Computers & Security

www.elsevier.com/locate/cose

Keyjacking: the surprising insecurity of client-side SSL[☆]

John Marchesini*, S.W. Smith, Meiyuan Zhao

Department of Computer Science, Hinman Box 6211, Dartmouth College, Hanover, NH03755, United States

Received 20 April 2004; accepted 22 June 2004

KEYWORDS
PKI;
SSL;
Authentication;
Usability;
Application security;
Spoofing;
API hijacking;
USB tokens

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 various client keystores. However, whether this works depends on whether what the machines do with the private keys matches what the humans think they do: whether a server operator can conclude from an SSL request authenticated with a user's private key that the user was aware of and approved that request. Exploring this vision, we demonstrate via a series of experiments that this assumption does not hold with standard desktop tools, even if the browser user does all the right things. A fundamental rethinking of the trust, usage, and storage model might result in more effective tools for achieving the PKI vision.

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

[☆] This report is a revised and extended version of TR2004-498 (February 2004), TR2003-443 (February 2003; available at www.cs.dartmouth.edu/~carlo/research/tr2003-443.pdf.) and our subsequent conference paper in the *2nd Annual PKI Research Workshop*, of April 2003.

* Corresponding author. Tel.: +1 603 646 9179.
E-mail addresses: carlo@cs.dartmouth.edu (J. Marchesini), sws@cs.dartmouth.edu (S.W. Smith), zhaom@cs.dartmouth.edu (M. Zhao).

0167-4048/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2004.06.014

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, we are beginning to see some of the necessary building blocks to achieve the client-side vision:

- Some modern operating systems (e.g., all flavors of Windows and Mac OSX) include a keystore and a set of *Cryptographic Service Providers* (CSP) which can be used by any application on the machine.
- Modern browsers which are designed to be used across multiple platforms (e.g., Netscape/Mozilla) now include 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. Some populations are even making plans to distribute USB tokens to users in order to store their key pairs.
- Providers of Web information services are starting to use client-side SSL as a better alternative than passwords or to authenticate users.

Here at Dartmouth, we are deploying a client-side PKI to enable users (primarily from IE/XP platforms, often shared) to access Web-based academic information services (such as changing class registrations and recording grades).

Using client-side PKI can alleviate the need for users to remember multiple passwords for multiple services. It also reduces the risk of an attacker capturing a user's password either by guessing or via a keyboard sniffer.

Does it work?

In previous work, we have examined the effectiveness of server-side SSL (Ye and Smith, 2002) and of digital signatures on documents (Kain et al., 2002). In this paper, we examine the question: *does this client-side PKI work*?

- When browsers use a private key 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 client was aware of and approved that request?

The perception of users—not only the users at the client, but also the application authors and administrators at the server—plays a critical role in determining whether client-side PKI works. If the natural mental models of the system do not match the actual system behavior, then users have no basis to make reasonable trust decisions. In security settings (such as client-side PKI), this inability to reason about the system can thwart the security efforts that the system's designers have implemented.

Our agenda

We wish to stress that we believe that PKI is a much better way than the alternatives to carry out authentication and authorization in distributed, multi-organizational settings. PKI does not require shared secrets or a previously established direct trust relationship between the two parties. Further, PKI permits many parties to make assertions, and allows for non-repudiation of those assertions—Bob can prove to Cathy that Alice authorized this request to Bob.

However, rolling out client-side PKI and migrating existing information services to use it requires considerable resources and effort. Weaknesses in the underlying technology risk undermining this effort. We provide a critical examination of the current client-side PKI approach precisely because we want the PKI vision to succeed.

This paper

In the next section, we examine the status quo. Then, we pose the question which drives our experiments. The later sections describe our experiments. We conclude in the last section.

The current state of affairs

Web information services and web applications

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 which 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.

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 applications

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 Niederst, 2001, chapter 15).

The 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 for a specific URL. If the submit action specified the GET method, the parameters are pasted onto the end of the URL; if the POST method was specified, the parameters are sent back in a second request part.

However, the 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.

Security mechanisms

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. The Web paradigm provides several standard avenues to do this.

Client address

For one example, 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 username 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 that sensitive data are protected in transit. Some of the common approaches include offering the entire service over an SSL channel, and 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!

