Stopping a Phishing Attack, Even when the Victims Ignore Warnings

Dinei Florêncio and Cormac Herley
Microsoft Research, One Microsoft Way, Redmond, WA

ABSTRACT

Several factors make phishing a very challenging security problem. First, the victim unknowingly assists the attacker, by typing her credentials into a spoofed web site. Second, it is hard to identify web sites as suspicious using a fixed algorithm: phishers adapt quickly, and it is difficult to anticipate the ingenuity of all future attackers with a fixed set of rules. Third, users tend to ignore popups or security warnings: a good detection system doesn't help if users "drive past" the alerts.

The scheme we propose overcomes these difficulties. We assume that victims will type their passwords at insecure sites, we assume that phishers will adapt, and we assume that many or most victims will ignore all the warnings we give. And yet, we save substantially all users. The scheme is very simple, and consists of a client browser plug-in and a server component. The client detects when passwords are re-used at unfamiliar sites, and reports this fact to the server. Only when several clients report suspicious re-use events against a target/phisher pair of sites does the phisher get added to a Unique Password Required list.

Our scheme does not assume that users have different passwords for different sites: we expect that they recycle a small number of passwords for all their accounts. Our scheme has no blacklist: no user is ever blocked from navigating to a site. We do have a Unique Password Required list, but the consequences of getting on this list are mild. Even if perfectly innocent sites end up on this list, nobody is prevented from logging into a pre-existing account. The harshest consequence of being mistakenly suspected is that those setting up new accounts will be forced to choose unique passwords. Finally, even users who typed their password at a phishing site *before* it was suspected can be saved: the credentials of all compromised accounts are sent to the site under attack.

1. INTRODUCTION

The problem of phishing has been well documented in the popular press. A phisher who wishes to attack, say BigBank.com, spams many users with an email purporting to come from Bigbank. The email says that there is a problem with their BigBank account, and directs them to a website to login and fix the problem. The email and the phishing website look as though they belong to BigBank, but in fact have no affiliation. Users who "login" are parting with their BigBank identity, which can be harvested by the phisher. See [17] for details of recent attacks and statistics on the enormous growth in the number of attacks since the phenomenon first emerged.

The problem differs from many other security problems in that we wish to protect users from themselves. The difficulty of finding a solution is compounded by a number of issues. Both false positives (where we falsely suspect a phishing attack) and false negatives (where we fail to detect one) are very expensive. False positives erode trust in the system and cause inconvenience and possible loss to websites that are erroneously classified as phishing. False negatives allow a phisher to grab a user's credentials in spite of our efforts.

A further difficulty is that of warning the user. Halting the browser connection (*i.e.* refusing to connect to the site) is not acceptable unless it is absolutely certain that the site is phishing. For example, suppose an innocent site is wrongly flagged as phishing; if the browser halts the connection, the innocent site has effectively been placed in an internet "black hole," without notice or appeal. Clearly, this can only be done when no doubt remains that the site in question is phishing. In the absence of certainty what should we do? Suppose, to be concrete, that we determine with high probability that a user has navigated to a phishing site. A pop-up warning would seem like the obvious remedy. However, the overuse of pop-ups has eroded their usefulness. Users have grown used to pop-ups that alert them to service packs, upgrades, mortgage refinances, offers of free software, enticements to visit questionable web sites, and claims that they have won prizes. This indiscriminate use has eroded their ability to draw attention to

very real security problems. Further, recall that the user we wish to warn has just responded to an email that (to them) appears to come from a trusted institution. A pop-up that purports to come from another trusted institution (*e.g.* the browser manufacturer or plug-in author) faces a likely credibility problem with the user. Thus any method that detects phishing sites must either be sure enough to block the connection or be prepared to have any warnings it delivers ignored. A recent study [20] confirms that even trained users fail to notice or ignore warnings delivered by phishing prevention plug-ins.

The scheme we propose circumvents these problems. While we present a pop-up alert when we suspect phishing we do not rely on our warnings being heeded to stop the attack. Even if every user "drives by" or ignores our alerts we can still stop the attack.

Further, even users who have typed their credentials into a phishing site can still be saved, since we send to the attacked institution the hashes of the userids of the compromised accounts. Finally, our scheme does not have a blacklist: *nobody will ever be blocked from accessing an existing account.* We do have a Unique Password Required list. This is not merely a name change: the only consequence of being on this list is that users who set up a new account at a site on the list much choose a password distinct from those already used.

1.1 The Basic Idea

Our architecture involves a plug-in running in the user's browser that detects when credentials from a *protected list* are entered in the browser, and a server that receives an alert when this has occurred. For the moment consider the credentials to be (*uid*, *pwd*, *dom*), where *uid*, *pwd* and *dom* are the userid, password and domain respectively. When the user enters *uid*, *pwd* at any site dom' where $dom' \neq dom$ and dom' is not on the *whitelist*, we suspect a possible phishing attack. At this point the client browser reports this suspicion to the server by sending (`hash(`*uid*`)`, *dom*, dom'). It is perfectly possible that the user is knowingly using the same userid and password at two different domains (and thus the suspicion that the user is being phished is false). However, the server is in a position to aggregate the suspicion reports from many users. A single user reporting that the credentials for *dom* have been entered into dom' might or might not be alarming, but several users doing so will indicate to the server that an attack is in progress. At this point the server can add dom' to a global *Unique Password Required List*, and contact the domain under attack, *dom*, and present it with a list of `hash(`*uid*`)`. Now *dom* can halt transactions on the compromised accounts. And from this point on new victims who type their passwords at listed sites will be prevented from submitting. Even phishing sites that relay passwords a key at a time will be stopped: they receive all but one character of a password only at the same time that *dom* is told to block the account.

The above is actually a considerable simplification; there are many details that need to be handled to defeat possible modified phishing attacks. But this is the spirit of our approach. Observe that the method does not depend on stopping every individual user from entering protected credentials into a phishing site: even if we deliver warnings we expect that many or most users will ignore them. The power of the approach is rather that the server seeing the information aggregated from several users is in a position to make a definitive determination and send the list of compromised accounts to the site being attacked.

In the next section we examine related and previous work. In Sections 3 and 4 we cover the scheme we are proposing in detail. In Section 5 we analyze its performance and effectiveness. Section 6 deals both with current phishing attacks, and the types of attacks that might evolve and how our system handles them. We also address the questions of whether the system can be used to mount DoS attacks.

