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

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 know the password, and the user has to provide the correct password before access is 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 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. Furthermore, 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 phishing 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 naive approach says that password strength is related to how strong a password is against brute force attacks [6]. Length and complexity are 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 challenge how the human brain works. In addition to the objective strength of the password, techniques making it harder for a computer to repeatedly try different passwords may be applied. Such techniques include CAPTACHAS [7] which are puzzles supposed to require a human to be able to solve, thus making brute force using a computer harder. Or techniques making it easier for the user to remember multiple strong passwords.

Yan et al. [8] investigate the trade-off between security of passwords versus memorability allowing humans to remember them. An important regarding this trade-off is that most sites applying advice and policies on how to create strong passwords, do not take into account if the recommended passwords are hard or easy to remember. There is no point in having a strong password if the user is going to forget it. Yan et al. 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. [9] investigate another matter; do strong passwords accomplish anything? The point is that no matter how long and complex password users chose they are still subject to the most dangerous and common threats (phishing, 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. Common meters use colored progression bars together with a word or short comment stating if the entered password is evaluated as "bad", "poor" or "strong". Ur et al. [10] found that password meters actually lead users into choosing longer and stronger passwords, but they also argue that enforcing such policies might frustrate users and possibly lead them into writing passwords down, use weak password management schemes or re-use the same password across many sites. The most common requirements used by passwords meters of known web services can be summarized as the following [11]:

– Length and character selection are part of most password meters. It is normal to disallow passwords shorter, and sometimes longer, than a given range, which may vary. A variation of different characters can be required, namely different kinds of symbols or capital letters. Spaces may be allowed in between other characters, at the start or end of the password, or not at all. Some sites checks for sequences of the same character as well.

– Personal information. Information registered by the user are evaluated by some meters, typically name, email and date of birth are checked against in original and transformed forms. This means that a password like "4nD3r?1991" ("anders1991" transformed) which look strong, will be evaluated as weak.

– Dictionary checks. Make sure that the password does not include any dictionary words by matching it with a dictionary of common words.

Entropy

write about entropy, [12, 13]

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 [14, 9], 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 is 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 strings 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* [15] 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 searches for a given hash in the set of end points, if no match is found the hash function is applied and a new search conducted. This process continues, until a match is found, the plain text is 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

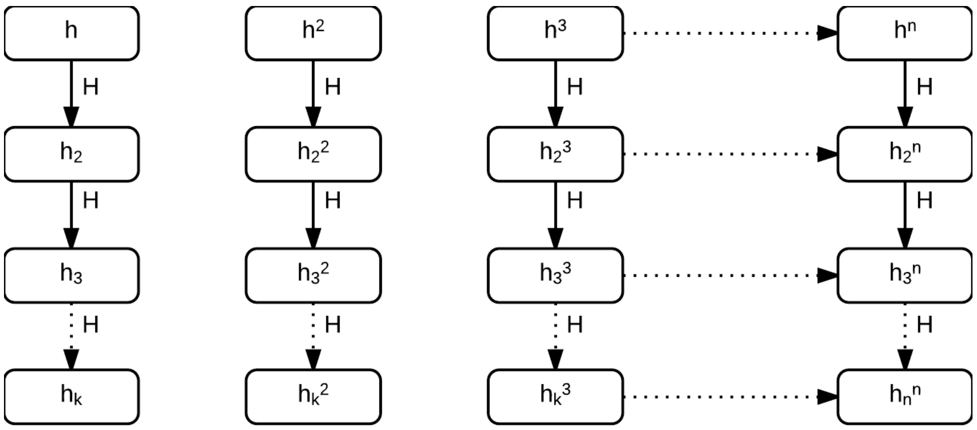

Figure 2.1: Rainbow table

2.1.3 Password Storage

hashing, salt etc..

2.1.4 Alternative authentication methods

alternative auth: graphical passface etc.

2.2 The User

"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 [16]. This data can not be from a random selection which is what a good password requires.

in progress: human behaviour

2.3 Password Management

As seen in the previous sections passwords introduce a dilemma as passwords are supposed to be hard to "guess" and thus hard to remember. The naive solution to this problem is to either use one password for many accounts or to write down passwords, even tough most users understand that this solution is bad, users will still do it if the password policy required is too complex. To make the process of managing passwords easier several techniques and tools have been suggested. The main techniques used to help manage passwords online are [17]:

- Password creation and memorization techniques [18]. One way to keep apparently strong unique passwords for all services is to use a base password and apply function using unique information from the different services [1]. The base can be any seemingly random word which is easy to remember, best practice would be to take a sentence or phrase and use the first letters of each word. “May the force be with you.” would yield "mtfbwy" as base, then a rule could be to add the first and last two letters of the service name. With this rule the password for facebook.com would be famtfbwyok.