Password-based systems have a number of risks. Users may pick bad passwords or share them across services; the authentication is not bound to the actual service (i.e., we have no non-repudiation); the adversary may mount online guessing attacks (Pinkas and Sander, 2002 has recently considered some interesting countermeasures here); users may not check that they are connected to correct server, making them vulnerable to bogus sites that look similar (i.e. ''Spoofing'', Felten et al., 1997; Ye and Smith, 2002; Ye et al., 2002).

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 a specific cookie; a properly functioning browser will automatically provide the cookie along with any request to a server that satisfies the policy. Many distributed Web systems—such as ''PubCookies'', and Microsoft's ''passport'' (Microsoft .Net Passport)—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., 2001 discuss many design flaws in cookie-based authentication schemes; PivX (Unpatched IE security holes) discusses many implementation flaws in IE which allows an adversarial site to read other sites' cookies.

Validating user input

Besides authenticating the user, another critical security aspect of providing Web services is ensuring that the input is correct. Failing to do so can lead to a number of issues. For example, 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. Another example occurs on the application level, where 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, and allows a malicious user to purchase an item for a lower price than the vendor intended.

Standard good advice is that the script writer thoroughly vet any *tainted* user input (Hamilton, 1999), and also verify that critical data being returned have not been modified (Preventing HTML form tampering, 2001).

Client-side PKI

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. In practical settings, many enterprises are adopting PKI technology because it allows single sign-on, minimizes the impact of keyboard sniffers, is trendy, and is being heavily marketed by PKI vendors.

As we mentioned in the introduction, using various keystores and client-side SSL is 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 many suppliers of VPN software.

On the application end, numerous players preach that 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'' (Stein and Stewart, 2002).
- 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'' (Digital IDs and the Netscape Enterprise Server 2.0).
- 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'' (Personal certificates).
- 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'' (Buyer's guide, 2001).

Recent research on user authentication issues also cite client-side SSL as the desired (but impractical) solution (Fu et al., 2001; Pinkas and Sander, 2002). The clear message is that Web services using password-based authentication would be much stronger if they used client-side SSL instead.

At the server

How does this work?

As noted earlier, SSL 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 via PKI as well. The server operator can require that a client authenticate via PKI, and 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.

At the browser

Different browsers take different approaches to storing keys and certificates. Our experiment focuses on the two browsers which are the most commonly used: Netscape and Internet Explorer.

Netscape stores its security information in a subdirectory of the application named `.netscape` (`.mozilla` in Mozilla). There are two files of primary interest: `key3.db` which stores the user's private key, and `cert8.db` which stores the certificates recognized by the browser's security module. Both these files are binary data, stored in the Berkeley DB 1.85 format. Additionally, the information in these files is encrypted with a keyphrase so that any application capable of reading the Berkeley DB format is still required to provide a password to read the plaintext or to 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 Henson (Netscape certificate data info), Henson (Netscape key database format), Huy et al. (2001), Mozilla (NSS security tools) and Netscape Communications Corp (2001) for details.

Internet Explorer relies on the Windows keystore and CSP to store the private key. One unfortunate result of this tight coupling between IE and the OS is that versions of IE which run on MacIntosh computers have no support for storing or using private keys.

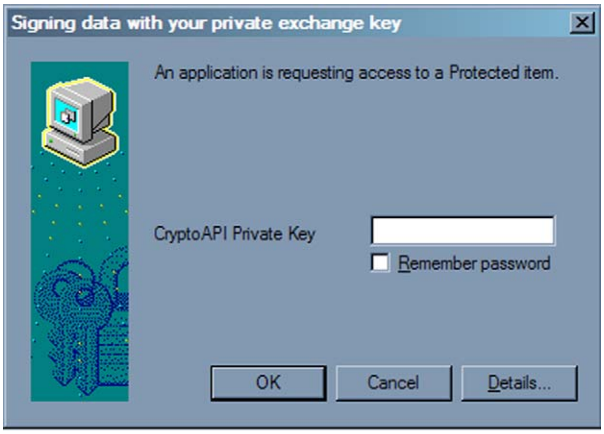
By default, Windows uses its own CSPs to store the private key, which generate *low-security keys* (i.e., not password-protected) by default. Many organizations (such as the DoD and even Microsoft) recommend against this behavior, noting that the key is only as secure as the user's account (Microsoft authenticode developer certificates). 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 `.pwl` file (for IE 3 and earlier) or a `.pfx` file which stores the key in PKCS#12 (for IE 4 to current versions). Once the key is exported, a user must then "import" it at a higher security level: *medium-security*, which prompts the user when the key is used; or *high-security*, which requires a password to use the key (assuming the user does not check the box marked "Remember password", which immediately demotes it to a low-security key; see Fig. 1).

