A Browser Plug-in Solution to the Unique Password Problem

Blake Ross  
blake@cs.stanford.edu

Collin Jackson  
collinj@cs.stanford.edu

Nick Miyake  
nfm@cs.stanford.edu

Dan Boneh*  
dabo@cs.stanford.edu

John C Mitchell  
jcm@cs.stanford.edu

Abstract

We describe a browser plug-in, called PwdHash, that improves web password security and helps defend against phishing attacks and some consequences of break-ins to low security web sites. Since the plug-in applies a cryptographic hash function to a combination of the user password, 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 PwdHash securely in a web browser 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 alternative techniques 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 over an SSL connection. This form of password authentication over an encrypted channel is vulnerable to two types of attacks:

- **Phishing scams**: By masquerading as a legitimate site, a phishing site obtains the user’s cleartext password for the legitimate site. Unfortunately, phishing is very effective at stealing user passwords, as documented in reports from the anti-phishing working group [APW].

- **Common password attacks**: Hackers exploit the fact that web users often use the same password at many different sites. Hackers break into a low security site – one that simply stores username/passwords in the clear – and directly pipe the retrieved password list into a high security site, such as a bank. This attack, which requires little work, often results in the theft of thousands of banking passwords.

While password authentication could be abandoned in favor of hardware tokens or SSL 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 an Internet Explorer browser plug-in for strengthening password authentication on the web. We believe that by providing something better than cleartext passwords over SSL, we can reduce the threat of the password attacks discussed above with **no server changes** and **little or no change to the user experience**. Since

*Supported by NSF.*
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.

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 of the user’s password using the domain-name as a salt. The browser plug-in captures user input to password fields and sends $\text{Hash}(\text{pwd, dom})$, where ‘pwd’ is the user’s password and ‘dom’ is the domain name of the remote site to which the form data (including the password) is sent. Since the hash computation is tailored to meet server password requirements, the resulting hashed password is handled normally at the server – no server modifications are required. This simple idea prevents phishing for passwords since the password received at the phishing site is not useful at any other site. More technically, the cryptographic hash makes it computationally difficult to compute $\text{Hash}(\text{pwd, dom2})$ from $\text{Hash}(\text{pwd, dom1})$, for any $\text{dom2} \neq \text{dom1}$. 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 lax security at high school reunion sites, for example.

Unfortunately, there are a number of difficulties associated with the design and implementation of a plug-in for password hashing. For example, the plug-in must recognize which user input to hash. If a user wishes to start using our plug-in, for example, she will have to enter both an unhashed password and a hashed password on the change-password dialog page for existing accounts. The plug-in cannot simply hash both password entries. There are also a number of security issues involving interaction with other browser functionality. A large part of this difficulty stems from the fact that a browser provides a form of operating system for scripts, plug-ins, and related code, without the kind of protection mechanisms that isolate independent processes in conventional operating systems. For example, JavaScript on phishing pages may 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 interaction with other browser scripts and plug-ins will also raise problems for a range of other possible plug-in projects, the obstacles we encountered and the solutions we developed may be relevant to other projects.

To summarize, our goals in the design and implementation of $\text{PwdHash}$ are: (1) no change to the user experience, and (2) strengthen password authentication using a browser plug-in without requiring any changes on the server side. Section 2 summarizes the main challenges in building $\text{PwdHash}$, while section 3 discusses the implementation solutions. Since some forms of password hashing have been used in other systems, we survey related work in Section 4.

2 Challenges in Implementing Password Hashing

We start with a description of the various challenges we faced when implementing password hashing in a web browser. Although our implementation is for Internet Explorer, these difficulties will come up in any modern browser.

- **JavaScript attacks.** How do we prevent JavaScript on a phishing page from stealing the user’s cleartext password?

- **Password reset.** After the plug-in is installed, it must help users reset their passwords at various web sites to the new hashed passwords.

- **Salting and encoding.** What parts of the domain name do we use when hashing? How do we encode the hashed value to comply with the site’s password requirements?
• **Roaming users and Internet Cafés.** Some users are not able to or are not permitted to install plug-ins. We must nevertheless enable these users to use their passwords.

• **Dictionary attacks.** Phishing sites obtain a hash of the user’s password that could be vulnerable to a dictionary attack. How do we prevent such dictionary attacks?

• **Auto-complete.** Our plug-in must be compatible with the password auto-complete database.

