A Convenient Method for Securely Managing Passwords

J. Alex Halderman
Princeton University
jhalderm@cs.princeton.edu

Brent Waters
Stanford University
bwaters@cs.stanford.edu

Edward W. Felten
Princeton University
felten@cs.princeton.edu

ABSTRACT

Computer users are asked to generate, keep secret, and recall an increasing number of passwords for uses including host accounts, email servers, e-commerce sites, and online financial services. Unfortunately, the password entropy that users can comfortably memorize seems insufficient to store unique, secure passwords for all these accounts, and it is likely to remain constant as the number of passwords (and the adversary's computational power) increases into the future. In this paper, we propose a technique that uses a strengthened cryptographic hash function to compute secure passwords for arbitrarily many accounts while requiring the user to memorize only a single short password. This mechanism functions entirely on the client; no server-side changes are needed. Unlike previous approaches, our design is both highly resistant to brute force attacks and nearly stateless, allowing users to retrieve their passwords from any location so long as they can execute our program and remember a short secret. This combination of security and convenience will, we believe, entice users to adopt our scheme. We discuss the construction of our algorithm in detail, compare its strengths and weaknesses to those of related approaches, and present Password Multiplier, an implementation in the form of an extension to the Mozilla Firefox web browser.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Authentication*; K.4.4 [**Computers and Society**]: Electronic Commerce—*Security*

General Terms

Design, Security, Human Factors

Keywords

Password security, website user authentication

1. INTRODUCTION

Logging in with usernames and passwords has become one of the most ubiquitous and most reviled rituals of the Internet age. On the web, passwords are used by publications (nytimes.com), blogs (slashdot.org), webmail providers (gmail.com), e-commerce sites (amazon.com, ebay.com), and financial institutions (etrade.com). Elsewhere, they serve as authentication for Internet service providers, email servers, and local and remote host accounts.

Users are asked to generate, memorize, and keep secret a growing number of passwords as they join new password-protected sites over time. Since at least 1997 [6], this trend has been recognized as both a nuisance and a security risk. Research has shown that typical users can be trained to select and remember a single secure password [8], but multiplying this dozens or hundreds of times is sure to push the physiological limitations of human memory. Users face a difficult dilemma: they can either stretch their limited mental storage by choosing simpler passwords or reusing them across sites, or they can resort to memory aids, such as writing passwords down or using password management software. Unfortunately, the inconvenience of available software has prompted many frustrated users (including, regrettably, the authors) to resort to the former, insecure strategy.

One common practice we have observed is to choose a single strong password and use it for many accounts. The danger, of course, is that an attacker could learn the password from one account and guess other places where the user is likely to use the same login information. Since a small number of web sites are used by a large fraction of people, sites like Amazon, EBay, Yahoo!, and large financial institutions would be very likely guesses. There are many possible avenues for opportunistically stealing passwords, including insider attacks, remote exploits of weakly secured sites, key logging on public terminals, and web site spoofing. Since sites that don't store financial or billing information are likely to have lower security standards (such as not using SSL for authentication), passwords can be stolen and reused to compromise sites that guard more sensitive data. Service providers can enforce policies about password strength and they can track failed logins, but this insecure password reuse behavior is very difficult to police.

Our solution avoids password reuse entirely while requiring the user to memorize only a single master password. Rather than store passwords chosen by the user, our scheme applies a type of strengthened hash function to deterministically generate high-entropy passwords every time they are needed. This approach, we will argue, is both convenient for users and highly secure, a combination not offered by previous designs.

Convenience. Since we use a deterministic method to calculate account passwords, users can run our software on different machines to access their passwords from almost

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

any location. We rely on very little input—the user's identity (perhaps an email address), the name of the account (e.g., the server domain name), and a single master password chosen and memorized by the user. This means users can transport in their heads all the inputs needed to regenerate their passwords. It also ensures that passwords for different accounts will appear to be unrelated.

Security. While the idea of hash-based password schemes is not new [6, 15], previous designs suffered from a potentially serious weakness. An adversary who stole one account password could conduct an offline brute force attack on the user's master password and thereby learn all the user's other passwords. The attack is simple; for every potential password the attacker applies the hashing algorithm for that particular site and compares it to the known site password. The rate of an attacker's search is limited by the time to compute the hash function, which can be as low as one microsecond in previous schemes. In this example, a conventional machine can search a password space of size one million in under one second. Given the low entropy of typical user passwords, this kind of attack constitutes a major threat.

To protect against such attacks, we apply techniques for "strengthening" user passwords to increase the cost of the hash computation so that it incurs an acceptably small delay for the user while substantially throttling brute force attacks. In some attack scenarios, this increases the time required to search a password space by a factor of approximately 10^8 compared to other approaches. We cache partial results on the local system so that passwords can be generated quickly after an initial computation time of about 100 seconds when our program is first installed.

