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. [BBD14], 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 **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Motivation 1

1.2 Related Work 1

1.3 Scope and Objectives 1

1.4 Method 1

1.4.1 Development 1

1.4.2 Experiments 1

1.5 Outline 1

2 Background **3**

2.1 Passwords 3

2.1.1 Password strength 4

2.1.2 Attacks 5

2.1.3 Password Storage 6

2.1.4 Alternative authentication methods 6

2.2 The User 7

2.3 Password Management 7

2.3.1 Password management schemes 7

2.3.2 Password manager software. 8

2.4 Usability Model 10

2.4.1 Definitions 10

2.4.2 Model 11

2.5 Security Model 12

3 Human Computable Passwords **13**

3.1 Password Management Scheme 13

3.1.1 Definitions and Notation 14

3.1.2 Human Computable Functions 15

3.1.3 Secure Human Computable Functions 16
3.2 Practical Usage 19
3.2.1 Setup procedure. 21
3.2.2 Authentication procedure. 22
3.3 Usability 22
3.3.1 Memorizing the secret mapping. 24
3.3.2 Rehearsing the secret mapping. 24
3.3.3 Computation Time. 25

4 Application **29**
4.1 Browser Extensions 29
4.1.1 Extension Security 30
4.2 Human Computable Passwords Chrome Extension 34
4.2.1 Applications Design 34
4.2.2 User interface 37
4.2.3 Implementation 38
4.2.4 Architecture 38
4.2.5 Demonstration 40
4.2.6 Discussion 41

5 Usability Experiment **47**
5.1 Experiment Objective. 47
5.2 Experiment Setup 47
5.2.1 Web Application 48
5.2.2 Results 48
5.2.3 Discussion 48

6 Conclusion **49**

References **51**

Appendices

A Demonstration slides **55**

B Extension class files **59**
B.1 Content Script 59
B.2 Controller 60
B.3 App.js file 61
B.4 View file (partial file) 62

List of Figures

2.1 Rainbow table 6
2.2 Rouge wifi landing page containing iframes with common sites, used to steal password form a autofilling password manager. 9

3.1 Number of challenge-response pairs required to recover mapping σ as a function of the size of the mapping n. 19
3.2 Number of accounts needed to recover the mapping σ. 20
3.3 Inequality plot of 3 different values for $\hat{a}$ 20

4.1 Chrome extension architecture. 30
4.2 Chrome extensions browser action and content scripts. 33
4.3 Angular data binding with controller, view and scope. Figure from AngularJs developer guide. 36
4.4 Wireframes illustrating the page schematics of the extension. 38
4.5 Class diagram of the human computable passwords chrome extension. . 43
4.6 Screen as seen by the user after launching the extensions while on a page without stored challenges. 44
4.7 Challenge screens as seen by the user while entering password. The challenges update when the user enter a new character in the password field. 45
4.8 Sequence diagram showing the flow in the system when a page is loaded and characters typed in a password field. 46

A.1 Demo slide 1. 55
A.2 Demo slide 2. 56
A.3 Demo slide 3. 56
A.4 Demo slide 4. 57
A.5 Demo slide 5. 57
A.6 Demo slide 6. 58

List of Tables

2.1 Visitation schedules. 12

3.1 Summary of notation. 17
3.2 [BBD14] Extra rehearsals required of the user during the first year to remember σ. Calculated using definition 2.3 with $t = 365$ and visitation schedules as in Table 2.1. 25
3.3 Layout template for displaying challenges. 26
3.4 Layout template for displaying challenges. 27

List of Algorithms

3.1 Generate mapping σ. 22
3.2 Create new challenge for account $A_j \in (A_1, \ldots, A_m)$ 23
3.3 Authentication process for account $A_j \in (A_1, \ldots, A_m)$ 23

Chapter 1
Introduction

1.1 Motivation

[Blo14]

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 [FH07] 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. [IWS04] 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 [ABK13], 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 [DMR09]. 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 [vABL04] 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. [YBAG00] 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. [FHC07] 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. [UKK^{+}12] 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 [dM14]:

- 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, [SBSB07, Sha01]

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 [SS09, FHC07], 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* [Bro11] 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

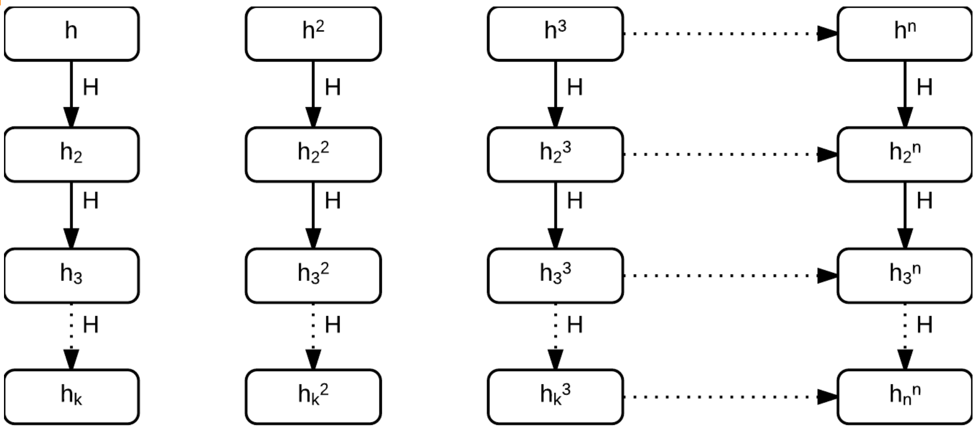


Figure 2.1: Rainbow table

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 [Mil56]. 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 [GF06]:

2.3.1 Password management schemes

Password creation and memorization techniques assists the user in remembering passwords, trying to circumvent the limitations of the human memory section 2.2. Blocki et al. [BBD13] consider 4 different examples of password managements schemes to illustrate how users might chose their passwords.

- Reuse Weak. When a user selects a random phrase or word w and reuse this as the password $p_i = w$ for all accounts. While maybe not very strong, this is the most simple example of a password management scheme.
- Reuse Strong. Same as reuse weak but the user chose four random words w_1, w_2, w_3, w_4 and reuse the concatenation of these as the password $p_i = w_1 w_2 w_3 w_4$ for all his accounts.
- Lifehacker. User choses three random words w_1, w_2, w_3 as a base password $b = w_1 w_2 w_3$ as well as a derivation rule d used to derive unique data from the site names for each password [1]. Example of a derivation rule could be the first and last three letters of the service name. The password for account A_i would then be $p_i = bd(A_i)$ with $d(A_i)$ being the string derived from the

[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

site name. In practice a password generated using the method might look like "facthreerandomwordsook".

– Strong Random and Independent. User choses new words $w_1^i, w_2^i, w_3^i, w_4^i$ for each account to be used as passwords $p_i = w_1^i w_2^i w_3^i w_4^i$.

It is clear that the three former schemes are much easier to use than the last one. Most user would prefer the first ones because they does not require much if any rehearsal while the one strong scheme would require too much effort in terms of rehearsing and memorization. This trade-off between usability and security is the main problem when designing password management schemes. For a scheme to be popular it cannot require too much extra rehearsal, while a secure scheme most of the time will require some.

2.3.2 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 master password protecting the user credentials against both server leakage and insiders accessing the data. Such systems usually provides a lot of extra features such as automatically changing of passwords and syncing between devices.

Built-in browser password managers. Most modern browsers provide "remember password" functions. These functions act similar to software like LastPass, by storing the users passwords in some fashion, then reproduce it when login in.

These kinds of systems and applications 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. Software aiding the user by storing passwords often also provide autofill-functions, automatically filling in the username and password of the associated site. Silver et al. [SJB$^+$14] propose attacks exploiting these autofill functions to extract passwords from the password manager, the most basic example attack is shown in 1. Despite of the obvious weaknesses in many password managers, they still argue that password managers can strengthen credential security if implemented correctly.

Example 1.
Consider a user connecting to a open wifi at a coffee bar or another public place. It is not unusual to present the user with a "landing page" asking for approval of some

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

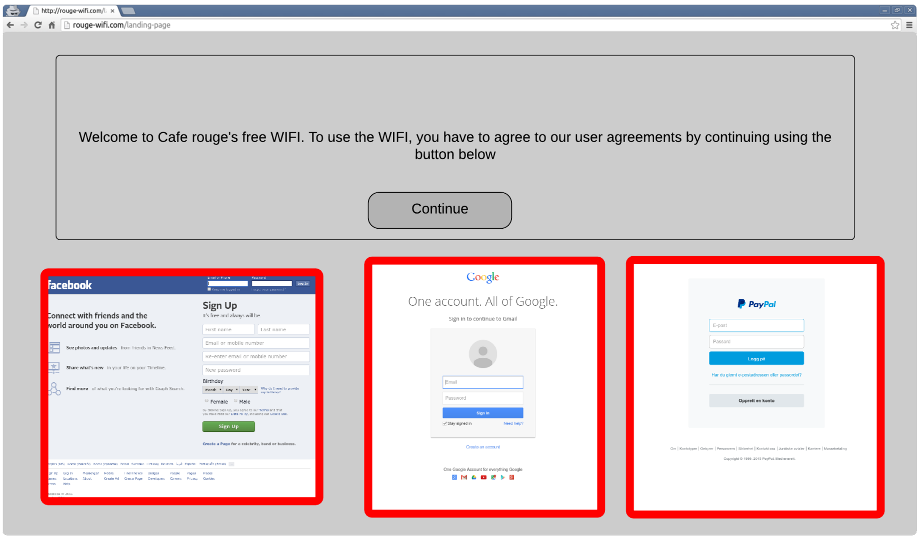

Welcome to Cafe rouge's free WIFI. To use the WIFI, you have to agree to our user agreements by continuing using the button below
rouge-wifi.com/landing-page
Continue
facebook
Sign Up
Connect with friends and the world around you on Facebook.
Google
One account. All of Google.
Sign in to continue to Gmail
Sign in
PayPal
Logg på
Opprett en konto

Figure 2.2: Rouge wifi landing page containing iframes with common sites, used to steal password form a autofilling password manager.

usage agreement. The rouge wifi provider could include multiple iframes[3] pointing to the login pages of common sites the user probably have stored credentials for. See Figure 2.2. By injecting javascript in the iframes the attacker can extract all the usernames and passwords autofilled by the password manager. Note that the iframes displayed in the figure are not visible to the user, to him it looks like a standard landing/welcome page. Silver et al. [SJB$^+$14] claim that six out of ten password managers were vulnerable to this simple attack.

Even if the password manager is secure against autofill-attacks, or if it does not include the feature, the manager might still be at risk. If the account storing all the usernames and passwords of a user where to be compromised, all the sites would be compromised, so it is important that the password manager is even more secure than the sites themselves. Zhao et al. [ZYS13] identified several vulnerabilities in the LastPass implementation, even though no known breaches has been reported. They investigate different types of attacks, including attacks on local decrypted credentials; request monitoring attacks which tries to intercept request between the password manager and related cloud-storage; as well as brute-force attacks trying to crack the master password. The conclusion is that password managers are double-edged swords, in theory they help make password authentication stronger, but if

[3]http://www.w3schools.com/tags/tag_iframe.asp

implemented slightly wrong may be a major vulnerability. Storing all the passwords at one place makes a obvious point of attack for adversaries since breaking the managers most of the time would break all accounts stored within.

2.4 Usability Model

This section presents the usability model defined by Blocki et al. [BBD13], which predicts the effort required of a user to keep a set of secrets memorized without forgetting it. It will be presented in the context of password management schemes, in particular the human computable password management scheme relied on in this project. The model will later be used to analyze the usability of the scheme and related applications.

