Problem Description

Human Computable Passwords - design and analysis.

Managing passwords is a significant problem for most people in the modern world. This project will be based around the paper "Human Computable Passwords" by Blocki et al. [1], proposing a method for humans to be able to re-compute passwords from public and reliable storage. Passwords are calculated using a memorized mapping from objects, typically letters or pictures, to digits; the characters of the passwords are then calculated in the users head, using a human computable function.

The main objectives of the project can be summarized as the following:

– Understand and compare the "Human Computable Passwords" scheme with other related password management schemes.

– Design and implement a password management scheme applying the ideas of the scheme.

– Analyze if the construction could be utilized to provide secure password management in practical situations.

– Validate if the scheme is feasible to use, comparing the user efforts required to the security rewards.

[1] J. Blocki, M. Blum, and A. Datta, “Human Computable Passwords,” CoRR, vol. abs/1404.0, 2014.

Assignment given: 12 January, 2015
Student: Anders Kofoed **Professor:** Colin Boyd, ITEM

Abstract

Preface

Contents

List of Figures **viii**

List of Tables **viii**

List of Algorithms **1**

1 Introduction **3**

1.1 Motivation . . . 3
1.2 Related Work . . . 3
1.3 Scope and Objectives . . . 3
1.4 Method . . . 3
1.4.1 Development . . . 3
1.4.2 Experiments . . . 3
1.5 Outline . . . 3

2 Background **5**

2.1 Passwords . . . 5
2.1.1 Password strength . . . 6
2.1.2 Attacks . . . 7
2.1.3 Alternative authentication methods . . . 7
2.2 Human Behavior . . . 8
2.3 Password Management . . . 8
2.4 Usability and Security Challenges . . . 8
2.5 Browser Extensions . . . 8
2.5.1 Extension Security . . . 9
2.6 Usability Model . . . 13
2.7 Security Model . . . 13

3 Human Computable Passwords **15**

4 Application Design **17**

4.1 Architecture . . . 17

5 Implementation **19**
5.1 HumanPasswordExtensionName 19
5.2 Experiment 19

6 Conclusion **21**

References **23**

List of Figures

2.1 Rainbow table 8
2.2 Chrome extensions browser action and content scripts. 11
2.3 Chrome extension architecture. 12

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

[2]

1.2 Related Work

1.3 Scope and Objectives

1.4 Method

1.4.1 Development

1.4.2 Experiments

1.5 Outline

Chapter 2
Background

2.1 Passwords

Passwords are the common way of authenticating users upon access to sites on the Internet, the idea is that only the user and the target service knows the password, and the user have to provide the correct password before access are granted. Passwords are a much discussed theme and claiming that passwords are usually not used in the correct manner is not an overreaction. The main problem seems to be the fact that good passwords and the human memory does not go well together. For passwords to be sufficient as authentication each user has to be forced into using long complex password, or even use one generated for them, with the problem being that it is easily forgotten. Further more, if a user was able to memorize *one* "good" password, he will probably use this for all of his accounts, so that if one of the services is compromised and user information leaked, all his accounts may be compromised. With all of this in mind it is easy to say that everybody should use complex, unique passwords for each account, but in practice this is not feasible. Florêncio and Herley [3] conducted a large scale study of passwords habits in 2007, revealing that a user on average has 25 different accounts protected by passwords. On average these sites are protected by about 7 distinct password, where 5 of them are rapidly re-used.

Password authentication requires the authenticating server to store something related to the password, if this is stolen the password will in most cases be compromised as well, even if the server did not store the clear text password, attackers will, in most cases, be able to retrieve the password eventually. After obtaining the username and password for one service the attacker would try this user data on other services and compromise these as well.

If a user was to have different passwords for each site, these might still easily be compromised. Ives et al. [4] discuss this "domino effect", where intrusion to one domain can compromise several others, if users have re-used passwords. A normal user will typically try to log in by trial-and-error [5], if the first password does not work, the user will try with another of his passwords. This way may passwords might be lost through phising attacks where a user is tricked into trying to log in to a fake site. It is thus clear that some kind of system is needed to allow a human user to manage strong passwords. The best case would be if each user, for each of his 25 accounts had a unique password of satisfying strength, this is of course not possible.