2. RELATED WORK

In its relatively short history the problem of phishing has attracted a lot of attention. Broadly speaking, solutions divide into those that attempt to filter or verify email, password management systems, and browser solutions that attempt to filter or verify websites.

2.1 Email and Spam Solutions

The email that induces a user to enter her credentials at a phishing site is a particular kind of spam. Machine Learning tools for filtering have achieved a lot of success against spam in recent years[19], and many phishing emails get caught as spam. However, the task is complicated by the fact that the phishing email purports to come from a trusted institution, and purports to contain very important information; so it is unlikely that spam filtering tools alone will solve the problem

Since spoofing the email origin is an important part of the phishing attack a number of approaches seek to verify either the email path or the sender. Today it is trivially easy for a phisher to make an email appear as though it comes from `accounts@bigbank.com` and the goal of verification systems is to make that difficult or impossible. For

example, Yahoo's Domainkeys proposal [8] proves the path of an email by having the originating mail server sign the message, and having that server itself certified by a DNS server.

Adida *et al.* [5, 6] also propose a trust architecture that allows detection of spoofed emails. However, it is a lightweight architecture and doesn't require a full public-key infrastructure (which is the main draw back of systems such as DomainKeys). The scheme is essentially a novel key distribution system coupled with an identity based signature scheme. It has several advantages in restoring a certain amount of verifiability to email; it would not however help with pharming or DNS hacking where a user is directed to a spoof site by some means other than email.

2.2 Password Management Systems

An early password management system was proposed by Gaber *et al.* [11]. The system used a master password when a browser session was initiated to access a web proxy, and unique domain-specific passwords were used for other web sites. These were created by hashing whatever password the user typed using the target domain name as salt. While the work of [11] pre-dates the earliest phishing attacks, the idea of domain specific passwords is a very powerful tool in protecting users. The original use was to ensure that password theft from a low security site would not yield useful information to an attacker, but it also combats phishing. Microsoft's Passport [1] allows users to sign in to the Passport site, which remembers their credentials and authenticates them at other sites that participate in the Passport program. This helps only with accounts at institutions that participate however, and thus will protect only some of any given user's accounts.

Ross *et al.* [18] propose a solution that, like [11], uses domain-specific passwords for web sites. A browser plug-in hashes the password salted with the domain name of the requesting site. Thus a phisher who lures a user into typing her BigBank password into the PhishBank site will get a hash of the password salted with the PhishBank domain. This, of course, cannot be used to login to BigBank, unless the phisher first inverts the hash. Their system has no need to store any passwords. To prevent browser scripts from confusing the plug-in that a password is being typed, they use a key tracker and perform a mapping on the keys which is undone only when the data is POSTed.

Halderman *et al.* [12] also propose a system to manage a user's passwords. Passwords both for web sites and other applications on the user's computer are protected. In contrast to [18] the user's passwords are stored, in hashed form on the local machine. To avoid the risk of a hacker mounting a brute force attack on the passwords a slow hash [15] is employed.

2.3 Browser Plug-ins

A number of browser plug-in approaches attempt to identify suspected phishing pages and alert the user. Chou *et al.* [7] present a plug-in that identifies many of the known tricks that phishers use to make a page resemble that of a legitimate site. For example numeric IP addresses or web-pages that have many outbound links (*e.g.* a PhishBank site having many links to the BigBank site) are techniques that phishers have used frequently. In addition they perform a check on outgoing passwords, to see if a previously used password is going to a suspect site. Earthlink's Scamblocker [2] toolbar maintains a blacklist and alerts users when they visit known phishing sites; however this requires an accurate and dynamic blacklist. Spoofstick [3] attempts to alert users when the sites they visit might appear to belong to trusted domains, but do not. Trustbar [13] by Herzberg and Gbara is a plug-in for FireFox that reserves real estate on the browser to authenticate both the site visited and the certificate authority.

Dhamija and Tygar [9] propose a method that enables a web server to authenticate itself, in a way that is easy for users to verify and hard for attackers to spoof. The scheme requires reserved real estate on the browser dedicated to userid and password entry. In addition each user has a unique image which is independently calculated both on the client and the server, allowing mutual authentication. A commercial scheme based on what appear to be similar ideas is deployed by Passmark Security [4]. The main disadvantage of these approaches is that sites that are potentially phishing targets must alter their site design; in addition users must be educated to change their behavior and be alert for any mismatch between the two images.

A very interesting recent study by Wu [20] confirms that users tend to ignore or fail to notice warnings. The study attempted to measure whether users notice the warnings provided by toolbars *even when the toolbar correctly detected the attack*. A large percentage of the participants did not change their behavior based on the warning.

2.4 Relation to our Method

While our client piece is implemented as a browser plug-in, we make no attempt to identify sites as suspicious based on their content or URLs. Using rules learned from recent phishing attacks would seem to be only a partial and temporary solution. As acknowledged in [7], once an attacker has access to the set of rules, it may take only a few

hours before he finds a way around it. It is unlikely that users can be persuaded to instal new versions of the plug-in each time a new attack strategy is discovered. Hence we begin with the belief that it is very difficult to defeat an active attacker with a static set of rules. In fact we believe there may be no reliable way to distinguish phishing sites from other login pages based on examining the URL and HTML. What distinguishes a phishing site from other sites is not numeric IP addresses, or outbound links, or low pagerank, or lack of traffic or reputation information. Some or all of these characteristics are shared by many sites that are not engaged in phishing. What distinguishes a phishing site from most other sites is that the phisher requires victims to type a password. And not just any password: it must be a password that the victim has previously used at another site. Thus, we focus our attention on the one thing that a phisher requires: he must get the victim to type a password at the phishing site. So we regard the first instance of a password being re-used at an unfamiliar site as an interesting event. In that respect, we start with similar ideas to the outgoing password check described in [7]. We also populate a list of protected credentials by examining the data POSTed when the browser submits data. However in contrast to [7] we do not rely on the same approach to detect passwords typed at suspicious sites: we track the user's key entries and constantly check against known passwords.

We solve the problem associated with storing the information, in ways that related to the slow hash techniques presented in [12] and [15]. We avoid Javascript attacks using keyboard tracking, similar to the technique used in [18]. In contrast to [7, 18, 12], we do not attempt to detect an attack at each client, but rather rely on aggregating information at the server. Thus a phishing site using the online mock field password field that Ross *et al.* explain would be stopped by our scheme. It bears mentioning that each of the approaches [18, 7, 12] attempts to protect users individually, while our approach aims at detecting the attack globally.