2.4.1 Definitions

A password management scheme generates and keeps track of passwords $p_1, \ldots, p_m \in P$ for all $m \in A$ accounts of a user. P is all possible passwords. Let $(\hat{c}, \hat{a})$ represent an association between an object (e.g. letter or picture) and a related mapping (typically a digit between 1 and 10). A user have to rehearse the association $(\hat{c}, \hat{a})$ to avoid forgetting it. Next two schedules are defined defining how often a user has to rehearse to not forget a association, and how often he visits an account.

Definition 2.1.
[BBD14] A rehearsal schedule for an object-mapping association $(\hat{c}, \hat{a})$ is a series of points in time $t_0^{\hat{c}} < t_1^{\hat{c}} < \ldots$. A rehearsal requirement for each $i \geq 0$ says that the object-mapping association pair must be rehearsed at least once in the time interval $[t_i^{\hat{c}}, t_i^{\hat{c}} + 1) = \{x \in \mathbb{R} | t_i^{\hat{c}} \leq x < t_{i+1}^{\hat{c}}\}$.

A rehearsal schedule as defined in definition 2.1 is said to be sufficient if a user can keep an association in his memory without forgetting it by following the schedule.

Visitation schedule [BBD14] is a series of numbers $\tau_0^i < \tau_1^i < ...$ representing the points in time when a user visits account A_i. This schedule cannot be known exactly so it is modeled using a random process with a parameter λ_i based on the average time between visits to account A_i - $E[\tau_{j+1}^i - \tau_j^i]$.

A rehearsal requirement can also be satisfied naturally if a user visits an account using the object $\hat{c}$ during the interval $[t_i^{\hat{c}}, t_i^{\hat{c}} + 1)$, as defined in definition 2.2,

Definition 2.2.
A rehearsal requirement $[t_i^{\hat{c}}, t_i^{\hat{c}} + 1)$ is naturally satisfied by a visitation schedule $\tau_0^i < \tau_1^i < \ldots$ if for any $j \in [m]$ and $k \in \mathbb{N}$ so that $\hat{c} \in c_j$ and $\tau_k^j i \in [t_i^{\hat{c}}, t_i^{\hat{c}} + 1)$. Let

$$ER_{t,\hat{c}} = |\{i | t^{\hat{c}}_{i+1} \leq t \wedge \forall j,k.(\hat{c} \notin c_j \wedge \tau^j_k \notin [t^{\hat{c}}_i, t^{\hat{c}}_i))\}|$$

denote how many extra rehearsals required, that are *not* satisfied by the visitation schedule, during time time interval $[0,t]$

2.4.2 Model

The core of the model is that usability depends on rehearsal required to remember all passwords, relative to the visitation schedule of the specific user. How hard it is for any given user to remember a relation between object may vary from person to person depending on mnemonic technique and genetic conditions. This is adjusted for in the model by the constant s representing the combined strength of mnemonic technique and the memory of a user. Next, consider two different rehearsal requirements specifying what is needed to maintain a memory.

Requirement 1.
Constant Rehearsal Assumption (CR)[BBD13]. The rehearsal schedule given by $R(\hat{c}, i) = is$ is sufficient to maintain the association $(\hat{c}, \hat{a})$ in memory.

Requirement 2.
Expanding Rehearsal Assumption (ER)[BBD13]. The rehearsal schedule given by $R(\hat{c}, i) = 2^{is}$ is sufficient to maintain the association $(\hat{c}, \hat{a})$ in memory.

The difference between these two assumptions about human memory is that CR assumes that the user have to keep rehearsing every s days for as long as he wants to make sure to not forget anything. This might be too pessimistic since it is reasonable to assume that it gets easier to rehearse for each rehearsal. This is what ER assumes, if a relation has been rehearsed i times it does not have to be rehearsed again in 2^{is} days. ER is the most intuitive assumption to make and is back up by experiments on how the human brain forgets over time [Squ89**?**].

Visitation Schedule

Every user will eventually have a unique visitation schedule which will vary greatly from user to user. The model uses a Poisson process to model the visitations schedule for a given site A_i, with parameter λ_i. The average time between visits, $\frac{1}{\lambda_i}$, is assumed to be known for each visitation schedule. A site visited every day would yield $\lambda_i = 1$ day, and $\lambda_i = \frac{1}{365}$ days for a site visited annually.

Next, the model use four different types of users which may have accounts of 5 different account types based on visitation frequency. The users can be: very active, typical, occasional or infrequent, while an account can be visited daily, every three days, every week, every month or annually. Table 2.1 defines how many of each type

Visitation schedule	1	$\frac{1}{3}$	$\frac{1}{7}$	$\frac{1}{31}$	$\frac{1}{365}$
Very Active	10	10	10	10	35
Typical	5	10	10	10	40
Occasional	2	10	20	20	23
Infrequent	0	2	5	10	58

Table 2.1: Visitation schedules.

the users have respectively. In example, an active user is said to have 10 accounts he visits daily and 35 he visits annually.

Extra rehearsals

If an object-mapping association $(\hat{c}, \hat{a})$ is not rehearsed through normal usage within the interval $[t_i^{\hat{c}}, t_{i+1}^{\hat{c}})$ the user would have to rehearse the association to prevent forgetting it. $ER_{t,\hat{c}}$ in definition 2.2 gives the number of extra rehearsals of $(\hat{c}, \hat{a})$ in a given time interval. From this it can be seen that $ER_t = \sum_{\hat{c}} ER_{t,\hat{c}}$ gives the number of rehearsals, in addition to natural rehearsal, needed to maintain all objects in a set of associations. In the context of a password management scheme it is clear smaller values for $E[ER_t]$ yield less effort required by the user.

Blocki et al.[BBD13] proves that, given a sufficient rehearsal schedule and a specific visitation schedule, ER_t, total number of extra rehearsals needed to keep all object-mappings in memory, can be predicted through Theorem 2.3.

Theorem 2.3. *[BBD13] Let $i_{\hat{c}}* = (argmax_x t_x^{\hat{c}} < t) - 1$, then*

$$E[ER_t] = \sum_{\hat{c} \in C} \sum_{i=0}^{i_{\hat{c}*}} exp\Bigg(- \Bigg(\sum_{j : \hat{c} \in C_j} \lambda_j \Bigg) (t_{t+1}^{\hat{c}} - t_i^{\hat{c}}) \Bigg)$$

Lemma 1. *[BBD13] The probability that $\hat{c}$ is not rehearsed naturally during the interval $[a, b]$ is $exp(-\lambda_{\hat{c}}(b - a))$, given $S_{\hat{c}} = \{i | \hat{c} \in c_i\}$ and $\lambda_{\hat{c}} = \sum_{i \in S_{\hat{c}}}$*

2.5 Security Model

[BBD13]

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. [BBD14]. 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 [BBD14] to describe the scheme. Next, human computable functions are introduced as this is the main component used in the password management scheme. How these functions can be used to generate and memorize unique passwords in practical cases will be presented. Finally, usability concerns rellated to the discussed password management scheme will be reviewed.

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 use a mapping and a function to calculate the response 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 the passwords. To create a new password, random challenges is generated, the user then computes the password from these using a memorized secret mapping. To reproduce the password later, the same challenges is displayed to the user, which then can calculate the same password.

This procedure is explained further in section 3.2, and in algorithms 3.3 and 3.2.

3.1.1 Definitions and Notation

Memory types considered are either *persistent* or *associative* memory [Bad97]. This project follow the settings of Blocki et al. [BBD13, BBD14] 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 [BBD13].

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 $\sigma : [n] \to \mathbb{Z}_d$. If $X_k \subseteq [n]^k$ is the space of ordered clauses of k variables, let $C \sim X_k$ be a clause chosen at random from X_k. C is now a set of k objects (e.g. $(2, 4, 7, 8)$). Now $\sigma(C) \in \mathbb{Z}_d^k$ is the mapped variables corresponding to challenge C. C can consist of any type of object, such as pictures letters or digits, with the mapping σ always being to digits.

Example 2.
If $\sigma(x) = x + 1 \quad mod 10$ and $C = (10, 25, 36)$ then $\sigma(C) = (1, 6, 7)$.

On of these challenges, C, will be referred to as a *single digit challenge*, which will consist of k ordered object chosen at random. The function $f : \mathbb{Z}_d^k \to \mathbb{Z}_d$ is a human computable function as discussed in the next section. The user the responds 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, which will produce 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 [Mil56], 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 3:

- 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 3.
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. [BBD14] 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 [Mil56]. 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 [AMS99].

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 3.
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. [BBD14] 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" [Blo14]. 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, memorizing a mapping of 100 object-digit mappings might be possible, but probably too hard for a "normal" user to bother doing. It might be more reasonable

$f(x_0, \ldots, x_{13})$	A human computable function used in the password management scheme investigated in this project. Defined in Theorem 3.2
$\sigma(x)$	Random mapping to be memorized by the user. It takes in an object of some sort and returns a digit between 0 and d, d can be assumed to always be 10.
C	A single digits challenge, this is a challenge consisting of 13 randomly chosen ordered objects. A single digits challenge in this project is a list of 13 random letters (e.g. ("B", "E", ...)). One of these will be used to compute *one* character of a user's password.
$\vec{C}$	A password challenge, consisting of t single digit challenges. A password challenge yields a complete password after calculating the response to all single digits challenges contained in it.

Table 3.1: Summary of notation.

to use a smaller set of mappings which will lower the security of the scheme, while making it more accessible for novice users.

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.

In addition to f, a mapping function σ is used. Definition 3.3 defines the composite function of f and σ which is used later in the actual password scheme.

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

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

Security parameters

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. It is not worth memorizing a large set of mappings only to store a few password, since there would not be enough mappings to "loose" for an adversary to recover even a small mapping set.

Another relation is 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 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 including a mapping function, $f \circ \sigma$, as defined in Theorem 3.3 and subsection 3.1.2, can be described through how many accounts $\hat{a}$ which needs to be compromised to recover the secret*

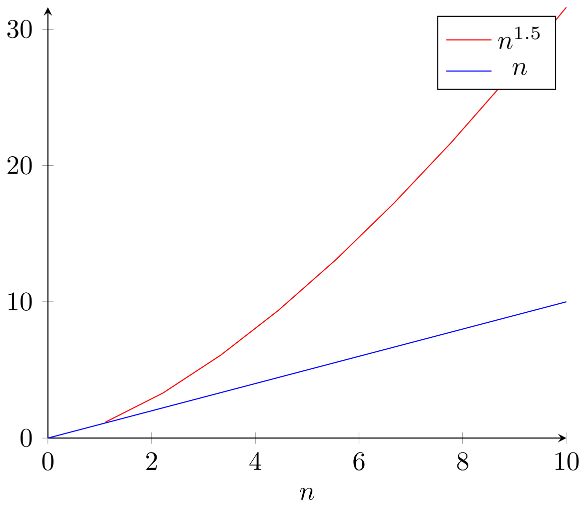


Figure 3.1: Number of challenge-response pairs required to recover mapping σ as a function of the size of the mapping n.

mapping σ, given passwords of length x and y mappings in σ. $\hat{a}$ is then defined as $\hat{a} < \frac{y^{1.5}}{x}$.

Example 4.
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.