While exporting and reimporting the private key may seem like a cumbersome process, it has become a standard practice in many organizations. In fact, the DoD guidelines for the Defense Message System outline the process in detail (DoD PKI Class 3 Certificates, 2003).

The question

We believe PKI is valuable and that secure Web information 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 is important to ask: *Does it work?*



Signing data with your private exchange key

An application is requesting access to a Protected item.

CryptoAPI Private Key

□ Remember password

Figure 1 The Microsoft CSP's password-prompt dialog.

Discussions of usability and security stress the importance the system behaving as the user expects (Yee, User interaction design for secure systems), and the dangers in creating systems whose proper use is too complex (Anderson, 1994; Whitten and Tygar, 1999). It is advertised that by using client-side PKI, a client can assume that his private key is used only to authenticate services he was aware of and intended, and a server operator can assume that the client was aware of and approved that request. In the common understanding of the Web, the user must choose to click in order to issue a request. With the protections promised by medium-security and high-security CSPs, we additionally expect the user to see and approve a warning dialog before a private key is used.

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 in order to evaluate the effectiveness of using the browser and client-SSL as a component of a client-side PKI. (However, some of our attacks have a wide range of applications, and could potentially be used to subvert other authentication schemes as well. We focus on PKI because it is claimed to be the strongest—and in theory, it could be.)

We were not 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—do they construct a mental model which matches actual system behavior?

Experiment 1: stealing keys

Historical vulnerabilities

This research began when we noticed some of the weakness of client-side PKI and browser keystores in the literature. Perhaps the most comprehensive list of problems with Microsoft's key storage system over the years comes from Gutmann (How to recover private keys for Microsoft Internet Explorer). Three vulnerabilities in particular caught our attention.

The first vulnerability applies to situations where the private key is stored without password protection (the Microsoft CSP's default behavior). With a tool such as the "Offline NT password & registry editor", it is possible for an attacker to access a user's account in a few minutes, given physical access to the computer on which the account resides. Since the private key is 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 Gutmann's or our modified version of OpenSSL to retrieve the key offline.

The second vulnerability comes from the format in which the private key is stored on disk once it has been exported (in a `.pwl` or `.pfx` file). Gutmann's *breakms* tool performs a dictionary attack to discover the password used to protect the file and outputs the private key.

The third vulnerability involves the `CryptExportKey` function found in the CryptoAPI, which Gutmann raised concerns about back in 1998. Specifically, with the default key generation (i.e., no password protection) and an exportable key, any program running under the user's privileges may call the `CryptExportKey` function and silently obtain a copy of the user's private key.

In the Microsoft "low-security" model of client-side PKI, it seems that one has to trust the entire system (OS, IE, the CSP, etc.) for the client-side vision to work. If an attacker compromises one piece of the system (e.g., an executable with a user's privileges), then because of tight coupling, the attacker can violate other parts the system (e.g., the user's private key).

Stealing low-security keys

We wondered if, with its new security emphasis, Microsoft had fixed some of these vulnerabilities, either directly or perhaps as a result of decoupling IE and the OS.

We began our experiments with the 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 in use. Immediately, we noticed that generating low-security keys is still the Microsoft CSP's default behavior.

We were curious to see if the latest versions of the `CryptoAPI` and CSP have remedied the CryptExportKey issue, perhaps by warning the user when their private key is being exported. 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.

Stealing IE medium-security and high-security keys

The Windows CryptoAPI does permit users to import 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 previous attack may not work; when the executable asks the API to export the key, the user may notice an unexpected warning window. So our attack strategy had to improve.