Each of [18, 7, 3, 2, 13] relies upon a pop-up, or a traffic light type signal to alert the user to problematic sites. All of these schemes depend (in order to stop a phishing attack) on users changing their behavior in response to a warning. While we do use a pop-up in certain cases we never rely on this; we assume that many or most users will ignore any warnings. Thus we believe our scheme is unique in that it requires changes neither in the sites that are possible phishing targets, nor in user behavior. Our method for stopping the attack relies on the accumulation of information at the server. This is a strength, in that information from many users is aggregated. But it also a weakness, in that our approach requires that the client plug-in is used by a great many users in order to be truly effective.

A further point of clear distinction with previous work is that unlike [7, 3, 2, 13, 9, 4] our plugin does not consume real estate on the browser; we have no need to further clutter the user's experience.

3. OUR SCHEME

Our goal is to halt an attack in which a Phisher lures many users to a website and asks them for userid and password information. The architecture of our scheme consists of a client piece and a server piece. The client piece has the following responsibilities:

1. identify and add important credentials *uid*, *pwd* to the protected list;
2. detect when a user has typed protected credentials into a non-whitelisted site;
3. warn the user and report instances of 2. to the server.
4. halt the connection if *dom* is on the UniquePwdRqd list.

The server has the responsibility of aggregating this information across users and determining when an attack is in progress. When it detects an attack it adds the phishing domain to a *UniquePwdRqd list* and sends the hashes of the compromised userids accounts to the target domain.

We assume that a full-featured browser that supports scripting and plug-ins is used on the client (the defender's task is a great deal simpler if the browser does not support scripting languages). We have implemented our client as a Plug-in for Microsoft's Internet Explorer but it could easily be adapted for other browsers such as Mozilla's Firefox. In this section we outline the main components of our algorithm, while Section 4 provides implementation details.

3.1 The Client Piece

We will explore the client's tasks in sequence. First, to produce a list of protected information; second, to detect that that information has been entered into another site; and, third, to report this to the server.

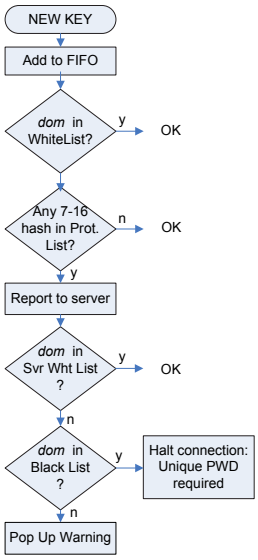

Figure 1: Keyboard analysis thread. Every key pressed gets added to a 16 character FIFO. Only if the hash of any of the last 7-16 characters matches the hash of a password and the domain is neither on the Cleared list nor Protected list is the server contacted. If the domain is on the server's UniquePwdRqd list the user will be instructed to choose a different password.

3.1.1 *Identifying credentials to be protected*

The password, *pwd*, and userid, *uid*, are easy to identify on any page that uses HTML forms (Section 4.1 for details), and the browser of course knows the domain, *dom*, to which it just connected. Unless *dom* is on the UniquePwdRqd list we add $[dom, uid, pwd]$ to the protected list. Since it would not be safe to store the credentials in the clear, what we actually store in the protected list is:

$$P_0 = [dom, H1, H2],$$

where *dom* is the domain, $H1$ is a hash of *pwd*, and $H2$ is a hash of *uid*. Section B will give details on computing H1 and H2. For now, it suffices to say that *pwd* is restricted to be between 7 and 16 characters long, and that we use salt that is specific to the client and to each table entry.

Observe that we add P_0 to the protected list without knowing whether the login is successful or not. We actually use the `BeforeNavigate2()` event handler provided by Internet Explorer to do all of the above processing, *before* the HTTP POST data gets sent. As described, this would mean that if a user mis-types her password, a new entry (with the wrong password) would be generated in the protected list. We arbitrarily restrict the protected list to 256 entries; this should be more than enough to store all of the important password sites for any user, even if the list ends up containing many spurious entries. We employ a Least Recently Used strategy for maintaining the list: thus we shift all entries $P_{k+1} = P_k$ for $k = 0, \cdots, 254$ before adding P_0 as above. We'll explore in Section 4.2 how to avoid polluting the protected list with spurious entries.

3.1.2 *Detecting when protected credentials are typed*

The near universal use of HTML forms makes populating the protected list relatively simple. It would be convenient if we could depend on phishers to use HTML forms; then we might just check the value of any password field submitted by a browser and see if it's on the protected list. It must be expected however that phishers will employ any means possible to conceal their intent from our defences. Using Javascript, for example, a phisher can present a page to the user that looks identical to the BigBank login page, but is code obfuscated in such a way that our plug-in cannot determine what data is being posted. An excellent account of several Javascript obfuscating techniques is given in [18].

To handle any and all tricks that phishers might employ we access the keystrokes before they reach the browser scripts. How this is implemented is obviously platform dependent. We use native calls in user32.dll on Microsoft Windows XP that allow our plug-in to get keystroke events directly. Similar calls are available under Linux and the major variations of Unix; the only real requirement is that we can be sure that no scripts running in the browser

can confuse the plug-in as to what was typed, or the order in which it was typed.

Figure 1 illustrates the procedure. For each key typed, we add the key to a FIFO buffer 16 characters long. Next we check to see which domain *dom* has control of the browser. If *dom* is in the whitelist, we do nothing (checking when credentials are to be added to the protected list is done in the separate thread detailed in Section 3.1.1). If *dom* is not in the whitelist, at each typed key we compute the hashes of possible passwords ending with the last typed character, and check against the appropriate hashes in the protected list. More specifically, since passwords can be between 7 and 16 characters, we need compute hashes of 10 strings. Since each entry in the protected list has a specific salt, we need to compute a total of $10 \times 256 = 2560$ hashes, and compare with the appropriate entries in the protected list. When one of the FIFO hashes matches a protected list H1 value, it means that the just-typed string matches a password on the protected list. Since we already determined that we are connected to a non-whitelisted domain this event is worth reporting to the server.