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 involving memorization of a randomly generated mapping as well as setting up all accounts with passwords calculated from challenges. Section 3.1.3 discussed how the scheme can be tweaked to fit different needs a user may have, depending on the required level of security and

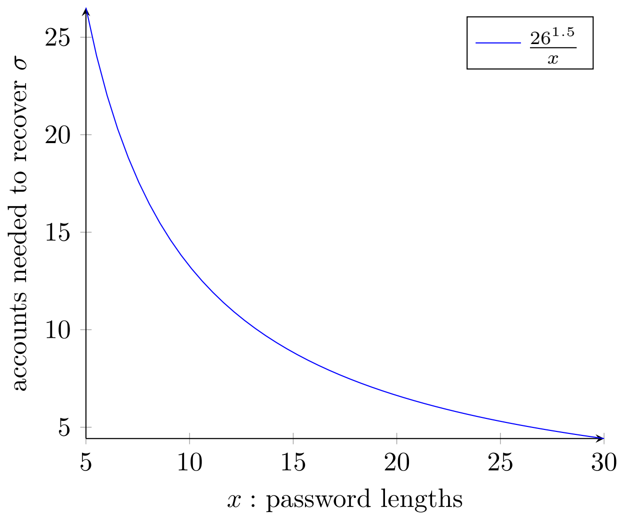


Figure 3.2: Number of accounts needed to recover the mapping σ.

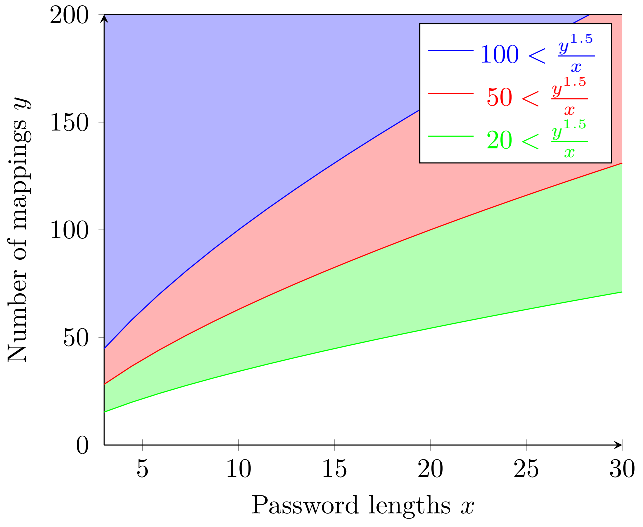


Figure 3.3: Inequality plot of 3 different values for $\hat{a}$

the amount of accounts and passwords a user may have. A summary of the setup and authentication procedures are presented next.

3.2.1 Setup 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 choose what type of objects to use, this can be an alphabet or a user chosen set of pictures. A system will then choose n of these objects and assign a random digit between 0 and d, d is normally 10. Algorithm 3.1 shows how the mapping would be generate from a set of objects by assigning random digits to each object. Remember that this function is supposed to only be stored in the memory of the user, and can thus not be evaluated anywhere else than in the mind of that user.

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. Active users will naturally rehearse and thus not forget their mapping as long as they keep using the scheme and regularly calculate passwords.

3. Passwords can now be generated for all the accounts of the user. Algorithm 3.2 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.

Algorithm 3.1 Generate mapping σ.

```
Require:
        – A base $d$.
        – $O_1, \ldots, O_n$ objects, typically letters or pictures.
1: for $i = 1 \rightarrow n$ do
2:     $k \sim \{0, d\}$
3:     $\vec{S} \leftarrow (O_i, k)$

4: function $\sigma(O_x)$
5:     search($O_x$  $in$  $\vec{S}$)
6:     return $(O_x, i)$

7: return $\sigma$
```

3.2.2 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 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.

Remark 3.
The notation $f(\sigma(\vec{C}))$ as used in the algorithms is equal to the composite function $f \circ \sigma(x_0, \ldots, x_{13})$ as used in definition 3.3.

3.3 Usability

Blocki et al. [BBD14] consider three usability parameters defining the usability of a human computable function. This section will discuss these requirements and how to influence them.

– The effort required to memorize the secret mapping.

– The extra rehearsal required of the user to not forget the secret mapping.

Algorithm 3.2 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 $i = 1 \to t$ 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 3.3 Authentication process for account $A_j \in (A_1, \ldots, A_m)$

```
Require:
        – Account $A_j \in (A_1, \ldots, A_m)$
        – Challenges $\vec{C} = (\vec{C_1}, \ldots, \vec{C_t})$ from account $A_j$.
        – Hash $h_j$ and hash function $H$.
1: for $i = 1 \to t$ 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
```

– How long it takes a human user to calculate the responses to a set of challenges, eventually producing the password.

All of these requirements might limit the usability of the scheme, and are thus worth discussing, but this project will focus mostly one the last requirement related to computation time. Later this computation time will be tested through implementing the scheme as a web app and having participants try it out while timing their efforts.

3.3.1 Memorizing the secret mapping.

As seen in the previous section, a user have to memorize a random mapping from object to digits before starting to use the scheme. This is most likely the biggest and most frightening barrier for any user considering to use the scheme. There are several techniques supposed to help memorizing relations easier, examples are the method of loci [Bad97] which is supposed to enhance memory by visualization. Mnemonic helpers showing objects merged together might help memorize relations as in the case of this project. Blocki et al. [BBD14] propose using mnemonic helpers if the mapping consist of letters to digits. These helpers would typically be a set of pictures showing a visual transition from a letter to a digit. This way might make it easier for the user to remember it instead of only being shown "A=1" etc. Some user might also feel that it is easier to memorize other things than letters, such as pictures. The user might even get to choose the set of pictures to be used themselves as long as the corresponding digits are chosen at random.

3.3.2 Rehearsing the secret mapping.

After memorizing the mappings, the user will have to rehearse it frequent enough to not forget it. Blocki et al. [BBD13] defines a model estimating the cost of this rehearsal, the model is described in section 2.4. Applying this model to the password management scheme gives insight to how much different types of users have to rehearse. The model predicts how long a user will remember a association between i and corresponding mapping $\sigma(i)$ without further rehearsal. If a user is about to forget a mapping according to the predictions, he should be reminded of this. Recall Theorem 2.3 and Table 2.1 from section 2.4. The formula for $E(ER_t)$ can be used to predict how many extra rehearsals different types of users will be required to do within a given period. Table 3.2 shows the expected number of extra rehearsals required by the different types of user given the length of the mapping function n, during the first year. It is computed using Theorem 2.3 with $t = 365$ and visitation schedules λ_i from each user type as seen in Table 2.1. For each account A_i a set of public challenges $\vec{C_i} \in (X_{14})^{10}$ are chosen at random using algorithm 3.2. The expanding rehearsal assumption as defined in [**?**] is used, which assumes that for each rehearsal i the user does not have to rehearse again for 2^{is} units of time. The

User	$n = 100$	$n = 50$	$n = 30$
Very Active	0.396	0.001	≈ 0
Typical	2.14	0.039	≈ 0
Occasional	2.50	0.053	≈ 0
Infrequent	70.7	22.3	6.1

Table 3.2: [BBD14] Extra rehearsals required of the user during the first year to remember σ. Calculated using definition 2.3 with $t = 365$ and visitation schedules as in Table 2.1.

variable s represents differences between user in terms of memory strength, this experiment uses $s = 1$. The values in Table 3.2 are calculated by generating 100 samples of $\vec{C_i} \in (X_{14})^{10}$, then calculating the average expected number of extra rehearsals required by the different user types. n is the number of mappings in σ.

The results in Table 3.2 clearly demonstrates that the human computable passwords scheme does not require much rehearsal at all if used frequently. In fact, for very active, typical and occasional users, memorizing the mapping is a one time cost. After memorizing it at the beginning, using the scheme will provide enough natural rehearsal to maintain the mapping in memory. When a user computes the response to a challenge $\vec{C}$ as described in subsection 3.2.2, he will have to recall the mapping of up to 5 $((\sigma(x_{11}), \sigma(x_{10}), \sigma(x_{12}), \sigma(x_{13}), \sigma(x_j)$, see definition 3.3) values of i for each character of the password. A password length of 10 would yield recalling, and thus rehearsing, 50 values of i. The same trade-off as discussed in section 3.1.3 can be observed here. The more complex the mapping is (larger values of n), the more effort is required when memorizing, but no extra rehearsals are necessary even with a larger number of mappings.

3.3.3 Computation Time.

The final requirement which may limit the usability of the scheme is calculation time. If a user can not compute the response to a challenge in a reasonably short amount of time the scheme would not be usable. How much time a user can tolerate is of course individual, but a too long computation time will directly effect the usability. To make the computation as easy as possible the challenges would have to be presented to the user in a practical way, facilitating fast and reliable calculation. Using the human computable function from definition 3.3, a challenge $C = (x_0, x_1, \dots, x_{13})$ could be displayed as shown in Table 3.3.

$$f \circ \sigma(x_0, x_2, \dots, x_{13}) = \big(\underbrace{\sigma(x_{\underbrace{(\sigma(x_{11}) + \sigma(x_{10})) \quad mod \quad 10}_{\text{Step 1, 2, \& 3}}})}_{\text{Step 4 \& 5}} \underbrace{+\sigma(x_{12}) + \sigma(x_{13})\big) \quad mod \quad 10}_{\text{Step 6 \& 7}}$$

To evaluate a challenge using this function and the layout template from Table 3.3 a user goes through the following steps:

1. Recall the mapping x_{10}
2. Recall the mapping x_{11}
3. Add the values from the two previous steps $i = \sigma(x_{10}) + \sigma(x_{11})$.
4. Locate the element x_i from the table.
5. Recall the mapping $\sigma(x_i)$.
6. Recall the mapping $\sigma(x_{12})$ and add this to the previous value, $z = \sigma(x_i)+\sigma(x_{12})$.
7. Finally recall the mapping $\sigma(x_{13})$ and add it to the previous value, obtaining the final sum $y = z + \sigma(x_{13})$k.
8. y is the response to the challenge C.

x_{10} x_{11}	x_{12}	x_{13}
$0 : x_0$	$5 : x_5$	
$1 : x_1$	$6 : x_6$	
$2 : x_2$	$7 : x_7$	
$3 : x_3$	$8 : x_8$	
$4 : x_4$	$9 : x_9$	

Table 3.3: Layout template for displaying challenges.

Each step depends on at most two earlier steps, allowing the user to do the calculation without having to store more than two values in his memory at any time. After Finishing steps 1-3 he will only keep one value in his memory, the previous intermediate results are now irrelevant and can be forgotten.

Example 5.
Table 3.4 shows the same layout using example objects. The example challenge is $C = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N\}$. Let the mapping function σ be the position in the alphabet $mod \quad 10$, $\sigma(A) = 1, \sigma(B) = 2, \sigma(J) = 0$ etc. The evaluation of this challenge would be the following.

1. $\sigma(K) = 1$, $\sigma(L) = 2$
2. $(11 + 12) \quad mod \quad 10 = 3$

3. $x_3 = D$
4. $\sigma(D) = 4$, $\sigma(M) = 3, \sigma(N) = 4$
5. $(4 + 3 + 4) \quad mod \quad 10 = \underline{\underline{1}}$

$K \quad L$	$M \quad N$
$0 : A$	$5 : F$
$1 : B$	$6 : G$
$2 : C$	$7 : H$
$3 : D$	$8 : I$
$4 : E$	$9 : J$

Table 3.4: Layout template for displaying challenges.

This chapter has presented the ideas behind the human computable password management scheme. It has shown how the scheme works with a secret mapping memorized by the user and a human computable function, which together allows the user to compute passwords from challenges stored in persistent memory. Algorithms describing the procedures has been shown and described, as well as discussing the usability challenges of such a scheme.

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 a chrome extension 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.