API hijacking

Before we discuss the specifics of stealing medium and high-security keys, a brief introduction to the general method of *API hijacking* is in order. The goal of API hijacking is to intercept (hijack) calls from some process (such as IE) to system APIs (such as the CryptoAPI).[1]

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*). We present a brief overview here (see Pietrek, 2000 for details).

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 "delay load" feature enabled, the linker generates a small stub containing the DLL and function name. This stub (instead of the function's address) is placed into the "imports" section of the calling process. 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.

DLL injection

So, how does an attacker use delay loading on a program for which he cannot build (possibly because he does not have the source code—i.e. IE)? 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 gets 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 a 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.

Hijacking the CryptoAPI

Using the techniques above, we were able to construct a couple of programs which, running at user privileges only, 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: the *parasite*—an attack DLL with the IAT redirection code and the delay loading stubs, and the

[1] Very recently, other researchers have suggested an attack that replaces the original DLLs (OS Security, Inc.). However, the countermeasures suggested for that attack don't defend against ours.

grappling hook—an executable to register a hook which is used to inject the attack DLL into IE's address space.

We implemented this attack as follows:

1. Get the parasite and grappling hook onto the victim's machine (perhaps through a virus or a remote code execution vulnerability in IE).
2. Get the grappling hook running with the user's privileges. This installs a Windows hook which gets the parasite injected into IE's address space.
3. The parasite changes IE's IAT so that calls to desired functions in the CryptoAPI (crypt32.dll and advapi32.dll) are redirected to the parasite.
4. 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 parasite, then every time IE calls this function (e.g. to do an SSL client-side authentication), control will pass to our code.
5. 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.[2]
   This could be remedied by hijacking the call which displays the warning. In fact, this would allow us to disable all such warnings, but we did not implement this.

Non-exportable keys
The bottom line is that with a DLL and small executable running with the victim's privileges, the private key—even with medium-security or high-security protections—can be stolen if it is exportable. The obvious solution is to make keys non-exportable, and we verified that this countermeasure prevents the attack.

Experiment 2: malicious use of keys via content-only attacks

Since making keys non-exportable stops outright theft, we had to revise our attack strategy. We wondered if we could just use the key at will without having to actually steal it, as this would be just as devastating and would work against non-exportable keys. So we began our second experiment with the question: "Can we exploit the complexity of the language of interaction in order to use the key as we wish, even if we are limited to just serving content to the browser? "

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 have found that a reliable technique is to use the HTML `frameset` construction, itself offered over server-side SSL. Fig. 2 sketches this scenario; Fig. 3 shows some sample HTML.

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 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 behavior 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.

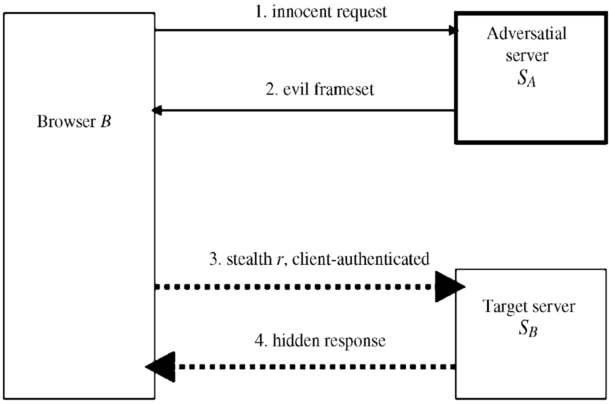


Figure 2 To borrow client-side authentication, the adversary needs to convince the browser's user to visit an SSL page at the evil server. Using the ordinary rules of Web interaction, the evil server can provide content that causes the browser to quietly issue an SSL request, authenticated with the user's personal certificate, to the victim server.

[2] In our demo, we fail the IE request, so the user sees a "404" error.

```
<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 3 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) appears 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.

We stress that this is different from full-blown cross-site scripting. S_A is not 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 (Ye and Smith, 2002), 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 $\mathrm{S_B}$ can easily be a GET request, forging response of a user to a Web form.

POST requests

Some implementors preach that no sane Web service should accept GET response to Web forms. However, services that use POST responses are also vulnerable. 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.

Fig. 4 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.

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), or perhaps S_B is offering email or class registration services, via `form` elements, to a campus population.

If S_B had used server-side SSL and required basic authentication or some other password scheme, then one might argue that a service can be executed 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, NSS security tools and default

```
<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 4 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.

