Chapter 1

Strongly authenticated URLs: Integrating of Web browsers and applications with strong authentication

Eddy Cheung, Andrew Goodchild, Hoylen Sue and Ben Fowler
Distributed Systems Technology Centre[†]
The University of Queensland, St. Lucia
Brisbane, Queensland, Australia 4072
{*cheung, andrewg, hoylen, bfowler*}@dstc.edu.au

Abstract

With the growing popularity of the Internet, the Web browser is emerging as perhaps the most widely used type of user interface. Many desktop applications made use of a Web browser as a key component in their overall system design. The most common approach to integrate a Web browser into an application system is to open a Web browser as an external process with a URL. Alternatively, some Web browsers are made available as library components that can be integrated as part of the custom application. While a custom application may maintain its own security model and mechanisms, it is rarely well integrated with a Web browser's security mechanism. As a result, when an application invokes a Web browser which is accessing a secure site, the user may be required to authenticate multiple times. This need for multiple authentication, can lead to a reduction in the system's overall usability. While there are a few standard techniques, such as unprotected private keys in the browser, security through obscurity, cookie hacking, browser plug-ins and authentication agents, none

[†]The work reported in this paper has been funded in part by the Co-operative Research Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Industry, Science & Resources).

of them are without their own problems. This paper describes a new simple method, which is usable and maintains strong authentication when utilizing a Web browser from a desktop application.

Keywords: Authentication, Usability, Integration, Web application design.

1.1 Introduction

The Web has significantly changed how information is delivered and how applications are developed. The Web browser has become the user interface most familiar to users, and creating Web content is easy compared to traditional programming. This has lead to an increased in the use of the Web browser as a key component in desktop applications.

Many standalone desktop applications have already made use of Web browsers as a way to interact with users. The incorporation of a Web browser into an application design has many benefits. Reports and documents can be generated as HTML pages, which are then displayed in the browser. The HTML format offers many rich display features that are relatively easy to work with. It is much easier to generate a HTML document than it is to create a proprietary format. Information presented in HTML format can also be viewed both locally as well as exported and published on the Web in a vendor and platform independent way. Existing third party documenting and publishing tools can also be used. Furthermore, desktop applications can immediately gain access to remote data and interact with a Web server. By using the Web browser as a component in the application's overall design, time, money and developer resources are saved.

Typically, a Web browser can be incorporated into the application in one of two ways. The first approach is to incorporate a Web browser as a library component within the desktop application. This is commonly done on the Microsoft Windows platform, where Microsoft's Internet Explorer is easily available to developers as a COM component or commonly known as the WebBrowser control [1]. This practice is also popular on UNIX based platform where HTML rendering engines like the Mozilla Project's Gecko component can be integrated into desktop applications [2]. The other approach, perhaps more common and easier approach, is to have the desktop application that starts a Web browser with a given URL. For example, a desktop application can provide an on-line registration feature by directing users to their Web site. Updates and live information can be provided in a similar manner by the application. Thus, the boundaries between a standalone custom application and the Web become blurred.

For applications designed to handle sensitive information, the overall security of the system design must be carefully evaluated. Although desktop applications can be made very secure, there are considerable security risks when information crosses between the desktop application and the Web browser.

This paper examines the security risks associated when desktop applications use a Web

browser as a component as part of a system designed to access remote data and presents a solution to the problems identified. Section 2 describes scenarios where integrating a Web browser with a custom application is useful, and describes their benefits and challenges. Section 3 describes earlier approaches traditionally used and why they are unsuitable. We propose a new method to overcome these limitations in section 4. The assumptions and limitations of this new method are described in section 5.