- Password manager software. Applications meant to keep passwords safe for the user. These applications can either be stand alone programs or, more common, browser extensions such as LastPass [2]. LastPass provides an user interface to generate and store passwords for online services, as well as form fillers to enter them when logging in. The passwords are encrypted using a

[1]How to Update Your Insecure Passwords and Make Them Easy to Use.http://lifehacker.com/5631203/how-to-update-your-insecure-passwords-and-make-them-easy-to-use

[2]LastPass https://lastpass.com/

master password protecting the user credentials against both server leakage and insiders accessing the data.

– Built-in browser functions. Most modern browsers provide "remember password" functions.

The two latter may seem the easiest to use but requires the user to trust that the implemented systems are secure enough to prevent adversaries, both insiders and outsiders, stealing credentials stored by the services. If an account storing all the usernames and passwords of a user where to be compromised, all the sites would be compromised. Zhao et al. [19] identified several vulnerabilities in the LastPass implementation, even though no known breaches has been reported.

password management: in progress

2.4 Usability Model

[20]

2.5 Security Model

[20]

Chapter 3 Human Computable Passwords

The previous chapter concludes that managing passwords for online accounts has become a major issue for the modern Internet user. It seems to be impossible to remember and maintain enough strong passwords to keep all accounts secured. The scheme presented in this chapter is designed to help the user maintain and remember multiple strong passwords, while also protecting these after multiple passwords breaches. Human computable passwords take advantage of the human brain allowing users to calculate passwords from public challenges, using their own mind to do so.

The Human Computable Password management scheme is proposed by Blocki et al. [1]. In addition to the scheme itself, the proposal introduce security and usability notions used to analyze the proposed scheme. This chapter will describe the scheme as well as associated security and usability concerns. The first section consists of definitions and notations as used in [1] to describe the scheme. Next, human computable functions are introduced as this is the main component used in the password management scheme which will be described after that. Finally the usability and security of the scheme is discussed and requirements defined.

3.1 Password Management Scheme

The main idea of the Human Computable Password management scheme is to have a set of challenges stored in persistent memory, typically on a computer or even a piece of paper. The user then uses a mapping and a function to calculate the responses to each challenge, which eventually gives the password. It is worth noting that this is different from other "traditional" passwords managers, in that the passwords are not stored, only challenges "helping" the user remember his passwords. To create a new password, a random challenge is generated, the user then computes the password from this using a memorized secret mapping. To reproduce the password later, the same challenge is displayed to the user which then can calculate the same password. This procedure is explained further in section 3.2, and in algorithms 3.2 and 3.1.

3.1.1 Definitions and Notation

Memory types considered are either *persistent* or *associative* memory [21]. This project follow the settings of Blocki et al. [20, 1] where persistent memory are equal to writing something down or somehow storing it reliably, but not securely. When talking about persistent memory, it can be assumed that this is publicly available, or at least that an adversary has undisclosed access to the data. This should be emphasized since this is a strength to the scheme, nothing needs to be kept secret after establishing the needed prerequisites.

Associative memory is the memory of each of the users, namely their human memory. This memory is different from the persistent memory in that it is totally private but needs to be rehearsed to not lose data. In a password management scheme rehearsing should optimally be part of the natural activity of a user. The best case would be if a user could rehearse and keep all his passwords in associative memory by simply logging in to his accounts as normal. This is a central challenge for all password schemes [20].