3.1.3 Warning the user and informing the server of a possible phishing attack

Whenever a hit is generated, we inform the server and, depending on the server response, warn the user. First we report to the server that a password from dom_1, on the protected list, was typed at dom_R, which is not on the whitelist, and request UniquePwdRqd list information on dom_R. What the client reports is:

$$C_{report} = [[(dom_1, H2_1), (dom_2, H2_2), \cdots], dom_R, h(IP)],$$

where $[(dom_1, H2_1), (dom_2, H2_2), \cdots]$ is the vector of domains with the conflicting password and their respective *uid* hashes, and $h(IP)$ is the hash of the IP address of the reporting computer. If dom_R is in the (more extensive) whitelist on the server, we proceed without issuing any warnings to the user. However, if dom_R is on the UniquePwdRqd list, (*i.e.* has been determined to be phishing as in Section 3.2 below), the client will halt the connection, and inform the user that they must choose a unique password if they wish to login to this site. Otherwise, if dom_R is on neither the whitelist nor the UniquePwdRqd list, the client plug-in presents the user with a warning. This warning can be very specific. For example: *"WARNING: You are submitting sensitive information to an unknown site. The server you are about to connect to is not affiliated with BigBank. We recommend you do not proceed".*

If the user ignores the warning and sends her credentials to dom_R, the client piece informs to the server that the credentials were indeed submitted to the site. The server will store the hash of the *uid* (i.e., H2) so that it has a list of compromised accounts if it subsequently emerges that dom_R is indeed phishing.

Recall that users commonly use the same password at several legitimate sites. This is not a problem to our scheme, and the user will not receive any warnings or pop-ups. First, most large password-using sites will already be included in the client's whitelist (and in this case we don't even check the passwords against the protected list). Second, even for smaller sites, not included in the client's whitelist, the client will consult with the server before reporting any suspicion to the user. The server has a much more extensive and dynamic white-list, and is likely to have even smaller legitimate sites in its list.

3.2 The Server Piece

The server has the role of aggregating information from many users. As detailed in Section 3.1.3, C_{report} is sent when protected credentials are typed into a non-whitelisted site; the server stores this record along with a timestamp. This is in itself of course, not proof that an attack is in progress. It means either that the user is knowingly using the same password at a protected and a non-whitelisted site, or that she has been fooled into entering the data by a phishing site. For example if dom_1 is "BigBank.com", and dom_R is "DanskinTriathlon.org" there might be nothing alarming (beyond the fact that the user is employing poor password habits by using same password at high and low security sites). However, if dom_R is "PhishBank.com" an attack might be in progress. Thus we must distinguish innocent cases of password re-use from phishing attacks. This task is much simplified by the fact that the server aggregates the data across many users. Having a single user typing her BigBank credentials into an unknown site may not be alarming, but having several users type BigBank credentials into *the same* unknown site within a short period of time is definitely alarming.

Recall that the server receives a report of every instance of a user typing protected credentials into a non-whitelisted site. In fact, it receives a vector with all legitimate domains that share that same password. The server collects this information for all non-whitelisted sites, and uses that to include the site on the black and white lists. For our initial evaluation, we assume a site is included in the black list if after receiving five reports for the same non-whitelisted site dom_R, there is at least one "phishable" site in the intersection of the five vectors. That is, each C_{report} on dom_R from given user contains the list of other domains associated with that password for that user. If one site *dom* is common to five users reporting on dom_R and *dom* is a financial institution we place dom_R on the UniquePwdRqd list. Recall that being on the UniquePwdRqd list does not mean that users cannot navigate to a site, or even login;

it merely means that they cannot type a password from their protected list into that site. It might appear that such a low threshold would make it easy to misuse our system to mount a Denial of Service attack; we explain that this is not so in Section 6.3.

The simple UniquePwdRqd list rule given suffices for the phishing attacks reported to date. However, the logic on the server might have to evolve as attacks evolve. One of the strengths of our system is that by making use of such reliable information (*i.e.* password re-use), aggregated across many users, the server is in a position to identify and stop an attack. To succeed with a distributed attack (see Section 6.2.1 and [14]) the phisher would have to make the pattern of client reports statistically insignificant.

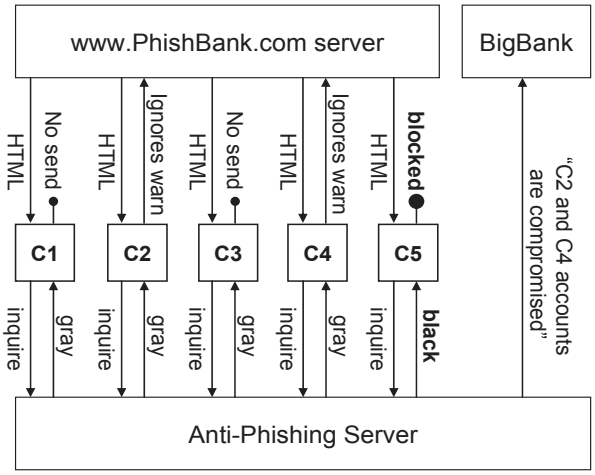

Figure 2: Users C1, C2, etc. are attracted to a phishing site, PhishBank.com. The first four consult with the server, and all receive a gray rating for the site (and a pop-up window). Say C1 and C3 decide not to submit the information, but C2 and C4 ignore the warning and proceed. By the time the fifth user (C5) consults with the server, the server is already in a position to declare the site as phishing, and blocks the connection. Furthermore, the server informs the attacked domain (BigBank.com) that accounts C2 and C4 may be compromised.

4. IMPLEMENTATION DETAILS

In the previous section we described the overall architecture of the proposed scheme. In the interests of flow, we ommitted a number of details, which we will now provide. Some of these are simply implementation details, some are targeted at increasing the robustness, some are targeted at reducing the risk associated with locally storing passwords hashes, and some are targeted at reducing the computational complexity of the search.

4.1 Finding the Credentials to be Protected on HTML form pages

We first point out that most websites use HTML forms to gather user input data. The "text" and "password" fields are almost universally used. For example the following snippet of HTML shows the relevant fields from a login page:

```
<inputtype="password" id=""name="login_pwd
"value=""class="homeInput">
```

This tells the browser that the variable login_pwd contains a password field data. Password fields get special treatment, so that they are not displayed, and the target site can specify whether they allow auto-complete or caching of their passwords. The HTTP Post data that is sent when the user submits this page looks as follows:

```
"login_email=clock10@hotmail.com & login_pwd
 =mickeymouse &submit.x=Log+In & form_charset
 =windows-1252\0"
```

the user input typed has been associated with the variable login_pwd telling the remote server that the password is "mickeymouse." Clearly this allows us direct access to the password itself.

