-alone password meter and generator services. It is undesirable to pass the actual passwords to such services. Although a password meter might not have the associated username, current fingerprinting techniques facilitate uniquely tracking browsers, allowing the identification of users [14]. A number of techniques such as autocomplete features open up for programmatically determining the usernames.

State of the art. The first part of our work is an analysis of the state of the art of password meters and generators on the web. We report on the state of affairs via an empirical study of password meters and generators reachable from the Bing search engine and top Alexa pages.

Unfortunately, the state of the art leaves much to be desired. Most strikingly, we find that the majority of password meters and generators lend their trust to third-party scripts. The current practice suffers from abusing the privileges of the script inclusion mechanism [36]. A recent real-life example is the defacement of the Reuters site in June 2014 [49], attributed to "Syrian Electronic Army", which compromised a third-party widget (Taboola [52]). This shows that even established content delivery networks risk being compromised, and these risks immediately extend to all web sites that include scripts from such networks.

77.9% of standalone password meters, 76.8% of standalone password generators, and 96.5% of password meters on service signup pages include third-party code (which runs with the same privileges as the main code). Figure 1 depicts the danger with trusting third-party code. A script from a third party has both access to the password and access to network communication to freely leak the password. Our findings (detailed in Section 2) include three websites that send passwords to such third-party sites as ShareThis [51] and Tynt [55].

Another unsettling finding is that password meters commonly send passwords over the network. This is unnatural because the purpose is to help the user with estimating the strength of such sensitive information as passwords. The principle of least privilege [50] calls for restricting the computation to the browser. Nevertheless, we observe that 16.35% of standalone password meters, 26.02% of standalone password generators, and 59.3% of password meters on service signup pages send the password over the network, of which 76.47%, 96.08%, and 3.92% send the password in cleartext


passive network attacker intercepts password
1. password sent over network in plaintext
Password:
Check
2. result is rendered
password is read and leaked to attacker
malicious JavaScript code is loaded and executed
Figure 1: Threats for state-of-the-art password meters

(over HTTP). Figure 1 illustrates the possible attacks. When HTTP is used, the passive network attacker might get hold of the password by sniffing the network traffic. When HTTPS is used, the second-party server (standalone password meter or generator) gets hold of the password, an undesired situation for the first-party service associated with the tested password.

Astonishingly, only one service from all the web services from our empirical study sends hashed passwords to the server. We will come back to this important point in the space of design choices.

Getting it right. With the identified shortcomings of the state of the art at hand, we argue for a *sandboxed client-side* framework and implementation for password meters and generators on the web. From the point of security, such an implementation honors the principle of least privilege: the password stays with the client with password strength estimation/generation executed by JavaScript within the browser. The sandboxing guarantees that the JavaScript code does not access the network. From the point of usability, this enables users to test their actual passwords rather than being forced to distort the original passwords (see the discussion below in the context of the case study). Finally, from the performance point of view, this allows entirely dispensing with client-server round trips for each request. This enables substantial speedup for processing password strength estimation.

Clearly, sending the password to the server can be reasonable for the password meters on service signup pages, where the implementations require that user passwords are stored on the server anyway. However, when it comes to standalone password meters and generators, we make a case for client-side deployment. One possible argument for involving the server in password strength estimation is that the server can check passwords against a dictionary of common words/passwords or a known database of leaked passwords. However, this only makes sense if the size of such a dictionary/database is significant (in which case the secure way to implement the service is to send salted and hashed passwords over HTTPS). We argue that commonly-used password meter libraries, such as CrackLib [10] and zxcvbn [67], are based on dictionaries of size that is susceptible to client-side checking.

Likewise, a reason to generate a password on the server side, is that JavaScript's built-in random number generator is not cryptographically secure on all browsers. The Web Cryptography API [63] will remedy this when it is standardized. In the meantime, there are JavaScript libraries, such as CryptoJS [12], that provide secure cryptographic algorithms to generate random numbers.

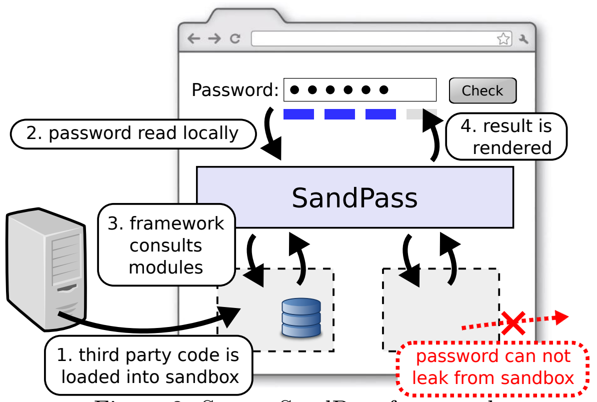

Password: ●●●●●●
Check
2. password read locally
4. result is rendered
SandPass
3. framework consults modules
1. third party code is loaded into sandbox
password can not leak from sandbox

Figure 2: Secure SandPass framework

Generic framework for sandboxing. As a concrete improvement of the state of the art, we propose SandPass, a general web framework that allows secure and modular porting of password meter and generation modules. The framework provides a *generic technique for secure integration of untrusted code* that operates on sensitive data, while stripped of capabilities of leaking it out. We show how to run password meter/generator code in a separate iframe while disabling outside communication and preventing possible password leaks. Figure 2 illustrates the security of the framework. Third-party code is loaded in isolated sandboxes without network access. The framework reads the password locally and consults the modules to score the password strength. Any databases with commonly used passwords or hashes are loaded into the isolated sandboxes as well. We demonstrate the usefulness of the framework by a reference implementation, where we show how to port such known password meter modules as CrackLib [10].

Case study. Following responsible disclosure, we have contacted the web sites that send out passwords and pointed out the vulnerability. One of our reports has resulted in a subsequent case study of a service by the *Swedish Post and Telecommunication Agency (Post- och telestyrelsen, PTS)* [45], a state agency that oversees electronic communications in Sweden. The case study is based on PTS' *Test Your Password* service *(Testa lösenord)* [54]. A quick Internet search of pages linking the service suggests that this service is often recommended by the Swedish organizations, including universities, and the media when encouraging users to check the strength of their passwords. According to PTS, over 1,000,000 passwords have been tested with the service [46].