Browser integration. We implemented our scheme as an extension to the Mozilla Firefox web browser. Browser integration of password management is important for multiple reasons. First, most of a user's passwords are likely to be used for web sites. Building our system into the browser allows it to work as conveniently and transparently as possible, minimizing the burden imposed on the user and increasing the chances that the system will be used in practice. Second, browser integration allows our system to offer some protection against spoofing and phishing attacks, since by default the password program will use the name of the server that will actually receive the submitted password, even if an attacker has tricked the user into believing she is connecting to a different site [15, 5].

Outline In Section 2 we reference prior work related to password systems and password strengthening. We describe our novel password computation scheme in Section 3 and discuss its security. In Section 4 we compare our approach to related proposals in terms of security and convenience. Section 5 focuses on a prototype implementation we have developed, including discussion of the challenges we faced. Finally, in Section 6 we conclude.

2. RELATED WORK

2.1 Password Systems

Recent years have seen a wide variety of proposed solutions to the management password problem. One approach is to create centralized, trusted authentication services, such as Microsoft's Passport initiative [1]. Users log in to these services using a single password and are then authenticated to participating web sites. Such systems have met limited success for two reasons. First, they require technical changes on the part of every site that uses them for authentication, something even Microsoft has lacked the clout to achieve on a widespread scale. Second, users have been cautious about placing so much trust and privacy-sensitive information under the control of a centralized system. Passport's security is analyzed in [12].

Several other schemes do not rely on server-side changes or trusted third parties. These proposals operate in two general ways: either they let users choose their own passwords and then store them somewhere safe, or they assign fixed passwords for each site or service that can be computed whenever they are needed. The Password Safe application [16] takes the former approach. It stores passwords and other sensitive data in an encrypted database on the user's machine, secured with a master password. The Lucent Personal Web Assistant (LPWA) [6] applies the latter approach. It operates as an HTTP proxy server that users access with a master username and password. They can then tag web site password fields to be automatically filled in with values derived from a hash-based function of the user's master password and the domain name of the web site. PwdHash, a recently released utility by Ross *et al.* [15], applies a similar hash-based technique on the client side. It functions as a web browser plug-in, seamlessly replacing values submitted via web site password fields with hashes of those values and the site's domain name. PwdHash is primarily intended to provide a defense against "phishing" or "spoofing" attacks by linking site passwords to the domain name of the server to which they are actually sent. In Section 4 we give a detailed comparison between the above schemes and our approach.

2.2 Strengthening Passwords

Researchers have long recognized that passwords guarded by one-way functions were vulnerable to brute force attacks, and that those functions should be made slower as a defense. The classic example is the Unix *crypt* function examined in 1979 by Morris and Thompson [14]. However, this early work was concerned with securing the password verification procedure. Here we wish to strengthen the passwords themselves.

There have been two primary techniques for strengthening low-entropy keys or passwords for protection against brute force searching attacks. Both approaches share the common goal of increasing the (average) amount of computation needed to confirm or eliminate a guess of a user's password. In both techniques the adversary has the output of a strengthened hash function and wishes to know the master password put into it.

The first kind of technique, as studied by Kelsey *et al.* [11], derives a new password by repeatedly iterating a hash function on the original master password. Assuming there are no shortcuts, an adversary's best attack is to apply the function to each guess herself. Therefore, if the scheme is parameterized to use k iterations an adversary will need to compute k hash functions for each guess.

The second type of technique is due to Abadi *et al.* [3] and Manber [13]. In these approaches, the password is concatenated with a random value known as a "password supplement" before it is hashed. When checking a password guess an attacker will need to perform a hash for each possible supplement until the space is searched. Thus, if the pass-

word supplement space is of size k the attacker will need to perform k hashes before she can completely eliminate one guess as a possible password.

The primary drawback of this second approach is that a regular user also needs to search through the space and needs a method for determining if the password supplement is correct. If a strengthened password is integrated into a login system, some extra data can be stored on the server and the server can perform the search. However, if we don't want the client to carry state and we want to be compatible with an oblivious web server, then the client will have no way to test each password supplement short of attempting a login. For these reasons we choose to build on the deterministic approach of Kelsey *et al.*, although it is possible that other techniques could be modified to meet our requirements.

2.3 Password-Authenticated Key Exchange

There has been recent interest in the subject of password-authenticated key exchange [10, 7, 4]. In these schemes we have two parties that authenticate each other only by a shared password and exchange a random session key. (We contrast this with SSL and SSH connections where one party, typically the server, has a known public key and the client authenticates himself with a password over a secure channel established using the public key.) The common goal of these schemes is to protect the shared password, which will typically have low entropy, from being computed by an attacker in an offline dictionary attack.