The following subsections describe how we addressed each of these problems. For completeness, we note that the hashed password is implemented as follows:

\[
\text{hash} = \text{PRF}_{\text{pwd}}(\text{dom})
\]

where PRF is a pseudo random function (such as AES), ‘pwd’ is the user’s password which is used as the PRF key, and ‘dom’ is the remote site’s domain name which is used as the input to the PRF. The ‘hash’ is then encoded as an eight character string that satisfies the site’s password encoding rules.

We emphasize that we are only concerned with attacks on our plug-in that originate on malicious phishing sites. Our plug-in is not designed to defend against spyware and keyboard loggers running on the user’s machine.

### 2.1 JavaScript attacks

Recall that HTML forms begin with a `<form action=URL>` that tells the browser where the form is to be submitted. Similarly, HTML password fields are marked as `<input type="password">`.

Naively, password hashing can be implemented as follows: (1) the plug-in listens for defocus events (events that trigger when focus leaves a field), and (2) upon defocus of a password field, the plug-in replaces the contents of the field with the hashed value (hashed with the domain name from the form action attribute). Thus, after the user enters his password, the password is replaced with the appropriate hash.

Unfortunately, this straightforward implementation of password hashing is fundamentally flawed and insecure. We describe a number of JavaScript-based attacks that illustrate the complexity of properly implementing password hashing. To mount these attacks the phisher need only embed a bit of JavaScript on the phishing page.

• **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, the browser sends the user’s password hashed with the banking domain name. 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 key stroke 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 plug-in leaves it unhashed. 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 just as the key is entered (Figure 4). The phisher thus obtains the password as it is typed in, as opposed to waiting until the web form is submitted.

These JavaScript attacks are presented in order of severity. Many other JavaScript attacks are possible, though these few examples are sufficient to make our point – implementing password hashing inside the browser securely is quite challenging given the overarching power of JavaScript.

### 2.1.1 Defenses

Our goal is to defend against these web scripting attacks *with minimal change to the user experience.* To do so, we designed our plug-in as follows:

1. The plug-in uses the low level Windows API to intercept all keyboard events sent to the browser window. Keyboard events sent to a non-password field are forwarded to the browser as usual.

2. The plug-in intercepts keyboard events sent to a password field and records internally the key pressed. The plug-in then changes the key value in the event to some “random” value and forwards the resulting event to the browser. Consequently, the browser and JavaScript running inside the browser, are never given access to the user’s password.

3. When the user clicks the submit button, Internet Explorer triggers a `BeforeNavigate2` event. Our plug-in intercepts this event and embeds the hashed password directly into the outgoing HTTP request. The hash of the user’s password is computed using the target domain as the salt.

This architecture prevents JavaScript keyboard monitoring attacks since JavaScript on the page never sees data entered into a password field. It also defends against domain rewriting attacks since the salt is determined only once the HTTP request is about to be sent out, at which point JavaScript can no longer change the target domain name.

Although this design prevents attacks on password fields, it does not defend against mock password fields (Figures 3 and 4). More generally, it does not defend against attacks that confuse the user into typing a password in an insecure location. We implemented three further defenses against such attacks. We believe the third defense, `password-key`, is the most interesting and the most effective.

**Password traffic light.** Our first defense is a new information toolbar in the browser window. The toolbar displays green when current focus is in a password field and displays red otherwise. Thus, when focus is an insecure location (such as a text field or a mock password field), the toolbar is red to inform the user that it is unsafe to enter a password. This feature is very helpful for security conscious users (like us). However, novice users are unlikely to look at the password toolbar every time they enter their password. Also, a sophisticated attacker might spoof the traffic light itself, tricking the user into entering an unprotected password. Starting with Windows XP Service Pack 2, scripts can no longer create pop-up windows outside the browser’s content area, but they still could simulate a pop-up window containing a fake traffic light, contained entirely within the browser’s content area.
<form>
<input type="hidden" name="secret" value=""/>
<input type="password" name="password"
onKeyPress="this.form.secret.value +=
    String.fromCharCode(event.keyCode);"/>
</form>

Figure 1: Keyboard monitoring

<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>

Figure 2: Domain rewriting attack

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

Figure 3: Mock password field

<input type="text" name="spoof" 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']); ?>

Figure 4: Online mock password field
Keystream monitor. Our second defense, the keystream monitor, contains two components: a recording component and a monitor component. The recording component records all passwords that the user ever types into a password field. It stores a one-way hash of these passwords on disk. The monitor component monitors the entire keyboard key stream. It looks for a consecutive sequence of keystrokes that matches one of the user’s passwords. If such a sequence is keyed in while focus is not on a password field, the user is alerted.