Since there is no special HTML field for userids, finding this field involves a little more work. First observe that userids are text fields, between 4 and 25 characters and they are whitespace free. We regard any such field that occurs on the same page as a password field as a possible userid. If it contains more than one, we regard both as possible userids, and all the possible userid/password pairs that occur on the page as credentials to be protected. We note that having spurious userids or passwords in the protected list carries almost no cost. In practice, even complicated registration pages where a user fills out name address, userid and password very seldom generate even a single spurious entry in the protected list.

4.2 Maintaining the Protected List

We adopt a Least Recently Used strategy to drop entries from the list. This means that we form the substitution $P_{k+1} = P_k$ for $k = 0, \cdots i$ when element P_i is used, and P_i replaces P_0. As we have stated, we arbitrarily restrict the size of the protected list to 256 entries.

Recall that cost of placing spurious entries in the protected list are small. In Section 3.1.1, for example, we were unconcerned that password typos might generate new entries in the protected list. Equally using a friend's computer to check an account would generate an entry in the friend's protected list, and using an internet kiosk would leave a hash of the credentials on the kiosk machine. To cut down on the ease with which credentials get onto the protected list we establish a *waiting list*. The first time a new set of credentials appears we simply put (*dom*,H8) in this list, where H8 is the first byte of H2. The second time the same (*dom*,H8) is typed, we check a flag, indicating that combination has already been typed twice. Finally, the third time we move the full entry to the protected list. This avoids adding typos to the protected list and it helps alleviate the risk of leaving traces at a public internet kiosk.

4.3 Our Client Implementation

Our client is implemented as a Browser Helper Object (BHO) for Microsoft Internet Explorer. It could be easily adapted for any of the other major Browsers, though the specifics would change. Identifying the credentials to be protected (Section 3.1.1) is performed using the BeforeNavigate2() event handler. This allows us to trap the connection before data is POSTed and examine the fields. Explorer provides hooks that allow us to examine the HTML document, the target domain and the fields to be POSTed.

The code to detect when credentials are typed (Section 3.1.2) is a separate thread. Using native event handlers provided by Windows in user32.dll we have a method that is called every time a key press event occurs and the browser has focus. This thread maintains the FIFO and does all of the checking against the protected list. The two threads do not need to communicate; the FIFO does however need to be accessible to the BeforeNavigate2() method that adds credentials to the protected list. It must check that a POSTed password was actually typed to prevent against a possible flushing attack (see section 6.1.1). Finally, if the browser has a password-save feature, the password may never be typed at all. Auto-completed passwords are easily added to the protected list, but we must check that the POSTed data is exactly what the auto-complete provided. Otherwise a variation on our flushing attack is possible. The current version of our plug-in does not support this check.

Using an implementation of SHA-1 provided by Windows security classes we perform 3890 SHA-1 hashes in 10ms on a 3.2GHz Pentium 4 PC with 512 MBytes of RAM. This can be taken as the number of times N we can iterate the hash as explained in Section B. The $2N + 2560$ hashes performed at every key press event cause no noticeable delay.

5. ANALYSIS OF PERFORMANCE

We now analyze the main aspects concerning performance. We look at minimum password sizes, false positive and false negative events, and try to quantify processing requirements at the client, and traffic generated to the server.

5.1 Minimum Password Size

Storing the hash of the password in the client opens the possibility of a exhaustive search attack on the password. To minimize that risk, we protect only password spaces of size at least $1e13$. For example, the password should satisfy at least one of the following criteria:

- *pwd* includes at least one number, one special character, one lower case, and one upper case letter, and is at least 7 characters long;
- *pwd* is at least 8 characters long, and include both numbers and letters;
- *pwd* is letters only, and is at least 9 characters long;

- *pwd* is numeric only and is at least 13 digits;

We also truncate passwords at 16 characters. Finally, we point out that shorter passwords can also be protected, by concatenating the short password with the userid. This requires a number of other modifications in the scheme, which are described in [10].

5.2 False Positives and Server Traffic

The scheme sends a message to the server only when a protected password is typed into a non-whitelisted site. There are three possible ways for this to happen. First, the user mistakenly typed the password for one account into another (e.g. absent mindedly entered her BigBank credentials on another login page). Second, the user is knowingly using the same credentials at a protected and a non-whitelisted site. Third, the user has been fooled into entering protected credentials into a phishing site. The first and second cases generate traffic to the server only if the domain receiving the credentials is not on the whitelist and is not on the users protected or waiting lists. Since the first login to a site will place it on the waiting list, this traffic only happen the first time the user logs in.

Thus, to be concrete, a user absent mindedly typing her BigBank credentials into the SmallBank login page (or any other white or protected list site) will generate no traffic. Using the same credentials as BigBank at, say, a community group site will generate traffic to the server only the first time the user logs in. The server will not conclude that the community group site is phishing BigBank, since a single alert is not indicative of an attack. If we assume an average user initiates a new login relationship at a non-whitelisted once a month, we should expect twelve messages to the server per user per year.

Since a site is not placed on the UniquePwdRqd list until several clients report the same site to the server, the false positive rate will depend on the heuristics running on the server. Even if we consider any traffic as false positive, the probability seems to be extremely low.

5.3 Roaming and Internet Cafes

Two problems arise when a user visits an internet kiosk (or a friend's computer). The first is that hashes of the user's account details can be left on the machine; the second is that the user's protected list is unavailable.

To solve the first we introduced the waiting list, presented in Section 4.2. The first two times an *uid* is typed at a site, we only store *dom* and a short (8 bit) hash of the *uid* in a waiting list. Note that 8-bits is not enough to compromise much information about the user. Only the third time the same *uid* is typed, we move it to the protected list, and store the full entry, which includes a full hash of the *pwd*. The waiting list has only 16 entries. Since the short hash is 8-bits long, the possibility of a 3-way collision is $(1/256)^2$. Ignoring the possibility of collision, the full hash would be stored only if the administrator did not disable the feature, you went to the same kiosk 3 times, used the same machine, and less than 16 other people used the machine to login on their accounts (on any site) since the first time you used the machine. This sequence of events seems very unlikely. Even if the full hash is stored in the machine, a hacker still has to break the hash, which may take years of computing time (see Section B.1).