Our ideas (as described in Section 3) can be used with password-authenticated key exchange if the password shared by a client and server is generated from a hash of the client's master password and server name. In this case we will still want to strengthen the password hashing function. The properties of password-authenticated key exchange protocols protect against a third party discovering the shared password. However, the server can still execute a dictionary search for the master password using the shared password that it legitimately knows.

3. OUR SCHEME

Master passwords in a hash-based password generation system can be particularly vulnerable to brute force attacks due both to the low-entropy of passwords memorable by humans and to the ability of an adversary to execute a low-cost offline password guessing attack. While there has been much effort put into alternate password entry techniques that increase memorable entropy (such as graphical passwords [9]), common character-based passwords are still by far the dominant mechanism. For this reason, our approach focuses on making guessing more time-intensive so as to raise the cost of a brute force attack.

A hash-based system is inherently vulnerable to an offline guessing attack of the master password by an adversary who has obtained one or more of the user's site specific passwords. This adversary can execute a simple brute force attack by enumerating the potential master passwords (possibly trying the most likely ones first) and for each one executing the hash algorithm for that particular site and comparing it to a compromised site-specific password. The amount of time to execute this attack will be the number of passwords tested multiplied by the execution time for the hashing algorithm. If the algorithm can be computed quickly, like SHA-1, this attack can be quite effective.

In our construction we limit the vulnerability to such an attack by increasing the amount of time needed to compute the hash function. In particular, we use the technique of Kelsey *et al.* [11] of "key-stretching" via repeated application of a hash function. The basic concept is that a regular hash function, $f()$, is replaced with a new hash function, $f^k()$, where $f^k()$ is computed by repeatedly applying the hash function k times. For example, $f^3(x)$ is equivalent to computing $f(f(f(x)))$. If P is the set of passwords the adversary wants to test and we conjecture that for all $p \in P$ the fastest way for an adversary to compute $f(p)$ is by repeated applying the function, $f()$, k times, then the cost of a brute force attack will increase by a factor of k.

Of course, a regular user will have to perform the same iterated hash computation when calculating a site-specific password. If the delay is very long, the system will be too inconvenient for most users. Thus, we must strike a balance between usability and security in our system design and choice of parameters.

3.1 Construction

In our scheme we use two levels of iterated hash computations. The first level is executed once when a user begins to use a new machine for the first time. This computation is parameterized to take a relatively long amount of time (around 100 seconds in our implementation), but its result will be cached and for future password computations by the same user.

The next level of computation is used to calculate site-specific passwords. It takes as input the calculation produced from the first level computation as well as the name of the site or account for which the user is interested in generating a password.

Let $f(x)$ be a secure hash function, then a site password is calculated in the following two steps:

1. Set a variable, V, as:

   $$V = f^{k_1}(\text{username} : \text{master\_password})$$

   In this function the concatenation of the username and master_password is iterated k_1 times. The concatenation should be performed such that it can be uniquely parsed into a username and master_password pair.

2. The site_password is then computed as:

   $$\text{site\_password} = f^{k_2}(\text{site\_name} : \text{master\_password} : V)$$

   In practice the raw output of the iterated hash function will be translated into an acceptable password for web sites and other accounts. As in the first step, the concatenation is performed such that it parses into a unique triple.

In the first step the iterated hash function is calculated on the master_password and username. The username can be any unique identifier a user chooses such as an email address. The purpose of the username is to thwart dictionary attacks where an attacker would compute and store the iterated hashes on a list of candidate passwords. In our scheme the username serves as salt and an attacker needs to compute attacks on a per-username basis.

Since the intermediate value, V, is independent of any particular website, it can be cached and used for computations for site specific passwords. In practice the value k_1 will

Attack Type	Offline	Hashes per Guess	Time per Guess
No Information	No	N/A	N/A
Stolen Site Password	Yes	$k_1 + k_2$	100.1 s
Stolen Cache	Yes	k_1	100 s
Stolen Cache and Password	Yes	k_2	0.1 s

Table 1: Difficulty of various brute force attacks against our scheme. The times per guess are for a fast modern PC and assume the values of k_1 and k_2 used in our implementation.

be set to be much larger than k_2. We can set our parameters such that whenever a user begins using the scheme she will incur a relatively high delay for the first step to be computed, but from that point on the value V can be cached such that any per-site calculation of step two will take a small amount of time that is unnoticeable to the user. In practice we set the parameters such that the first step takes around 100 seconds on a fast modern machine and the second takes around a tenth of a second. In this manner our scheme strives for the best balance between security and usability.

