Keyjacking: Risks of the Current Client-side Infrastructure

John Marchesini, S.W. Smith, Meiyuan Zhao*
Department of Computer Science
Dartmouth College

Technical Report TR2003-443
{carlo,sws,zhaom}@cs.dartmouth.edu

February 10, 2003

Abstract

In theory, PKI can provide a flexible and strong way to authenticate users in distributed information systems. In practice, much is being invested in realizing this vision via client-side SSL and browser-based keystores. Exploring this vision, we demonstrate that browsers will use personal certificates to authenticate requests that the person neither knew of nor approved (and which password-based systems would have defeated), and we demonstrate the easy permeability of these keystores (including new attacks on medium and high-security IE/XP keys). We suggest some countermeasures, but also suggest that a fundamental rethinking of the trust, usage, and storage model might result in a more effective PKI.

1 Introduction

Because public-key cryptography can enable secure information exchange between parties that do not share secrets a priori, PKI has long promised the vision of enabling secure information services in large, distributed populations.

In the last decade, the Web has become the dominant paradigm for electronic access to information services. The *Secure Sockets Layer* is the dominant paradigm for securing Web interaction. For a long time, SSL with *server-side authentication*—where, during the handshake, the server presents a public-key certificate and demonstrates knowledge of the corresponding private key—was perhaps the most accessible use of PKI in the lives of ordinary users.

However, in the full vision of PKI, all users have key pairs—not just the server operators. Within the SSL specification, a server can request *client-side authentication*—where, during the handshake, the client also presents a public-key certificate and demonstrates knowledge of the corresponding private key. The server can then use this information for identification, authentication, and access control on the services it provides to this client.

An emerging client-side PKI exploits the natural synergy between these two scenarios. Because the Web is the way we do business and client-side SSL permits servers to authenticate clients:

- modern browsers[1] now include personal keystores, for a user's key pairs;
- enterprises (and other distributed populations) are arranging for users to obtain certified key pairs to live in these keystores;

*The authors have received support from the Mellon Foundation, the NSF, AT&T/Internet2, and the U.S. Department of Justice (contract 2000-DT-CX-K001). The views and conclusions do not necessarily reflect the sponsors.

[1] Admittedly, due to Microsoft's contention that the browser is part of the operating system, one might argue that the Internet Explorer (IE) keystore on Windows is really part of Windows.

- providers of Web services are starting to use client-side SSL as a better alternative than passwords or to authenticate users;
- and even non-Web applications may typically expect to find and use the key pair resident in the browser keystore.

In previous work, we have examined the effectiveness of server-side SSL [29] and of digital signatures on documents [13]. In this paper, we examine the question: *does this client-side PKI work*?

- When browser-based keystores are used in contemporary desktop environments, is it reasonable for the user at the client to assume that his private key is used only to authenticate services he was aware of, and intended?
- Is it reasonable for the user at the server to assume that, if a request is authenticated via client-side SSL, that that client was aware of and approved that request?

This Paper First, we lay out the background. Section 2 introduces how Web services work; Section 3 discusses (pre-PKI) user authentication; Section 4 discusses the push to use SSL client-side PKI. Then, we discuss our exploration. Section 5 frames the basic questions; Section 6 and Section 7 report the experiments. Finally, in Section 8 and Section 9, we consider the implications.

2 Web Services

Currently, the Web is the dominant paradigm for information services. Typically, the browser issues a request to a server and the server responds with material the browser renders.

Language of the Interaction From the initial perspective of a browser user (or the crafter of a home page), these "requests" correspond to explicit user actions, such as clicking a link or typing a URL; these "responses" consist of HTML files.

However, the language of the interaction is richer than this, and not necessarily well-defined. The HTML content a server provides can include references to other HTML content at other servers. Depending on the tastes of the server operator and the browser, the content can also include executable code; Java and Javascript are fairly universal. This richer content language provides many ways for the browser to issue requests that are more complex than a user might expect, and not necessarily correlated to user actions like "clicking on a link."

As part of a request, the browser will quietly provide parameters such as the browser platform and the `REFERER` (sic)—the URL of the page which contained the link that generated this request.

Issues such as caching at the browser site or an intermediate firewall can complicate this model further. [5]

In the current computing paradigm, we also see a continual bleeding between Web interaction and other applications. For example, in many desktop configurations, a server can send a file in an application format (such as PDF or Word), which the browser happily hands off to the appropriate application; non-Web content (such as PDF or Word) can contain Web links, and cause the application to happily issue Web requests.

Web Services Surfing through hypertext documents constituted the initial vision for the Web—and, for many users, its initial use. However, in current enterprise settings, the interaction is typically much richer: users (both of the browser and server) want to map non-electronic processes into the Web, by having client users fill out forms that engender personalized responses (e.g., a list of links matching a search term, or the user's current medical history) and perhaps have non-Web consequences (such as registering for classes or placing an Amazon order).

In the standard way of doing this, the server provides an HTML `form` element which the browser user fills out and returns to a *common gateway interface (CGI)* script (e.g., see Chapter 15 in [18]).