On the positive side, PTS' service avoids including third-party scripts. However, it sends (over HTTPS) the actual passwords to the server. PTS realizes that this might be problematic, which is manifested by encouraging the users on the web page *not to use their actual passwords* [46]. Not only does this make the service insecure (the users' passwords or their derivatives are leaked to PTS) but also severely limits its utility (the users are forced to distort their passwords and guess the outcome for the real passwords). In addition, the performance of the service is affected by communication round trips to the server on each request.

To help PTS improve the service, and with our reference implementation as the baseline, we have implemented a service that improves the security, utility, and performance of the Test Your Password service. The security is improved as already illustrated by Figure 2 in contrast to Figure 1. The utility is improved by enabling the users to test their real passwords. We have also made the service more interactive, providing feedback on every typed character instead of the original service where the users type the entire password and press a submit button. Due to the volume of JavaScript, our load-time performance increases with the order of 2.5x (unnoticeable for user experience). However, the speedup for the actual password processing is in the order of 34x because it is unnecessary to communicate with the server.

Contributions. A brief summary of the contributions is:

- Bringing much needed attention of the security community to the problem of design and implementation of password meters and generators on the web.
- The first large-scale empirical study of security of web password meters, password generators, and account registration pages.
- Uncovering unsatisfactory state of the art: we point out unnecessary trust to third-party servers, second-party services, and the network infrastructure.
- Development of a generic sandboxing framework that allows code to operate on sensitive data while not allowing leaks out of the sandbox.
- Design and implementation of SandPass, a secure modular password meter/generator framework. We demonstrate security with respect to both the web and passive network attacker.
- Case study with a password meter by the Swedish Post and Telecommunication Agency to improve the security, utility, and performance for a widely used service.

The code for SandPass and case study are available online [58].

2. STATE OF THE ART

To gain insight in password meters and password generators, we performed an extensive Internet search to find standalone instances of them. In addition to occurrences in the wild, they also occur on account signup pages. Since no instances of password generators were observed on signup pages, we do not consider those.

All experiments are based on a common setup which, besides the Firefox browser, also incorporates PhantomJS and mitmproxy.

PhantomJS [4] is a headless browser based on WebKit, scriptable through a JavaScript API. PhantomJS will load a page, render text and images, and execute JavaScript as any regular browser. Interaction with a loaded page can be scripted through a JavaScript API, allowing a user to automate complicated interactions with a web application and process the response. In our experiments, PhantomJS was used to render screenshots of websites once they were loaded and had their JavaScript code executed.

Mitmproxy [3] is a man-in-the-middle proxy which can be used to log, intercept, and modify all HTTP and HTTPS requests and responses passing through it. A CA SSL certificate can be installed in browsers making use of mitmproxy, allowing it to also intercept and modify encrypted traffic without the browser noticing. Python scripts can register hooks into mitmproxy, which are triggered on requests and responses, and which can perform custom actions not originally implemented into mitmproxy. In our experiments, we use mitmdump, a version of mitmproxy without a UI, together with custom hooks that trigger certain actions when a special URL is visited.

The typical workflow of any of our manual experiments is driven by a control-loop which launches a clean Firefox instance and opens an URL to investigate. All traffic is monitored and logged while the user interacts with the loaded webpage. Bookmarklets [35] are used to log information about the visited webpage and transfer that information from Firefox through mitmproxy into the control-loop.

2.1 Stand-alone password meters

Setup. We queried Bing for typical keywords associated with password meters, e.g. "password strength checker", "website to test password strength", "how secure is my password", ... and stored the top 1000 returned URLs for each set of keywords. This resulted in a total dataset of 5900 unique URLs. A number of these webpages are related to password meters in some way, but do not actually contain a functional password meter. To filter those from the dataset, we rendered screenshots for all URLs using PhantomJS, classified them manually and only retained the functional password meters.

Each of the password meters was visited manually using the common setup, and interacted with to input a 20 character password. The response of the webpage was observed to determine whether visual feedback about the strength of the given password was given. During this interaction, all HTTP and HTTPS network traffic was intercepted and logged by mitmdump.

This traffic was then analyzed to see whether any form of the password was transmitted over network. Because some forms might truncate the entered password to a shorter length, we searched for the first 8 to 20 characters of the password. To make sure the password was not sent in an encoded form, we also looked for the MD5, SHA1, SHA224, SHA256, SHA384, SHA512 hashes as well as the Base64 encoding of the different versions of the password.

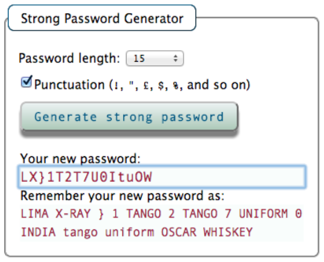
Results. In the set of 5900 URLs returned by Bing, we found 104 functional password meters. Of those 104, 98 included JavaScript of which 88 were over an insecure HTTP connection, and 81 included JavaScript from a third-party host, with 73 over HTTP. 86 password meters gave visual feedback about the strength of the given password without the user having to press a submit button.

While interacting with the password meters, 17 sent out the password over the network and 13 did so over an unencrypted HTTP connection. Of those 17, 15 required a submit button to be pressed, but two did not and sent the password to a server in the background. Only one of those 17 (`http://www.check-and-secure.com/passwordcheck/`) after having pressed submit, sent the password in a hashed format over the network instead of in plaintext, using both the MD5 and SHA256 hash formats.

None of the observed password meters submitted the password to a third-party host.

2.2 Stand-alone password generators

Setup. We again queried Bing, this time for keywords associated with password generators, e.g. "password generator", "passphrase creator", "create password online", ... and stored the top 1000 returned URLs for each set of keywords. This resulted in a total dataset of 8150 unique URLs.