2.1.1 Password strength

How to measure the strength of passwords is a well known and discussed problem, but the general idea is that password strength is related to how strong a password is against brute force attacks. Length and complexity is the most thought of parameters to measure such strength. A perfect password would thus be one as long as allowed by the system consisting of random characters from all possible characters, this one would also be changed frequently. All these characteristics despite how the human brain works. Yan et al. [6] investigate the trade of between security of passwords versus memorability allowing humans to remember them. An important regarding this trade-off is that most sites applying advices and policies on how to create strong passwords, does not take into account if the advices passwords are hard or easy to remember. Point being that there is no point in having a strong password if the user is going to forget it. They suggest that passwords should appear random but be constructed using a mnemonic structure such ass passphrases. The idea here is to generate a random looking password by memorizing a familiar sentence and using the first letters of each word as the password. Florêncio et al. [7] investigate another matter; do strong passwords accomplish anything? The point being that no matter how long and complex password users chose they are still subject to the most dangerous and common threats (phising, keylogging or access attacks), as discussed in the previous section. The reason for enforcing strong passwords seems to be to protect against bulk guessing attacks, against other attacks, typically offline attacks, shorter passwords is usually sufficient.

Password strength meters are a common way used by many sites on the Internet to aid their users when selecting passwords.

in progress: password strength meters

The conclusion on what "good" passwords are, is not clear, but the one thing agreed upon seems to be that re-use of passwords are the biggest threat. It is a fact that the human brain is not capable of remembering different passwords for each account on the Internet, thus the need for an aiding application such as the one discussed in this project.

2.1.2 Attacks

Passwords are often the only barrier stopping adversaries from directly accessing the accounts of a user. The combination of user name and password are the easiest point of entry to access, and thus the first logical point of attack. There are several methods used to attack password authentication, trying to retrieve passwords. The most important attacks and their respective mitigation technique [8, 7], will be discussed in the following section.

Capturing of passwords directly from the server responsible for the authentication involves the attack acquiring password data through breaking into the data storage, eavesdropping on communication channels or through monitoring the user by other means. The first most basic threat are to simply steal the stored password from an insecure server, this would require a weak configured server storing the passwords in plain text. This is mitigated by storing only cryptographic hashes of passwords, which allows the server to authenticate users while preventing attackers from determining the actual passwords without *cracking* the hashes.

Cracking requires the attacker to go through several steps. First acquiring the hash of a user account or a whole file of hashes for a site. Next, one would try to find a sequence of string yielding the same hash as the actual password. How hard it is to crack a hash depends on the strength of the password and can be mitigated by choosing strong passwords and changing these frequently. *Rainbow tables* [9] are a technique employed by attackers to speed up this process. Rainbow tables are precomputed table of hashes, allows the attacker to compute a set of hashes once and use these values several times, thus providing a space-time trade-off. This involves using more space, since all the computed hash values would have to be stored somewhere, but allowing a much shorter computation time to brute-force a hash. The technique stores chains of hashes as shown in figure 2.1, storing only the first and last value of the chain. The attacker then search for a given hash in the set of end points, if no match are found the hash function is applied and a new search conducted. This process continues, until a match is found, the plain text is then then computed from the start value of the chain, applying the hash function the same amount of times it took to find a match in the chain.

rewrite - more precise

2.1.3 Alternative authentication methods

alternative auth: graphical passface etc.