This `form` element can contain `input` tags that (when rendered by the browser) produce the familiar elements of a Web form: boxes to enter text; boxes (with a "browse") tag to enter file names for upload; radio buttons; checkboxes;

etc. For each of these tags, the server may specify a `name` (which names the parameter being collected from the user) and a default `value`. The server content associates this form with a `submit` action (typically triggered by the user pressing a button labeled "Submit"), which transforms the parameters and their values into a request to specific URL. (If the `submit` action specified the `GET` method, the parameters are pasted onto the end of the URL; if the `POST` method, the parameters are sent back in a second request part.)

However, this submit URL specifies an executable script, not a passive HTML file, in the "Web directory" at the server. When a server receives a request for such a script, it invokes the script; the script can interrogate request parameters, such as the form responses, interact with other software at the server side, and also dynamically craft content to return to the browser.

3 Authentication and Security

3.1 Authenticating the User

In enterprise settings, the server operator may wish to restrict content only to browser users that are authorized. In a situation where the browser user is requesting a service via a `form`, the server operator may wish to authenticate specific attributes about the user, such as identity and the fact that the user authorizes this request.

Client Address The Web paradigm provides several standard avenues to do this. Via `.htaccess`, the server may restrict requests to client machines with specific hostname or IP address properties.

Passwords With *basic authentication* (or the *digest authentication* variant), the server can require that the user present a userid and password, which the browser collects via a special user interface channel and returns to the server. The server requesting the authentication can provide some text that the browser will display in the password-prompt box. Alternatively, the server may also collect such authenticators as part of the `form` responses from the user.

With these various forms of password-based authentication, the server operator would be wise to take steps to ensure the passwords and other sensitive data are not exposed in transit, such as:

- by offering the entire service over an SSL channel;
- by having the form submitted by the POST method, so the responses are not cataloged in histories, logs, REFERER fields, etc..

Indeed, if neither the user nor server otherwise expose a user's password, and if the user has authenticated that he is talking to the intended server, then a strong case can be made that a properly authenticated request requires the user's awareness and approval. The password had to come from somewhere!

Clearly, password-based systems have other risks. Users may pick bad passwords or share them across services; the authentication is not bound to the actual service (that is, we have no non-repudiation); the adversary may mount online guessing attacks (Pinkus et al has recently considered some interesting countermeasures here [22]).

Cookies The server can establish longer state at a browser by saving a *cookie* at the browser. The server can choose the contents, expiration date, and access policy for this cookie; a properly functioning browser will automatically provide this cookie along with any request to a server that satisfies the policy. Many distributed Web systems—such as PubCookies [24]—use one of the above mechanisms to initially authenticate the browser user, and then use a cookie to amplify this authentication to a longer session at that browser, for a wider set of servers.

Cookie-based authentication can also be risky. Fu et al [6] discuss many design flaws in Cookie-based authentication schemes; PivX [27] discusses many implementation flaws in IE which allows an adversarial site to read other sites' cookies.

3.2 Validating User Input

Besides authenticating the user, another critical security aspect of providing Web services is ensuring that the input is correct.

Issues here can occur on two levels:

- An adversarial user can exploit server-side script vulnerabilities by carefully crafting *escape sequences* that cause the server to behave in unintended ways. The canonical example here is a server using user input as an argument in a shell command; devious input can cause the server to execute a command of the user's choosing.
- On an application level, an adversarial user can change the request data, such as form fields or cookie values. The canonical example here is a commerce server that collects items and prices via a form.

Standard good advice is that the script writer thoroughly vet any *tainted* user input [8], and also verify that critical data being returned has not been modified [23].

4 Client-Side PKI

4.1 Overview

When prodded, PKI researchers (such as ourselves) will recite a litany of reasons why PKI is a much better way than the alternatives to carry out authentication and authorization in distributed, multi-organizational settings. For example:

- PKI does not require shared secrets.
- PKI does not require a previously-established direct trust relationship between the two parties.
- PKI permits many parties to make assertions.
- PKI permits non-repudiation of assertions—Bob can prove to Cathy that Alice authorized this request to Bob.

As we mentioned in the introduction, browser-based keystores and client-side SSL are a dominant emerging paradigm for bringing PKI to large populations. Some organizations currently using client-side SSL include Dartmouth College, MIT, the Globus Grid project, IBM WebSphere and VPN software.

On the application end, numerous players preach the client-side SSL is a better way to authenticate users than passwords. We cite a few examples culled from the Web:

- The W3C: "SSL can also be used to verify the users' identity to the server, providing more reliable authentication than the common password-based authentication schemes." [26]
- Verisign: "Digital IDs (digital certificates) give web sites the only control mechanism available today that implements easily, provides enhanced security over passwords, and enables a better user experience." [12]
- Thawte: "Most modern Web browsers allow you to use a Personal Email Certificate from Thawte to authenticate yourself to a Web server. Certificate-based authentication is much stronger and more secure than password-based authentication." [20]
- Entrust: ..."identify or authenticate users to a Web site using digital certificates as opposed to username/password authentication where passwords are stored on the server and open to attacks." [4]

Recent research on user authentication issues also cite client-side SSL as the desired (but impractical) solution. [6, 22]

The clear message is that Web services using password-based authentication would be much stronger if they used client-side SSL instead.

4.2 At the Server

How does this work?