Password length: 15

✓ Punctuation (!, ", £, $, %, and so on)

Your new password:

LX}1T2T7U0ItuOW

Remember your new password as:

LIMA X-RAY } 1 TANGO 2 TANGO 7 UNIFORM 0 INDIA tango uniform OSCAR WHISKEY

Figure 3: Example password generator

Just as with the raw "password meter" dataset, this set of URLs contained a number of pages related to, but not containing a password generator. We again rendered screenshots for all URLs and classified them manually.

Each password generator was then visited using our common setup, and interacted with to generate a password. As Figure 3 suggests, users often have to interact with a password generator to customize its parameters and generate a strong password. The generated password was logged through a bookmarklet so its presence could be detecting in incoming or outgoing network streams. Again, all network traffic generated during each of the visits was logged with mitmproxy.

The network traffic captured during the visit of each password generator was then analyzed to see whether the password, or any truncated or encoded form of it, was transmitted over network either in the requests or their responses.

Results. In the set of 8150 URLs returned by Bing, we found 392 functional password generators. Of those 392, 117 did not require user input to generate a password. In total, 351 of them included JavaScript, of which 332 were over an unencrypted HTTP connection, and 301 included JavaScript from a third-party host, of which 283 were over an unencrypted HTTP connection.

We have contacted the owners of several password generators in order to determine how often their service is used. The three replies we received indicate between 50 and 115 page views on average per day.

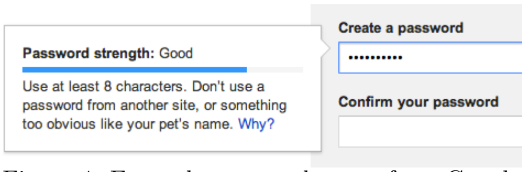
After interacting with the password generators, 100 of them generated a password on the server side and transmitted it back to the browser. 96 of those responses happened over an unencrypted HTTP connection.

Surprisingly, six password generators also transmitted the generated password over the network from the client side. Two of those had generated the password locally, while the remaining four received it from a server. While three of the six sent the password back to a server in their own top-level domain, the other three sent the password to two popular JavaScript widgets which enable and track content-sharing on webpages: ShareThis [51] and Tynt [55].

2.3 Password meters on registration pages

Setup. For each domain in the Alexa top 250, we visited the topmost webpage (e.g. `http://example.com` for example.com) and searched for an account signup form by following links and instructions on that webpage. If a signup page was found, and it allowed us to signup for an account freely and easily (e.g. without having to enter a social ID, a credit card number, waiting for an invitation e-mail or other), the URL of the signup page was kept as being usable for this experiment.

We then visited each usable signup page manually using our common setup and typed in a strong 20 character password in


Password strength: Good

Use at least 8 characters. Don't use a password from another site, or something too obvious like your pet's name. Why?

Figure 4: Example password meter from Google

the password field, but we did not click the submit button to complete the signup procedure. Figure 4 shows the password meter in action during our visit to the Google signup page, without having to click a submit button. Again, all HTTP and HTTPS network traffic generated during the visit was logged with mitmproxy.

The network traffic of each visit was analyzed to see whether the password, or any truncated or encoded form of it, was transmitted over network.

Results. From the top 250 Alexa domains we included in our experiment, we discovered 186 usable signup forms. Of the 186 signup pages, 86 use a password meter to give instant visual feedback to the registering user about the strength of the chosen password. Of those 86 signup pages with a password meter, 83 include third-party JavaScript code and 51 transmitted the entered password to a remote server in the background. Of those last 51 password-transmitting password meters on signup pages, two sent the password over unencrypted HTTP.

None of the signup pages sent the password to a host on a third-party domain.

2.4 Discussion

The most insightful results from the previous experiments with regard to our threat model, are summarized in Figure 5, Figure 6, and Figure 7.

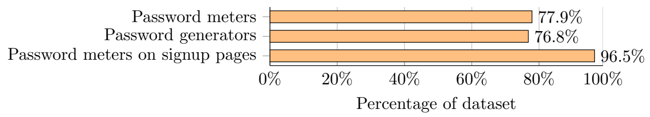


Figure 5: Fraction of dataset including 3^{rd} party JS

Third-party web attacker. Figure 5 shows that the majority of webpages in all three datasets include third-party JavaScript in a JavaScript environment that has access to the password field: 77.9% of standalone password meters, 76.8% of standalone password generators and 96.5% of password meters on account signup pages.

The inclusion of third-party JavaScript can pose a real threat when that JavaScript is under the control of a *third-party web attacker* [36]. Even if the author of third-party JavaScript code is not malicious, the host on which this code is located might be compromised. In that case nothing prevents the attacker from creating JavaScript to read all entered passwords and leak them to the Internet.

Nikiforakis et al. [36] show that close to 70% of the top 10,000 Alexa domains include Google Analytics. We believe that our similar result does not diminish our findings because it indicates that the developers of password meters and generators are unaware of the security implications of including third party JavaScript code.

Although we did not observe any malicious scripts that are actively intercepting and stealing passwords, we have found three cases of standalone password generators from which the generated passwords are leaked by third-party JavaScript designed to monitor content sharing.

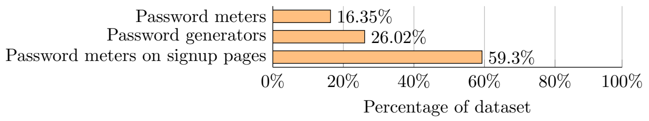


Figure 6: Fraction of dataset transmitting the password

Second-party web attacker. Figure 6 shows that 16.35% of standalone password meters, 26.02% of standalone password generators and 59.3% of password meters on account signup pages transmit passwords over the network to a remote server. This behavior is not isolated to lesser-known websites, but also occurs in highly Alexa-ranked domains. E.g. the password meter on Google's account signup page transmits the password over the network when this password exceeds seven characters.

Despite the availability of client-side solutions for the implemented services, there is a significant fraction that opts to send the password over the network and either check it on a remote server, or generate it on a remote server. It is hypothetically possible that these services use resource-intensive computations that are impractical to implement in client-side JavaScript. However, it is just as well possible that these services have been implemented by *second-party web attackers* with the purpose of tricking visitors into revealing their password and logging them. Nothing distinguishes these two possibilities for the user.