This approach is quite effective at warning users when they type a password in an insecure location. However, it has three limitations. The most severe weakness 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 already obtained all but the last character of the user’s password. Another problem is that storing hashes of user passwords on disk could lead to an offline password dictionary attack if the user’s machine is stolen (however, the same is true for the browser’s auto-complete password database). Furthermore, since novice users tend to choose poor passwords, it is possible that their passwords will occur naturally in the keystream, leading to false alarms that cause the user to disregard monitor warnings. As a result, we consider the keystream monitor a partial defense at best.

Password key. The discussion above suggests that a new mechanism is needed for explicitly informing the desktop that a password is being entered. The desktop can then take steps to protect the password. After thinking about this for some time, we converged on the following clean solution we call password-key. The idea is to introduce a dedicated keyboard key called a “password-key.” Our plug-in trains users to always press the password-key just before entering their passwords. We imagine that future keyboards might have a dedicated key marked “password,” but for now we use the ‘F2’ key, which is not used by IE, Firefox, or Opera. The semantics of the password-key inside our plug-in are as follows:

- Our plug-in monitors all keyboard events using the low level Windows API. Normally, it passes all keyboard events to the browser. However, it blocks all password-key (F2) events from reaching the browser.

- When the user presses the password-key while focus is in a password field, the plug-in switches into password mode and does the following: (1) it internally records all subsequent key presses, and (2) it replaces the user’s keystrokes with random keys and passes the resulting events to the browser. This continues until focus leaves the password field, at which point the plug-in reverts back to normal mode. In other words, all keystrokes entered from the moment the user presses F2 until focus leaves the field are hidden from the browser and from scripts running inside the browser. Hence, JavaScript keyloggers (Figure 1) cannot steal the cleartext password. As before, our plug-in embeds the hashed password directly into the HTTP request once the BeforeNavigate2 event is triggered.

- If the user ever presses the password-key while focus is on a non-password field, our plug-in alerts the user not to enter a password. This protects users from attacks that confuse them into entering a password in an insecure location.

- Finally, if the user types text into a password field without first pressing the password-key, the plug-in pops up a message saying: “for your protection, please always press F2 before typing in your password.” This trains users to always press the password-key prior to typing in their password.
Thus, users continue to enter their passwords into password fields as they currently do. However, the password-key protects them from malicious scripts downloaded from the remote site.

We emphasize that since we implemented the password-key in a user-level application (a browser plug-in), we can only defend against malicious web scripts found at phishing sites. We do not defend against spyware and keyboard loggers already installed on the user’s machine. Keyboard loggers can directly listen to keyboard events (using the same Windows interface as our plug-in) and obtain the user’s cleartext password. Although this is orthogonal to the focus of this paper, we note that one potential solution is to implement the password-key mechanism inside the OS kernel. That is, the kernel protects user passwords and embeds them directly into outgoing HTTP requests. We leave this as a promising direction for future research.

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. 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 introduces a considerable change to the user experience. 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. Our design enables web users to safely enter passwords in the browser window as they currently do.

Problem sites. We conclude this section by noting that we found a small number of sites, such as yahoo.com, that implement their own password hashing using JavaScript on their login page. 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. We discuss a solution for these sites in Section 2.3.

2.2 Password reset after plug-in install

Once users install our plug-in, they can no longer login to their web accounts: the browser will now send password hashes instead of cleartext passwords. Hence, the user must manually reset his password at every site where he has an account. This can be done gradually over time; there is no need to update all web accounts at once. At each site, the new password will be set to the hash of the user’s password, using that site’s domain name as the salt.

Our plug-in helps the user with the password reset process by using IE’s auto-complete database. After installing the plug-in, when the user visits a web site, our plug-in checks whether there is an existing auto-complete entry for this web site. If so, it suggests that the user update her password. Once the user updates the password, the plug-in updates the auto-complete database so that this warning is not issued again. We discuss the auto-complete database in more detail in Section 2.7.

Unfortunately, there is an extra complication. 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 since the email password is entered in a password field, it will be hashed by our plug-in. The resulting hash will be sent to the site where it will be rejected since it does 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. The plug-in sends a hash of the current password, and as a result, the password change request is rejected.
We resolve this problem by providing the user with the means to disable password hashing when necessary by double clicking on the password field whose contents should not be hashed. As discussed above, this toggle is mostly needed for password resets.