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.

We note that this authentication-borrowing differs 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. In our scenario, the user's browser silently authenticates to any site of the adversarial site's choosing.

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).

Browser configurations

The answer to our question for this experiment was: ''Yes, with most standard out-of-the-box configurations, we can use the key without the user's permission.'' The seemingly natural defense to such attacks is to properly configure the browser to avoid them, and indeed actions such as disabling Javascript help.

Experiment 3: malicious use of keys via API attacks

IE on Windows is by far the dominant client platform. In trying to establish such a proper browser configuration for IE, we noticed that IE would only prompt for a password (on our high-security key) once per visit to a particular domain. Specifically, we would visit site A, perform a client-side authentication which prompted us for the password, leave site A, and then return—*only we were never prompted for the key's password again*. Furthermore, we could not find any browser configuration which would enforce this behavior (even the DoD guidelines leave browsers susceptible DoD PKI Class 3 Certificates, 2003), and we eventually discovered that Microsoft considers the advertised behavior to be a bug (Microsoft Knowledge Base Article—821574).

The inability to configure our browser so that the advertised behavior of a high-security key (which reads ''Request my permission with a password when this item is to be used'') led us to believe that the flaw must be at a lower level. So we began our third experiment with the question: ''Can we use some of our previous techniques such as API hijacking to understand what is happening and then to use the key?''

The default CSP is broken

The first step was to convince ourselves that IE was really using our high-security key to perform client-side authentication without requesting our permission, and watching network traffic with a sniffer confirmed our suspicion. We then attempted to reproduce the behavior we observed. Using API hijacking, we were able to attach a debugger to IE and watch the parameters it passes to the

CryptoAPI. Reverse engineering in this way allowed us to build a standalone executable which made the same sequence of calls to the CryptoAPI as IE does and uses the same parameters.

Our program opens the same keystore IE uses during a `CryptAquireContext`. Our code sits in an infinite loop taking a line of input from the command line. It then mimics the sequence of calls that IE makes to the `CryptoAPI` in order to get data signed: `CryptCreateHash`, `CryptHashData`, and `CryptSignHash`. Since our key is a high security, the first call to `CryptSignHash` prompts for a password, as expected. However, no subsequent calls prompt for a password, even if the data are completely different. Thus, the CSP is failing to "request my permission with a password when this item is to be used".

The magic button

In all of our explorations of the various IE configuration options, we came across one button (in "Internet Options") labeled "Clear SSL State". API hijacking showed that this button will make IE call `CryptRelaseContext` in the CryptoAPI, resulting in a password prompt the next time the key is used. We also found that restarting the browser will result in a prompt the next time the key is used (in contrast, we were initially surprised that restarting the Web server did *not* result in another prompt).

These are more extreme measures than simply configuring the browser to behave reasonably, but they were the best we could find, and are recommended by Microsoft Knowledge Base Article—290345.

Exploiting the CSP to get around the magic button

Armed with a little information as to how the CSP and IE work, we were curious to see if there was a way to defeat the magic button and browser restarts. Our goal was to make a program—running only with user privileges—which waits for IE to prompt the user to arm his high-security key with the password, and then use the key to sign arbitrary messages—even after IE has been closed, or SSL state has been cleared.

Using our previous technique of API hijacking, we reused the grappling hook from the attack in Experiment 1. However, the parasite is slightly different than the one mentioned in Experiment 1. In this attack, the parasite will spawn an *agent* process (called `iexplorer.exe`; the real IE is `iexplore.exe`) which communicates with the parasite over a named pipe. When IE goes away, the agent will persist and be able to use the key without prompting for the password.

The initial stages of the attack are identical to the one in Experiment 1. The attacker begins by getting the grappling hook and parasite on the victim's machine. Once the code is in place, the grappling hook begins executing which will get the parasite injected into IE address space. Upon injection, the parasite changes IE's IAT so that calls to desired functions in the CryptoAPI (advapi32.dll and crypt32.dll) are redirected to the parasite.