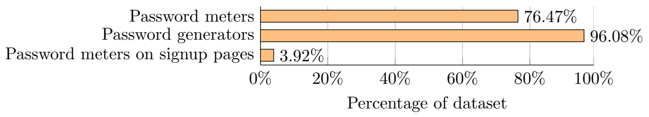


Figure 7: Fraction of password transmissions in the clear

Network attacker. Assuming that a second-party web attacker is not involved, there may be a need to send the password over the network. However, it would be unwise to send these passwords over the network in plaintext, without using encryption via HTTPS. Yet, as Figure 7 shows, the majority of standalone password meters and generators (respectively 76.47% and 96.08%) do not use encryption when transmitting the password. On the other hand, only 3.92% of the account signup pages, with a password meter, from the top 250 Alexa domains transmit the password without encryption. This data shows that a 96.1% majority of the Alexa top 250 website providers, in contrast to the providers of the standalone password meters and generators, better understand the dangers in sending password over an unencrypted connection. The handful of account signup pages in our dataset that do not use encryption when transmitting a password, can have their user's passwords intercepted by a *passive network attacker*.

3. CLIENT-SIDE FRAMEWORK

3.1 Framework

Based on our observations in the web and the attacker model, we identify requirements for the implementation of secure password meters/generators. To support web developers to fulfill these requirements in practice, we design SandPass, a JavaScript framework for secure client-side password meters/generators.

Requirements. The current state of the art for password meters and generators is vulnerable to attacks as described in our threat model in Section 1. The wide use of unencrypted HTTP connections, especially when transmitting passwords in plain text, allows for passive network attacks. But even with encrypted connections, second- and third-party web attackers can be successful by stealing the password from the webpage or tricking the client to send data over the network. However, completely banning third party code from a web page is usually not a realistic option. Also, preventing a website from sending any data over the network at all proves impractical. For example, a registration page with an integrated password meter must be able to send the user credentials to the server to complete the registration process.

It is therefore desirable to have a client-side service which on the one hand allows the inclusion of existing third party solutions for password metering and password generation, while on the other hand restricting the code's capabilities so that it cannot leak any password information. The concrete requirements for a framework to support such a service are as follows:

Client-side only: To prevent a password from being leaked, the password meter/generator does not require server access in order to provide the service. Thus, all password meter/-generator related code must be executed on the client side.

Small code base: The framework code is as small as possible to allow easy revision by web developers integrating the framework in their web page.

Code inclusion: The framework allows the inclusion of third-party code for password metering/generation.

Code isolation: To prevent JavaScript code from interfering with code of other modules or the main page, each module is isolated from the rest of the web page.

No network access: Included JavaScript code cannot send or leak password information over the network.

Result validation: The results of each module are validated before they are used in the main page to avoid content injection attacks.

Safe integration: The framework follows the current best practice for secure web implementations (e.g. the guidelines given by OWASP [39]), i.e. the framework is not the "weakest link" in an otherwise securely programmed web page.

Architecture. The architecture of SandPass is general enough to use it for both password meters and password generators.

For password meters, we assume a setting as illustrated in Figure 2. A user can type in the password in an input field on the main page which the framework then passes to the password meter code for analysis. For password generators, we assume a similar setting with the difference that the user can specify password generator options instead of supplying a password to be tested. SandPass then passes the generator options to the password generator code.

The result of the password/generator is then shown to the user on the same web page. The framework code itself is directly included in the main page and handles the collection of the input data, running the password meter/generator code, and calling the routines for updating the web view (steps 2–4 in Figure 2). These steps are executed every time a password has to be checked or generated.

The program code which actually performs the password metering/generation is downloaded by the framework and integrated in the web page as so called *modules* (step 1 in Figure 2). The purpose of modules is to isolate the third-party code from the web page as well as to restrict its network access.

3.2 Reference implementation

The reference implementation of SandPass respects the requirements and uses the architecture as described in the previous section. Additionally, we avoid using non-standard libraries to prevent dependencies on third-party code which could open security breaches in the framework. Instead, SandPass uses only standard browser features and JavaScript APIs as specified for HTML5 [62].

Standard browser features. The HTML5 *iframe* [17] element allows the embedding of web pages within others. Browsers limit access between iframes according to the *Same-Origin Policy* (SOP). With the `sandbox` attribute set, a browser assigns a unique origin to the iframe, strengthening the SOP access restrictions. By default, the sandbox attribute also disables scripts, forms and popups, which can be re-enabled using the respective keywords.

The JavaScript browser API method `postMessage` [43] provides a cross-origin communication channel for sending data between browser contexts, e.g., an iframe and its host page. A browser context can add an event listener for receiving and handling messages. Besides the actual data, the message contains a `source` attribute which can be used for sending response messages to the dispatcher.

The *Content Security Policy* (CSP) [9] specifies the sources a web page is allowed to access and which protocols to use. The main purpose of CSP is to mitigate the risks of content injection attacks. It therefore prohibits by default inline scripts and the JavaScript `eval` function. These restrictions can be lifted by using the keywords `"unsafe-inline"` and `"unsafe-eval"`, respectively. Though usually defined on the server side, the policies are enforced completely in the client's browser.

```
 1 <!-- fetch framework code from server -->
 2 <script language="javascript" src="pwdmeter.js" />
 3
 4 <script language="javascript" />
 5   /* respective callback functions */
 6   function resultHandler(res) { ... };
 7
 8   /* module inclusions */
 9   include("http://example.com/m.js", resultHandler, "
         check");
10
11   /* running a password strength analysis */
12   runSandPass("myPassword");
13 </script>
```

Listing 1: Example code for including SandPass

SandPass is fully implemented in JavaScript. After downloading the framework and module scripts, all code is executed in the browser without any further server interaction.

The framework can be added to the main web page through common JavaScript inclusion techniques, e.g., through the HTML script element. Listing 1 shows an example web page snippet including the framework in line 2.