Problem description

Most Web browsers and clients use the Secure Socket Layer (SSL) [3] protocol to secure their communication with remote services. SSL's robust protocol design and ability to support strong authentication using public key-based technology makes it an ideal security mechanism. When correctly configured, it allows service providers and users to mutually authenticate each other and establishes a secure encrypted communication channel. It ensures both the confidentiality and integrity of the information transmitted between the Web browser and the Web server. For mutual authentication using public key technology, both parties must hold a public and private key pair. Each party must also recognise and trust the digital certificate presented by the engaging party. To successfully establish a SSL session, each party must prove it holds the corresponding private key of the certified public key by using it to decrypt the message during the key exchange of the SSL handshake. Thus, it is important to securely store the private key. Typically, this is done by storing the private key protected by a passphrase/PIN either on the user's machine or within a secure token, such as a smart card. However, this has the unfortunate side effect that the intrusiveness of the passphrase or PIN dialog prompt to unlock the private key may impact on the overall system's usability. It also reduces the management of the authentication information problem back to traditional password and PIN management problems.

This usability problem becomes significantly apparent when two or more applications such as the desktop application and the Web browser require public key technology. As the Web browser's security model and mechanisms are not integrated with the desktop application, each will attempt to retrieve and use the private key independently. As a result, users can be prompted a passphrase/PIN every time the desktop application requests the Web browser to securely engage with the remote Web server. For example, a mobile bank loan agent may need to access a customer's account information during a consultation. Using a Web browser, the bank's financial application will open a particular URL to the bank to access the customer's information. To ensure that only valid users can access the information, the bank's Web server is secured using SSL with mutual authentication. Thus, the agent must present the passphrase to unlock the stored private key during the SSL handshake. Subsequently, when the agent wishes to submit a loan request on behalf of the client, the agent must use the private key to sign the request before it is submitted electronically to the bank. As the financial application cannot fully integrate with the browser's security, it can only use another copy of the private key stored by the application. As a result, the bank's

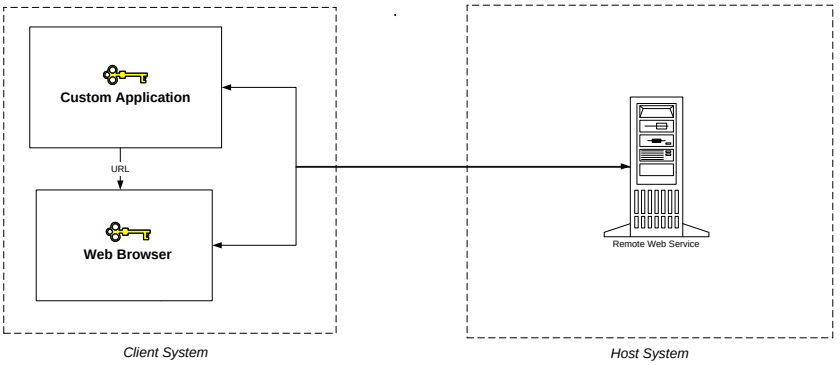

Figure 1.1: Logical view of the custom application and Web browser accessing the remote service

agent must present the user with another passphrase dialog prompt. The intrusiveness of the passphrase dialog can significantly decrease the overall system's usability.

Examples like the one described in figure 1.1 above are becoming increasingly common. Web enabled client applications such as financial, banking, healthcare, enterprise application integrations of legacy application, or even point-of-sale terminals are all examples of applications where both usability and security are important to both users and service providers. If a system is not user-friendly, the application may be rejected by its users, or face failure in the marketplace. As a consequence, developers are forced into a balancing act between usability and security.

1.2 Current approaches

There are currently several approaches for addressing the balance of security and usability in developing Web enabled desktop applications. However, these methods are limited and fail to fully resolve the issue. These approaches and their associated limitations are outlined below.

Unprotected private key in Web browser

One of the popular approaches to maintain both usability and security is to store the private key unprotected within the Web browser. This places lighter demands on users to remember passphrases/PINs and reduces the inconvenience and user interface clutter of additional passphrase dialogs.

At first glance, this is an attractive approach, as the overall usability of the application is maintained while the communication between the user and the remote Web server is secured. However, this approach introduces a new vulnerability into the system. The private key

stored unprotected within the browser becomes vulnerable to unauthorised users and trojan horse. Moreover, the usability of this approach is also less than ideal. Users are still faced with a dialog prompt from the Web browser to select the correct key and certificate when attempting to establish a SSL session with the remote Web server.