Once the parasite is set up, it watches for IE to make a specific sequence of calls: `CryptAcquireContext`, `CryptCreateHash`, `CryptSetHashParam`, followed by two calls to `CryptSignHash`. This sequence indicates that IE is using a private key for the first time, which will result in a password prompt. As each of these calls occur, the parasite intercepts the call before the CryptoAPI has a chance to handle it, packs all of the arguments into a binary data structure, and passes them to the agent over the named pipe. The idea here is that the agent is mirroring the exact sequence of calls (with the exact arguments) that IE is making.

Of particular interest is the second call to `CryptSignHash` as this is the call that spawns the password prompt. When this call occurs, the parasite does not actually let the call pass through to the CryptoAPI; it has the agent sign the data instead. The result is that the agent is the program which is requesting the user password, so it may use the key indefinitely—*with no further password prompts*. IE gets the correct signature, so the SSL handshake continues as normal. Examining the "Details" of the signing operation show that "iexplorer.exe" is using the private key instead of "iexplore.exe"; this subtle name change is the only means of detection.

At this point, the user can close IE and the agent still has an "armed" key, which it can use indefinitely. Our example agent puts itself into command line mode, allowing us to sign and decrypt arbitrary messages with the victim's key. A real attack would most likely have the agent act as a Trojan where it binds to a port and awaits remote signing and decryption commands.

The punchline: no configuration prevents this attack

We were unable to find any browser configuration which stops this attack because the problem is below the browser—it is with the CSP. The attack is possible because the system is designed with the assumption that the entire system is trusted. If one small malicious program with user privileges (such as can happen by a user clicking on an unknown attachment) finds its way into the system, the security can be undermined—even with high-security non-exportable keys, and even assuming everyone does the right thing, no matter how awkward: browser users clear SSL state or kill the browser after each session, and server application writers use forms with hidden nonces.

Experiment 4: malicious use of keys on a USB token

Many in the field suggest getting the private key out of the system altogether and placing it in a separate secure device of some sort. Taking the key to a specialty device (such as an inexpensive USB token) would seem to reduce the likelihood of key theft as well as shrink the amount of software which has to be trusted in order for the system to be secure. Specifically, at first glance, it would appear that the just the device and the software which provides access to the device (i.e., its CSP) need to be trusted.

We had a couple of these devices (the Aladdin eToken and the Spyrus Rosetta USB token), so we decided to have a critical look at these. Since the keys on the devices we had were non-exportable, key theft seemed impossible (assuming we leave ''rubber hose cryptanalysis'' and hardware attacks out of our attack model), but we wondered if we could use the key as in the previous attacks.

Spyrus Rosetta USB

The Spyrus CSP was the most verbose one in all of our experiments. It was the only CSP which prompted every time the key is used. In our opinion, these devices work too well—it was not uncommon to get multiple password prompts while loading one page. (This suggests a further line of inquiry: why does the actual usage of a client private key in such a session depart so radically from the user's perception of it? That is: visiting a site ''once'' should generate one warning, not an endless barrage.)

While this is the CSP which allows users to render the best mental model, the model it renders is not a particularly usable one. Our hypothesis is that we could simply ask for the password outright, and would probably get it because users are so trained to enter their password for this device. We could probably just hide in the noise.

Upon contacting Spyrus about the issue, they pointed us to the ''Spyrus Rosetta CSI library''. The library supports a ''Policy Console'' feature which allows the CSP to use the key for a user-specified time interval without asking for permission. This feature clearly enhances the usability of the device.

Aladdin eToken

The Aladdin eToken did not give us any option as to how often we wanted to be prompted for a password, and experiments showed that the Aladdin CSP's default behavior seems to follow a policy of one password authentication per application. This is virtually the same behavior we saw with the default Microsoft CSP and the high-security key. In fact, it is a bit worse than the default CSP in that the ''Clear SSL State'' button has no effect on the token whatsoever. Within a few minutes, we were able to replicate the attack in Experiment 3, allowing us to use the key even after the intended application (e.g. IE) has been shut down.

Upon discussing the issue with Aladdin, they suggest setting a ''Secondary Authentication'' to prompt upon application request. This does indeed prompt for authentication each time the key is used. As discussed previously, the result is possibly multiple password prompts per page load.