In our current implementation a user at an internet kiosks will not be protected. By this we mean that typing her BigBank credentials at a phishing site will generate no report to the server. A main advantage of our scheme, however, is that by aggregating across many users the server can detect an attack quickly and the UniquePwdRqd list can be expected to respond very quickly. Note however, that a user on her home computer who types BigBank credentials into the PhishBank site *before* the server places it on the UniquePwdRqd list will generate a report to the server. When the server determines that PhishBank is in fact phishing, and places it on the UniquePwdRqd list it sends H2, the hash of the userid of the known victims to BigBank. Thus, the first few victims of a phishing attack can probably be saved since BigBank can halt all activity on their account. However, if any of those victims were unfortunate enough to respond to the attack from an internet kiosk there is no record. By storing each users protected list at a server (much as [11] did) we can overcome even this difficult; details are in [10].

BigBank is also likely to start action to block or redirect the site PhishBank, for protection of users not using our scheme. By the time BigBank is successful in blocking the site, everyone will be protected. Thus, the early detection enabled by our scheme will protect roaming users, as well as users not using our scheme.

5.4 Privacy Concerns

The list of protected credentials is stored on the client. We deal in Section B.1 with the risk of the credentials being hacked, but what about exposure to having the user's spouse, children, parents etc figure out that she has an account with xxx.com ? While we've assumed that the domain names are stored in the clear, it is probably wise to at least obfuscate the domains so they are not easily readable. So long as it is as hard to see the domain names as it is to check a browsers address history, there would appear to be no additional privacy exposure.

Regarding the data sent to the server, recall that we send only the hash of the userid, the IP address, and the two domains (original and suspicious). The server gets this information only when there is either a false positive (addressed in Section 5.2) or the user is phished. Note in particular that we never send any information regarding the password.

5.5 Results and Efficacy

As in any security task, efficacy is hard to figure out, as it will depend on the ingenuity and evolution of the attacks. In the course of our research, we have investigated a few dozen actual phishing attacks, and our scheme blocks 100% of them. Those included very elaborate attacks, some of which could not be detected even by security experts looking at them.

Nevertheless, examining past attacks is not a good measure of efficacy. Once deployed, we can expect attacks to evolve and exploit any flaws in the system. We have tried to foresee possible ways to circumvent the scheme, and have introduced hooks to block every single future attack we could imagine. There is, of course, no guarantee that we have thought of everything. Our method has been presented, and subjected to security review by over one hundred colleagues. Although we believe the system is at this point in a very mature stage, we are looking forward to expand the range of the scrutiny.

Finally, we point that there are a few clear ways of circumventing the system. An obvious one is to ask the victim to use a screen keyboard to enter the password. Another is to send e-mail asking the victim to call 1-800-PHISHME. Nevertheless, we think the further the paradigm comes from what the user is expecting, the less likely an attack is to succeed. We believe the proposed method has the potential to block all attacks where a user is asked to type his password.

So, in summary, our method would address essentially all past phishing attacks. And has already stand the scrutiny of over one hundred security-related experts.

6. ATTACKS

There are two main ways of succeeding with a phishing attack on the system. First, the phisher may try to prevent clients from reporting to the server. Second, he may try to prevent the server from detecting an attack from the reports it receives by hiding below any suspicion threshold the server may have. There are two other important attacks, unrelated to phishing that we must consider. First, a hacker may try to access the password by inspecting the hash stored in the client. Finally, a vandal may try to use the system to mount an denial of service attack at a legitimate site.

6.1 Preventing the client from reporting

We kept the logic of the client piece as simple as possible, so that it is hard to prevent it from reporting entry of protected credentials at non-whitelisted sites. This means that, once deployed, the client piece probably does not need to be updated as phishing attacks evolve. To make the client piece as durable as possible we have considered several attacks.

6.1.1 Flushing the protected list

A Phisher might try to circumvent the protection by removing some (or all) of the passwords from the protected list. For example, since the protected list has only 256 entries, a phishing site could submit (using HTML forms) 256 strings as passwords to a random site. As described in Section 3.1.1 this would effectively "flush" everything from the protected list because of the Least Recently Used maintenance rule. To avoid this attack, before accepting a new entry, from the HTML form data (as in Section 3.1.1), we match the password with the keyboard buffer, effectively requiring that the password have been actually typed at the site. It is unlikely that a Phisher can induce a victim to actually type hundreds of password-like strings.

6.1.2 Hosting on a whitelisted domain

A phisher might attempt to host on an existing, legitimate site. For example putting the phishing page up on an ISP member site, like members sites on AOL or MSN, or a small site like a community group association, or by employing a Cross-Site Scripting (CSS) attack. Each of these is handled by proper design of the client whitelist.

It is easy to handle ISP member sites by including the ISP, but excluding certain sub-domains from the whitelist. Small groups like community associations cannot be depended upon to prevent break-ins. Thus the client whitelist should contain only very large commercial domains. Recall that a site can be on a users protected list, without being on the whitelist. CSS attacks actually host the phishers page on a target domain. For this reason, only sites that

can be depended upon to maintain basic levels of security should be permitted to populate the client's whitelist.

6.2 Preventing the server from detecting an attack

As we stated in Section 3.2 the logic employed by the server to detect an attack from the client reports might need to evolve with time.

6.2.1 Distributed Attack

A possible approach for a phisher who seeks to evade detection is to distribute the attack. Rather than phish BigBank by directing victims to PhishBank the phisher may use many domains, or numeric IP addresses. Thus when clients report sending their BigBank credentials, it fails to trigger an alert at the server. For this reason, we believe the server logic needs to adapt as phishers adapt. For example, while the destination for several BigBank credentials may be distributed, a large increase in the number of reports for a given whitelisted domain is in itself worthy of suspicion.

6.2.2 Redirection Attack

Similar to the distributed attack is a Redirection attack where a phisher directs all victims to a single URL, but each victim is redirected to a different (possibly unique) address to be phished. For example the phishing URL might originally redirect to IP_1, but as soon as the first victim is caught it redirects to IP_2 and so on. This might appear to confuse our system by distributing the client reports one at a time among many addresses. To circumvent this we include in the client report any URLs visited in the last minute. By intersection, the redirecting (phishing) URL can be detected.

6.2.3 Looking at the Phishing html?

Including a site on the UniquePwdRqd list should be taken only when no doubt remains about a site being a phishing site. For legal and liability reasons it may even be necessary to have a person actually look at a site before making a final determination on UniquePwdRqd list inclusion. If that's the case, rather than visit the suspect site it is better to receive directly from the reporting client the HTML content that it received. The reason for this is that a phisher might easily present each phishing victim with a unique url, so that the first visitor to the url (the victim) would see the page requesting userid and password, while the second and subsequent visitors (*e.g.* someone at the server checking the site) would see a completely innocent page. This is easily accommodated by adding the HTML as a record to C_{report}.