The hashing toggle is also useful for disabling hashing for fields of type “password” that do not contain a password. For example, sites which ask for a person’s social security number or numerical PIN often provide a password field so that the number is not echoed on the screen. Clearly, this field should not be hashed. The user can double click on such fields to manually disable password hashing. However, we rely on an easier solution based on the password-key mechanism described in the previous section. Recall that we train the user to press the password-key prior to entering a password. If the user enters text into a password field without first pressing the password-key, the plug-in says, “To enable PwdHash protection, please press F2 before typing your password. Click Don’t Hash if you want to allow the site to view your secret data.” If the user presses the Don’t Hash button, password hashing for that field is disabled.

We also propose an alternate, cleaner solution that we have not implemented because it requires a slight extension to HTML. The idea is to add a new field type called “secret.” Fields of type password (which exists in HTML today) and fields of type secret (which is a new type) will not echo user input to the screen. The difference between the two is that fields of type password will be hashed, whereas fields of type secret will not. Thus, password fields will be strictly used for passwords, whereas secret fields will be used for other types of private information. We note that there is no added security risk in introducing this new field type – the password-key and password toolbar will prevent users from entering their passwords into a field of type secret.

Finally, we note that giving users the ability to disable password hashing by double clicking the field is a potential vulnerability. Phishers could create a spoof password reset page that asks for the user’s current password. The user is thus fooled into disabling password hashing and sending the cleartext password to the phisher. Note, however, that disabling hashing when entering the current password is only needed the first time that the user resets her password after installing the plug-in. On subsequent password resets, hashing the current password should be enabled. Consequently, spoofing the password reset page should not present a serious threat over time.

2.3 Salting and encoding problems

Recall that we use the target domain name as the salt when hashing the password. We found a few problematic sites that need special attention. The two difficulties are as follows:

- **What salt to use?** At some sites, the user resets their password at some domain $A$ but the login page is at some different domain $B$. For example, at passport.net the password reset page is located at memberservicesnet.passport.net whereas the login page is at login.passport.net. In principle the two domains could be completely different. Then after password reset, the user’s password is set to $H_A = H(\text{pwd}, A)$. However, during login, the browser sends $H(\text{pwd}, B)$, which will be rejected since it does not equal $H_A$.

- **How to encode the hash?** 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 encoding algorithm must depend on the target site.

The plug-in addresses both issues using a preset configuration file. The configuration file consists of a short sequence of rules where each rule has the following format:

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

8
For example, a rule might look like

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

The rule means that for domains that match "*.com" we use the two top-level domains as the salt and use encoding algorithm number 1. 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 plug-in contains five hash encoding algorithms that seem to cover the variety of password requirements enforced on the web. The default encoding algorithm, `encode-alg-1`, satisfies most sites. Furthermore, `encode-alg-0` means no hashing at all (i.e. sending the cleartext password). This encoding algorithm is only used for sites like `yahoo.com` that implement their own password hashing algorithm using JavaScript on their login page. Other encoding algorithms satisfy other password requirements by including at least one alphabet and one numeric character, by including one non-alphanumeric character, and so on.

### 2.4 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 on a machine at an Internet Cafe. Still, we need to provide the means for them to compute their password hashes.

We address this problem by providing a web page to help such end users. See Figure 5. Users are asked to enter both the domain name where they want to login and their password. Then JavaScript on the page computes the password hash and pastes the result into the clipboard. The user can then paste the hashed password into the desired password field. We stress that this all takes place on the user’s machine so that the user’s cleartext password is never sent on the network.
2.5 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. An extreme version, using ultra-slow hash functions, is proposed in [HWF05].

- **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 plug-in would have to make multiple login attempts with the user’s password. However, the plug-in often cannot tell whether a particular login request succeeded or failed. Furthermore, web sites often lock up after several failed login attempts.

We chose to defend against dictionary attacks using an optional feature we call *global plug-in password*. The user can specify a global plug-in password that is incorporated into the salt of all the hashes that the plug-in produces. Thus, in order to mount a dictionary attack, a phisher has to guess both the user’s web password and her global plug-in 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 plug-in 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. Sadly, they are also mired by patents.

2.6 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 distinct passwords, one for every security level (e.g. one password for all financial sites, one password for all news sites, etc). The PwdHash plug-in ensures that a break-in at one financial site will not expose the user’s password at all other banks.

2.7 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 stores the hashed password 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.

3 Implementation

We implemented our prototype plug-in as an ActiveX Browser Helper Object for Microsoft Internet Explorer. The plug-in registers three new objects: an entry in the Tools menu (for user-configurable options), a new toolbar (the “traffic light”), and the password protection service.
Internet Explorer provides sinks that allow Browser Helper Objects to react to events on an HTML page. 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 IE allows page elements to react to these events before they “bubble” up to the plug-in at the top level. Since our plug-in must intercept keystroke events before browser scripts, we intercept keystrokes using low-level Windows keyboard hooks.