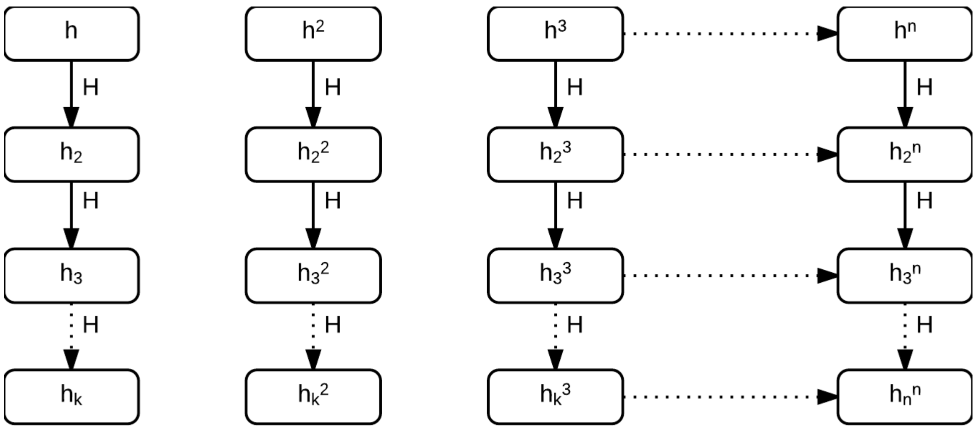

Figure 2.1: Rainbow table

2.2 Human Behavior

"Good passwords" as discussed in 2.1.1 does not go well with the human memory. The first limitation which will be an important property later are the limitation to how much data we can store in immediate memory, this limit was showed to be 7 chunks of data at once [10]. This data can not be from a random selection which is what a good password requires.

in progress: human behaviour

2.3 Password Management

2.4 Usability and Security Challenges

2.5 Browser Extensions

Modern computer users shift towards doing more and more work through their web browsers. Web applications have become popular due to the ubiquity of browsers, thus allowing web apps to run anywhere. A web app can run at any platform running a web browser, allowing the application to run on multiple platforms as well as different devices. Updates can be applied quickly without having to distribute patches to a possibly huge amount of devices.

Browser extensions add additional features to the web browser allowing the user to tweak the experience of the web pages visited. Typical examples are extensions adding to, or tweaking already present features of the browser such as changing how bookmarks are managed, or adding additional features such as blocking advertisements. Lately browser extensions have been extended even further allowing standalone applications to be developed running as native applications [11]. This allows developers to create desktop apps using the same technology as in web apps, mainly HMTL5, Javascript and CCS.

This chapter will present Google chrome browser extensions, including architecture and security mechanisms.

2.5.1 Extension Security

Browser extensions introduce some security concerns which must not be forgotten while developing applications using this environment. Chrome extensions run in the browser with access to both the DOM of the active page as well as the native file system and connected devices. The overall architecture of the application is summarized in Figure 2.3 and described in the chrome extensions documentation [12]. This section will describe the architecture considering security concerns relevant when developing chrome extensions which handles sensitive data such as passwords.

Earlier extensions written for IE and Firefox ran in the same process as the browser and shared the same privileges. This made extensions an attractive entry point for attackers, since a buggy extension could leave security holes leaking sensitive information or even provide an entry point to the underlying operating system. For these browsers several frameworks for security have been proposed [13, 14], trying to mitigate vulnerabilities is browser extensions. The chrome extensions architecture is built from scratch with security in mind, chrome uses a permission system following three principles [15]; least privileges, privilege separation and strong isolation.

Least privileges specifies that extensions should only have to privileges they need functions, not share those of the browser. The privileges of each extension are requested in the *manifest* file [16]. This json file needs to be included in all chrome extensions, and consist of all the permissions needed by the extension as well as some meta data and version information. This is done to prevent compromised extensions from exploiting other permissions than those available at runtime. An example of a manifest file can look like this:

```
{
    "name": "Example extensions",
    "description": "An example extensions to demonstrate how the
                   manifest file works.",
    "version": "1.2",
    "manifest_version": "2"
    "background_page": "main.hmtl",
    "permissions": [
        "bookmarks",
        "storage",
        "https://*.ntnu.no"
    ]
}
```

This extension has specified access to the bookmarks API, chrome local storage and all sub domains of ntnu.no. Extensions can request different permissions in the manifest file including web site access, API access and native messaging [16]. If an extension contains weaknesses it will not compromise any other parts of the system not covered by the specified privileges. For the least privileges approach to work properly each developer should only request the permissions needed, Barth et al. [17] examined this behavior and concluded that developers of chrome extensions usually limit the origins requested to the ones needed.