The password management scheme uses a random mapping between a set of objects to single digits which has to be memorized by the user. This mapping is denoted as $\sigma : [n] \to \mathbb{Z}_d$. Random challenges will be generated and stored in persistent memory. Let $C \in X_k$ be a random challenge chosen from X_k, the space of k possible variables. Now $\sigma(C) \in \mathbb{Z}_d^k$ is the mapped variables corresponding to challenge C. The set of challenges C can be any type of object, such as pictures letters or digits, with the mapping σ always being to digits. One of these challenges, C, will be referred to as a *single digit challenge*. The function $f : \mathbb{Z}_d^k \to \mathbb{Z}_d$ is a human computable function as discussed in the next section. The user will now be able to calculate the response to a challenge C by computing $f(\sigma(C))$. A complete password challenge, $\vec{(C)} = (C_1, \ldots, C_t) \in (X_k)^t$, will consist of t separate, single digit challenges. The response to $\vec{(C)}$, namely $f(\sigma(\vec{(C)}))$, is the complete password.

The password management scheme works by generating one challenge, $\vec{(C)}$, for each of a user's accounts $A_1, \ldots, A_m$. The challenges $\vec{(C_1)}, \ldots, \vec{(C_m)} \in (X_k)^t$ are stored in persistent memory. When a user wants to log in to a service he is shown the challenge corresponding to that account, the user then calculate the responses to all the single digit challenges, yielding the password.

3.1.2 Human Computable Functions

At the core of the scheme is a human computable function f and the memorized mapping σ. The scheme require the composite function of these two $f \circ \sigma$ to be *human computable*, which simply means that the function should be easily computable in the head of the user. To fulfill this requirement the function can't involve many operations, since the complexity and thus computation time would be too high. As shown by Miller [16], a human can only store 7 ± 2 pieces of information at a given time, on the other hand humans are quite good at simple operations such as addition modulo 10. In example "1+6+5+3+8+9+3+1+4+6+7+7+6 mod 10" would be easy for most humans to compute by simply doing one operation at a time, updating the answer after each. With this approach only one piece of information would be stored in memory of the user at any time. The problem with such an expression is the amount of terms.

The requirements needed for a function to be human computable can thus be summarized as the following, and formalized in Requirement 1:

– Can only involve "simple" operations, mainly addition and recalling from long-term memory.

– Limited amount of terms.

– Limited amount of operations.

Remark 1. All operations used in the human computable functions discussed in this project are modulo 10, this is the most natural for most humans.

Requirement 1. Function f is said to be $\hat{t}$-human computable if a human can compute it in his head in $\hat{t}$ seconds.

Blocki et al. [1] believe that a function f is human-computable if it can be computed using a fast streaming algorithm, meaning that the input is presented as a sequence of objects that only can be evaluated once. The algorithm would have to be simple since humans are not good at storing intermediate values [16]. Typical operations fast enough for the human to compute in his head is addition modulo 10 which is natural for most humans to do quickly, and recalling a mapped value $\sigma(i)$ from memory.

Definition 3.1. A function f is $(P, \tilde{t}, \hat{m})$-computable if there is a space $\tilde{m}$ streaming algorithm computing f using $\hat{t}$ operations from P.

Remark 2. Space $\tilde{m}$ means that the algorithm requires no more than $\tilde{m}$ memory slots during calculation. Slots are typically used for storing values and executing primitive operations such as addition [22].

As for the primitive operations in P, the following are considered:

– Add takes two digits x_1 and x_2, and returns the sum $x_1 + x_2$ mod 10.

- Recall returns the secret value $\sigma(i)$ corresponding to an input index i. The mapping σ is memorized by the user, allowing the recall operation to be done quickly in the users head.
- TableLookup involves looking up the x'th value from a table of 10 indices.

Example 1. The function $f \circ \sigma(i_1, \ldots, i_5) = \sigma(i_1) + \cdots + \sigma(i_5)$ is $(P, 9, 3)$-computable, since it requires 9 operations from P, 5 recall operations and 4 add operations. $\tilde{m} = 3$ since a sequence of additions $i_1 + \cdots + i_n$, requires one slot for storing the sum, one slot for storing the next value in the sequence and one slot to execute the addition.