SandPass provides an `include` function for the inclusion of modules. The function parameters are a list of all URLs of the script file included in the same module, the result handler function, and the name of the module's main function, i.e. the function called to later execute the module. When `include` is called, the framework fetches the module code from the given sources and creates the respective module.

The *result handler* is a JavaScript function which is called after the associated module returns a result. Its main purpose is to present the result to the user by updating the main web page. Since the demands for the result handler vary for each individual page design, the web developer is completely free to implement this function as she sees fit. The example in Listing 1 defines a result handler in line 6 which is used in line 9 when including modules in the framework.

The framework's `runSandPass` function triggers the password metering or generation (line 12 in Listing 1). The function does not implement any metering/generation logic itself but uses the `postMessage` to call the respective main function of the included modules and to provide the necessary data, e.g. the password.

When a module returns a result to the main page, the framework calls the respective result handler for providing feedback to the user.

Modularity. SandPass modules are implemented as iframes which create a new and secure execution context for included JavaScript code. Each iframe enables the `sandbox` attribute which limits the access permissions to the *Document Object Model* (DOM) of the sandboxed code to its own unique browser context. Since the purpose of a module is to run JavaScript code, the framework also uses the `"allow-scripts"` keyword to re-enable scripts in the sandbox.

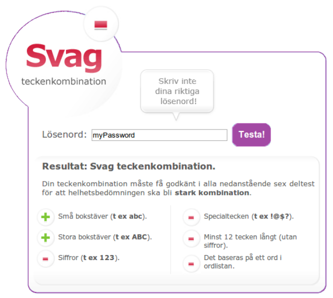
Each module contains a basic HTML document which defines the most restrictive CSP rule, prohibiting access to any network resource from within the iframe.

The framework core and a module communicate through the `postMessage` API function. A module therefore contains a message receive handler. On receiving a message, it calls the modules main function and sends its result back to the framework, again using `postMessage`.

The framework imposes no restrictions on the included JavaScript code, i.e. a web developer can include code from any source as she sees fit in the sandbox. This allows SandPass to be utilized for both password meters and password generators.

4. CASE STUDY

The case study is based on the password meter by the Swedish Post and Telecommunication Agency (PTS). Their password meter web page, shown in Figure 8, contains an input field in which a user can type the password. When the submit button ("Testa!") is clicked, the password is sent to PTS for the actual checks. The reply is an updated web page with feedback based on the results of the algorithms run on server side.



Lösenord: myPassword

Resultat: Svag teckenkombination.

Din teckenkombination måste få godkänt i alla nedanstående sex deltest för att helhetsbedömningen ska bli **stark kombination**.

- + Små bokstäver (**t ex abc**).
- + Stora bokstäver (**t ex ABC**).
- − Siffror (**t ex 123**).
- − Specialtecken (**t ex !@$?**).
- − Minst 12 tecken långt (utan siffror).
- − Det baseras på ett ord i ordlistan.

Figure 8: PTS passwordmeter

Besides syntactical checks, e.g. for the usage of upper- and lower-case letters, PTS uses the open-source library *CrackLib*. CrackLib checks if a password is somehow derivable from any word within a given dictionary. It applies transformations to the given password and checks the result for existence in the dictionary. For example, CrackLib substitutes all digits in "p455w0rd" with their respective *leet speak* [26] counterparts and transforms it to "password" which can be found in a common English dictionary.

CrackLib is fully written in C. For inclusion as a module in SandPass it has therefore been necessary to translate it to JavaScript. Additionally, we've implemented a separate script for the syntactic checks. We have then modified the PTS service to include SandPass, replacing the transmit action of the submit button with the `runSandPass` function of the framework. To provide the same results as the server-side approach, the JavaScript version of CrackLib and the script for syntactic checks have been included as modules. The respective result handler functions have been implemented to update the web page to match the layout of the original service.

As a positive side effect, the good performance of SandPass has allowed us to enable checks on every keystroke made by the user and we have therefore even improved the user experience through immediate feedback. Before, the password had had to be sent to the server first for feedback.

5. EVALUATION

SandPass implements the general requirements and architecture presented in Section 3.1. We have evaluated the framework to see how it prevents the attacks from the attacker model, i.e. the passive network attacker and the second- and third-party web attacker. We've also looked at the practical implications of SandPass for security and performance.

5.1 Security evaluation

Security guarantees. SandPass is a framework which is designed to support the implementation of fully client-side password meters and password generators. Client-side code execution renders leaking a password for analysis to the server or requesting password generation from the server redundant. In fact, the framework defines a CSP rule for included code which completely forbids any network traffic. As an implication, no password information can be leaked to a second party web attacker. Additionally, a passive network attacker cannot sniff for transmitted passwords, which is in particular the case when data is sent over only HTTP.

The framework modules are implemented as sandboxed iframes which are treated by browsers as if their content comes from a unique origin. This behavior in combination with the SOP, implemented and enforced by browsers, prevents the code of a module from accessing the DOM of the main page or even other modules. Therefore, the JavaScript code included in a module is isolated and cannot tamper with the rest of the web page. As mentioned before, the framework prevents communication with external resources by implementing the most restrictive CSP for each module, i.e. it forbids any network traffic from within a module. Therefore, a module cannot leak a password to a server. As a result, modules mitigate the threat imposed by third party web attackers and a web developer can include untrusted scripts as modules without compromising the web page's security. Note that our policy for modules affects only modules but not the rest of the web page and a web developer retains all freedom in its design.

Security considerations. Though SandPass comes with the above security guarantees, there are some security considerations which must be addressed when using web frameworks in general.

Firstly, the administrators of a web server must ensure and maintain the security properties for their servers. For SandPass this means that the integrity of the framework's source code must be guaranteed. Otherwise, an attacker can easily disable security features or even replace the framework code entirely. For password meter/generator scripts, *Cross-Origin Resource Sharing* (CORS) [60] must be allowed to permit web pages of other domains to download the source codes and include them as modules. Otherwise, the scripts will be blocked by the SOP enforcement mechanism in the client's browser.