Privilege separation . Chrome extensions are as mentioned divided into components; content scripts, extension core and native binaries. The addition of native binaries allows extensions to run arbitrary code on the host computer, thus posing a serious security threat. This project does not use this permission, this component will thus not be mentioned from now on. Content scripts are javascript files allowing extensions to communicate with untrusted web content of the active web page. This scripts are instantiated for each visited web page and has direct access to the document object model (DOM) of these, allowing both monitoring and editing of DOM elements. To be able to inject content scripts to a visited page, the origin of the site has to be added to the manifest file. Other than this permission, content script are only allowed to communicate with the extension core. It is important that the privileges of these scripts are at the minimum level since they are at high risk of being attacked by malicious web sites [18], due to the direct interaction with the DOM.

The extension core is the application interface responsible for interaction with the user as well as long running background jobs and business logic. The core is written in HTML and javascript and are responsible for spawning popups and panels, as well as listening for browser action. The typical way to activate a extensions is by clicking a icon in the navigation bar, which then activates either a popup or a detached panel. The core is the components with the most privileges as it does not interact with any insecure content directly, only through direct messaging to a content script or using http requests if the target origin is defined in the manifest. In addition to this the core has access to the extension APIs, these are special-purpose interfaces providing additional features such as alarms, bookmarks, cookie and file storage. The APIs are made available through the manifest file and only those specified there can be used. Figure 2.2 illustrates the interaction between the background page, content scripts, panels and active web page. The information flow starts by clicking the extension's icon in the navigation bar which launch the background page spawning a panel in the browser. A content script is injected in the current web site (google.no in the example), the script now have access to the DOM of this site and can communicate with the background which in turn can update the panel.

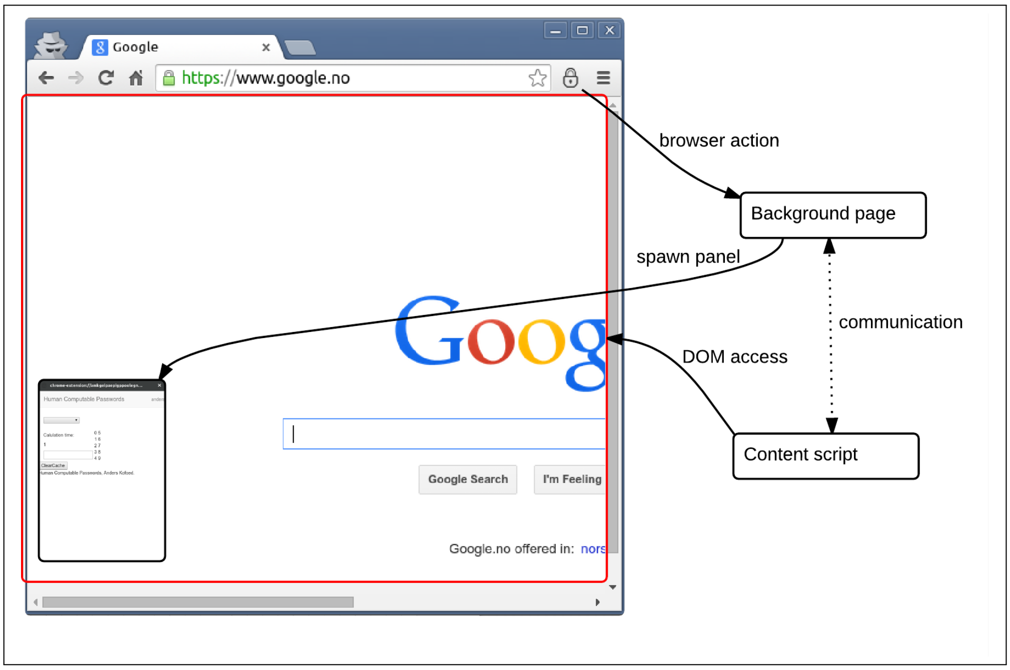

Figure 2.2: Chrome extensions browser action and content scripts.