Security through obscurity

This approach attempts to circumvent the security issue by replacing the need for client authentication with an obscured URL constructed by the desktop application and passed into the Web browser. Thus, effectively, the obscured URL becomes the authentication information, a shared secret between the client and the remote Web server.

While this approach seems practical, it weakens the overall security of the system. By using an obscured URL as a shared secret for authentication purpose, it can be easily captured and exploited. URLs are generally monitored, and logged for many different reasons. For example, URLs are captured and stored in the browser's history by default. Furthermore, many Internet service providers implement proxy servers to cache requested data. These proxy servers may be explicitly known or transparent to users. It is common for those proxy servers to capture and log URL requests for caching purposes. Thus, it is unsuitable to use URL as a shared secret for authentication purposes between the client system and the remote Web server. Moreover, this method is susceptible to a replay attack. Therefore, this approach is unsuitable if data privacy and strong authentication are required.

Cookie hacking

Some developers have taken an approach to manually modify the cookie kept by the Web browser to streamline authentication. Described in RFC 2965, cookies are a mechanism to create stateful sessions with HTTP requests and responses [4]. Cookies are textual information usually kept in files read by the Web browser. This text file can be easily deleted, modified or captured by an external program. Thus, a developer could externally modify the Web browser's cookie file to include authentication information for the remote Web server. The remote Web server can then subsequently retrieve the modified cookie containing the authentication information, relieving the need for the user to authenticate to the remote Web server.

Cookie hacking offers seamless integration, and maintains usability. However, like the previous mechanism, it introduces new vulnerabilities into the system because cookies are not designed to hold authentication information [4]. Such information is highly vulnerable to perusal and theft. Cookies are also susceptible to modification or removal by other programs such as "cookie cleaners" that are designed to remove stored cookies. Moreover, this approach has limited portability as the location of the cookie file can vary between browsers, platforms, vendors as well as browser's own user profiles. In addition, the problem of locat-

ing the correct cookie to modify in a computer where multiple browsers are installed can be an interesting challenge in itself.

Authentication agent

An authentication agent is an application that runs on the user's desktop or trusted device. It is designed to hold authentication information and authenticate to remote entities on the behalf of the user. The purpose of the authentication agent is to relieve the user of having to manage multiple sets of authentication credentials, as well as decreasing the intrusiveness of authentication dialog prompts. At the start of a session, the passphrase is entered once to unlock the protected authentication information. Subsequently, the agent uses the unlocked authentication information to authenticate to remote entities when necessary. It addresses both the overall desktop usability while maintaining a high level of security. Secure Shell's (SSH) authentication agent is a well-known example of such an application [5]. A similar approach can be used when security and usability is essential to both the desktop application and Web browser. However, the problem with this approach is that no browsers currently work with authentication agents and it is not likely to happen in the short term, as the interface to authentication agents has not been standardised.

Browser's Plugin

Another alternative solution to ensure both usability and security is maintained is to utilise the Web browser's plugin technology. By using the browser plugin architecture, the developer can have the opportunity to intercept and modify incoming and outgoing data to and from the Web browser. The browser plugin can then access the private key stored by the custom application to incorporate authentication information into the request. As a result, the user does not need to input the passphrase/PIN multiple times. The Minotaur Project [6] is one such example of ensuring both usability and security are maintained when accessing information remotely. However, like previous approaches, a significant effort is required in setting up a Kerberos Key Distribution Center (KDC) and for porting and maintaining the browser plugin for the different vendor's Web browsers, browser versions and operating systems.

Authenticated URLs

This paper presents an alternative approach, called authenticated URLs, for authentication when a Web browser is used by a desktop application. This method aims to maintain both strong authentication and usability.

With the authenticated URL method, the desktop application controls and manages its security model and mechanism. It is responsible for storing and managing its private key (or it may rely on an authentication agent to manage the key). The Web browser is not required to store another copy of the private key for authentication purposes. Like the obscured URL approach, this approach uses the URL for authentication purposes. When a user requires access to information located on the remote Web server securely via the Web browser, the desktop application constructs a URL that is recognised by the Web server. However, the URL constructed does not have an embedded shared secret between the user and the remote Web server.