Secondly, the integrity of the framework code must not only be ensured on the server side, but also during the transmission over the network. To limit the risks of attacks there, the server can be configured to always use encrypted connections, i.e. to use HTTPS.

```
function evilFun(pwd) {
  return "<img src='evil.com/img.png?"+pwd+"' />";
}
```

Listing 2: Example script code for malicious module

```
...
<script language="javascript" />
  function resHandler(result) {
    document.getElementById("myElem").innerHTML =
        result;
  }
  include("http://example.com/m1.js", resHandler, "
      evilFun");
</script>
<p id="myElem"></p>
...
```

Listing 3: Example code vulnerable to code injection

Thirdly, since CSP restricts the modules' network access, it is important that a module can not simply navigate its containing iframe to a page without such a restriction by e.g. manipulating `document.location`. To prevent this, the web page in which SandPass is integrated, must restrict the contents of the module iframes by setting the `child-src` CSP directive to e.g. `self` or `none`.

Finally, though every module is isolated through a sandboxed iframe, the framework allows data to flow to the main page through calls of the `postMessage` function. On receiving the data, the framework runs the respective result handler function, i.e. the handler is executed in the context of the main page. Thus, malicious modules can attack the main page through content injections if the result values are not verified properly. Listing 2 shows a possible attack scenario in which a module's main function named `evilFun` returns a string containing HTML code for an `img` element. In Listing 3, the module's result handler directly assigns the returned value to the element `myElem` on the main page. When executed, this creates a `img` element inside the web page's body. On loading the image source, the password is leaked to `evil.com` as part of the URL. This attack can be avoided by, e.g., validating the return value in the result handler or by assigning the return value to the safer HTML element property `textContent` [61] instead of `innerHTML`.

Besides that a wary administrator considers most of the above security issues for all of the services, using SandPass has the benefit that the code for the modules does not need to be hosted, analyzed for malware, updated, or otherwise maintained. The SandPass consists of a small trusted code base (76 LOC), which can be easily reviewed. The modules can be included safely from third parties in a similar way as it is common practice for libraries such as jQuery.

5.2 Performance evaluation

Our performance evaluation [58] (See Appendix A for more detail) indicates a 106ms overhead in loading time over the baseline of 72ms, mostly due to Cracklib's built-in dictionary. The microbenchmark indicates a factor 34x improvement over the delay experienced during a single password check in our PTS use case, and still a factor 2.5x improvement when the server-side password meter is on localhost. Because loading the password meter only needs to happen once, and will be cached by the browser afterwards, the load time delay is negligible. Combined with the results from the microbenchmarks and security evaluation, using a client-side password meter is beneficial for both security and performance.

6. RELATED WORK

Service providers encourage users to select stronger passwords by guidelines to improve the password entropy [16, 31]. The general problem of defining password strength is addressed by a large body of work, based on both estimating password entropy [8, 7] and on empirical password-guessing techniques and tools [44, 64, 23] that might have access to passwords that have been leaked in the past.

Egelman et al. [15] have studied the impact of password meters on password selection in experiments with user groups. The conclusion is that password meters are most useful when users are forced to change passwords.

de Carné de Carnavalet and Mannan [13] analyze password strength meters on popular web sites. They mention a classification of web sites into client-side, server-side, and hybrid meters, but the focus of their study is the password strength metrics and consistency of outcomes. As mentioned earlier, determining password strength is orthogonal to the goals in this paper. Our focus is on secure deployment of password meters and generators in the web setting.

Among the password meters we discuss in Section 2, a popular one is Dropbox's client-side password meter [67] that includes a number of syntactic and dictionary checks but provides no modular architecture or code isolation. It can be easily plugged into SandPass as a module. Another noteworthy project is Telepathwords [48] that attempts guessing the next character of a password as the user types it.

Language-subset JavaScript sandboxing techniques as [28, 11, 42] require the JavaScript code to be written in a safe subset of JavaScript. Such sandboxes place restrictions on JavaScript code, which third-party code providers are often hesitant to follow. Other JavaScript sandboxing techniques [33, 19, 32, 47, 22] require remote JavaScript code to be rewritten or instrumented on the server. These assume that a developer has access to an execution environment on the server, on which to perform the rewriting. Yet other JavaScript sandboxing techniques as [30, 57, 27] require modifications to the browser, which is a drawback for such a dynamic environment as the Internet, without tight control over browser vendors and versions.

There are approaches to JavaScript sandboxing [53, 41, 29, 1, 18, 24, 66, 21] that require neither server-side modification of code nor specially added client-side support. Instead, they use existing security features available in the browser. Some of these [53, 24, 66, 21] do not offer any means to block network traffic generated by the sandboxed JavaScript, and might allow data to leak out this way. Those sandboxes that can restrict network traffic [41, 29, 1, 18] introduce wrapper code around basic DOM functionality, which can be controlled by a fine-grained control mechanism. SandPass does not require such custom fine-grained control over basic DOM functionality, and uses standard browser functionality instead: the modules execute in a sandboxed iframe with a unique origin and CSP blocks all network traffic. Because of the usage of standard browser functionality, SandPass's codebase is small and can easily be code-reviewed.

7. CONCLUSION

We have presented a large-scale study of web-based password meters and generators. To our knowledge, this is the first such study that addresses secure deployment of password meters and generators on the web. It is alarming that services that are trusted to handle sensitive password information take the liberty to extend the trust to third-party web sites. We find that the vast majority of password meters and generators are open to third-party attacks. Further, we show that some password generators actually leak passwords to third-party web sites via JavaScript. We also find that online password meters are not widely adopted on account registration pages, but most of them also follow unsafe practices allowing credentials to leak away. Another finding is that a substantial fraction of password meters sends passwords to the network, sometimes in plaintext.

As a concrete step to advance the state of the art, we have designed and implemented SandPass, a modular and secure web framework for password meters and generators. By appropriately tuning the CSP policy for iframes, we achieve code isolation for password meter/generator code, enabling security, usability, and performance improvements. We show the usefulness of the framework with a reference implementation that indicates that client-side deployment is advantageous even in cases when password meters include dictionary checks. To further demonstrate the benefits of the framework, we perform a successful case study that allows improving the security, usability, and performance of the password strength meter provided by PTS.