Process isolation are a set of mechanisms shielding the component from each other and from the web. Usually when javascript is loaded from the web the authority of the script is limited to the origin from where the script is loaded [17]. Since the scripts used by the extensions is loaded from the file system, it does not have a origin in the same sense, and thus needs to be assigned one. This is done by including a public key in the url of the extension, allowing a packaged extensions to sign itself, freeing it from any naming authority or similar. The public key also enables usage of persistent data storage, since the origin of the extension can stay the same throughout updates and patches. This wouldn't be possible otherwise since the chrome local storage API relies on origin.

The different components also run in different processes. The content scripts are injected and ran in the same process as the active web page, while the core run in its own process started when the extension is initiated.

paragraph not finished. Cross-origin js and malicious web site operators.

Finally content scripts are ran in a separate javascript environment isolating it from the possibly insecure environment of the web site. The environment of the content scripts are called isolated world, which in practice is a separate set of javascript objects reflecting the ones of the underlying DOM of the web page. This means that the content script can read and edit the DOM of the page it is injected into, but not access variables or javascript functions present in the web page. Both the page and the content scripts sees no other javascript executing in their own isolated world, but they share the same DOM [19].

Figure 2.3 illustrates the architecture of chrome extensions with process isolation and isolated worlds.

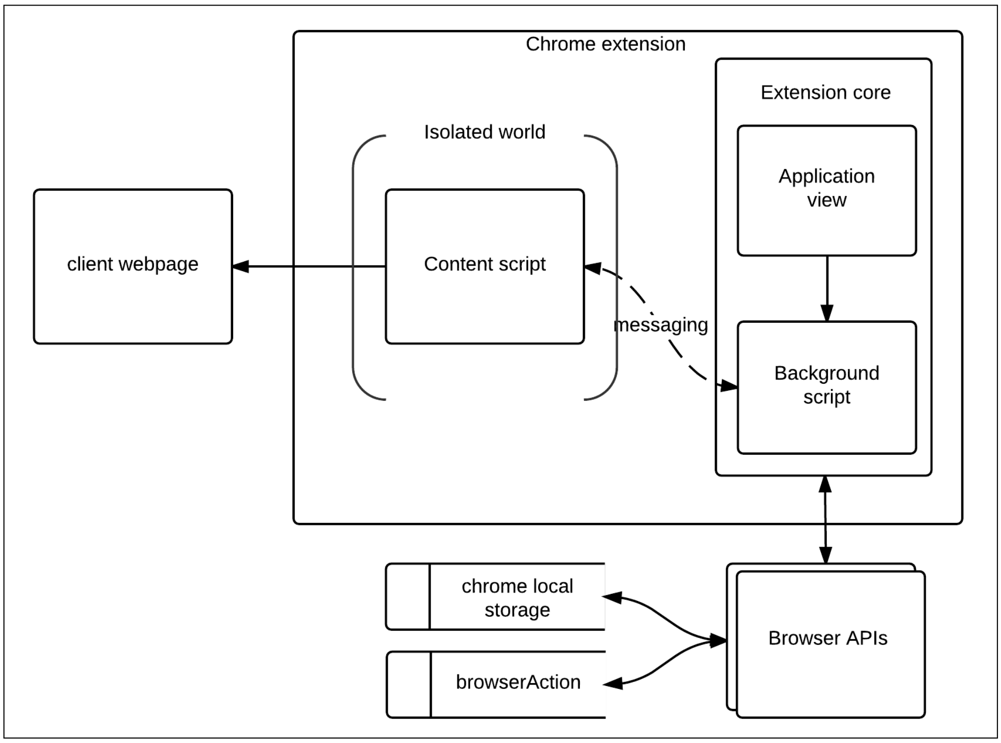

Figure 2.3: Chrome extension architecture.