The trust boundaries do not shrink

Unfortunately, just putting the private key on a token isn't enough. The token's CSP is still interacting with the whole system (the OS and CryptoAPI), and the entire system still has to be trusted. Putting the private key on a token gives some physical security and makes it harder to steal the key (physical violence notwithstanding), but it doesn't protect against malicious use, and it doesn't increase usability.

Conclusions

For client-side PKI to be usable, it must behave as expected—it must only allow transactions which the client is aware of and approved. If we trust the entire desktop, and users ''clear SSL state'' or kill their browsers after each session, and application writers include and verify hidden nonces, then we might conclude that client-side PKI works. But these are not reasonable assumptions—and as we've demonstrated, relaxing them even a little yields security trouble.

Usability

It should be easy for a browser user to perceive and approve of the use of their private key; it should be easy for an application writer to build on this. To cite just a few design principles: (Yee, User interaction design for secure systems)

- "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.

In order for client-side PKI to work, these principles should apply to both the client user, as well as the IT staffer setting up a Web page.

The minimum trust boundary

Clearly, we would like to find the minimum number of components which have to be trusted, as this shrinks the number of potential targets. How can we shrink the trust boundary so that buggy desktops which have almost weekly "Critical Security Updates" are not the cornerstone of our secure systems? Trusting just the kernel doesn't solve the problem. Trusting a separate cryptographic token doesn't solve the problem.

We do not have a clear answer yet, but we discuss some of our initial thoughts.

Trusted paths

One natural area for further attention is a *trusted path*. Our earlier work (Ye and Smith, 2002) 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 specification and require that a user type a value into a `file` input tag. (Without this feature, malicious servers can provide content that quietly uploads a file of their choosing.)[3] Wouldn't an `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. Indeed, in the online literature, we see that earlier versions of Netscape provided a *signed form* facility in Java that forces some user involvement (Netscape form signing, 1999) (this feature is gone in current versions of Netscape, and IE never provided native support); we also see some brief discussion for authentication tags (Lawrence and Leach, 1999).

Until then, ongoing work (Pinkas and Sander, 2002) 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 provides some descriptive freetext might help. Instead of "hostname wants you to authenticate," the browser window might give some context, e.g., "...in order to change your class registration—are you sure?". (Netscape's Signed Forms went in this direction, but it permitted the server to provide HTML content that can enable some types of signature spoofing.)

To rephrase a point from our earlier work (Ye and Smith, 2002), the community insists on strict access controls protecting the client file system from server content, but neglects access controls protecting the user's perception of the client user interface.

Tokens with UI

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.

Many researchers have long advocated that private keys are too important to be left exposed on a general-purpose desktop. We concur. However, in light of our experiments, we might go further and assert that the user interface governing the use of the private key is too important to be left on the desktop—and too important to be left to the sole determination of the server

[3] Actually, some versions of Safari on OSX appear to be susceptible; we plan further tests.

programmer, through a content language not designed to resist spoofing.

Rethinking

Our experiments show that the natural mental model which arises for client-side PKI is not representative of the actual system's behavior. This fact, coupled with the underlying assumption that all of the system's components are trusted, creates opportunities for a number of devastating attacks. 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 PKI paradigm. We humbly suggest that some of this investment might be better spent rethinking the basic model.

Acknowledgments

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

The authors have also 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.

References

Anderson R. Why cryptosystems fail. Communications of the ACM November 1994;32–40.

Berkeley DB, <http://www.sleepycat.com>.

Buyer's guide—Web portal security solution, <http://www.entrust.com/resources/pdf/buyers_guide.pdf> November 2001.

Felten E, Balfanz D, Dean D, Wallach D. Web spoofing: an Internet con game. In: 20th National information systems security conference; 1997.

Fu K, Sit E, Smith K, Feamster N. Dos and don'ts of client authentication on the Web. USENIX Security 2001.

Gutmann P. 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? <www.cs.auckland.ac.nz/~pgut001/pubs/breakms.txt>.

Hamilton J. CGI programming 101, <www.CGI101.com> 1999.

Henson S. Netscape certificate database info, <http://www.drh-consultancy.demon.co.uk/cert7.html>.