SandPass enables a general technique for modular and secure sandboxing of untrusted code. There is a number of independently interesting applications scenarios for this type of sandboxing. For example, a loan or tax calculator needs access to users' private financial information, which the users might not like to leave the browser.

On the side of practical impact, we are currently in contact with PTS to help improve the current service [54] with our case study as the base.

Acknowledgements. This work was partly funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR. It was also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE).

8. REFERENCES

[1] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC*, 2012.

[2] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a formal foundation of web security. In *CSF*, 2010.

[3] Aldo Cortesi. mitmproxy. `http://mitmproxy.org`.

[4] Ariya Hidayat. PhantomJS. `http://phantomjs.org`.

[5] Badpass: password strength indicator. `https://addons.mozilla.org/en-US/firefox/addon/badpass/`.

[6] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *S&P*, 2012.

[7] W. E. Burr, D. F. Dodson, W. T. Polk, and D. L. Evans. Electronic authentication guideline. In *NIST Special Publication*, 2004.

[8] L. S. Clair, L. Johansen, W. Enck, M. Pirretti, P. Traynor, P. McDaniel, and T. Jaeger. Password exhaustion: Predicting the end of password usefulness. In *ICISS*, 2006.

[9] Content security policy 1.0. `http://www.w3.org/TR/CSP/`.

[10] CrackLib. `http://cracklib.sourceforge.net/`.

[11] D. Crockford. ADsafe – making JavaScript safe for advertising. `http://adsafe.org/`.

[12] CryptoJS. `https://code.google.com/p/crypto-js/`.

[13] X. de Carné de Carnavalet and M. Mannan. From very weak to very strong: Analyzing password-strength meters. In *NDSS*, 2014.

[14] P. Eckersley. How unique is your web browser? In *PET*, 2010.

[15] S. Egelman, A. Sotirakopoulos, I. Muslukhov, K. Beznosov, and C. Herley. Does my password go up to eleven?: The impact of password meters on password selection. In *SIGCHI*, 2013.

[16] Google password help. `https://accounts.google.com/PasswordHelp`.

[17] Html - living standard: The iframe element. `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html`.

[18] L. Ingram and M. Walfish. TreeHouse: JavaScript sandboxes to help web developers help themselves. In *USENIX ATC*, 2012.

[19] Jacaranda. Jacaranda. `http://jacaranda.org`.

[20] C. Jackson and A. Barth. Forcehttps: protecting high-security web sites from network attacks. In *WWW*, 2008.

[21] C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *WWW*, 2007.

[22] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW*, 2007.

[23] P. Kelley, S. Komanduri, M. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *S&P*, 2012.

[24] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *WWW*, 2008.

[25] D. V. Klein. Foiling the cracker: A survey of, and improvements to, password security. *USENIX Security*, 1990.

[26] Leet. `http://en.wikipedia.org/wiki/Leet`.

[27] T. Luo and W. Du. Contego: capability-based access control for web browsers. In *TRUST*, 2011.

[28] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *CSF*, 2009.

[29] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In *Nordsec*, 2010.

[30] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *S&P*, 2010.

[31] Create strong passwords. `https://www.microsoft.com/security/pc-security/password-checker.aspx`.

[32] Microsoft Live Labs. Live Labs Websandbox. `http://websandbox.org`.

[33] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized JavaScript. Technical report, Google Inc., June 2008.

[34] R. Morris and K. Thompson. Password security - a case history. *Commun. ACM*, 22(11):594–597, 1979.

[35] Mozilla. Use bookmarklets to quickly perform common web page tasks. `https://support.mozilla.org/en-US/kb/bookmarklets-perform-common-web-page-tasks`.

[36] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *CCS*, 2012.

[37] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *CRYPTO*, 2003.

[38] Openwall. John the ripper password cracker. `http://www.openwall.com/john/`.

[39] OWASP. HTML5 Security Cheat Sheet. `https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet`.

[40] OWASP. Password storage cheat sheet. `https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet`.

[41] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *ASIACCS*, 2009.

[42] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: type-based verification of JavaScript Sandboxing. In *USENIX Security*, 2011.

[43] Html - living standard: Posting messages. `http://www.whatwg.org/specs/web-apps/current-work/multipage/web-messaging.html`.

[44] R. W. Proctor, M.-C. Lien, K.-P. L. Vu, E. E. Schultz, and G. Salvendy. Improving computer security for authentication of users: influence of proactive password restrictions. *BRMIC*, 34(2):163–9, 2002.

[45] Swedish Post and Telecommunication Agency. `http://www.pts.se/`.

[46] A million tested passwords. `http://www.pts.se/en-GB/News/Press-releases/2012/A-million-tested-passwords/`.

[47] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. In *OSDI*, 2006.

[48] M. Research. Telepathwords: Preventing weak passwords by reading your mind. `https://telepathwords.research.microsoft.com/`.

[49] Syrian Electronic Army uses Taboola ad to hack Reuters (again). `https://nakedsecurity.sophos.com/2014/06/23/syrian-electronic-army-uses-taboola-ad-to-hack-reuters-again/`.

[50] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *IEEE*, 1975.

[51] Sharethis. `http://www.sharethis.com/`.

[52] Taboola. `https://www.taboola.com/`.

[53] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *USENIX Security*, 2010.

[54] Test your password (testa lösenord). `https://testalosenord.pts.se/`.

[55] Tynt. `http://www.tynt.com/`.

[56] B. Ur, P. G. Kelley, S. Komanduri, J. Lee, M. Maass, M. L. Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer, N. Christin, and L. F. Cranor. How does your password measure up? the effect of strength meters on password creation. In *USENIX Security*, 2012.

[57] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: least-privilege integration of third-party components in web mashups. In *ACSAC*, 2011.

[58] S. Van Acker, D. Hausknecht, and A. Sabelfeld. Password meters and generators on the web: From large-scale empirical study to getting it right – full version and code. `http://www.cse.chalmers.se/~andrei/SandPass/`.