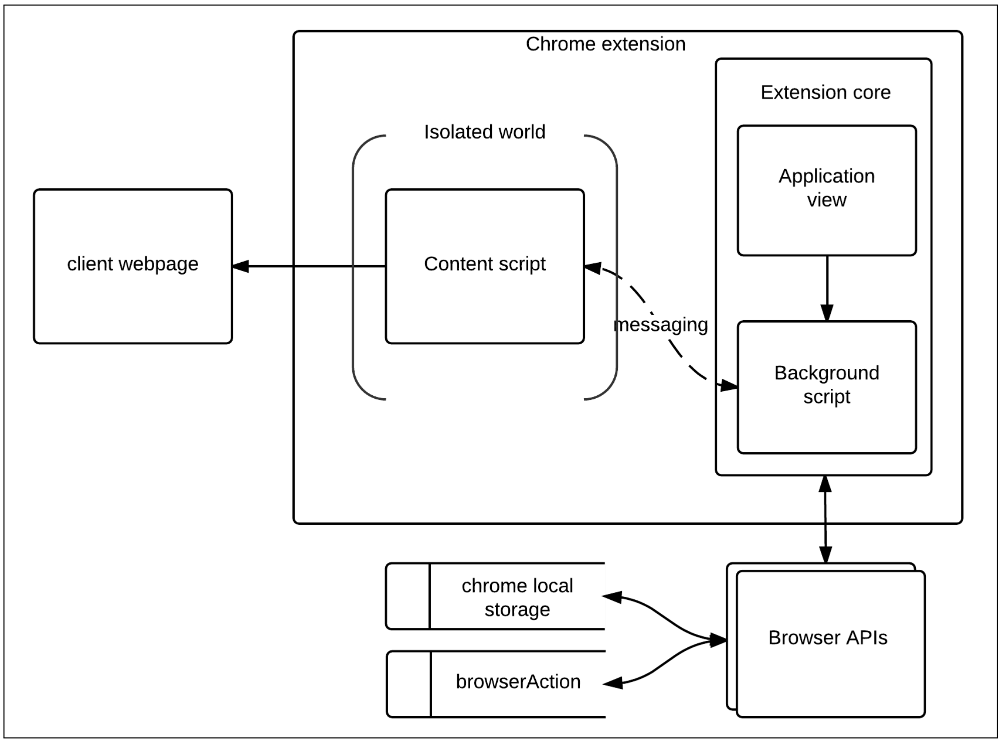


Figure 4.1: Chrome extension architecture.

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 point for attackers, since a buggy extension could leave security holes leaking sensitive

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

information or even provide an entry point to the underlying operating system. For these browsers several frameworks for security have been proposed [LLV08, DG09], 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 [LZC+]; *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. 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. [BFSB10] 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 [BCJ+14], 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 is 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 [BFSB10]. 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,

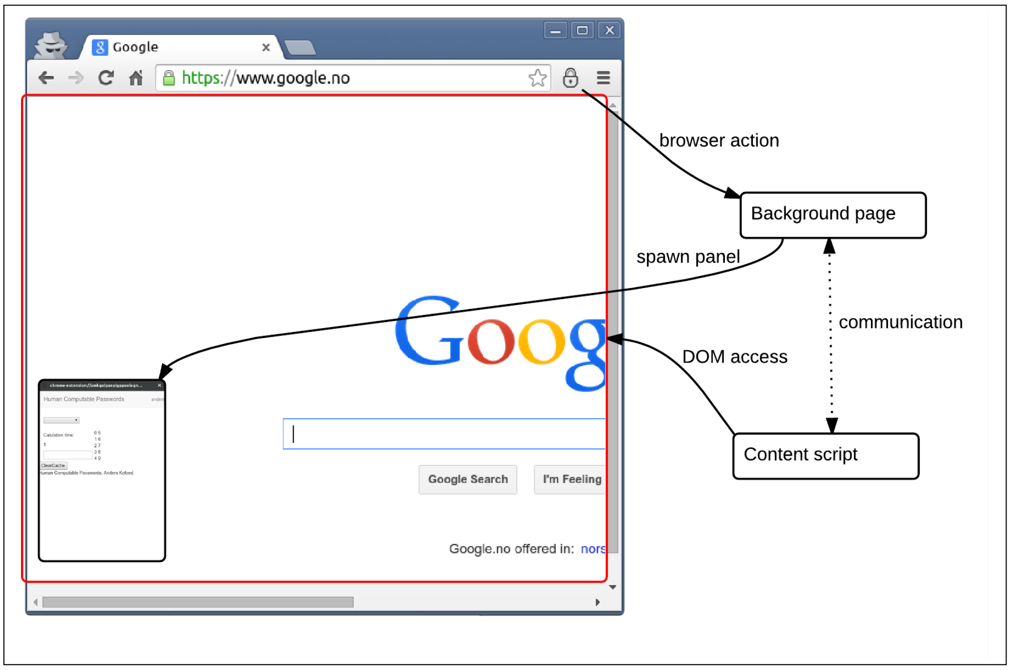


Figure 4.2: Chrome extensions browser action and content scripts.

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. Each extension will have a private/public key pair, a hash of the public key is used as id and each updated version uploaded will be sign with the private key. By doing this the id stays the same throughout versions and can be verified by when uploading and by chrome storage after releasing a new version.

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. This protects against javascript injections from malicious web sites[BZW13]. Since the content script executes in the same environment as the active web page, users may visit websites hosting content meant to exploit extensions[CFW12], possibly stealing sensitive information or issue fake requests.

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].

4.2 Human Computable Passwords Chrome Extension

Chrome extensions are very useful in that they can be ran from any computer with Google chrome installed, thus on any operating system and on any computer. An extension makes it possible to run applications while browsing the web, which in the case of an password management scheme is very useful. An application meant to help the user recall complex passwords should preferably be visible simultaneously with the password field. Popular password management software today are usually either web applications, mobile applications or native desktop applications[5], some of these might include plug-ins such as browser extensions. All of these password managers are reliably storing all the passwords as described in subsection 2.3.2, then they are either auto-filled into the login fields or through copy pasting manually, which introduce several security issues[BFSB10, BZW13, BCJ$^+$14, CFW12, LZC$^+$, SJB$^+$14]

This section will present the design and prototype implementation of the human computable password scheme (chapter 3, [BBD14]. The design evolves around the fact that the scheme does not have to store anything securely, it is though important to make it as easy as possible for the user. The architecture will be similar to the one explained in section 4.1 using content scripts to monitor the password fields of the active browser session. The user interface will be presented through a "panel" in the browser. Panels are windows that stays in focus will interacting with other windows or applications [6].

4.2.1 Applications Design

The extension implementing human computable passwords chapter 3 will be an extension helping the user with storing and managing of challenges for different accounts. The generation of secret mappings will not be part of the application, this should be done through a separate program on the user's local computer. Such a program would follow algorithm 3.1. The requirements for the extension are the following:

- Provide an user interface making it easy for the user to calculate his password.

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

[5]Five Best Password Managers. http://lifehacker.com/5529133/five-best-password-managers

[6]"Panels"- The Chromium Project. https://www.chromium.org/developers/design-documents/extensions/proposed-changes/apis-under-development/panels

– The application should keep track of the active site, displaying challenges for the correct site without user interaction.

– Add new sites to the system easily.

– The displayed challenge should update seamlessly while typing the password.

– The user should be able to type his password directly in the password field of the active site.

– If the user is about to forget a secret mapping, according to the usability model 2.4, the application should notify about it and advice the user to rehearse.

AngularJs

The front-end is built and updated using AngularJS[7], which is an open source, client-side javascript framework. Angular is built using a variation of the model-view-controller architecture [Dea09], though the creators of angular states that angular is a model-view-whatever framework[8], point being that what the architecture is called isn't important.

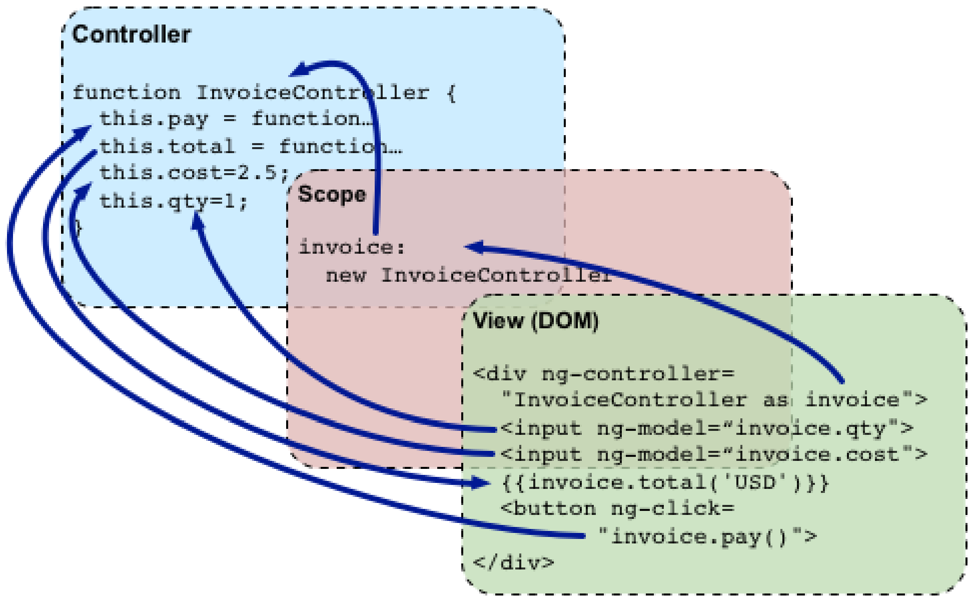
The *views* in angular is templates written entirely in HTML, making it easy to read and update. The *controller* contains all the business logic used by the view. The views and controllers are connected using a shared object called *$scope*, variables or functions on this object is usually defined in the controller and accessed by the view by using double curly brackets in the view (e.g. {{name}} to access the name variable on $scope). Figure 4.3 shows how scope is used to share variables between the controller and the view. See "Angular Essentials - Rodrigo Branas"[Bra14] for a step-by-step introduction to AngularJS.

The main benefits of using angular in this project is the data-binding which makes it easy to update the HTML shown to the user. The extension will take advantage of this when updating the challenges seamlessly while the user calculates his password. The content of the extension will change according to the data filled in a password field without reloading the extension.

Chrome Storage.

Local storage[9] is a way for applications to store data persistently and securely in the browser of clients. It is meant as an improvement of cookies which is the usual way of storing user data across sessions. The biggest problem with cookies is that they

[7]AngularJS, https://angularjs.org/
[8]Model-View-Whatever - https://plus.google.com/+AngularJS/posts/aZNVhj355G2
[9]Local storage - http://diveintohtml5.info/storage.html



```
function InvoiceController {
  this.pay = function…
  this.total = function…
  this.cost=2.5;
  this.qty=1;
}
```

```
invoice:
  new InvoiceController
```

```
<div ng-controller=
  "InvoiceController as invoice">
  <input ng-model="invoice.qty">
  <input ng-model="invoice.cost">
  {{invoice.total('USD')}}
  <button ng-click=
       "invoice.pay()">
</div>
```

Figure 4.3: Angular data binding with controller, view and scope. Figure from AngularJs developer guide.

are sent with every HTTP request and thus slow down the applications using them. Local storage allows applications to store (key, value)-pairs in the browser.

Chrome storage[10] is close to the same as local storage, differences being that chrome local storage allows applications to store data in what is called "chrome.storage.sync". This specific storage saves data locally, but also syncs it with the currently active chrome account, thus allowing a user to log into his account in any chrome browser and access the same application data. Chrome storage also allows storage of objects compared to local storage which only allows storing strings. This project will use chrome storage to persistently store challenges across sessions, and also provide backup. This way each user will be able to keep challenges for all their account within the storage of their google account. This is a important feature since it allows access to the challenges from remote locations as well.

Content Scripts.