6.3 Denial of Service on a Site

We now explore the possibility that a vandal tries to fool our system that a legitimate site is phishing. Specifically, suppose a disgruntled MyCornerStore customer types BigBank credentials at the MyCornerStore login site. Recall, the vandal need not use real BigBank credentials, since populating the protected list was very easy. What are the consequences if the server wrongly concludes that MyCornerStore is phishing and places it on the UniquePwdRqd list?

First, note that we track the IP of the reporting client; a single client reporting the site repeatedly will have no effect. Second, no site that is on the client whitelist (of large institutions) or the server's much larger whitelist can ever be placed on the UniquePwdRqd list. The most powerful defense, however, is that if MyCornerStore does get on the UniquePwdRqd list, *it does not mean that nobody can navigate to that site.* Rather, it means that those who attempt to login to MyCornerStore for the first time (*i.e.* it is not on their protected or waiting list) and use a password already used at at a site on their protected list would have their login blocked. Users who have existing MyCornerStore credentials in their protected list would be unaffected (since the client will not even consult the server). Users logging in for the first time who choose passwords not on their protected list would be unaffected. Even those users who are initially blocked, can login by simply choosing a unique password for MyCornerStore.

7. CONCLUSION

We proposed a scheme which have the potential of neutralizing most phishing attacks. The client piece stores hashes of important personal information (e.g., passwords), and reports to the server whenever such information is typed at non-whitelisted sites. The server aggregates those reports to produce a very responsive black list. Although specific results will depend on the extension of the deployment, we expect to detect an attack as soon as five potential victims are lured to the site and type their passwords. Even if the victim ignores the warning and proceed to submit

the sensitive information to the phisher, we may be able to rescue him, by informing the financial institution about the compromised accounts.

Acknowledgements: the authors wish to acknowledge numerous discussions with Robert Rounthwaite, Geoff Hulten, Manav Mishra and members of the MSN TCS team. Cem Paya provided several suggestions and attacks that have materially improved the scheme.

8. REFERENCES

[1] `http://www.passport.com`.
[2] `http://www.scamblocker.com`.
[3] `http://www.spoofstick.com`.
[4] `http://www.passmarksecurity.com`.
[5] B. Adida, S. Hohenberger, and R. L. Rivest. Fighting phishing attacks: A lightweight trust architecture for detecting spoofed emails. 2005.
[6] B. Adida, S. Hohenberger, and R. L. Rivest. Separable identity-based ring signatures: Theoretical foundations for fighting phishing attacks. 2005.
[7] N. Chou, R. Ledesma, Y. Teraguchi, D. Boneh, and J. Mitchell. Client-side defense against web-based identity theft. *Proc. NDSS*, 2004.
[8] M. Delany. Domain-based email authentication using public-keys advertised in the dns. 2004. `http://www.ietf.org/internet-drafts/draft-delany-domainkeys-base-01.txt`.
[9] R. Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. *Symp. on Usable Privacy and Security*, 2005.
[10] Dinei Florêncio and Cormac Herley. Stopping phishing attacks. *MSR Tech. Report.*
[11] E. Gaber, P. Gibbons, Y. Matyas, and A. Mayer. How to make personalized web browsing simple, secure and anonymous. *Proc. Finan. Crypto '97.*
[12] J. A. Halderman, B. Waters, and E. Felten. A convenient method for securely managing passwords. *Proceedings of the 14th International World Wide Web Conference (WWW 2005).*
[13] A. Herzberg and A. Gbara. Trustbar: Protecting (even naive) web users from spoofing and phishing attacks. 2004. `http://eprint.iacr.org/2004/155.pdf`.
[14] M. Jakobssen and A. Young. Distributed phishing attacks. 2005.
[15] J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Secure applications of low-entropy keys. *Lecture Notes in Computer Science*, 1396:121-134, 1998.
[16] P. Oechslin. Making a faster crytanalytical time-memory trade-off. *Advances in Cryptology - CRYPTO 2003*, 2003.
[17] Anti-Phishing Working Group. http://www.antiphishing.org.
[18] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. A browser plug-in solution to the unique password problem. *To appear in Proceedings of the 14th Usenix Security Symposium, 2005.*
[19] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A bayesian approach to filtering junk email. *Learning for Text Categorization*, 1998.
[20] M. Wu. Users are not dependable: How to make security indicators to better protect them. *Trustworthy Interfaces for Passwords and Personal Information*, 2005.

APPENDIX

A. OTHER ISSUES AND COMMON QUESTIONS

In describing this work certain questions or misconceptions occur frequently. Here we try to address some of the most common.

A.1 Nobody is blocked from logging into an existing account

A natural question is to wonder under what circumstances a legitimate user can get locked out of a legitimate account. First observe that this can never happen on a machine that a user regularly uses to login to a particular site: on this machine the user's Protected List will ensure that no report to the server is ever generated. Thus, even if a site is placed on the UniquePwdRqd list, users who have this site on their protected list will be unaffected so long as they have access to their Protected List. Equally, if they access their MomPop.com account from another machine (such as an internet cafe) where their Protected List is not present no password re-use event will be recorded, thus

no report to the server, and thus they will not be blocked. Thus users with existing accounts are not locked out, whether they have access to their Protected List or not.

A.2 Password changes

If a user elects to change her password she is generally directed to a page with three password fields. She enters the old password once, and the new one twice for confirmation. When this happens the new password gets entered at the top of the Protected List. The old password is also refreshed, so is also at the top of the list. However, assuming it does not get used again, is slowly descends the list until, following the LRU strategy, it drops offs.

A.3 Auto-complete function on browser

Most modern browsers offer to auto-complete certain fields (including passwords) for users. General HTML text fields (such as userid) will often be auto-completed whenever a field of the same name appears on a web page. Thus, for example, suppose a HTML text field on the BigBank site has `name` set to `‘‘userid’’`. Then for any other site the user visits that has a HTML text field with the same name the user's BigBank userid will be offered as a possible auto-completion. Note that the same is not true of password fields. The browser will offer to auto-complete the password only if the user has selected the auto-complete passwords option, and it will only auto-complete on the same site as previously used.