3.1.3 Secure Human Computable Functions

Blocki et al. [1] suggest a family of human computable functions defined as follows.

$$f_{k_1,k_2}(x_0, \ldots, x_{9+k_1+k_2}) = x_j + \sum_{i=10+k_1}^{9+k_1+k_2} x_i \quad mod \quad 10,$$

$$\text{with } j = \sum_{i=10}^{9+k_1} x_i \quad mod \quad 10 \quad \text{and} \quad k_1 > 0,\, k_2 > 0$$

This project will use one of these functions, with $k_1 = k_2 = 2$. From now on this will be the function referred to as f, the function is defined in definition 3.2. For an in depth analysis of the function see "Usable Human Authentication: A Quantitative Treatment" [2]. Blocki argues that an adversary would have to see $\tilde{\Omega}(n^{1.5})$ challenge-response pairs to be able to start recovering the secret mapping σ. A realistic mapping σ would probably consist of no more than 100 object to digit mappings. A secret mapping consisting of $n = 100$ mappings would require an attacker to steal 1000 challenge-response pairs (100 accounts given password length of 10) to recover the secret mapping. In practice this might be the tricky part of the scheme, since memorizing a mapping of 100 object-digit mappings might be possible, it might be more reasonable to use a smaller set of mappings which will lower the security of the scheme. An example mapping which could be feasible in practice is characters to single digits, with characters from the alphabet and digits between 1 and 10. This mapping would yield $n = 26$ which would require an attacker to recover significantly less challenge-response pairs. With $n = 26$ the amount is down to 133 compared to the 1000 with $n = 100$. Still, this would require to fully compromise 13 or more accounts with password lengths of 10 characters.

Write about how human computable function f is used together with mapping function σ to create accounts and authenticate, algorithms 3.1, 3.2 and 3.3

Definition 3.2. $f(x_0, x_2, \ldots, x_{13}) = x_{(x_{11}+x_{10} \quad mod \quad 10)} + x_{12} + x_{13} \quad mod \quad 10$

Definition 3.3. $f \circ \sigma(x_0, x_2, \ldots, x_{13}) = \sigma(x_{(\sigma(x_{11})+\sigma(x_{10}) \mod 10)}) + \sigma(x_{12}) + \sigma(x_{13}) \mod 10$

Security parameters

fitting tittle?

There are some interesting trade-offs related to the parameters of the human computable passwords scheme. A bigger set of mappings makes it increasingly hard to recover the secret mapping, but it becomes equally hard to memorize and rehearse it. It is reasonable to say that complexity of a mapping grows linearly with the number of mappings n, and the resistance versus attackers grows polynomially, thus much quicker than the complexity, see Figure 3.1. In other words, for each mapping added to σ n is increased with one and the security multiplied 1.5 times. The trade-off which would have to be evaluated for each user is then; how much effort is the user willing to put into memorizing the mappings versus how secure he wants it to be. This should be evaluated in regards to how "important" the passwords and the accompanying accounts are, and how many accounts the user plans on having.

Another trade-off is actually between password length and number of accounts which would have to be stolen. Assume that a set of mappings with $n = 26$ is used, the number of accounts needed to recover the mapping is then a function of the password length as seen in Figure 3.2. If the passwords are very long only a few logins would have to be stolen to recover σ. This is important to take note of since one of the main strengths of the password scheme is that even if one account is compromised all the others are still secure since each site has a different, "unrelated" password. If the revelation of only a few accounts could compromise the secret mapping, all the passwords of the user might easily be lost.

A user requiring very secure passwords might generate very long passwords of 20+ characters for each of his accounts. If, by chance, the mapping was shorter than suggested, all these "strong" passwords might be lost if only a few of them was to be compromised through a password breach. Users are not advised to use shorter passwords, longer passwords are always better and the possibility of a password being cracked drastically decreases with the length of the password. The point is that the secret mapping needs to be long enough to support the length and number of passwords a user wants to generate using the scheme.

The number of accounts needed to recover the mapping can be used as a practical way of describing the security of the scheme, Theorem 3.4 defines this parameter as an inequality reliant on the password lengths x and the number of mappings y. Figure 3.3 illustrates the relationship between password lengths and number of

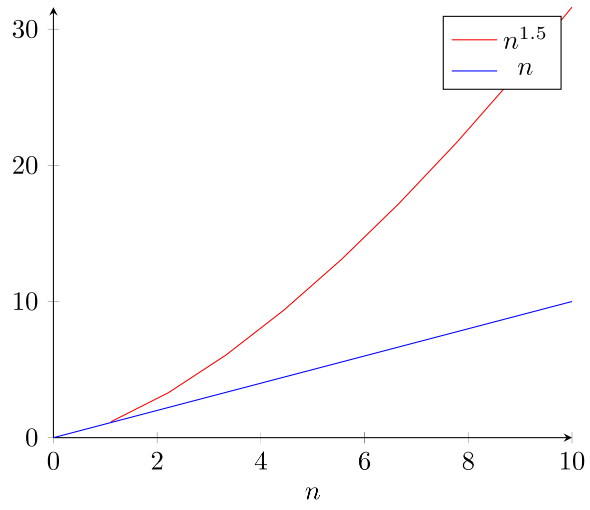

Figure 3.1: Number of challenge-response pairs required to recover mapping σ as a function of the size of the mapping n.

mappings needed to achieve different levels of security. How high the parameter $\hat{a}$ depends on how long and how many passwords the user intends to use.