3.2 Security Analysis

We will now analyze the effort needed by an attacker to learn a user's password for a particular site. We characterize four different types of attackers by the type of information they are able to obtain prior to the attack. The first attacker has no information except the user's username for the targeted site. The second type of attacker additionally knows the user's site password for a site other than the targeted site. The third type of attacker has the result of the first level of computation by a user, but does not know any site-specific passwords. The final type of attacker has both the result of the first level of computation and a site-specific password. Our analysis follows:

(1) No Information The first type of attacker has essentially no information about a user. This type of adversary is limited to performing an online attack against the user's password at the targeted site. This adversary will typically be limited by a velocity control mechanism at the site which will constrain the number of login attempts allowed per unit time and might even disable the account if too many unsuccessful attempts are logged. Since the adversary needs to mount an active attack she faces a high risk of detection. If successful, she will only gain a site-specific password, which cannot be directly used to attack the user's password for a second site. However, with this site-specific password she can then become the second type of attacker.

(2) Stolen Site Password The second type of attacker we consider has access to the targeted user's site password for at least a single site. The adversary might obtain this information by convincing the user to log in to a malicious website using a generated password, or else she might compromise an insecure site. The attacker can then perform an offline attack on the user's master password. She will need to spend $p(k_1 + k_2) > pk_1$ hash computations to search for the master-password where p is the number of attempts needed to guess a password. Since this is an offline attack, the attacker can execute it undetected. In practice, we believe, this is the most important adversary to guard against, because an opportunistic attacker can easily learn some site passwords without being detected. Since our scheme forces the adversary to spend over a minute of computation time per attempted password, the attack will be rate limited.

(3) Stolen Cache In the third type of attack we assume that the attacker steals or compromises a user's machine that has an intermediate value V stored locally. The adversary can execute an offline attack where she searches for the master password by executing the first step in the algorithm for each possible password and compares its output to the stolen value. Each guess will take k_1 hash iterations to test. Since $k_1 >> k_2$, this attack should not be significantly more successful than the previous one. Moreover, if the user is able to detect that her machine was compromised or stolen, then she would likely have enough time to change the passwords on her accounts before the adversary could break into them.

(4) Stolen Cache and Site Password For the final attack we consider an attacker who obtains the user's cached intermediate value and at least one site password. This could occur if the adversary who stole the machine also owned a malicious site that the user visited, or more likely, if the stolen machine also stored site-specific passwords. The adversary will only need to compute k_2 hashes for each guess. Since k_2 will be set significantly lower than k_1, the adversary will need less time to search through a password space. However, the iterated hash function will still require much more adversarial effort than a traditional hash-based system.

Our scheme provides a high level of protection for Scenarios 1–3 and a reasonable amount for the strong attack of Scenario 4. We note that it is important for users of our scheme to prevent their site-specific passwords from being cached locally so that an adversary who steals the machine has to execute attack Scenario 3 rather than Scenario 4. Indeed, our scheme replaces much of the utility of locally stored passwords.

We summarize the results of our analysis in Table 1.

3.3 Iterated Hashing

The security of our scheme rests upon the belief that for the adversary to search a password space of size n she must compute $n \cdot k$ hashes. This assumption is heuristic in nature and is similar to viewing the underlying hash function as a Random Oracle. There has been some evidence that compressing the iteration of hash functions is difficult. Kelsey

et al. [11] prove that if an adversary can compute a 2^m iterated hash with effort $2^m/\ell$ then the adversary can find a hash function collision about $\ell/2$ times as fast as a naive birthday attack.

While the result of Kelsey *et al.* is encouraging, in practice it must hold that the adversary cannot amortize the cost of computing k iterations across all n passwords she is interested in. Unfortunately, if we extend the Kelsey *et al.* proof in the natural way, the best we can show is that if an adversary can compute a set of n iterated hash functions in time $2^m/\ell$ then that adversary can be used to find a collision about $\ell/(2n)$ times as fast as a naive birthday attack.

In practice this reduction is not very strong (especially when n approaches the number of iterations). In fact, even this weak reduction requires the stronger assumption that the adversary would be able to tell which input hashes corresponded to the output hashes, which would not necessarily be the case in practice. Therefore, it seems we need to assume that the hash function behaves similarly to a Random Oracle from the point of view of the adversary. We leave it as an open problem to explore the difficulty of amortizing the cost of iterated hashing.

4. COMPARISON WITH OTHER APPROACHES

In this section we compare our approach to the client-side password manager schemes described in Section 2: Password Safe, LPWA, and PwdHash. We focus on their ease of use and their security against various kinds of attacks.

4.1 Usability

To be widely adopted, a password manager system needs to impose a minimal burden on end users, both when creating and retrieving passwords (the typical usage case) and in less common, but potentially trickier situations, like changing passwords and migrating from unmanaged passwords.