When the password-key (F2) key is pressed, the plug-in determines whether the active element is a password field. If it’s not a password field, the user is warned that it is not safe to enter his password. If it is a password field, the plug-in 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 plug-in maintains a translation table for each of these password fields, mapping mask characters back to the original keystrokes. This masked characters approach allows the user to backspace and delete characters at arbitrary positions within the password field without confusing the plug-in.

### 3.1 User Interface

The main user interface of PwdHash consist of the menu, the password traffic light icon, and double-clicking on password fields.

- **Menu** Once the plug-in is installed, a PwdHash option is added to Internet Explorer’s Tools menu. Using this menu item, the user can turn the plug-in on or off and control several options.

- **Password traffic light** PwdHash adds a small traffic light icon to the toolbar. The icon acts as a simple indicator that informs the user whether or not it is safe to enter his password into the currently active field. The icon turns green when the focus is in a password field and hashing is turned on, and stays red otherwise. Clicking on the traffic light icon displays information about the current page, including what salt will be used.

- **Double-clicking password fields** PwdHash allows the user to toggle the hashing state for a password field by double clicking it.

### 3.2 Limitations

The implementation of password hash currently has the following limitations:

- **Other applications.** The mshtml.dll HTML parser is used in various applications other than the IE browser. For example, it is used to render HTML within Outlook and used with the AOL shell. Some of these applications do not support a plug-in infrastructure, and hence we cannot, currently implement PwdHash in all applications that render HTML. To fully implement PwdHash the plug-in would have to be more closely integrated with the HTML engine.

- **Spyware.** As mentioned earlier, PwdHash is designed to defend against scripts on phishing sites. It does not protect user passwords from spyware and keyloggers. Clearly, if an attacker has sufficient access to the user’s machine to install a keylogger, the attacker could just as easily disable or modify PwdHash. Along the same lines, a recent phishing attack works 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). PwdHash would not defend against this. However, if hackers have sufficient access to change the hosts file, they could just an 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.

4 Related Work

Password hashing with a salt is an old idea. However, it 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:

\[
\text{digest} = \text{Hash}(pwd, \text{realm}, \text{nonce}, \text{username}, \ldots)
\]

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 site A. The user’s response provides the phisher with a valid password digest for A. Password hashing implemented in Kerberos 5 has a similar vulnerability.

The first proposals we are aware of that properly use password hashing are the Lucent Personal Web Assistant (LPWA) [GGMM97] 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, most 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 plug-in so that we have access to 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 directly into a browser password fields. 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 [PMa] plugin for Mozilla Firefox provides a toolbar where users can obtain hashed passwords that must be manually typed or pasted into the password fields on the page. This approach combines the functionality of the PwdHash roaming page with the convenience of a browser toolbar. However, the process of typing passwords into the toolbar and then re-entering them into the page is a major departure from the usual user experience. Also, because Password Maker hashes using the current page domain rather than the form action domain, it would provide different passwords for different sites that used the same domain to process the password.

We emphasize that PwdHash does not preclude other anti-phishing solutions. For example, SpoofGuard [CLTM04] is a browser plug-in that alerts the user when the browser navigates to a suspected phishing site. These two techniques complement one another nicely. Similarly, the Passmark [Pas] web personalization method for fighting phishing could co-exist with PwdHash.

A closely related project [HWF05] at Princeton studies how to secure password hashing from dictionary attacks by using ultra-slow hash functions. Our focus is very different. We are 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 provide convenient password management [PSa]. These applications store the user’s web passwords on disk, encrypted under some master password. When the user tries to login 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. These applications have two drawbacks compared to PwdHash:

- **Roaming.** The user can only use the web on the machine that stores his password.
- **Crash recovery.** In case of a system crash the user could lose all her passwords.

These problems do not exist when using PwdHash. On the plus side, these systems provide stronger protection against dictionary attacks.

5 Conclusions

We presented a browser plug-in, PwdHash, designed to improve password authentication on the web with minimal change to the user experience. Deployment requires no work on the server side.

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. We are currently running user experiments and will report the results in the full version of the paper.

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

Acknowledgments

We thank Aaron Emigh, Burt Kaliski, Cem Paya, Eric Rescorla, and Jim Roskin for helpful discussions about password phishing and this work.

References