Theorem 3.4. *The security of human computable function together with a mapping function $f \circ \sigma$, as defined in Theorem 3.3 and subsection 3.1.2, can be described through how many accounts $\hat{a}$ that needs to be compromised to recover the secret mapping given passwords of length x and y mappings in σ. $\hat{a}$ is then defined as $\hat{a} < \frac{y^{1.5}}{x}$.*

Example 2. A user plans on having passwords of length 20 for all of his many important accounts, and wants these to be securely stored even if it requires him to use more time on rehearsal. In this case assume that the user wants his accounts be secure even if 100 accounts was leaked. Using Theorem 3.4 with $x = 20$ and $\hat{a} = 100$, gives $100 < \frac{y^{1.5}}{20} \implies y > 159$. This means that the user would have to memorize at least 159 unique random mappings to achieve the desired level of security against leakages.
If the user was to save only a couple of shorter passwords, in example requiring only security allowing loss of only 20 accounts before possibly revealing the mapping and passwords of length 15, he would need to memorize at least 45 mappings.

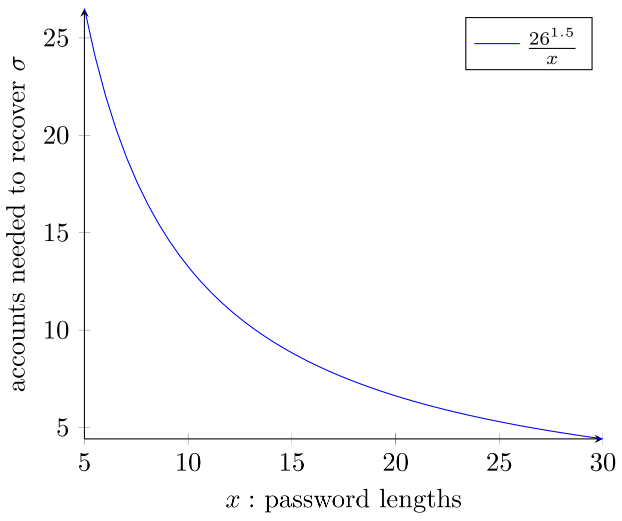

Figure 3.2: Number of accounts needed to recover the mapping σ.

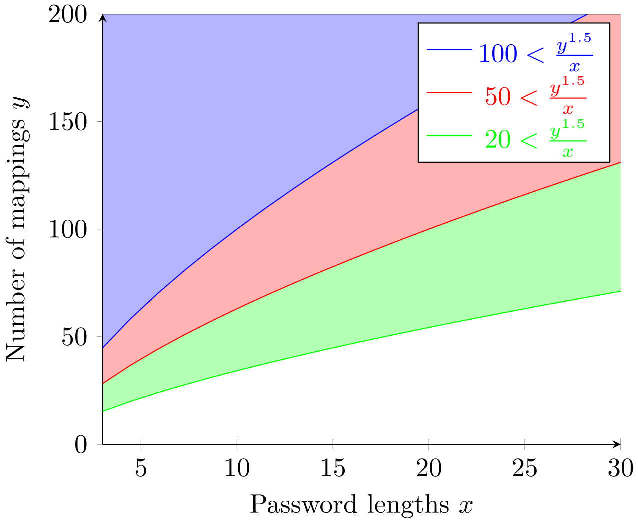

Figure 3.3: Inequality plot of 3 different values for $\hat{a}$

3.2 Practical Usage

This section will illustrate how a human computable password scheme works in practice, using the principles described in the previous sections. Before using the scheme a user have to go through a setup procedure. section 3.1.3 discuss how the scheme can be tweaked to fit different needs a user may have. A summary of the setup and authentication procedures are presented next.

Setup procedure. setup procedure, configuration procedure, ??