Content scripts are javascript files running in the context of the active web page, as browsed by the user. These scripts has direct access to the content of the active site and can thus monitor and attach event listeners to the content of the page. The content scripts are isolated from the extension itself as describe in subsection 4.1.1,

[10]chrome.storage - https://developer.chrome.com/extensions/storage

thus protecting the extension from possibly harmful sites trying to exploit weaknesses in the content script. Because of this the content scripts has to communicate with the extension through google chrome's built-in message passing system. Chrome message passing[11] allows scripts to listen for and respond to messages. One side set up an event listener listening for messages, when a message is sent from the other side this event triggers and the message can be received and parsed.

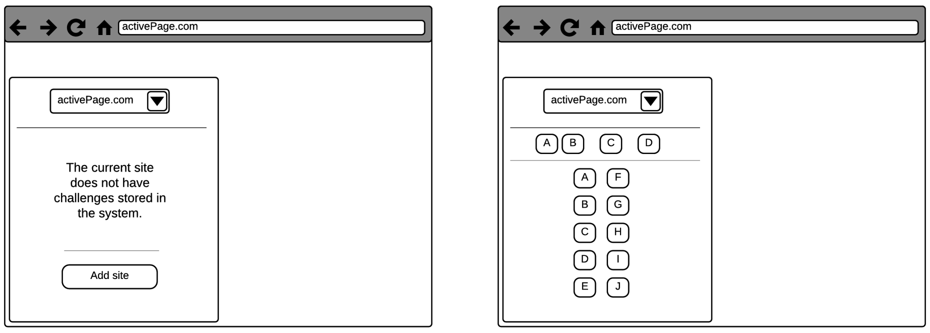
The content scripts and the message passing system are likely points of attack for adversaries. The content script can monitor the value of the password field and thus, in theory, steal these if a malicious script was able to trick it into leaking theme. The communication channel is not particularly prone to attacks since even if an adversary eavesdropped all data sent on it, the only information leaked would be the current length of the password. It is though important that the design is like this since the mistake of sending the whole password string, which might seem like a solution, would be potentially dangerous.

Typical usage of content scripts in this application will be to attach event listeners to the password fields of the pages visited by the user, and message the extension when the value of the password field changes. When a change happens, the content script send a message containing the current length of the typed password, so that the extension can display the correct challenge. In example if the user has entered 4 characters of his password the extension should display the 5th challenge etc. The content script also keep the extension updated on the URL of the current page, by sending a message every time a page is loaded. The extension then displays the challenges corresponding to the password of that site. If the site does not have challenges stored by the application, the user can generate new ones and store these in the system through clicking a button.

4.2.2 User interface

The user interface will be very simple, there will be two possible screens displayed to the user. Either, challenges associated to the current page will be shown, or a dialog asking the user to generate new challenges. The most important feature of the user interface is that it will automatically update depending on what site the user is currently browsing, and display the correct challenge when typing passwords. Wireframes illustrating the page schematics of the extension are shown in figures 4.4a and 4.4b.

[11]Chrome message passing - https://developer.chrome.com/extensions/messaging



(a) Screen seen by the user when launching the extension while visiting a page that is not stored in the system.

(b) Screen seen by the user when loading a site with challenges stored in the system.

Figure 4.4: Wireframes illustrating the page schematics of the extension.

4.2.3 Implementation

The implementation follows the design described in the previous section, this section will present how the extension are implemented and demonstrate the actual extension.

4.2.4 Architecture

Figure 4.5 shows the complete architecture of the system. The content script attach an event listener to the password field of the active site. If the content of the password field changes, the script sends a message using chrome message passing containing the current length of the password. The controller receives the new password length from the content script, and update the challenges displayed to the user. When the user visits a new site, the url is sent to the controller, which then updates the view with new challenges corresponding to that site.

The classes seen in figure 4.5 are all represented in the implemented version of the extension. The construction and responsibilities of the classes are summarized in the following list. The code of the implementation for each component is explained in more detail in appendix B.

– **Controller** is responsible for the business logic related to the information shown to the user in the extension. The controller first tries to load the user data stored in chrome storage, if no user has been created, typically if it is the first time the extension is loaded, a new user is created. The controller keeps track of the url of the current active page and the current length of the password field. It listens for messages from the content scripts and updates the

URL and password length variables with the info received. The controller class used in the extension is described in appendix B.2.

– **ChromeStorage**[12] is a utility resource used to access chrome local storage[section 4.2.1]. This module is essentially a wrapper making it easier to retrieve data from chrome storage, while also providing useful debugging and cache management functions.

– **Content script** is responsible for attaching event listeners to the password field of the currently active page.. It also listens to the onload event, sending url updates to the controller every time a new page is loaded. The content script of the extension built in this project can be seen appendix B.1.

– **main.html** is the view of the application, responsible for the user interface. The view receives updates from the controller through the $scope variable. When the controller updates in example the current password length, this is reflected in the view where different objects are shown. If the current url is not in the user's list of saved sites, the view will show a dialog allowing the user to add it. When adding a new site the user should update the password of that site using the extension. A snippet of the view is shown in appendix B.4.

– **App.js** is the focal point of the application, responsible for initializing the angular app and routing of views. In this project this file also includes some helper functions which can be seen in appendix B.3.

Launching the application is done by clicking an icon in the browser toolbar which will launch the extension in a panel floating on top of the other browser windows and tabs. This behavior is specified in the *background page*[13]. The background page is the "launcher" of the extension, it waits for the user to click the extension icon, firing a browser action event. On catching this event, the script spawns a panel, with the angular application as content. Listing 4.1 shows the background page of this extension. After spawning the panel, the background page is standby doing nothing, everything now happens through the angular application running inside the panel.

```
chrome.browserAction.onClicked.addListener(function() {
    chrome.windows.create({
        url: chrome.extension.getURL('main.html'),
        type: 'panel',
        focused: true,
        height: 520,
        width: 400,
    });
```

[12]angular-chrome-storage - https://github.com/infomofo/angular-chrome-storage
[13]Background pages - https://developer.chrome.com/extensions/background_pages

```
});
```

Listing 4.1: Background page.

4.2.5 Demonstration

After launching the extension, as previously explained, the user is presented with the window show in figure 4.6. In this example the active site does not have a record in the user's challenge database, thus the "add new" dialog. By clicking the button, new challenges are generated and stored in the database as a new site. The site is automatically added with the current site domain as key together with randomly generated challenges. Next time the user visits the same page, challenges will appear. After adding a new site, it is the users responsibility to change the password of this site so it matches the challenges.

Figures 4.7a-d shows how a user would use the extension when login in to a site. When launching it, the system receive the current sites domain and loads the first challenge corresponding to that site. The user then calculates the first character using the challenges displayed, when this is entered in the password field, new challenges appear until the whole password is calculated.

Data flow

Figure 4.8 illustrates the life cycle of the application. When the user visit a site, an event is triggered and caught by the content script, which in turn message the controller about the newly loaded site. The controller then try to load challenges for that site from the storage; if there is a record, the view is updated with the first challenge, if not the "add new"-dialog is loaded. Next event is triggered when the user enters a character in the password field, on receiving notice from the content script the controller updates the view with a new challenge depending on the length of the password. This will then go on until the user has entered the entire password, eventually closing the extension panel.

4.2.6 Discussion

The approach of this project when designing the human computable passwords extension was to make an clean and intuitive application making the password management scheme feasible to use. The extension does not require any user interaction, except from when adding new sites, this was done so the user could focus on doing the calculation correctly without having to keep track of the challenges. The scheme itself is also quit complicated so it is important for the user to focus on as few operations as possible. (The simplest approach would be to have a "next"-button for the user to click between every character calculated.)

The choice of a browser extension was done with the same mind set, making the application helpful and not distracting. It should be easy to start using the application, requiring as little configuration and managing as possible. The design and construction makes it so that the only real "configuration" required is the changing of passwords when adding sites, which essentially can not be circumvent. Panels is another very useful feature, since the panels floats on top of the other browser windows and keeps in focus while typing. Without panels the user would have to switch between windows when calculating a character and typing it in the password field, which would be a huge drawback both in terms of time and usability.

One apparent problem with using extensions is that they are not supported in mobile browsers, and probably will never be [14]. The solution to this would be to make an additional application for mobile, and sync the data to another service available on mobile as well. This could be done quit easily since the data is stored as text, the only change needed would be to substitute the storage module with some other cloud storage service. The problem if the application was built for mobile would be the lack of space to display challenges while typing the password.

[14] Multidevice FAQ - https://developer.chrome.com/multidevice/faq

The decision to make a browser extension was made on behalf of the mentioned pros and cons, with the most important parameter being disturbance. Only extensions allow a seamless, non-interfering user interface. The other options, namely web application and mobile application, would require switching between windows or at least switching of focus, as well as requiring the user to manually chose the correct site and browse through the different challenges. This is all done automatically with the construction presented in this chapter. The main goal of the design was to include the least amount of unnecessary features, with a clean and unobstructive interface. A summary of the strengths of the chosen design is listed next.

– Easy to use, require less to no user interaction after configuration.
– Panel that stay on top while entering the password allows for quick and easy calculation without stopping to update and manage challenges.
– Chrome.storage.sync automatically synchronize the stored challenges to the users google account allowing persistent usage across devices and accounts.
– All data is stored as text strings, this makes it easy to integrate with other services in the future. The user could also store the challenges elsewhere if need be, e.g. as text locally or even print the challenges on paper in the extreme case.