As noted earlier, the *secure sockets layer* permits the browser and user to establish an encrypted, integrity-protected channel over which to carry out their Web interaction: request, cookies, form responses, basic authentication data, etc. The typical SSL use includes server authentication; newer SSL uses permit the browser to authenticate as well. The server operator can require that a client authenticate via SSL, can restrict access based on how it chooses to validate the client certificate; server-side CGI scripts can interrogate client-certificate information, along with the other parameters available.

4.3 At the Browser

Typically, browser-based storage relies on some form of database system, such as Berkeley DB 1.85 [3], to store both certificates and private key material in a "secure" manner.

Netscape/Mozilla Netscape stores its security information in a subdirectory of the application named *.netscape* (*.mozilla*). There are two files of primary interest: *key3.db* which stores the user's private key, and *cert7.db*[2] which stores the certificates recognized by the browser's security module.

Both of these files are stored in a Berkeley DB 1.85 [3] format, meaning that they contain binary data and are unreadable by humans. Additionally, these files are password protected so that any application capable of reading the Berkeley DB format is still required to provide a password to read the plain text or modify the files without detection.

A detailed description of the techniques used to securely store user's keys is beyond the scope of this paper, but we point readers to [17, 16, 11, 10, 9] for details.

Internet Explorer/Windows Older versions of IE used a similar approach as Netscape, in that the keys were stored in the file system automatically (in a password protected .pfx or .pwl file which stores the private key in PKCS#12 format).

More recent versions of IE store the private key and certificate as a binary "blob" in the registry by default [2]. This absolves the key pair creator from having to worry about key management issues on the machine. This approach makes the private key and certificate as secure as the underlying operating system, in that the operating system is responsible for allowing/denying access to the registry.

Microsoft recommends against this behavior, noting that there is no password protection on the private key by default, and that the key is only as secure as the user's account [15]. This implies that if an attacker were to gain access to a user's account or convince the user to execute code with the user's privileges, the attacker would be able to use the private key at will, without having to go through any protections on the key (such as a password challenge).

One way to remedy the lack of password protection is to "export" the private key, placing it in a password protected .pfx file which stores the key in PKCS#12. Additionally, Windows 2000 and XP provide a means for displaying a warning or a password prompt to be displayed when the private key is being used.

Further, the latest versions of Microsoft Windows (Win2000 and XP) give applications an interface for protecting data called the *Data Protection API (DPAPI)*. In short, DPAPI provides OS-level data protection services to applications, allowing them to use the OS to store things like private keys and passwords. Applications use DPAPI via two functions which are a part of the CryptoAPI.

4.4 Historical Vulnerabilities

Netscape/Mozilla's keystore has remained fairly static, and to the best of our knowledge, historical vulnerabilities are also current vulnerabilities.

[2] In December 2002, NSS 3.7 introduced cert8.db, but it is nothing radically different.

Microsoft's IE, on the other hand, has gone through a number of revisions. Perhaps the most comprehensive list of problems with Microsoft's key storage system over the years comes from Peter Gutmann.

The first vulnerability comes from the format in which the private key is stored in some older versions of IE, and is exploited by a tool name *breakms*, available from Gutmann's web site [7]. The tool performs a dictionary attack to discover the password used to protect the file and outputs the private key.

Some versions of IE store private keys and certificates in the registry by default. This is disastrous. With a tool such as the "Offline NT Password & Registry Editor" [19], it is possible for an attacker to access a user's account given physical access to the computer on which the account resides in a few minutes. Since registry-stored keys are not password protected, an attacker can use the private key of the account's owner at will for as long as they are logged on. Additionally, an attacker could export the key to a floppy disk (password protecting it with a password that the attacker chooses), and then use tools like Peter Gutmann's or our modified version of OpenSSL to retrieve the key offline.

Prior to our work, we have not seen vulnerabilities demonstrated in DPAPI and in IE/XP medium-security and high-security keys.

5 The Question

We believe PKI is valuable and that secure Web services are important. We also realize that any deployment will require considerable effort and user education (as we participate in such a deployment here at Dartmouth). Hence, we believe that it's important to ask: *Does it work?*

If we encourage user populations to enroll in client-side PKI, and encourage service providers to migrate current services to use client-side SSL authentication and to roll out new services this way, have we achieved the desired goals: that service requests are authenticated from user A only when user A consciously issued that request?

To this end, we carried out a series of experiments.

Discussions of usability and security stress the importance the system behaving as the user expects [31], and the dangers in creating systems whose proper use is too complex [1, 28]. In the case of client-side PKI, we have two classes of users to consider:

- The user of the client browser, who requests services
- The user of the server, who sets up and deploys the Web application that provides these services.

As a consequence, we weren't focused on bizarre bugs (or extremely carefully constructed applications), but on general usability. If users on either end follow the "path of least resistance"—standard out-of-the-box configurations and advice—does it work?

Section 6 and Section 7 describe our experiments. Section 8 will consider countermeasures and implications.

6 Our Experiments: Usage of Keys

A basic assumption underlying client-side SSL is that the client's certificate and private key are used only for SSL requests that the client user was actually aware of and approved.

Is this true?