1. A secret mapping is the first prerequisite required before the scheme can be used. A random mapping of length n is generated from a set of objects chosen by the user, to digits in $0, 1, \ldots, d$. The user will typically chose what type of objects to use, this can be an alphabet or a user chosen set of pictures. A system will then chose n of these objects and assign a random digit between 0 and d, d is normally 10.

2. Memorization of the mapping is the next, and most costly procedure. The user basically have to learn the mappings by heart, and be comfortable he will not forget it. After memorizing, the mapping is deleted and not stored anywhere else than in the mind of the user. After finishing this step, there is no way to recover the mapping if the user forgets it. This might seem like a barrier making the scheme unusable, it will be shown later that this might not be that much of an effort after all.

3. Passwords can now be generated for all the accounts of the user. Algorithm 3.1 describes the process of creating a new password for an account.

   a) First the user chose the desired length of the password, t, to be generated.

   b) t random challenges are generated and shown one by one to the user, which calculates the responses to these. Each of the responses are one character in the new password.

   c) The calculated password is then sent to the server which should store the salted hash of the password. In the algorithms used here, it is assumed that the sites store the password hashes properly as described in subsection 2.1.3.

4. the same procedure (1-3) can be done for all the accounts the user wants to include in the password scheme.

Authentication procedure.

1. Authenticating with a site, which password was previously generated using the scheme, start of by selecting the correct site. The corresponding challenges will then be displayed starting with the first one.
2. The user calculates the response to each challenge, the same way he did when generating the password. If the calculations are done correctly, the same result should be the same password.
3. After calculating the response to all t challenges, the password can be submitted to the server which checks if the hashed value is the same as the stored one. If it is the user is authenticated.

Algorithm 3.1 Create new challenge for account $A_j \in (A_1, \ldots, A_m)$

```
Require:
    – $t$ desired length of password.
    – $\sigma$ secret mapping memorized by the user.
    – $f$ a human computable function.
    – $O_1, \ldots, O_n$ objects, typically letters or pictures.
1: for $i = 0 \to t$ do
2:     $k \sim [0, n]$
3:     $\vec{C_i} \leftarrow \{O_k\}^{14}$

4: $\vec{C} \leftarrow (\vec{C_1}, \ldots, \vec{C_t})$
5: (User) Computes $(p_1, \ldots, p_t) = f(\sigma(\vec{C}))$
6: (Server) Store $h_j = H(p_1, \ldots, p_t)$
7:
8: return $\vec{C}$
```

Algorithm 3.2 Authentication process for account $A_j \in (A_1, \ldots, A_m)$

```
Require:
    – Account $A_j \in (A_1, \ldots, A_m)$
    – Challenge $\vec{C} = (C_1, \ldots, C_t)$ from account $A_j$.
    – Hash $h_j$ and hash function $H$.
1: for $i = 0 \to t$ do
2:     Display $C_i$ to the user
3:     User Compute $p_i \leftarrow f(\sigma(C_i))$
4:                 ▷ $p_i$ is the $i$'th character of the password for account $A$
5: $\vec{P} = (p_1, \ldots, p_t)$
6: if $h_j = H(\vec{P})$ then                                   ▷ (Server)
7:     Authenticated on account $A_j$
8: else
9:     Authentication failed
```

3.3 Usability

Chapter 4

Application

This chapter will describe how browser extensions are built in google chrome, focusing on architecture and security features. Browser extensions can be utilized to build an application implementing "human computable passwords" as described in chapter 3. The scheme is different from other traditional password managers since it does not *store* the password, but challenges *helping* the user to remember strong passwords. The idea by using browser extensions to implement this is to have an extension monitor the password fields of the sites a user visits and update the challenges depending on the current state of the active site. This technique will be described and a prototype extension demonstrated.

4.1 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 [1]. This allows developers to create desktop apps using the same technology as in web apps, mainly HMTL5, Javascript and CCS.

[1] A new breed of Chrome apps, http://chrome.blogspot.no/2013/09/a-new-breed-of-chrome-apps.html - accessed: 2015-03-02

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

4.1.1 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 document object model (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 4.1 and described in the chrome extensions documentation [2]. This section will describe the architecture considering security concerns relevant when developing chrome extensions which handle sensitive data such as passwords.