Typical Usage PwdHash is the easiest scheme to use in the typical case. Its operation is so seamless that were it installed in every browser since the foundation of the web, users would notice virtually no difference aside from improved security. The other approaches require a bit more user interaction for each password operation. In our scheme and Password Safe a user must explicitly interact with an application, and users in LPWA must type in a special character string in a password field to signal the scheme to fill in the password. The applications also have slightly different usable scopes. LPWA and PwdHash are limited to web site passwords, our approach protects user-selectable passwords in general, and Password Safe additionally secures arbitrary secrets like credit card numbers and PINs.

Password Changes and Migration All the hash-based schemes (LPWA, PwdHash, and our scheme) have difficulty with the transition from unmanaged passwords. Since these schemes generate rather than store passwords, users need to manually execute a password change for each password that will be managed.

PwdHash, LPWA, and our scheme face similar challenges dealing with password changes. In these techniques, web passwords are deterministically computed from the site's domain name and information provided by the user (such as a master password) that is global across every site the user visits. If this password is compromised or is required to be changed periodically as a matter of policy, the user must take additional steps to generate an alternate password. We describe how we mitigate this problem in Section 5.

Password Safe most successfully copes with these situations. Since it stores encrypted passwords, it can accept existing passwords rather than forcing the user to change them, and it can handle future password changes in a similarly straightforward manner.

Transportability To take the place of memorized passwords, password manager software needs to be easy to use not only in the sense of simplicity but also that of availability. The people most likely to need such a program—active Internet users with accounts on many web sites—are particularly likely to demand access to their passwords from multiple locations, such as from desktops at home and at work, from laptops while traveling, and from public terminals. The ease and reliability with which passwords can be transported and accessed from remote locations is an important criterion for these applications.

Password Safe is not designed to be particularly transportable; it can only be used where its encrypted password store is accessible. The user is forced to carry this data with her in a storage device (which can be inconvenient and raises synchronization issues) or else to access it over a network (which introduces a host of other security risks).

In contrast, the hash-based mechanisms were designed with transportability as a central concern. Since they compute passwords on demand rather than storing them, users can shift from one machine to another by moving only a very small amount of state—typically a single secret small enough for the user to carry in her head. The hashing schemes are distinguished by the means of accessing each software application. LPWA, a web-based service, might be expected to be the most accessible, but in fact it has been totally unavailable since 2000, when the service was taken offline. Since the technology is proprietary, the scheme is effectively no longer available to the public. This incident highlights the need for temporal as well as spatial availability. PwdHash and our scheme are distributed with complete source code, and they are based on simple algorithms that can be expected to outlast any particular implementation. Both these schemes require the software to be distributed securely using trusted servers, verified hashes, or cryptographic signatures. Unlike the other schemes, our design imposes an initial delay of about 100 seconds when a user runs it for the first time on a new machine. While this may hinder availability in some instances, we believe it is a worthwhile trade off for greatly increased security.

PwdHash also supports a secondary "password" that can be stored locally on the user's machine and used for all web sites. Users can trade transportability for security by setting this value to a long secret string. While this would make attacks on the master password much more difficult, the value would need to be explicitly stored and transfered when moving to another machine.

4.2 Security

Since these password management schemes are intended to solve a security problem, perhaps the most important single criterion by which to evaluate them is their resistance to attacks. At the very least, they should be more successful

Scheme	Stolen Password	Stolen Data	Stolen Data and Password
Password Safe	N/A	35.2 (7.5)	$\geq$ 35.2 (7.5)
LPWA	43.5 (9.3)	N/A	43.5 (9.3)
PwdHash	$\leq$ 45.8(9.7)	45.8 (9.7)	45.8 (9.7)
Our Scheme	19.2 (4.1)	19.2 (4.1)	25.9 (5.5)

Table 2: Bits of password entropy (or number of random lower case letters) needed to resist attack for an average of one year under three attack scenarios.

Scheme	Stolen Password	Stolen Data	Stolen Data and Password
Password Safe	N/A	74.6 secs	$\leq$ 74.6 secs
LPWA	0.5 secs	N/A	N/A
PwdHash	0.1 secs	0.1 secs	0.1 secs
Our Scheme	116 days	116 days	2.8 hours

Table 3: Resistance to a dictionary attack under three attack scenarios—Times to test 100,000 password guesses using a fast modern PC.

than a naive user who selects the same password for every account, while the best we can hope for is to approach the security of a user who memorizes and keeps secret a unique random password for each account.

We compare the schemes under the three *offline* attack scenarios described in Section 3. These are: (2) the adversary has access to a stolen site password; (3) she has access to stolen data from the user's machine; and (4) she has stolen both local data and a site password. In each case the adversary wants to learn the target user's password for an additional site. To quantitatively compare the schemes, we consider an example in which the adversary is willing to devote one year of computation (on a typical fast PC) to the attack, and we determine how many bits of password entropy are required to fend her off at least half the time.