For the purpose of this paper, the original URL constructed by the desktop application is referred to as the basic URL. To allow a URL to be used for authentication, the basic URL is appended with additional authentication information. The resulted URL is digitally signed before it is submitted to the remote Web server. Due to the asymmetric nature of the public key technology, the URL does not contain a shared secret that can be easily captured and exploited.

To ensure this approach is not susceptible to replay attacks, additional authentication parameters are included with the basic URL. It is important to recognise that digitally signing a URL request does not provide adequate security. Like the obscured URL approach, a digitally signed URL is also susceptible to replay attacks. Malicious users can simply capture the URL and re-transmit to the remote server and imitate the legitimate user. Thus, to prevent replay attacks, additional authentication parameters such as timestamps must be included with the basic URL before it is digitally signed. Other information such as user identifier is also required to enable the remote Web server to verify the digital signature should also be included. Ideally, the user's certificate should also be attached. However, due to the size of a typical digital certificate, for practical reason, only the issuer's distinguished names and the certificate's serial number are included. This assumes that the Web server can retrieve the user's certificate securely from another channel, which is acceptable in most situations.

This approach replaces the need by the Web browser to perform client authentication. It is designed to be used with server-side authenticated SSL for mutual authentication purpose. Once authenticated, subsequent data transmitted is secured by the SSL. The authenticated URL has the following format.

```
https://<URI>?timestamp=<timestamp value>&SignAlg=<signature
algorithm used>&CertIssuer=<Issuer's Distinguished Name>
&CertSerialNum=<certificate's serial number>
&Signature=<Signature value>
```

As illustrated in figure 1.2 below, the authenticated URL approach consists of the following steps:

1. The desktop application constructs the basic URL. For example:

   ```
   https://www.example.com/index.html
   ```

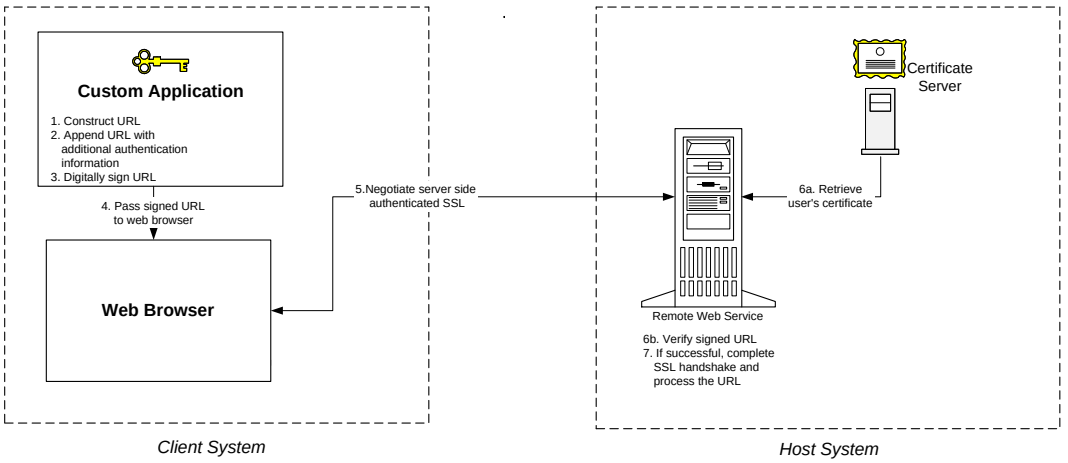

Figure 1.2: The authenticated URL approach

2. The basic URL is appended with additional authentication parameters: a timestamp, user identifier, and an identifier specifying the digital signature algorithm used. For example:

```
https://www.example.com/index.html?timestamp=10516788112
54&SignAlg=RSAwithSHA1&CertIssuer=CN%3Dme%2C+O%3DDSTC%2C+c%3
DAU&CertSerialNum=1021
```