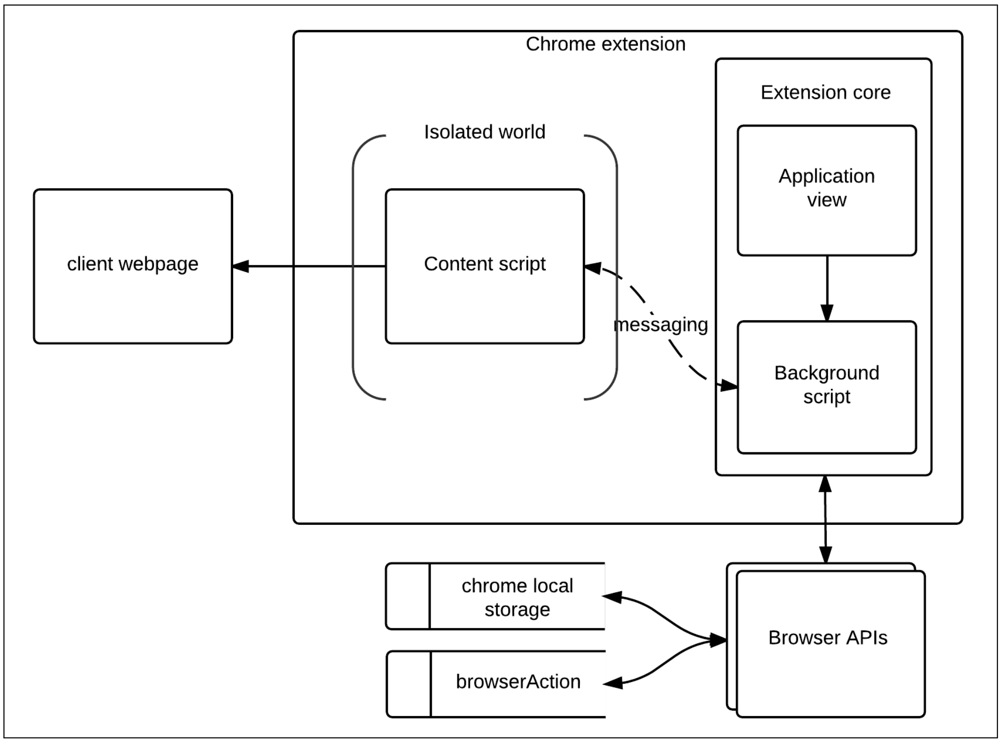

Figure 4.1: Chrome extension architecture.

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

[2]What are extensions?, https://developer.chrome.com/extensions/overview - accessed 2015-03-02

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 [23, 24], 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 [25]; *least privileges*, *privilege separation* and *process isolation.*

Least privilege 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 [3]. 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 [26]. 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. [27] examined this behavior and concluded that developers of chrome extensions usually limit the origins requested to the ones needed.

[3]Manifest file format, https://developer.chrome.com/apps/manifest - accessed 2015-03-04

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. These scripts are instantiated for each visited web page and has direct access to the 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 [28], 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 is responsible for spawning popups and panels, as well as listening for browser action. The typical way to activate a extensions is by clicking an 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 4.2 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 launches 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.

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 [27]. Since the scripts used by the extensions are loaded from the file system, they do not have a origin in the same sense, and thus need to be assigned one. This is done by including a public key in the url of the extension, allowing a packaged extension 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.