This might seem like a lot of power to devote to attacking one user, but we believe it is a realistic worst-case assumption. An intelligent adversary would choose her targets carefully, perhaps by stealing demographic information such as income level from a compromised web site. She might also determine ahead of time whether the potential target had accounts on the most desirable sites, such as banks. To speed things up, she could parallelize the attack using a distributed network of compromised machines. Breaking the password manager's security would hand the adversary the keys to a user's entire digital identity, so costly and sophisticated attacks should be expected.

Password Safe, which stores an independent password for every site, is not vulnerable in Scenario 2. Attacks are possible in Scenarios 3 and 4, when the adversary has access to the encrypted password database. Existing software [17] attempts to discover the master password by performing a dictionary attack against a password verification code stored at the beginning of the database. The code is computed by repeated applications of the Blowfish cipher to a hash of the password. This attack can be executed at a rate of about 1340 tests per second on a modern PC. To thwart an adversary using this software for an average of a year, the user would need to select a password with 35.2 bits of entropy, equivalent to a randomly selected password of 7.5 lower case letters. Even faster attacks may be possible in Scenario 4 using one or more site passwords.

LPWA computes passwords using a hash-based function and does not store any data to disk, so it is equally vulnerable in Scenarios 2 and 4, and Scenario 3 is not applicable. Since we do not have access to the LPWA software, it is more difficult to estimate the cost of a brute force search. However, if we assume the authors implement the pseudo-random functions in the LPWA algorithm as HMACs instantiated with the SHA-1 function, then we can find a simple attack that takes five applications of the SHA-1 function per guess (or roughly 5 microseconds per guess on a modern computer). At this rate, an attacker could test 3.6×10^{12} guesses in a year, and users would need an average of 43.5 bits of entropy to remain secure. This is equivalent to a 9.3 character password of random lower case letters.

We need to consider the security of PwdHash in two different situations. The first is if the user does not use a long random string stored locally. Then the scheme will be vulnerable to an offline brute force attack in Scenarios 2, 3, and 4. PwdHash uses an MD5-based HMAC computation to generate site passwords. This takes approximately one microsecond to compute on modern machines, so the adversary could test 3.15×10^{13} passwords a year, and an average of 45.8 bits of entropy (or 9.7 randomly selected lower case letters) would be needed to thwart her attack.

If a long, random local string is used, then the scheme can be very secure against an attacker of Scenario 2 (but at the expense of the transportability problems mentioned above). However, since this second "password" is stored on disk, an attacker in Scenario 3 or 4 can execute the brute force attack discussed in the previous paragraph as if no secondary password were present.

Finally, for our system, in Scenario 2 the adversary would need to run a brute force attack against the entire strengthened hash function, which takes approximately 100.1 seconds to compute on a modern PC. That means that a brute force attacker could try at most about 316,000 passwords in a year. To ensure that an average attack takes more than a year, a user would need to select a password with 19.2 bits of entropy, equivalent to a randomly selected password of 4.1 lower case letters. In Scenario 3, the attacker could steal the cached intermediate value and use it to check the results of a brute force attack on the master password.

This would reduce the time for a single guess to 100 seconds, requiring about the same entropy as in Scenario 2. In Scenario 4, the adversary could use the stolen intermediate value to make guesses much more quickly and could check the results against the stolen site password. This attacker could test a password in about 100 milliseconds on a typical machine, so it would take about 25.9 bits of password entropy to fend off her attack for an average of a year. This is equivalent to a random password of 5.5 lower case letters.

Based on this analysis (as summarized in Tables 2 and 3), we see that users under our scheme need to memorize up to approximately 45% fewer bits of password data than Password Safe, 58% fewer bits than PwdHash users and perhaps 56% fewer bits than LPWA users for a comparable resistance to brute force attacks. Since passwords chosen by typical users usually have fewer bits of entropy than random passwords of the same length, this reduction is likely to be a significant security benefit in practice. With minimal training, users can easily create and remember passwords with entropy equivalent to 4–6 random characters [8], but it would be a substantial burden to handle as many as 10 random characters, as required for security in some other schemes.

As an additional security benefit, our scheme can be adjusted in the future to increase the amount of strengthening in the hash function to compensate for advances in computational power. If the adversary's computational power doubles about every 1.5 years, users will need to add $6\frac{2}{3}$ bits of password entropy every decade to maintain the same level of security. Our scheme could be recalibrated every few years to keep the required password entropy within a fixed range (at the cost of requiring users to change all their site passwords).

As we have seen, our approach offers a very high degree of security while remaining easy to use and conveniently portable. We believe that this low cost, high benefit combination will be appealing to users.

5. IMPLEMENTATION

To test our system, we implemented the construction described in Section 3 as an extension to the Mozilla Firefox web browser. Our program, called Password Multiplier, is available online at:
`www.cs.princeton.edu/~jhalderm/projects/password/`.