To consider a concrete example: suppose a phisher mirrors the BigBank site in every respect, including the name of the HTML form fields. When a victim visits the phishing site the browser will auto-complete the userid, since this field matches that on the legitimate site. It will not auto-complete the password; a password has to be entered before at that site before password auto-completion occurs. Since a victim is always visiting the phishing site for the first time there is no possibility that auto-complete will give the phisher the password without having the user type it.

A.4 Visual Keyboards and other input devices

Entering the password by any means other than typing will not be detected by our system. For example, a phisher might direct victims to a page with an image of a keyboard and ask them to enter their password by pointing with the mouse to the appropriate keys. Equally, a phishing email might merely direct victims to call a particular phone number and give their credentials to the operator. Social engineering attacks can be very sophisticated. Our expectation is that if phishers are forced to use password acquisition schemes that deviate substantially from the paradigm that users are familiar with their yield will fall very substantially.

A.5 Keylogging

It is important not to confuse the key capturing mechanism we employ with “key-logging” software. First, essentially every application that accepts user input must have a key event handling system and buffer just as our plugin does, and many of them use the same user32.dll callback that we do. This includes the majority of consumer PC applications, and utilities such as Notepad, WinEdt, Firefox. Applications such as WinEdt, for example, would not be able to offer real-time spell checking capability, if they not employ a key event callback just as we do. Thus the key event callback mechanism is not merely common, it is almost ubiquitous in modern software applications.

A.6 Sloppy password re-use habits are just fine

A frequently point of misunderstanding is to assume that sloppy password habits by users will cause many false positives. This is not the case. Even many clients reporting password re-use among the pair dom, dom' will not cause dom' to be added to the UniquePwdRqd list unless that pattern appears anomalous. Each domain can be expected to have password re-use reports that occur because of users' habits. It is a deviation from the expected report pattern that will cause dom' to be placed on the UniquePwdRqd list.

A.7 Password re-use information is not disclosed

As we mentioned before, providing information about password re-use across different sites would be a security risk. We solved this problem by inserting a entry-specific salt. With this modification, verifying password re-use would require pre-imaging a hash, which is too computational for a relatively small value information.

B. THE HASHING ALGORITHM

Each new entry in the protected list contains three fields. The domain, *dom*, is stored in the clear, or is obfuscated for privacy. The other two fields, H1 and H2, are the hashes of *pwd* and *uid*, respectively. We simply store a SHA-1

hash of the userid, using *dom* as salt. Passwords, however, must be handled very carefully. To minimize the risk of a dictionary attack, we use a slow hash, and require the password space to be at least $1e13$ (see Section 5.1 for details). Dictionary attacks are considered in B.1. Even with a large password space, a hacker could use a pre-computed table to find the password, e.g., mounting an attack similar to [16]. To avoid this, we add a *client-specific* salt.

Finally, storing passwords from different sites in a table might give a hacker access to a user's password usage pattern. For example, a hacker might see that the same password is used at a bank and low-security site. He might get the low-security site *pwd* (which may even be stored in the cache), and use it at the bank. To avoid that, we insert an *entry-specific* salt to the hash of each password. In other words, each of the 256 entries in the table have a specific salt. This is in addition to the client specific salt.

We will now describe in detail how exactly we compute the H1 hash. We use a hashing algorithm based on the SHA-1, but adapted for our application. First, recall that passwords used in H1 are limited to between 7 and 16 characters. Second, note that, at each key entry we must compute 2560 different hashes (we have 10 possible password lengths and each entry in the protected list has an entry-specific salt as explained in Section 3.1.2). However, these 2560 are not hashes of independent strings. We will use that fact to design a hash function where these 2560 hashes can be computed in approximately the same time it will take an attacker to compute a single hash.

First we describe how H1 is calculated at the time of entry to the protected list, and then we describe how the list is checked at each keystroke. When the plug-in is installed, we measure N, the number of SHA-1 hashes can be computed in 10 ms on the client CPU. To compute the hash of *pwd*, we start by concatenating the last seven characters of *pwd* with the *client-specific* salt[1]. We then recursively compute the SHA-1 of this string N times. The resulting hash is concatenated with the eighth character, and we compute the recursive hash $N/2$ times. We then concatenate the ninth character, and compute the recursive SHA-1 $N/4$ times. This continues until all characters in *pwd* are included. We then concatenate the last result with the *entry-specific* salt, and compute the SHA-1 one last time. This is the H1 entered in the protected list as described in Section 3.1.1. Note that, regardless of the length of *pwd*, this whole process requires fewer than $2N$ calls to SHA-1.

Now, when a key press event occurs, as in Section 3.1.2, we take the last 7 characters in the FIFO, concatenate with the *client-specific* salt, and recursively compute the SHA-1 of this string N times. Call this string $H1_7$. We then concatenate $H1_7$ one at a time with each of the 256 *entry-specific* salts of the protected list, compute SHA-1 of the string and compare with the protected list entry. Next we concatenate $H1_7$ with the 8-th last character in the FIFO, recursively compute the SHA-1 of this string $N/2$ times, and call this string $H1_8$. We then concatenate $H1_8$ one at a time with each of the 256 *entry-specific* salts of the protected list, compute SHA-1 of the string and compare with the protected list entry. This proceeds until we calculate $H1_{16}$ and all characters in the FIFO are involved. Observe that the total number of calls to SHA-1 is less than $2N + 256 \cdot 10$.

B.1 Dictionary attack on the password hash

As we have mentioned before, we use a number of measures to avoid a exhaustive search attach on the hash of the password stored in the client. The client specific salt should take care of pre-computed attacks, and the slow hash should make it very computational intensive to do a search. More specifically, given the way we compute the hash (see Section B) we can assume computing the hash of a single 7 characters string should take approximately 10ms. The first additional character can re-use some of the previous computation, and only adds 5ms per hash. Omitting some of the details, full search over an M character password space, where each character is drawn from an alphabet of size Z would require a total time of:

$$T = Z^M * (1/2)^{M-7} * 10ms \tag{1}$$

This implies that, for each of the four minimum password requirements in Section 5.1, exhaustive search time would be 3180 years, 447 years, 430 years, and 50 years, respectively. One could probably reduce this time by using faster computers, using a more efficient (dictionary) search, or distributing the task. Nevertheless, we expect that the processing cost would be bigger than the expected profit one could obtain from the illicit use of the password.

Note that the risk of an attacker observing information about password re-use across sites was solved by using entry-specific salt in calculating H1.

[1] note that this salt can be random, and do not need to be secret. It simply needs to vary from client to client, to avoid the pre-computed hash pre-imaging attack.