more details

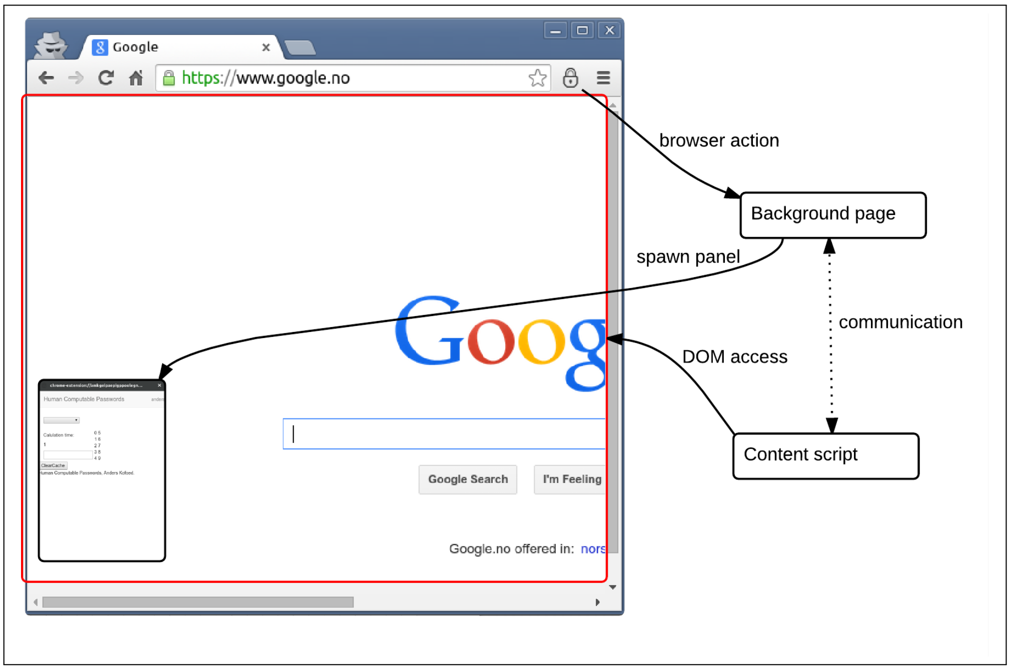

Figure 4.2: Chrome extensions browser action and content scripts.

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 [4].

Figure 4.1 illustrates the architecture of chrome extensions with process isolation and isolated worlds.

[4]Content Scripts, https://developer.chrome.com/extensions/content_scripts - accessed: 2015-03-05

Chapter 5 Experiment

This chapter will present the design and execution of an experiment trying to measure the performance of the human computable password management scheme. The experiment is design as a web application implementing the scheme allowing participants to test how the scheme would work in practice while measuring how fast the computation is performed. The application thus acts as both a demonstration app and a tool for gathering performance data. It has four sections designed to help the participants understand the scheme and get familiar with the computation technique. First as demonstration video is displayed explaining how to compute a password from a challenge, then the user is asked to enter some demographic data. Next a practice section

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] M. Dell'Amico, P. Michiardi, and Y. Roudier, "Measuring Password Strength: An Empirical Analysis," *CoRR*, vol. abs/0907.3, 2009.

[7] L. von Ahn, M. Blum, and J. Langford, "Telling humans and computers apart automatically," *Commun. {ACM}*, vol. 47, no. 2, pp. 56–60, 2004.

[8] 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.

[9] 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.

[10] 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 *Proceedings of the 21th {USENIX} Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pp. 65–80, 2012.

[11] X. de Carné de Carnavalet and M. Mannan, "From Very Weak to Very Strong: Analyzing Password-Strength Meters," in *21st Annual Network and Distributed*

System Security Symposium, {NDSS} 2014, San Diego, California, USA, February 23-26, 2013, The Internet Society, 2014.

[12] S. Komanduri, R. Shay, P. G. Kelley, M. L. Mazurek, L. Bauer, N. Christin, L. F. Cranor, and S. Egelman, “Of passwords and people: measuring the effect of password-composition policies,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 2595–2604, ACM, 2011.

[13] C. E. Shannon, “A mathematical theory of communication,” *Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

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

[15] 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.

[16] 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.

[17] S. Gaw and E. W. Felten, “Password management strategies for online accounts,” in *Proceedings of the 2nd Symposium on Usable Privacy and Security, {SOUPS} 2006, Pittsburgh, Pennsylvania, USA, July 12-14, 2006*, pp. 44–55, 2006.

[18] A. Wildenhain, “Comparison of usability and security of password creation schemes.” http://www.cs.cmu.edu/~jblocki/Anne_Wildenhain_2012.htm. Accessed 2015-03-18.

[19] R. Zhao, C. Yue, and K. Sun, “A Security Analysis of Two Commercial Browser and Cloud Based Password Managers,” in *Social Computing (SocialCom), 2013 International Conference on*, pp. 448–453, IEEE, 2013.

[20] J. Blocki, M. Blum, and A. Datta, “LNCS 8270 - Naturally Rehearsing Passwords,” pp. 361–380, 2013.

[21] A. D. Baddeley, *Human memory: Theory and practice.* Psychology Press, 1997.

[22] N. Alon, Y. Matias, and M. Szegedy, “The Space Complexity of Approximating the Frequency Moments,” *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 137–147, 1999.

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.