6.1 GET Requests

The language of Web interaction—even when restricted to HTML only, and no Javascript—makes it very easy for a server S_A to send content to a browser B, that causes the browser to issue an arbitrary request r to an arbitrary server.

If one wants this request r to be issued over SSL, we've found that a reliable technique is to use the HTML `frameset` construction, itself offered over server-side SSL. Figure 1 sketches this scenario; Figure 2 shows some sample HTML.

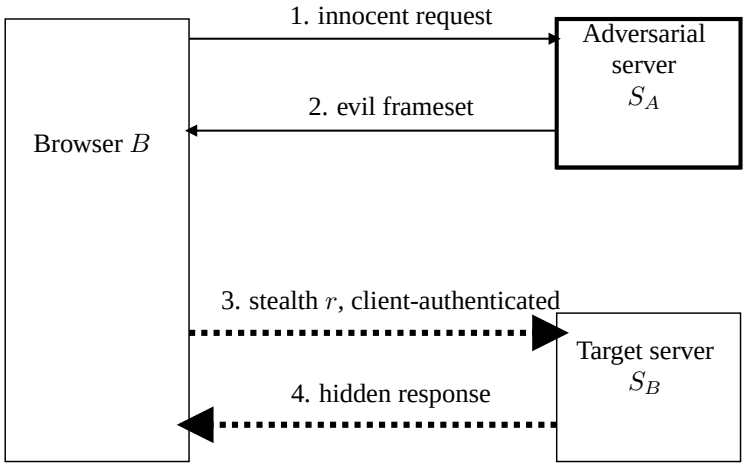

Figure 1: To borrow client-side authentication, the adversary needs to convince the browser's user to visit an SSL page at the evil server. Using the ordinary rules of Web interaction, the evil server can provide content that causes the browser to quietly issue a SSL request, authenticated with the user's personal certificate, to the victim server.

Basic Techniques A `frameset` enables a server S_A to specify that the browser should divide the screen into a number of frames, and to load a specified URL into each frame. The adversarial server can specify *any* URL for these these frames. If the server is careful with frame options, only one of these frames will be visible at the browser. However, the browser will issue all the specified requests.

(This appears to violate the well-known security model that "an applet can only talk back to the server that sent it" because this material is not an applet.)

We stress that this is different from full-blown cross-site scripting. S_A is not is using a subtle bug to inject code into pages that are (or appear to be from) other servers. Rather, S_A is using the standard rules of HTML to ask the browser to itself load another page.

Framesets and SSL In previous work [29], we noticed that if server S_A offers a frameset over server-side SSL, but specifies that the browser load an SSL page from S_B in the hidden frame, then many browser configurations will happily negotiate SSL handshakes with both servers—but (in the cases we tried) the browser will only report the S_A certificate.

So, we wondered what would happen if S_B requested client-side authentication.

- In Mozilla 1.0.1/Linux (RedHat 7.3 with 2.4.18-5 kernel), using default options, the browser will happily use a client key to authenticate, without informing the user.
- In IE 6.0/WindowsXP, using default options and any level key, the browser will happily use a client key to authenticate, without informing the user, if the user has already client-side authenticated to S_B.

  If the user has not, a window will pop-up saying that the server with a specified hostname has requested client-side authentication; which key, and is it OK?
- In Netscape 4.79/Linux (RedHat 7.3 with 2.4.18-5 kernel), using default options, the browser will pop-up a window saying that the server with a specified hostname has requested client-side authentication; which key, and is it OK? Then the browser will authenticate.

The request to S_B can easily be a `GET` request, forging response of a user to a Web form.

```
<html>
<frameset rows="*,1" cols="*,1" frameborder="no">

<frame src="f0.html" name="f0" scrolling="no">
<frame src="blank" name="b0" scrolling="no">
<frame src="blank" name="b1" scrolling="no">
<frame src="https://cobweb.dartmouth.edu:8443/cgi-bin/test.pl?
    debit=1000&
    major=None%3B%20I%27m%20withdrawing%20from%20the%20college"
   name="f1" scrolling="no">
</frameset>
<noframes> no frames </noframes>
</html>
```

Figure 2: HTML permits an adversarial server to send a frameset to a browser. The browser will then issue requests to obtain the material to be loaded into each frame. A deviously crafted frameset (such as the one above) appear to be an ordinary page. If an adversarial server includes a form response in the hidden frame, the browser will submit an SSL request to an arbitrary target server via GET. In many scenarios, browsers will use client-side authentication for the GET; with the devious frameset, the user may remain unaware of the request, the use of his personal certificate, and the response from the target.

6.2 POST Requests

Some implementors preach that no sane Web service should accept `GET` response to Web forms.

If we extend the adversary's tools to include Javascript, then the adversarial page can easily include a `form` element with default values, and an `onload` function that submits it, via an SSL `POST` request, to S_B. Figure 3 sketches this code.

Sending this page via a hidden frame further hides the request and the response.

Again, browsers will use the user's personal certificate to authenticate this request.

6.3 Implications

As we noted earlier, it is continually touted that client-side SSL is superior to password-based authentication.

Suppose the operator of an honest server S_B offers a service where authorization or authentication are important. For example:

- Perhaps S_B wanted to prove that its content was served to particular authorized parties (and perhaps to prove that those parties requested it—one thinks of Pete Townshend or a patent challenge).
- Perhaps S_B is offering email or class registration services, via `form` elements, to a campus population.