2.6 Usability Model

2.7 Security Model

Chapter 3

Human Computable Passwords

Chapter 4 Application Design

4.1 Architecture

Chapter 5
Implementation

5.1 HumanPasswordExtensionName

5.2 Experiment

Chapter 6

Conclusion

References

[1] J. Blocki, M. Blum, and A. Datta, “Human Computable Passwords,” *CoRR*, vol. abs/1404.0, 2014.

[2] J. Blocki, *Usable Human Authentication: A Quantitative Treatment.* PhD thesis, School of Computer Science, Carnegie Mellon University, 2014.

[3] Z. Liu, Y. Hong, and D. Pi, “A Large-Scale Study of Web Password Habits of Chinese Network Users,” *JSW*, vol. 9, no. 2, pp. 293–297, 2014.

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

[5] T. Acar, M. Belenkiy, and A. Küpçü, “Single password authentication,” *Computer Networks*, vol. 57, no. 13, pp. 2597–2614, 2013.

[6] J. Yan, A. Blackwell, R. Anderson, and A. Grant, “The memorability and security of passwords: some empirical results,” *Technical Report-University Of Cambridge Computer Laboratory*, p. 1, 2000.

[7] D. Florêncio, C. Herley, and B. Coskun, “Do Strong Web Passwords Accomplish Anything?,” in *2nd {USENIX} Workshop on Hot Topics in Security, HotSec’07, Boston, MA, USA, August 7, 2007*, {USENIX} Association, 2007.

[8] K. Scarfone and M. Souppaya, “Guide to enterprise password management (draft),” *NIST Special Publication*, vol. 800, p. 118, 2009.

[9] G. Brose, “Rainbow Tables,” in *Encyclopedia of Cryptography and Security, 2nd Ed.* (H. C. A. van Tilborg and S. Jajodia, eds.), pp. 1021–1022, Springer, 2011.

[10] G. A. Miller, “The magical number seven, plus or minus two: some limits on our capacity for processing information.,” *Psychological review*, vol. 63, no. 2, p. 81, 1956.

[11] “A new breed of chrome apps.” http://chrome.blogspot.no/2013/09/a-new-breed-of-chrome-apps.html, 2013.

[12] Google, “What are extensions?.” https://developer.chrome.com/extensions/overview. Accessed 2015-03-02.

[13] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan, “Enhancing web browser security against malware extensions,” *Journal in Computer Virology*, vol. 4, no. 3, pp. 179–195, 2008.

[14] M. Dhawan and V. Ganapathy, “Analyzing Information Flow in JavaScript-Based Browser Extensions,” in *Twenty-Fifth Annual Computer Security Applications Conference, {ACSAC} 2009, Honolulu, Hawaii, 7-11 December 2009*, pp. 382–391, 2009.

[15] L. Liu, X. Zhang, V. I. H. R. Center, G. Yan, and S. Chen, “Chrome Extensions: Threat Analysis and Countermeasures,”

[16] Google, “Manifest file format.” https://developer.chrome.com/apps/manifest. Accessed 2015-03-04.

[17] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, “Protecting Browsers from Extension Vulnerabilities,” in *Proceedings of the Network and Distributed System Security Symposium, {NDSS} 2010, San Diego, California, USA, 28th February - 3rd March 2010*, The Internet Society, 2010.

[18] L. Bauer, S. Cai, L. Jia, T. Passaro, and Y. Tian, “Analyzing the dangers posed by Chrome extensions,” in *{IEEE} Conference on Communications and Network Security, {CNS} 2014, San Francisco, CA, USA, October 29-31, 2014*, pp. 184–192, 2014.

[19] Google, “Content scripts.” https://developer.chrome.com/extensions/content_scripts. Accessed 2015-03-05.