Henson S. Netscape key database format, <http://www.drh-consultancy.demon.co.uk/key3.html>.

Huy P, Lewis G, Liu M. Beyond the black box: a case study in c to java conversion and product extensibility. Technical report, Carnegie Mellon Software Engineering Institute; 2001.

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>.

Information about the clear SSL state option in Windows XP, <http://support.microsoft.com/?kbid=290345>. Microsoft Knowledge Base Article—290345.

Installing DoD PKI Class 3 Certificates into Windows 2000 for use with Microsoft products; May 2003.

Kain K, Smith SW, Asokan R. Digital signatures and electronic documents: a cautionary tale. In: Advanced communications and multimedia security. Kluwer Academic Publishers; 2002.

Lawrence S, Leach P. User agent authentication forms, <http://www.w3.org/TR/NOTE-authentform>; February 1999.

Microsoft .Net Passport, <http://www.passport.net/Consumer/default.asp>.

Microsoft authenticode developer certificates, <http://www.thawte.com/getinfo/products/devel/msauthenticode.html>.

Mozilla . NSS Security Tools, <http://www.mozilla.org>.

Netscape form signing, <http://developer.netscape.com/tech/security/formsign/formsign.html>; 1999.

Netscape Communications Corp . Command-line tools guide: netscape certificate management system. 4.5 ed. October 2001.

Niederst J. Web design in a nutshell (2/E). O'Reilly; 2001.

Offline NT password & registry editor, <http://home.eunet.no/~pnordahl/ntpasswd>.

OS Security, Inc. . Round one: "DLL Proxy" attack easily hijacks SSL from Internet Explorer, <http://www.securityfocus.com/archive/1/353203/2004-02-09/2004-02-15/2>.

Personal certificates, <http://www.thawte.com/html/COMMUNITY/personal/>.

Pietrek M. Under The Hood. Microsoft Systems Journal February 2000.

Pinkas B, Sander T. Securing passwords against dictionary attacks. ACM Computer and Communications Security 2002.

Preventing HTML form tampering, <http://advosys.ca/papers/form-tampering.html>; August 2001.

Pubcookie: open-source Software for intra-institutional Web authentication, <http://www.washington.edu/pubcookie/>.

Stein L, Stewart J. The world wide web security FAQ, <http://www.w3.org/Security/Faq/>; February 2002.

Unpatched IE security holes, <http://www.pivx.com/larholm/unpatched/>.

Whitten A, Tygar JD. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. USENIX Security 1999.

Windows prompts you for your password multiple times when you use outlook if strong private key protection is set to high, <http://support.microsoft.com/?kbid=821574>. Microsoft Knowledge Base Article—821574.

Ye E, Smith SW. Trusted paths for browsers. USENIX Security 2002.

Ye E, Yuan Y, Smith SW. Web spoofing revisited: SSL and beyond. Technical report TR2002-417, Department of Computer Science, Dartmouth College; 2002.

Yee K-P. User interaction design for secure systems, <http://zesty.ca/sid/uidss-may-28.pdf>.

John Marchesini is currently a Ph.D. candidate in the Computer Science Department at Dartmouth College. His advisor is Sean Smith, and his research interests are security, distributed systems, and PKI. Before going to Dartmouth, he worked as a software developer for the BindView Corporation and earned a B.S. in Computer Science from the University of Houston (Summa Cum Laude).

Sean Smith is currently on the faculty of the Department of Computer Science at Dartmouth College, and serves as director of the Cyber Security and Trust Research Center at Dartmouth's Institute for Security Technology Studies. His current research focuses on how to build trustworthy systems in the real world. He previously worked as a scientist at IBM T.J. Watson Research Center, doing secure coprocessor design, implementation and validation; and at Los Alamos National Laboratory, doing security designs and analyses for a wide range of public-sector clients. Dr. Smith was educated at Princeton (B.A., Math) and CMU (M.S., Ph.D., Computer Science).

Meiyuan Zhao is currently a Ph.D. candidate in the Computer Science Department at Dartmouth College. Her research interests include security, PKI systems, distributed systems, routing protocols, and system simulation. Before going to Dartmouth College, she received her B.E. degree in computer science and engineering from University of Electronic Science and Technology of China.