If S_B had set up their site with server-side SSL, and required basic authentication or some other password scheme, then one might argue that a service can be carried out in a user's name only if that user authorized it, or shared their password.

However, suppose S_B uses "stronger" client-side SSL. With Mozilla and default options, a user's request to S_B can be forged by a visit to an adversarial site S_A. With IE and default options, a user's request can be forged if the user has already visited S_B.

With Netscape or IE, a user's request to S_B can probably be forged without the user noticing if the adversarial site simply claims that S_A *also* requires client-side authentication. With this bogus claim—which sounds quite reasonable in a campus enterprise environment—then the user will expect to see the password prompts, etc. (and probably won't notice any fine print).

```
<html>
<head>
<SCRIPT LANGUAGE=javascript>
  function fnTemp()
   {
     document.myform.submit();
   }
</script>
</head>
<body onload="fnTemp()">

<form name="myform" method="post"
  action="https://cobweb.dartmouth.edu:8443/cgi-bin/test.pl">
<input name="debit" value="1000">
<input name="major" value="Hockey">
<input type="submit" value="Submit Form">
</form>
</body>
</html>
```

Figure 3: A web page such as this uses Javascript to cause the browser submit an SSL request to an arbitrary target server via POST. In many scenarios, browsers will use client-side authentication for the POST. If an adversarial server specifies that this page be loaded into a hidden frame, then the user may remain unaware of the request, the use of his personal certificate, and the response from the target.

We note that this is different from the standard single-sign-on risk that, once a user arms their credential, their browser may silently authenticate to any site the user consciously visits—because here, the user's browser silently authenticates to any site of the adversarial site's choosing.

Limits We could not demonstrate a way for the adversary, using the tools of sending standard HTML and Javascript to users with standard browsers, to forge a response to a file upload `input` tag (see further discussion below) or to forge `REFERER` fields (although `telnet` links look promising).

7 Our Experiments: Storage of Keys

In Section 6, the adversary can only borrow use of the client's private key, under constrained circumstances. It would be much more interesting to steal the key outright.

7.1 Stealing Netscape/Mozilla Keys from Foolish Users

In Netscape and Mozilla, the private key is stored in the *key3.db* file, as discussed in Section 4.3. Knowledge of this file and the user's keystore password enables easy extraction of the private keys.

There is a significant amount of information available describing the algorithms used to store the private keys and certificates in the Netscape/Mozilla browser. As a result, there are a number of tools which can be used to view and modify Netscape's and Mozilla's key stores.

The NSS Security Tools (available from Mozilla [16]) allow users to add and delete the key and certificate databases directly. While it is possible to vandalize key databases, none of the tools seemed to give direct access to the private key. NDBS 2.0, a free tool available from Carnegie Mellon University allows programmatic access to Netscape's key and certificate databases [11]. The tool is extremely powerful in that it enables applications to capture the private key in a number of formats and do what they wish with it—store it to a file, use it to generate signatures, post it on a web site, etc. NDBS is really a set of Java classes which enable programmers to write code which accesses the key stores.

We have devised a simple but effective attack which allows us to capture users' private keys. The attack relies on a misleading interface presented on a web page to trick users into sending us the file containing their private keys, the file containing their certificate, and the password needed to access those files. Once collected, the items are given to a program which uses NDBS to discover the private keys and forge the digital signatures.

The attack begins by users pointing their web browser to an official looking site that claims to be the Dartmouth Authentic Really Secure Service (DARSS), which is advertised as some special service that requires PKI. The user is told that the DARSS must verify the key pair, and ensure that the user is actually authorized to use the key. This is done by filling out a form which points the browser to the files containing the private key and certificate [3] and prompts the user for the password protecting the files. Note that the password `input` tag can have the `type=password` option, so that the characters don't echo on the string, in order to increase the user's feeling of security.

This is a flat-out lie. The DARSS is not verifying anything, and the password is not used to check authorization. In actuality, the files and password are shipped to a directory and archived, so that another program can discover and collect the private keys and forge digital signatures.

The system has two main components, the front end which is the SSL site responsible for advertising the DARSS and the form tricking the user into sending us their files and password, and the back end which is the system which uses either NDBS (for Netscape keys) or OpenSSL (for *exported* IE keys) to open the files and capture the private key.

7.2 Stealing Netscape Keys from Wiser Users

The attack of Section 7.1 above requires that the users are vulnerable to the social engineering technique of simply asking for their password and keystore paths. Can we modify this attack to be effective against more cautious users?

Our lab has some Web spoofing experience [30, 29]. Drawing on these tricks, we easily constructed a server page that opens a window that looks and acts very much like the standard Netscape classic-skin keystore password prompt. (Our initial foray into Mozilla left a "Javascript" warning at the top; using the browser's own `alert` pop-up did not disable echoing of the password prompt.) Thus, for many Netscape users, the adversary can easily obtain the keystore password.