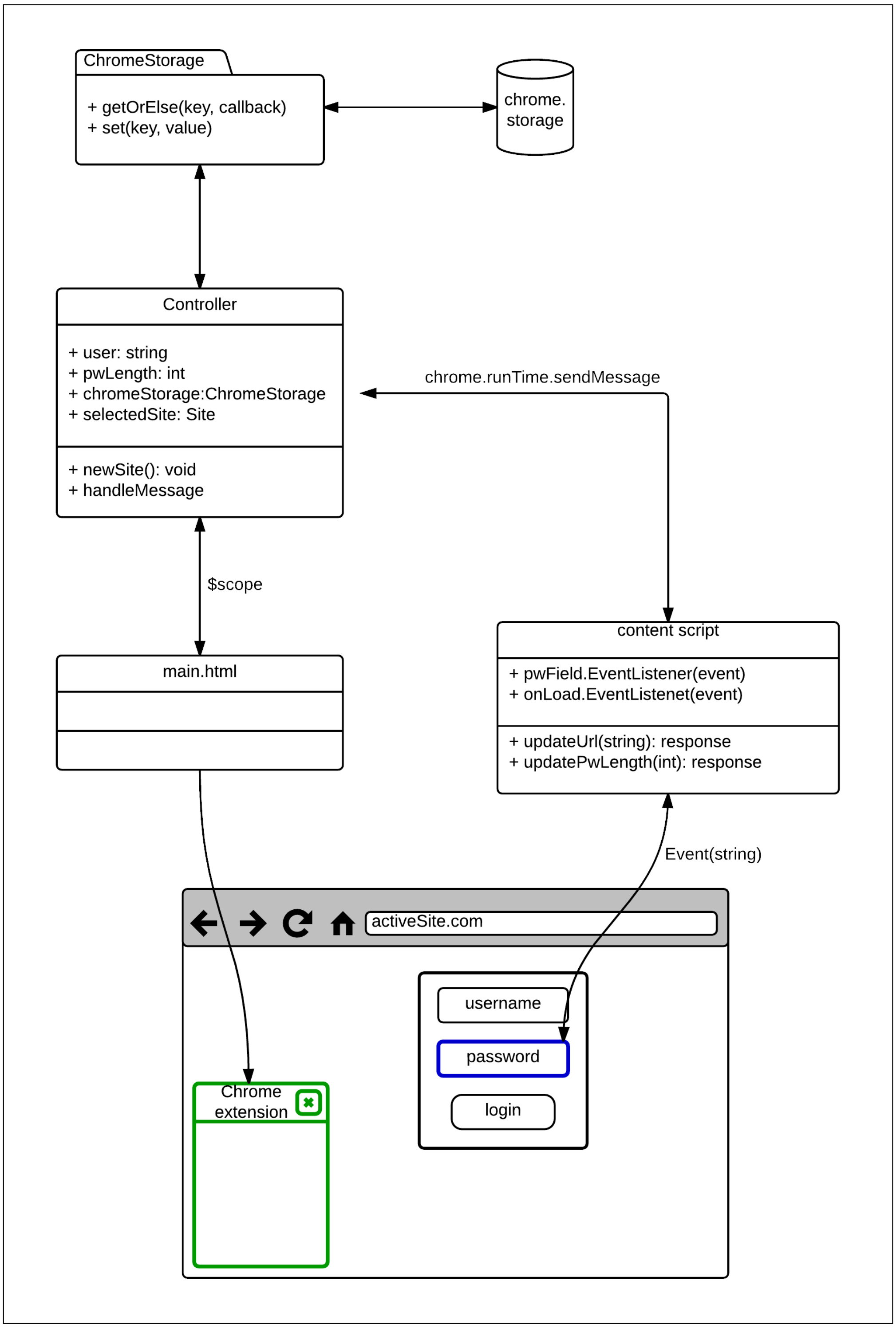
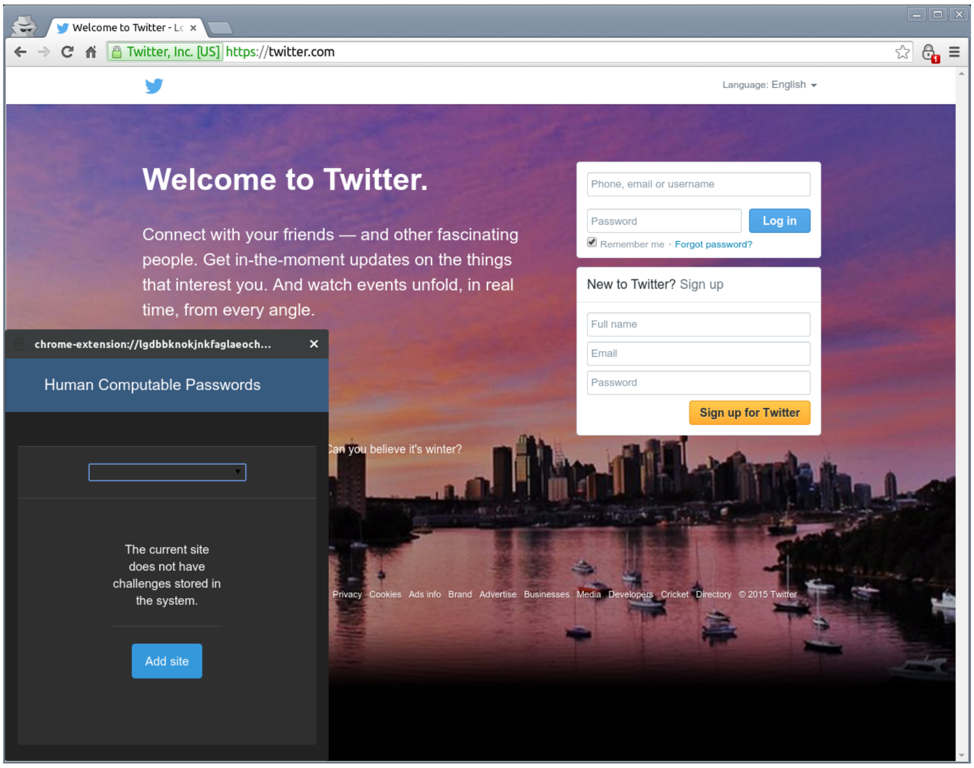


Figure 4.5: Class diagram of the human computable passwords chrome extension.



Welcome to Twitter.

Connect with your friends — and other fascinating people. Get in-the-moment updates on the things that interest you. And watch events unfold, in real time, from every angle.

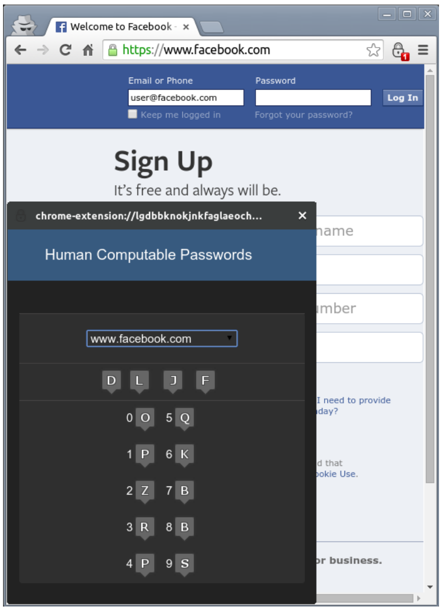
chrome-extension://lgdbbknokjnkfaglaeoch...

Human Computable Passwords

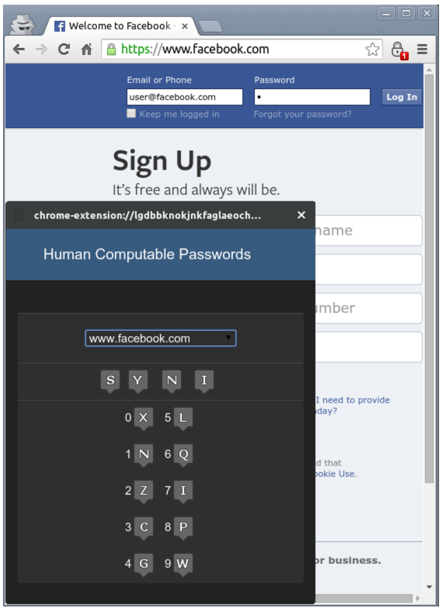
The current site does not have challenges stored in the system.

Welcome to Twitter - Lc
Twitter, Inc. [US] https://twitter.com
Language: English
Phone, email or username
Password
Log in
Remember me · Forgot password?
New to Twitter? Sign up
Full name
Email
Password
Sign up for Twitter
Can you believe it's winter?
Add site
Privacy Cookies Ads info Brand Advertise Businesses Media Developers Cricket Directory © 2015 Twitter

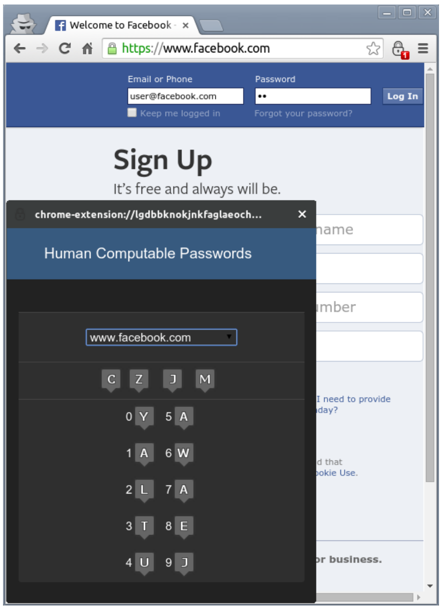
Figure 4.6: Screen as seen by the user after launching the extensions while on a page without stored challenges.



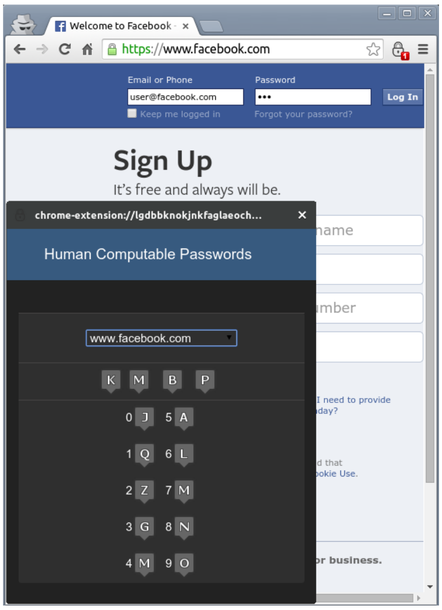
(a)



(b)



(c)



(d)

Figure 4.7: Challenge screens as seen by the user while entering password. The challenges update when the user enter a new character in the password field.

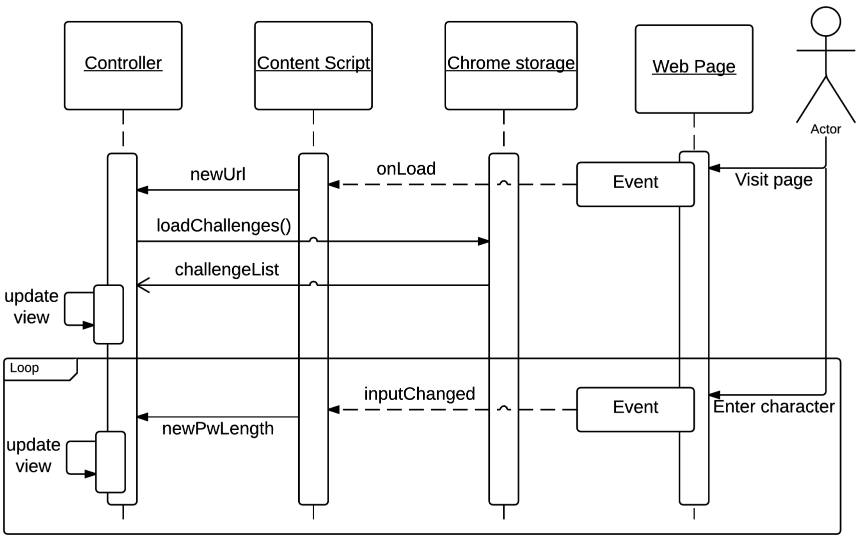


Figure 4.8: Sequence diagram showing the flow in the system when a page is loaded and characters typed in a password field.

Chapter 5

Usability 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 is the practice section, where the user is supposed to practice doing the calculation until being able to solve challenges without error. Finally is the experiment part, which is timed and correctness monitored. After finishing the calculations the user will submit the data and can chose to continue doing more experiments.

5.1 Experiment Objective.

Explain what the goal of the experiment is. What results I'm looking for/what the purpose is.

NOTE: The experiment application is not only recording data, it works as a presentation of the scheme, trying to make it easy to understand so that most people should be able to use it.

5.2 Experiment Setup

What data will be recorded, choices regarding the secret mapping etc.

5.2.1 Web Application

Design and architecture of the web application

Discuss if a similar application could be used as an actual password manager. What are the advantages/disadvantages versus the extension implementation.

5.2.2 Results

Presentation of the results gather through the experiment (calculation time, failure rate etc.)

Different plots of the data. Not sure what and how to plot the different data??

5.2.3 Discussion

Discuss how the results fit in the usability model, plots $\hat{t}$ for the different participants

What use cases would require a faster or slower calculation time?

Chapter 6 Conclusion

References

[ABK13] Tolga Acar, Mira Belenkiy, and Alptekin Küpçü. Single password authentication. *Computer Networks*, 57(13):2597–2614, 2013.

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency Moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[Bad97] Alan D Baddeley. *Human memory: Theory and practice.* Psychology Press, 1997.

[BBD13] Jeremiah Blocki, Manuel Blum, and Anupam Datta. LNCS 8270 - Naturally Rehearsing Passwords. pages 361–380, 2013.

[BBD14] Jeremiah Blocki, Manuel Blum, and Anupam Datta. Human Computable Passwords. *CoRR*, abs/1404.0, 2014.

[BCJ$^+$14] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, and Yuan 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*, pages 184–192, 2014.

[BFSB10] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron 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.

[Blo14] Jeremiah Blocki. *Usable Human Authentication: A Quantitative Treatment.* PhD thesis, School of Computer Science, Carnegie Mellon University, 2014.