Password Multiplier uses Mozilla's cross-platform scripting tools and user interface language (XUL), which allow it to integrate neatly into the browser and to match its look and feel on each operating system. We support Firefox's three primary platforms—Windows, Linux, and Mac OS X.

One of the principal challenges we faced was achieving the proper balance between attack resistance and user convenience. A brute force attacker would choose the fastest available hash implementation. To make the attacker's job more difficult, we also need to choose a fast implementation so that we can perform as many hash iterations as possible within the longest delay we deem acceptable—100 seconds for the one-time initialization and 100 milliseconds for each subsequent password generation. Mozilla's interpreted scripting language is not efficient enough to meet these performance requirements, so we implemented the iterated hash functions in compiled platform-specific XPCOM objects. These employ the highly-optimized SHA-1 hash function code from the OpenSSL [2] library. After testing performance on various modern machines, we parameterized the system to use 10^8 iterations during the initialization phase (k_1) and 10^5 during password generation (k_2).

Figure 1: Before using Password Multiplier for the first time on a new system, the user needs to "authorize" it. This involves a one-time computation on the user's email address and master password that takes around 100 seconds. The result is cached for future use.

Here is how a typical user experiences Password Multiplier. Before using the program for the first time on a new machine, the user needs to "authorize" it using the interface depicted in Figure 1. This dialog box prompts the user for her email address (which serves as a unique identity) and her master password, which must be at least eight characters in length and is entered twice to guard against errors. The program gives advice on picking a strong password to encourage users to follow good security practices. Users who have used the application before on a different machine are instructed to enter the same address and password to ensure that the correct passwords are computed.

With this information, the program performs the initialization process, in which it computes the hard part of the strengthened hash function as described in Section 3. The result is cached to disk so that this process only has to be performed once per user per system.

After it has been initialized, Password Multiplier can be invoked by double-clicking on any web site password field, or by highlighting a password field and pressing Alt+P. It presents the Multiply Password window (shown in Figure 2), which asks the user to re-enter her master password and confirm the domain name of the web site for which the password is used (we automatically fill in the name of the server that will receive the HTML form submission). The generated site-specific password is copied into the site's password field.

Password Multiplier supports both HTML form password inputs and standard HTTP authentication prompts. In both cases, it interacts correctly with the browser's internal saved-password manager. The passwords computed by our pro-

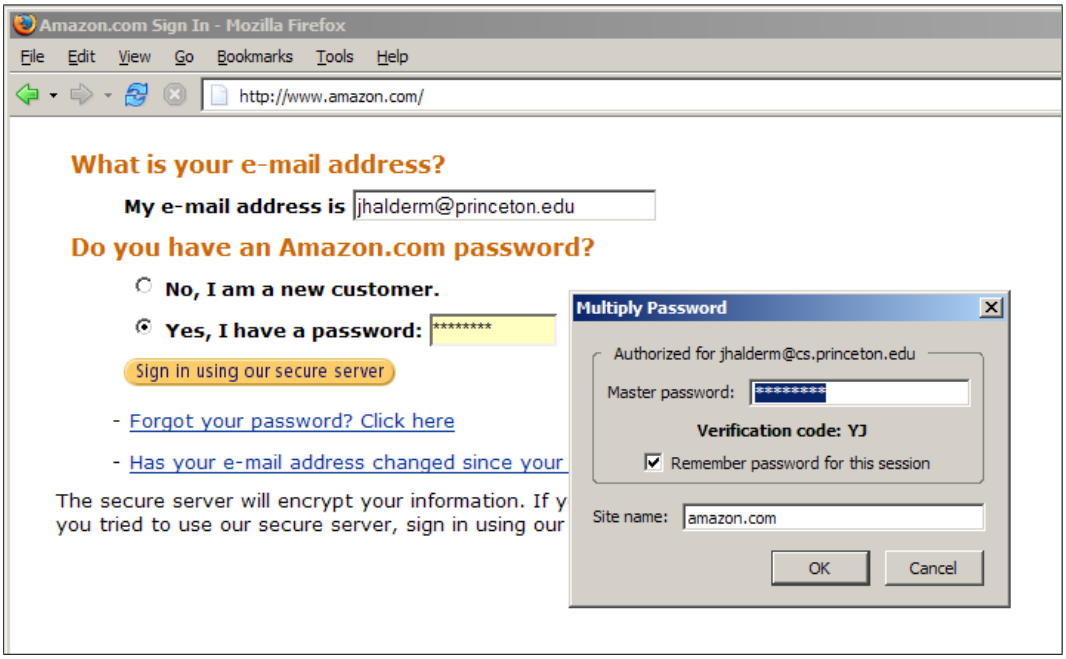

Figure 2: Double-clicking a password field (or highlighting it and pressing Alt+P) invokes the Multiply Password window shown here. The site name is filled in automatically. After the user enters her master password, the program pastes her site-specific password into the site's password field.

gram consist of eight case-sensitive alpha-numeric characters derived by treating the output of the hash function as a 160-bit little endian number, converting it to base 62, and mapping it to the character set `[0-9A-Za-z]`. Passwords of this form contain about 47 bits of entropy (or about 41 bits if the case is ignored), so they will normally be at least as hard to guess as the master password itself.