The next step is to get the encrypted password files. The HTML spec permits the server to specify a default value for `input` tags of `type=file`; and Netscape leaves keys in a known place, so that's a start. A bit of experimentation revealed a way to create submittable forms with `type=file` fields whose text boxes and "Browse" buttons were not visible to the user (even without Javascript). However, then we ran into a stumbling block: the standard browsers deliberately disregard the HTML spec, and do *not* actually render default values for `type=file` upload tags. Except with explicitly buggy browser versions that were quickly patched, the user must actually type something. (It appears we have hints of a trusted path from user to server!)

To get around this, we need to upgrade the adversary's toolkit to include an executable running on the user's platform. This executable locates and sends back the encrypted key file. (It has been rumored that the password itself can be found by careful inspection of the browser process's address space, but we did not try that.)

The permeability of modern computing environments, plus the general lack of a trusted path from browser to user, makes Netscape a risky place to leave client private keys.

7.3 Stealing IE Low-Security Keys

In older Windows platforms, obtaining client private keys was fairly straightforward, as discussed in Section 4.4. Modern versions of Windows and IE have added features which make obtaining the client's private key a bit more challenging. Examples include:

- providing medium and high security options to the key generation functions, resulting in either a warning or a password prompt when the private key is accessed by an application.

[3] In Netscape, the keys are in a default place relative to the user's home directory, and the home directory is implicit if the user types in a relative path.

- giving all applications a means to request that the OS securely store data (such as a private key or password) via the DPAPI.

However, these are just features, some applications (e.g. legacy applications) do not use these newer security features. Of particular interest to us is the ability to generate a "low-security key"—a key which can be used by any application running with the user's privileges without warning the user that the key is use. Of further disturbance is the fact that this is the CryptoAPI's default behavior.

Peter Gutmann raised serious concern over the `CryptExportKey()` function found in the CryptoAPI back in 1998[7]. Specifically, with the default key generation, any program running under the user's privileges may call this function and obtain a copy of the user's private key.

We were curious to see if the latest versions of the CryptoAPI have remedied this issue. Our conclusion: "no". We were able to construct a small executable which, when run on a low-security (default) key, quietly exports the user's private key with no warning.

7.4 Stealing IE Medium-Security and High-Security Keys

The Windows CryptoAPI does permit users to bring in keypairs at "medium" or "high" security levels. With both of these levels, use of the private key will trigger a warning window; in the high-security option, the warning window requests a password.

Consequently, the attack of Section 7.3 may not work; when the executable asks the API, the user may notice an unexpected warning window. So our attack strategy has to improve.

7.4.1 API Hijacking

Before we discuss the specifics of stealing private keys, a brief introduction to the general method of *API Hijacking* is in order. The goal of this attack is to intercept (hijack) calls from some process (such as IE) to system APIs (such as the CryptoAPI).

Delay Loading API Hijacking uses a feature of Microsoft's linker called "Delay Loading". Typically, when a process calls a function from an imported Dynamically Link Library (DLL), the linker adds information about the DLL into the process (in what is referred to as the "imports section"). This topic is discussed in depth by Matt Pietrik [21], but we present a very brief overview.

When a process is loaded, the Windows loader reads the imports section of the process, and dynamically loads each DLL required. As each DLL is loaded, the loader finds the address of each function in the DLL and writes this information into a data structure maintained in the process's imports section known as the Import Address Table (IAT). As the name suggests, the IAT is essentially a table of function pointers.

When a DLL has the "DelayLoad" feature enabled, the linker generates a small stub containing the DLL and function name. This stub is placed into the imports section of the calling process instead of the function's address. Now, when a function in the DLL is called by a process for the first time, the stub in the process's IAT dynamically loads the DLL (using `LoadLibrary()` and `GetProcAddress()`). This way, the DLL is not loaded until a function it provides is actually called—i.e. its loading is delayed until it is needed.

For delay loading to be used, the application must specify which DLLs it would like to delay load via a linker option during the build phase of the application. So, how does one use delay loading on a program for which they can not build (possibly because they don't have the source code—i.e. IE)?

DLL Injection The answer is to redirect the IAT of the victim process (e.g. IE) to point to a stub which implements the delay loading *while the process is running*.

The strategy is to get the stub code as well as the IAT redirection code into an attack DLL, and *inject* this DLL into the address space of the victim process. Once the attack DLL is in the process, the IAT redirection code changes

the victim's IAT to point to the stub code. At that point, all of the victim process's calls to certain imported DLLs will pass through the attack DLL (which imported DLLs are targeted and which functions within those DLLs are specified by the attack DLL—i.e. the attacker get to choose which DLLs to intercept). This implements a software man-in-the-middle attack between an application and certain DLLs on which it depends.

The Windows OS provides a number of methods for injecting a DLL into an process's address space (a technique commonly referred to as "DLL Injection"). The preferred method is via a "Windows Hook", which is a point in the Windows message handling system where an application can install a routine which intercepts messages to a window.

7.4.2 Hijacking the CryptoAPI

Using the techniques above, we were able to construct a couple of programs which allowed us to intercept function calls from IE to the CryptoAPI. This is particularly useful for stealing medium or high security private keys which display warning messages when used (in a client-side SSL negotiation, for example).