[Bra14] Rodrigo Branas. *AngularJS Essentials.* Packt Publishing Ltd, 2014.

[Bro11] Gerald Brose. Rainbow Tables. In Henk C A van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 1021–1022. Springer, 2011.

[BZW13] A Barua, M Zulkernine, and K Weldemariam. Protecting Web Browser Extensions from JavaScript Injection Attacks. In *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*, pages 188–197, 2013.

[CFW12] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An Evaluation of the Google Chrome Extension Security Architecture. In *USENIX Security Symposium*, pages 97–111, 2012.

[Dea09] John Deacon. Model-view-controller (mvc) architecture. *Online][Citado em: 10 de mar{ç}o de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*, 2009.

[DG09] Mohan Dhawan and Vinod 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*, pages 382–391, 2009.

[dM14] Xavier de Carné de Carnavalet and Mohammad 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.

[DMR09] Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier. Measuring Password Strength: An Empirical Analysis. *CoRR*, abs/0907.3, 2009.

[FH07] Dinei A F Florêncio and Cormac Herley. A large-scale study of web password habits. In Carey L Williamson, Mary Ellen Zurko, Peter F Patel-Schneider, and Prashant J Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web, {WWW} 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 657–666. ACM, 2007.

[FHC07] Dinei Florêncio, Cormac Herley, and Baris 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.

[GF06] Shirley Gaw and Edward 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*, pages 44–55, 2006.

[IWS04] Blake Ives, Kenneth R Walsh, and Helmut Schneider. The domino effect of password reuse. *Commun. {ACM}*, 47(4):75–78, 2004.

[LLV08] Mike Ter Louw, Jin Soon Lim, and V N Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3):179–195, 2008.

[LZC$^+$] Lei Liu, Xinwen Zhang, Vuclip Inc Huawei R&D Center, Guanhua Yan, and Songqing Chen. Chrome Extensions: Threat Analysis and Countermeasures.

[Mil56] George A Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[SBSB07] Richard Shay, Abhilasha Bhargav-Spantzel, and Elisa Bertino. Password Policy Simulation and Analysis. In *Proceedings of the 2007 ACM Workshop on Digital Identity Management*, DIM '07, pages 1–10, New York, NY, USA, 2007. ACM.

[Sha01] Claude E Shannon. A mathematical theory of communication. *Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[SJB$^+$14] David Silver, Suman Jana, Dan Boneh, Eric Yawei Chen, and Collin Jackson. Password Managers: Attacks and Defenses. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd {USENIX} Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 449–464. {USENIX} Association, 2014.

[Squ89] Larry R Squire. On the course of forgetting in very long-term memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 15(2):241, 1989.

[SS09] Karen Scarfone and Murugiah Souppaya. Guide to enterprise password management (draft). *NIST Special Publication*, 800:118, 2009.

[UKK$^+$12] Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie Faith 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*, pages 65–80, 2012.

[vABL04] Luis von Ahn, Manuel Blum, and John Langford. Telling humans and computers apart automatically. *Commun. {ACM}*, 47(2):56–60, 2004.

[YBAG00] Jianxin Yan, Alan Blackwell, Ross Anderson, and Alasdair Grant. The memorability and security of passwords: some empirical results. *Technical Report-University Of Cambridge Computer Laboratory*, page 1, 2000.

[ZYS13] Rui Zhao, Chuan Yue, and Kun Sun. A Security Analysis of Two Commercial Browser and Cloud Based Password Managers. In *Social Computing (SocialCom), 2013 International Conference on*, pages 448–453. IEEE, 2013.

Appendix A

Demonstration slides

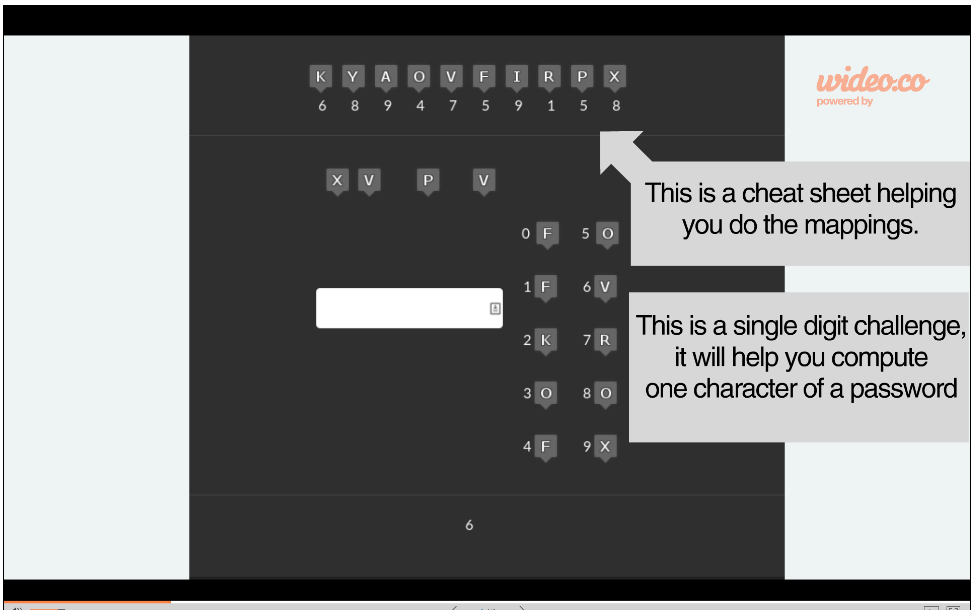
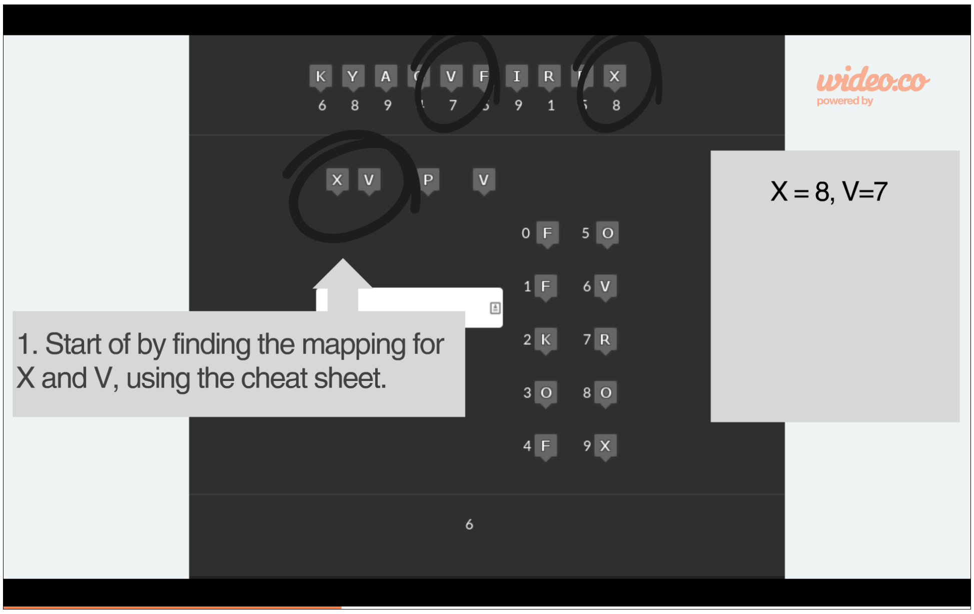

K Y A O V F I R P X
6 8 9 4 7 5 9 1 5 8
X V P V
0 F 5 O
1 F 6 V
2 K 7 R
3 O 8 O
4 F 9 X
6
wideo.co
powered by
This is a cheat sheet helping you do the mappings.
This is a single digit challenge, it will help you compute one character of a password

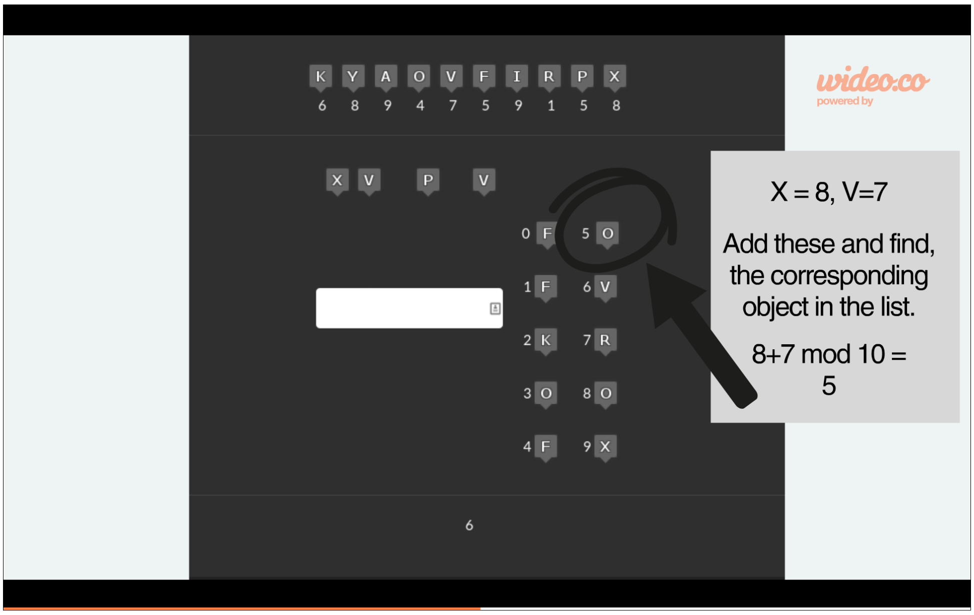
Figure A.1: Demo slide 1.



1. Start of by finding the mapping for X and V, using the cheat sheet.

X = 8, V=7

Figure A.2: Demo slide 2.



X = 8, V=7

Add these and find, the corresponding object in the list.

8+7 mod 10 = 5

Figure A.3: Demo slide 3.

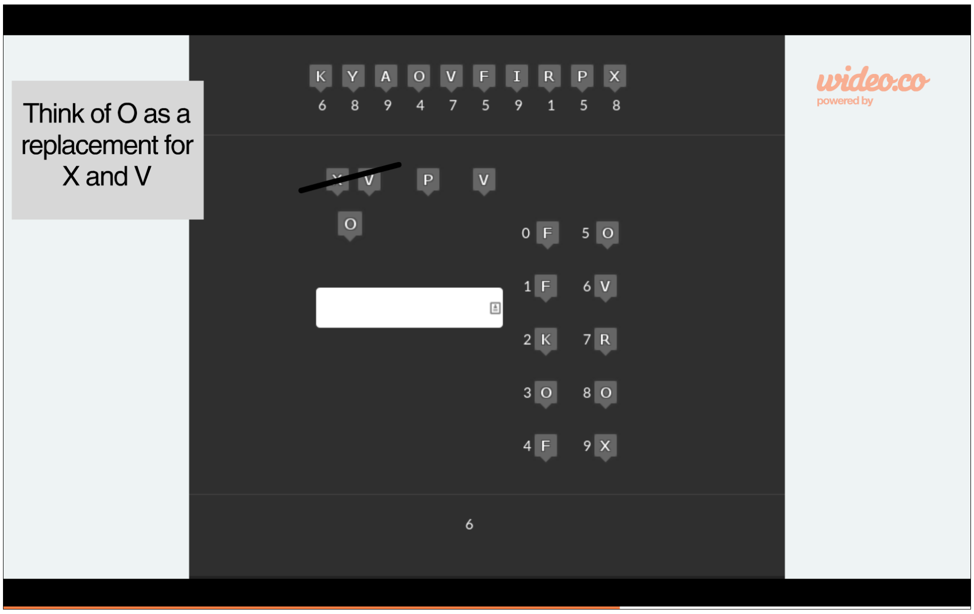

Think of O as a replacement for X and V

Figure A.4: Demo slide 4.