[59] Verizon. 2014 data breach investigations report. `http://www.verizonenterprise.com/DBIR/2014/`.

[60] W3C. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`.

[61] W3C. Document Object Model Core – textContent. `http://www.w3.org/TR/DOM-Level-3-Core/core.html#Node3-textContent`.

[62] W3C. W3C Standards and drafts - JavaScript APIs. `http://www.w3.org/TR/#tr_JavaScript_APIs`.

[63] Web Cryptography API. `http://www.w3.org/TR/WebCryptoAPI/`.

[64] M. Weir, S. Aggarwal, M. P. Collins, and H. Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *CCS*, 2010.

[65] J. J. Yan, A. F. Blackwell, R. J. Anderson, and A. Grant. Password memorability and security: Empirical results. *S&P*, 2004.

[66] S. Zarandioon, D. Yao, and V. Ganapathy. Omos: A framework for secure communication in mashup applications. In *ACSAC*, 2008.

[67] zxcvbn: realistic password strength estimation. `https://tech.dropbox.com/2012/04/zxcvbn-realistic-password-strength-estimation/`.

APPENDIX

A. PERFORMANCE EVALUATION

We evaluated the performance of our reference implementation by measuring the loading time of the entire web page and the speedup of a single password check. We use the PTS password meter as the *baseline*, and compare it against a modified version which makes use of SandPass, which we name the *improved version*.

A.1 Loading time benchmark

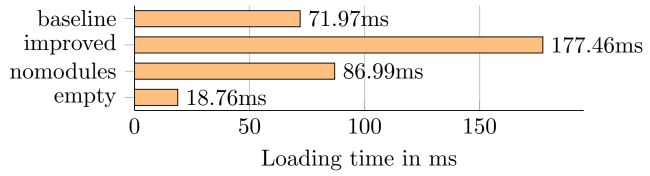


Figure 9: Measurements of the loading time of the baseline PTS password meter and one outfitted with SandPass

To measure the effect of SandPass on the loading time of a webpage using it, we set up the following series of experiments.

We load a webpage into an iframe 1000 times. After each single load, the page inside the iframe sends a message to the outside frame using postMessage to indicate that the loading has finished. When the parent detects this, the next load starts. By recording the time before and after the 1000 loads, an average loading time can be calculated.

All pages are loaded locally to eliminate noisy measurements due to possible temporary network problems on the Internet, and browser caching was disabled.

The experiment is repeated four times for: the original PTS password meter ("baseline"), the PTS password meter outfitted with SandPass implementing the same functionality as the PTS password meters ("improved"), the PTS password meter outfitted with SandPass but without any actual modules ("nomodules") and an empty page ("empty").

The results of these experiments are depicted in Figure 9. The "baseline" loading time is 71.97ms ± 2.1ms (2.1ms being the standard deviation), the "improved" loading time is 177.46ms ± 0.9ms, the "nomodules" loading time is 86.99ms ± 1.0ms and the "empty" loading time is 18.76ms ± 0.6ms.

A.2 Micro-Benchmarks

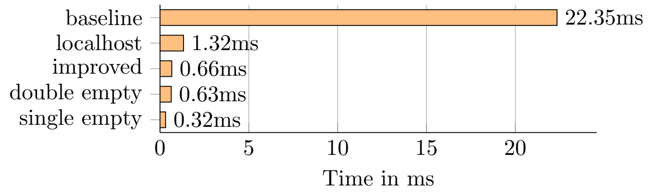


Figure 10: Measurements of the micro-benchmarks comparing the baseline PTS password meter against one improved with SandPass

To measure the speedup gained by SandPass over the baseline, we performed two series of experiments.

First, we measured the network delay experienced in the baseline, by sending 1000 requests to the real PTS password meter using XMLHttpRequests and measuring the average response time. This experiment is called "baseline". While the PTS password meter website might be very responsive to people in Sweden, results may differ in other parts of the world. Therefore, we repeated this experiment and used localhost as the target to get the fastest possible average response time possible for a password meter implemented as the PTS password meter. This experiment is called "localhost".

Secondly, we performed 10000 password evaluations using SandPass and again measured the average delay. We set up three variations of this experiment. The "improved" variation uses SandPass together with two modules, which together implement the same functionality as the PTS password meter. The "double empty" variation has two empty modules, meaning no measurements are performed on the given password. Finally, the "single empty" variation has just a single empty module.

The average response times measured in these experiments are shown in Figure 10. The "baseline" response is 22.35ms ± 0.2ms, the "localhost" response is 1.32ms ± 0.1ms, the "improved" response is 0.66ms ± 0.03ms, the "double empty" response is 0.63ms ± 0.03ms and the "single empty" response is 0.32ms ± 0.01ms.

A.3 Discussion

Fitting a password meter with SandPass comes at a cost, but it is acceptable. The performance evaluation of SandPass clearly shows an improvement.

The PTS case study shows that the loading time increases by about 106ms for an equivalent client-side only password meter implementation. This extra loading time is mainly due to the 53k-words dictionary (622KB) coming with the CrackLib implementation (in total 672KB), which is more than 200 times the size of SandPass' code-base (less than 3KB). By eliminating this heavy implementation from the benchmarks, the "nomodules" measurement shows only a 21% increase in loading time. In addition, the loading of SandPass needs only happen once, and will be cached by the browser afterwards so that there is no noticeable delay for the end user.

For both security and performance reasons, it makes sense to have a client-side password meter instead of a server-side password meter. Comparing the numbers in the PTS case study, the delay caused by the server-side password meter is 34 times larger than the delay experienced trough SandPass. However, our measurements also show that this delay is still 2.5 times larger even when the server-side password meter is on localhost, the best possible location. In essence, CrackLib is based on string modifications and dictionary lookups which is efficiently implemented for common JavaScript engines. As a result, checking a single password using CrackLib with SandPass takes on average only 0.66ms, which allows for checking more than 1500 passwords every second, which is more than adequate for an interactive password meter.