The idea is to wait for IE to use the key (hence, displaying the warning or prompting for a password), and then get a copy of the private key for ourselves—*without triggering an extra window that might alert the user.*

The Attack Essentially, the attack code is two programs: an attack DLL with the IAT redirection code and the delay loading stubs, and one executable to register a hook which is used to inject the attack DLL into IE's address space.

The strategy is:

- Get the attack DLL and executable onto the victim's machine (perhaps through a virus or a remote code execution vulnerability in IE).
- Get the executable running. This installs a Windows hook which gets the attack DLL injected into IE's address space.
- Change IE's IAT so that calls to desired functions in the CryptoAPI (crypt32.dll) are redirected to our attack DLL.
- At this point, we have complete control and are aware of what IE is trying to do. For example, if we specify `CryptSignMessage()` to be redirected in our attack DLL, then every time IE calls this function (e.g. to do an SSL client-side authentication), control will pass to our code.
- We know that the user is expecting to see a warning in this case, so we take advantage of the opportunity to do something nefarious—like export the private key. In our current demo, the adversarial code exports the private key, so the warning window will say "exporting" instead of "signing" at the top[4].

  This could be remedied by hijacking the call which display the warning. In fact, this would allow us to disable all such warnings, but we did not implement this.

It should be noted that if the user has a low-security key, we can steal the private key with no information being passed to the user.

In sum, API Hijacking can lead to a number of quite effective attacks that can undermine many underlying security mechanisms. In addition to getting the private key (as in our demo), it would be possible (and easier) to simply "use" the private key to forge signatures.

[4]In our demo, we fail the IE request, so the user sees a "404" error; however, more polite adversarial code might carry out the requested cryptographic operation and return the response to the user.

8 Analysis

8.1 Short-Term Countermeasures

Many institutions, including are own, are working on trying to deploy client-side PKIs and Web services that use them.

As we have demonstrated, it is very easy to roll out Web services (particularly if one is migrating password-based services) where client-side authentication with a personal certificate does not imply the person was aware of or approved this request. Those using client-side PKI as a replacement for password-based schemes should take note.

One colleague suggested the use of hidden form fields. This is a good idea. At a minimum, we recommend that providers of services intending to use personal certificate authentication do a two-step process: first, authenticate the requester and construct a form that includes hidden field containing the name of requester, a timestamp, all signed (or MAC'd); then, check for this field on the form submission. So far, we have not been able to figure out how an adversarial server can fool such a service (without using an implementation bug, such as code-injection).

We also recommend that deployers very carefully explore the certificate use options in the browser platforms in their user population, and educate users to choose wise configurations and pay attention to the implications. In particular, many of the browsers we saw, when configured to warn of key use, only warned with the hostname—so deployers would be wise not to mix different security domains on the same host.

Forcing browsers to periodically re-negotiate with the server via the server's `Keep-Alive` can be used to kill SSL sessions. This would make it a little trickier for the adversary to borrow authentication—e.g., more likely to trigger warnings.

To combat key theft, we recommend that deployers insist on medium-security and high-security keys for IE (since this complicates the attack), and consider making keys non-exportable.

As news continues to show [14], code injection and other malicious executables remain a constant risk in contemporary networked desktop environments. Using long RSA moduli does not help if the underlying platform is easily permeable. We recommend that deployers not overlook the basics of aggressively maintaining system security on machines housing private keys.

8.2 Long-Term

On a deeper level, one might argue that the term "personal certificate" is a misnomer. In the best case, using a browser-stored key for client authentication (via SSL) authenticates two HTTP endpoints. No person need be involved at all. Even though much conventional wisdom implies client-side SSL can replace weaker authentication, SSL designers (e.g., [25]) state that client-side SSL is simply not intended to replace application-level security. With client-side SSL (and perhaps many single sign-on schemes), "what the user knows" has been replaced by "what some complex software on the user's desktop does in response to complex stimuli."

Consequently, it would be interesting to re-consider the client Web authentication system, from the perspective of security and usability. To cite just a few design principles: [31]

- "The path of least resistance" for users should result in a secure configuration.
- The interface should expose "appropriate boundaries" between objects and actions.
- Things should be authenticated in the user's name only as the "result of an explicit user action that is understood to imply granting."

One might quip that it has hard to find a principle here that the current paradigm does *not* violate.

One natural area for further attention is a *trusted path.* Our earlier work [29] built trusted paths from the browser to the user. We also need trusted paths in the other direction (e.g., a Web equivalent of the "secure attention key") and an easy way for Web service writers to invoke that. This may not be as much of a stretch as one might think; already, the standard browsers depart from the HTML spec and require that a user type a value into a `file` input tag. Wouldn't a

`authenticate` input tag be much easier than trying to work through cryptographic hidden fields? Adding another level of personal certificate that only was invokable via such a tag (and perhaps even signed something) would help.

Until then, ongoing work [22] in using *reverse Turing tests* to defeat robotic probing could assist a server in setting up a trusted path.

Another area for further attention is the user's mental model of Web interaction. For this new `authenticate` tag (or even current warning windows) to be effective, the screen material to which it applies should be clearly perceivable by the user. Even adopting the "Basic Authentication" model of letting the server demanding the authentication provide some descriptive freetext might help—instead of "hostname wants you to authenticate," the browser window might say "...in order to change your class registration—are you sure?".