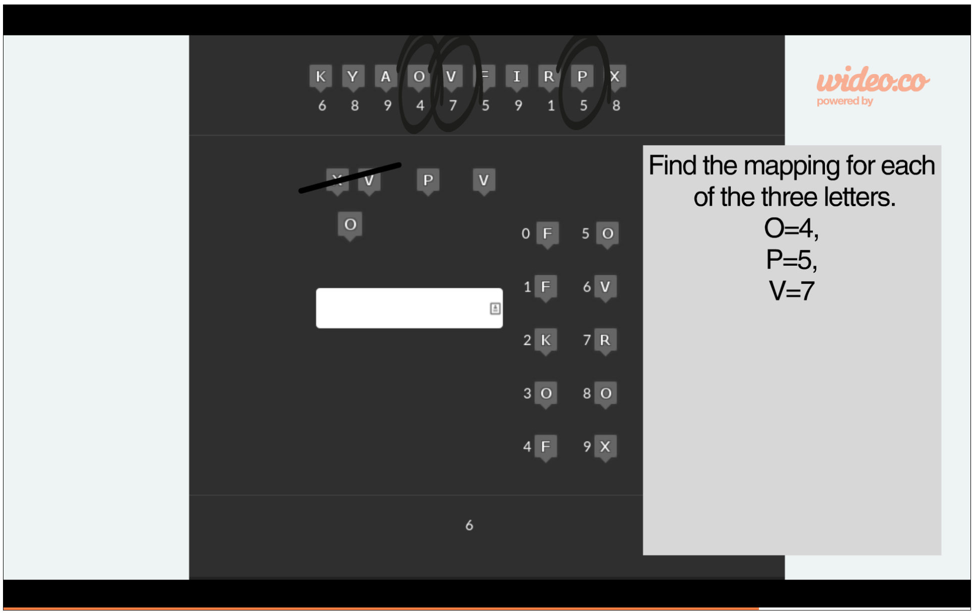
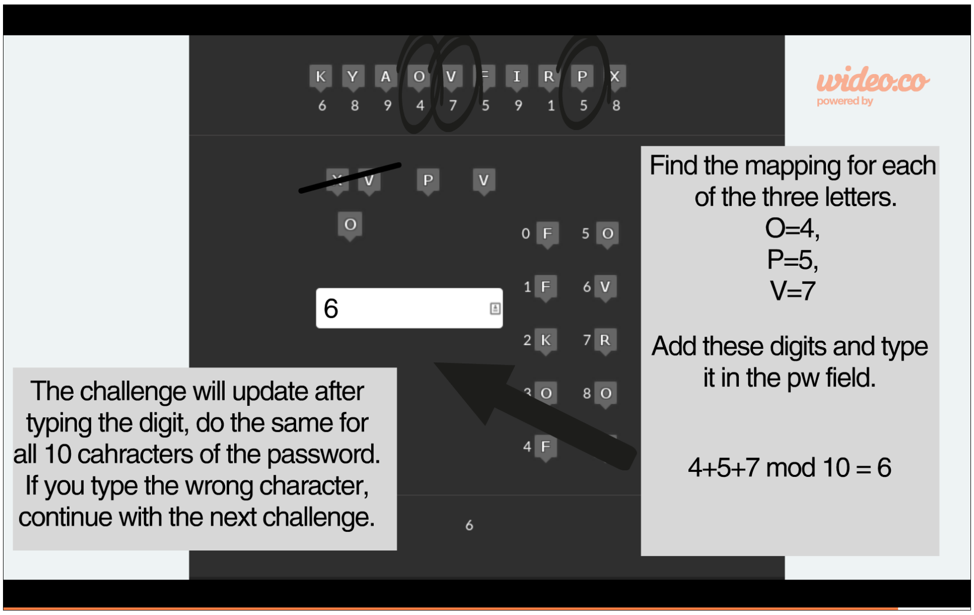

Find the mapping for each of the three letters.
O=4,
P=5,
V=7

Figure A.5: Demo slide 5.



Find the mapping for each of the three letters.
O=4,
P=5,
V=7

Add these digits and type it in the pw field.

4+5+7 mod 10 = 6

The challenge will update after typing the digit, do the same for all 10 cahracters of the password. If you type the wrong character, continue with the next challenge.

Figure A.6: Demo slide 6.

Appendix B
Extension class files

B.1 Content Script

```
1  window.onload = setTimeout(updateUrl(),500);
2  var passwords = getPwdInputs();
3  var pwField = passwords[0];
4
5  if(pwField){
6      pwField.addEventListener('input',
7      function (callback){
8
9          console.log("input content: " + pwField.value.length);
10         chrome.runtime.sendMessage({pwValue: pwField.value.length+1},
11         function(respons){
12             console.log("Respons pw changed: " + respons);
13         });
14
15     });
16 }
17
18 function getPwdInputs() {
19        var ary = [];
20          var inputs = document.getElementsByTagName("input");
21          for (var i=0; i<inputs.length; i++) {
22              if (inputs[i].type.toLowerCase() === "password") {
23                          ary.push(inputs[i]);
24              }
25          }
26            return ary;
27 }
28
29 function updateUrl(){
30     chrome.runtime.sendMessage({newUrl: window.location.hostname},
31     function(respons){
32         console.log("URL changed to: "+ window.location.hostname);
33     });
34 }
```

Listing B.1: Content script file.

The content script listens for the onload event triggered by the windows object when a new page is loaded. When receiving this event the updateUrl function(line 27) is called, which sends an update containing the *window.location.hostname* which essentially is the hostname of the current page. Hostname is used since login forms may be located at different locations at different domains.

Next the script search the DOM for input fields of type "password" using the *getPwdInputs* function(line 16). This function iterates through all the input fields looking for password fields. If a password field is found, an event listener is attached to the field, listening for events of type "input" which are sent when the field changes[1]. When the password field changes a message containing the new length of the password is sent to the controller.

B.2 Controller

```
angular.module('human-computable-pws.controllers', [])
.controller("MainController",
    function($timeout, $scope, chromeStorage){
    //initial values
    $scope.user = "";
    $scope.pw=0;
    $scope.selectedSite=null;

    //function adding new site to the user
    $scope.newSite = function(){
        var newSite = new Site($scope.url);
        $scope.user.sites.push(newSite);
        $scope.selectedSite = newSite;
        chromeStorage.set("user", $scope.user);
        $scope.newSiteButton = false;
    }

    //try to load the user object from storage,
    //if no user is present, create new.
    try{
        chromeStorage.getOrElse("user",
        function(){
            var newUser = {
                name: "user",
                sites: [
                    new Site("accounts.google.com"),
                    ]
            };
            chromeStorage.set("user", newUser);
            return newUser;
        }).then(function(keyValue){
            $scope.user = keyValue;
        });
    }
    catch(err){
        console.log("Not run as extension")
    }

    //receive messages sent from the content script
    //if url or password length changes
    var tmp;
    function handleMessage(){
        if(tmp.pwValue){
            $scope.pw = tmp.pwValue-1;
```

[1]https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener

```
            console.log("pw: %o", $scope.pw);
            console.log("ombjects:   %o", $scope.user)
        }
        if(tmp.newUrl){
            $scope.url = tmp.newUrl;
            var site = searchList($scope.url, $scope.user.sites);
            if(!site){
                $scope.newSiteButton=true;
                $scope.selectedSite = null ;
            }else{
                $scope.newSiteButton=false;
                $scope.selectedSite = site;
            }
        }
    }
    //add eventhandler listening for messages from content script
    try{
        chrome.runtime.onMessage.addListener(
        function(message,sender){
            tmp = message;
            $timeout(handleMessage);
        });
    }
    catch(err){
        console.log("not extension")
    }
});
```

Listing B.2: Angular controller.

B.3 App.js file

```
var hcp = angular.module('human-computable-pws', [
    'human-computable-pws.controllers',
    'ngAnimate',
    'chromeStorage',
    'ngRoute',
    ]);

hcp.config( ['$compileProvider',
    function( $compileProvider  ) {
        var currentImgSrcSanitizationWhitelist =
            $compileProvider.imgSrcSanitizationWhitelist();
        var newImgSrcSanitizationWhiteList =
            currentImgSrcSanitizationWhitelist.toString().slice(0,-1)
            + '|chrome-extension:' +currentImgSrcSanitizationWhitelist.
            toString().slice(-1);

        console.log("Changing imgSrcSanitizationWhiteList from "+
            currentImgSrcSanitizationWhitelist+" to "+
            newImgSrcSanitizationWhiteList);
        $compileProvider.imgSrcSanitizationWhitelist(
            newImgSrcSanitizationWhiteList);
    }
]);

hcp.config(['$routeProvider',
function($routeProvider){
    $routeProvider.
        when('/',{
            templateUrl: 'partials/content.html',
            controller: 'MainController'
        }).
        otherwise({
            redirectTo: '/'
        });
}]);

function searchList(site, sites){
    if(sites == undefined) return false;
    for(var i = 0; i<sites.length; i++){
        if(sites[i].name == site){
            return sites[i];
        }
    }
    return false;
}



function Site(name){
    this.name = name;
    this.challenges = [];

    for(var j=0; j<14; j++){
        this.challenges.push(randomChallenge());
    }
}

function randomChallenge(){
    var letters = "abcdefghijklmnopqrstuvwxyz";
    var ch = [];

    for(var z=0; z<14; z++){
        ch.push(letters.charAt(Math.floor(Math.random()* 26)));
    }
    return ch;
}
```

Listing B.3: Angular launcher file.

B.4 View file (partial file)

```
<table
    align="center" class="challenges"
    ng-init="site = user.sites[user.sites.indexOf(selectedSite)]">
    <tr>
        <td>
            <img ng-src="letters/letter_{{site.challenges[pw][10]}}.png"/>
            <img ng-src="letters/letter_{{site.challenges[pw][11]}}.png"/>
        </td>
        <td>
            <img ng-src="letters/letter_{{site.challenges[pw][12]}}.png"/>
        </td>
        <td>
            <img ng-src="letters/letter_{{site.challenges[pw][13]}}.png"/>
        </td>
        <td></td>
    </tr>
</table>
<hr class="divider" style="margin: 0;">
<table align="center" class="challenges">
```

```
    <tr>
        <td>
            0<img ng-src="letters/letter_{{site.challenges[pw][0]}}.png"/>
        </td>
        <td>
            5<img ng-src="letters/letter_{{site.challenges[pw][5]}}.png"/>
        </td>
    </tr>
    <tr>
        <td>
            1<img ng-src="letters/letter_{{site.challenges[pw][1]}}.png"/>
        </td>
        <td>
            6<img ng-src="letters/letter_{{site.challenges[pw][6]}}.png"/>
        </td>
    </tr>
    <tr>
        <td>
            2<img ng-src="letters/letter_{{site.challenges[pw][2]}}.png"/>
        </td>
        <td>
            7<img ng-src="letters/letter_{{site.challenges[pw][7]}}.png"/>
        </td>
    </tr>
    <tr>
        <td>
            3<img ng-src="letters/letter_{{site.challenges[pw][3]}}.png"/>
        </td>
        <td>
            8<img ng-src="letters/letter_{{site.challenges[pw][8]}}.png"/>
        </td>
    </tr>
    <tr>
        <td>
            4<img ng-src="letters/letter_{{site.challenges[pw][4]}}.png"/>
        </td>
        <td>
            9<img ng-src="letters/letter_{{site.challenges[pw][9]}}.png"/>
        </td>
    </tr>
</table>
```

Listing B.4: Angular view. Only included the table showing the challenges for readability.