5.1 Handling Password Changes

A challenge faced by all generator-style password managers is the need to support account password changes. A new password may be needed if the existing password was compromised, or system administrators may require users to change their passwords regularly as a matter of policy. While we believe these situations are uncommon for typical usage with web site accounts, it would be helpful for implementations to provide a mechanism for handling them.

Generating an alternate password requires extra inputs in addition to the user's identity, her master secret, and the identity of the target account. This means users must carry additional "state" when moving between machines, but implementations can be designed to minimize this inconvenience. Users could indicate a password change by incrementing a password index presented along side the account name. The index value could be stored with the name of the account on the local system, so there would be no need to set it again unless the user moved to a different machine. The index would be appended to the account name in the generation algorithm to produce a completely different password. Upon switching machines, the user would need only remember how many times she had changed her password for a particular account, rather than what her new password was. Unlike the password itself, this information would not need to be kept secret, so the user would be free to write it down, post it online, or use other reminders.

If a particular password was required to be changed regularly, the name of the period for which the password was valid could be used instead of a numerical index. For example, a password changed monthly might use the string `2005-Jan`, appended to the domain name, as the input to the generation algorithm. The password program could use an internal calendar to track when password changes were required and remind the user automatically.

The password change features in this section have not yet been added to our implementation but are planned for an upcoming release.

6. CONCLUSION

Many users (including those who should know better) fail to take adequate steps to protect their passwords. Often the cause is not a failure to understand that strong passwords are important, but rather frustration with the difficulty of doing the right thing. In our study we attempted to make strong password management more convenient. Whereas previous schemes were lacking in either transportability for mobile users or security against brute force attacks, our design achieves a balance of the two by using password strengthening techniques. Our implementation, Password Multiplier, is available on the web. We encourage novices and experts alike to try it.

Acknowledgment

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

7. REFERENCES

[1] Microsoft Passport service. http://www.passport.net.

[2] OpenSSL: The open source toolkit for SSL/TLS. http://www.openssl.org.

[3] Martín Abadi, T. Mark A. Lomas, and Roger Needham. Strengthening passwords. Technical Report 1997 - 033, 1997.

[4] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT*, pages 139–155, 2000.

[5] E. Felten, D. Balfanz, D. Dean, and D. Wallach. Web spoofing: An Internet con game. *Proc. 20th National Information Systems Security Conference*, 1997.

[6] Eran Gabber, Phillip B. Gibbons, Yossi Matias, and Alain J. Mayer. How to make personalized web browsing simple, secure, and anonymous. In *Financial Cryptography*, pages 17–32, 1997.

[7] Rosario Gennaro and Yehuda Lindell. A framework for password-based authenticated key exchange. In *EUROCRYPT*, pages 524–543, 2003.

[8] J. Jeff, Y. Alan, B. Ross, and A. Alasdair. The memorability and security of passwords – some empirical results, 2000.

[9] Ian Jermyn, Alain Mayer, Fabian Monrose, Michael K. Reiter, and Aviel D. Rubin. The design and analysis of graphical passwords. 1999.

[10] Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Efficient password-authenticated key exchange using human-memorable passwords. In *EUROCRYPT '01: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques*, pages 475–494. Springer-Verlag, 2001.

[11] J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Secure applications of low-entropy keys. *Lecture Notes in Computer Science*, 1396:121–134, 1998.

[12] David P. Kormann and Aviel D. Rubin. Risks of the Passport single signon protocol. In *Proc. 9th international World Wide Web conference on computer networks*, pages 51–58. North-Holland Publishing Co., 2000.

[13] U. Manber. A simple scheme to make passwords based on one-way functions much harder to crack, 1996.

[14] Robert Morris and Ken Thompson. Password security: A case history. *CACM*, 22(11):594–597, 1979.

[15] Blake Ross, Collin Jackson, Nicholas Miyake, Dan Boneh, and John C. Mitchell. A browser plug-in solution to the unique password problem, 2005. Technical report, Stanford-SecLab-TR-2005-1.

[16] Bruce Schneier et al. Password Safe application. http://www.schneier.com/passsafe.html.

[17] Joe Smith. Password Safe cracker utility. http://members.aol.com/jpeschel3/recovery.htm.