On a system level, we recommend that further examination be given to the module that stores and wields private keys: perhaps a trustable subsystem with a trusted path to the user. As a device which has a very rich and complex interaction with the rest of the world, browsers can often behave in unexpected and unclear ways. Such a device should not be the cornerstone of a secure system.

9 Conclusions

One might look at this work from a narrow or broad perspective.

From a narrow perspective, here's what we do:

- We explicitly demonstrate that client-side authentication with personal certificates does not necessarily authenticate the person. (In the hoopla surrounding client-side PKI, we had not seen this issue raised. Indeed, the opposite is preached.)
- We explicitly demonstrate that browser-based keystores—even IE medium-security ad high-security keys—are easily permeable in modern desktop environments. (We had not seen attacks on these keys, nor had we seen use of our approach applied to cryptographic APIs.)

However, we intend this paper to have a broader perspective as well. We believe in PKI. Much work is being done in many places to try to bring PKI to users; considerable investment of effort is being focused on the client-side browser paradigm. We humbly suggest that some of this investment might be better spent rethinking the basic model.

Acknowledgments

The authors are grateful to our many colleagues—both in the Dartmouth PKI Lab and in the greater Internet2 community—for their help and advice.

References

[1] R. Anderson. Why Cryptosystems Fail. *Communications of the ACM*, pages 32–40, November 1994.

[2] AspEcrypt. Opening a Certificate Store. `http://www.aspencrypt.com/task_certs.html`.

[3] Berkeley DB. `http://www.sleepycat.com`.

[4] Buyer's guide - web portal security solution. `http://www.entrust.com/resources/pdf/buyers_guide.pdf`, November 2001.

[5] E. Felten and M. Schneider. Timing Attacks on Web Privacy. In *ACM Computer and Communications Security*, 2000.

[6] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and Don'ts of Client Authentication on the Web. In *USENIX Security*, 2001.

[7] Peter Gutmann. How to recover private keys for Microsoft Internet Explorer, Internet Information Server, Outlook Express, and many others - or - Where do your encryption keys want to go today? `http://www.cs.auckland.ac.nz/~pgut001/pubs/breakms.txt`.

[8] J. Hamilton. *CGI Programming 101*. CGI101.com, 1999.

[9] S. Henson. Netscape Certificate Database Info. http://www.drh-consultancy.demon.co.uk/cert7.html.

[10] S. Henson. Netscape Key Database Format. http://www.drh-consultancy.demon.co.uk/key3.html.

[11] Pisey Huy, Grace A. Lewis, and Ming-hsun Liu. Beyond the Black Box: A Case Study in C to Java Conversion and Product Extensibility. Technical report, Carnegie Mellon Software Engineering Institute, 2001.

[12] Implementing web site client authentication using digital ids and the netscape enterprise server 2.0. http://www.verisign.com/repository/clientauth/ent_ig.htm#clientauth.

[13] K. Kain, S.W. Smith, and R. Asokan. Digital Signatures and Electronic Documents: A Cautionary Tale. In *Advanced Communications and Multimedia Security*. Kluwer Academic Publishers, 2002.

[14] Microsoft security bulletin ms03-004. http://www.microsoft.com/technet/treeview/default.asp?url=/technet/secu%rity/bulletin/ms03-004.asp, February 2003.

[15] Microsoft Authenticode Developer Certificates. http://www.thawte.com/getinfo/products/devel/msauthenticode.html.

[16] Mozilla. NSS Security Tools. http://www.mozilla.org.

[17] Netscape Communications Corp. *Command-Line Tools Guide : Netscape Certificate Management System*, 4.5 edition, October 2001.

[18] J. Niederst. *Web Design in a Nutshell (2/E)*. O'Reilly, 2001.

[19] Offline NT Password & Registry Editor. http://home.eunet.no/~pnordahl/ntpasswd.

[20] Personal certificates. http://www.thawte.com/html/COMMUNITY/personal/.

[21] Matt Pietrek. Under the hood. *Miscrosoft Systems Journal*, February 2000.

[22] B. Pinkas and T. Sander. Securing Passwords Against Dictionary Attacks. In *ACM Computer and Communications Security*, 2002.

[23] Preventing html form tampering. http://advosys.ca/papers/form-tampering.html, August 2001.

[24] Pubcookie: open-source software for intra-institutional web authentication. http://www.washington.edu/pubcookie/.

[25] E. Rescorla. *SSL and TLS - Designing and Building Secure Systems*. Addison Wesley, 2001.

[26] L. Stein and J. Stewart. The world wide web security faq. http://www.w3.org/Security/Faq/, February 2002.

[27] Unpatched ie security holes. http://www.pivx.com/larholm/unpatched/.

[28] A. Whitten and J.D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *USENIX Security*, 1999.

[29] E. Ye and S.W. Smith. Trusted Paths for Browsers. In *USENIX Security*, 2002.

[30] E. Ye, Y. Yuan, and Smith S.W. Web Spoofing Revisited: SSL and Beyond. Technical Report TR2002-417, Department of Computer Science, Dartmouth College., 2002.

[31] K.-P. Yee. User Interaction Design for Secure Systems. http://zesty.ca/sid/uidss-may-28.pdf.