3. The private key stored within the desktop custom application subsequently digitally signs the resulting URL. For example:

```
https://www.example.com/index.html?timestamp=10516788112
54&SignAlg=RSAwithSHA1&CertIssuer=CN%3Dme%2C+O%3DDSTC%2C+c%3
DAU&CertSerialNum=1021&Signature=lifoJ0ZRf%2BIfNB26LjuCqxke6
tS1TNGueoJ6znNqPwAGB6EHBuoU9DItFmULuVL6pY63FCY47eKhJoQNar%2B
jSRZiY6%2Be%2BFg92qtCSjuCOhS09BbosCH20DOzTiohh0R9VqFqk0oKqat
e9BOMh09gQybKy7xnO%2BRVSGSRNgAEnOA%3D
```

4. The authenticated URL is then passed onto the Web browser.

5. The Web browser negotiates a server side authenticated SSL session with the remote Web server and submits the URL request.

6. Before the SSL session is successfully established, the Web server authenticates the user using the information included in the URL.

   (a) Using the embedded user identifier, the Web server retrieves the user's certificate from the user directory and verifies the digital signature of the URL.

   (b) If the digital signature successfully verifies, the Web server checks that the timestamp provided lies within the acceptable timeframe.

7. Once the client is successfully authenticated, the Web server completes the SSL session establishment and processes the basic URL.

The advantage of the authenticated URL approach over the traditional approaches is the ability to maximise usability without compromising security. At the same time, it alleviates the need for a developer to customize a user's browser or restrict the user's operating environment. Although this approach requires the developer to customize the authentication module within the remote Web server, however, it is believed that the engineering effort required at the remote Web server is insignificant compared to the client side approaches.

1.3 Assumptions and Limitations

The authenticated URL approach places several assumptions on the users and the remote Web server. Firstly, it is assumed that the desktop application can securely protect the user's private key or it relies on another application for this task, such as an authentication agent. Secondly, it is assumed that the Web browser supports server side authenticated SSL and additionally recognises and trusts the certificate presented by the Web server. Thirdly, this approach assumes the time difference between the end-user's computer and the remote Web server is small, and within the acceptable limits to minimise the effectiveness of replay attacks (although this could be alleviated by using NTP [7]). It is assumed that the time required for the URL request to travel to the remote Web server is within the acceptable timeframe. Finally, the checking of the digitally signed URL assumes the Web server has access to the legitimate certificate or public key, and that they are secured from tampering from unauthorised users.

Since the nature of the solution requires modification to the Web server to recognise authentication information embedded in the URL, it is envisaged that this approach is only suitable to an environment that is willing to accept these assumptions.

A potentially major limitation identified with the authenticated URL method is the issue of client re-authentication. Once the SSL session has terminated or expired, the client will need to re-authenticate itself to the server. Since the authenticator is constructed outside of the web browser, the user must therefore re-authenticate through the desktop application. This could potentially reduce the overall usability, especially when SSL session has a short seession time.

Future Work

Future work on the authenticated URL focuses on challenging the assumptions identified in the previous section. In particular, while the risk of replay attacks may be small, however,

malicious attacks can still exploit the vulnerability with some significant effort. Moreover, significant time difference between the user and the remote Web server can result in the denial of service. The authenticated URL approach will be subject to a thorough security analysis and several prototypes will be developed in an attempt to validate our assumptions and discover issues impacting the usability, security and practicality of our approach.

Conclusion

The Web browser provides a media rich content interface that is familiar to most users. The ability to use a Web browser as a component in the development of a desktop application is attractive to developers. Integration of the Web browser in the overall system design can greatly reduce the time to market. However, traditionally, using Web browsers in this fashion has implied trade-offs between user convenience and security, and therefore, it has been restricted to applications that do not require a high level of security. The authenticated URL approach described in this paper attempts to address these problems. It is designed to provide strong authentication to the remote Web server when Web browser is incorporated into an application while maintaining the overall application's usability. It avoids the significant customisation and integration effort required by the end-user side of the application. The approach is designed to be generic, portable and vendor independent.

Bibliography

[1] Edwards, M., and Roberts, S. (1998) *Reusing Internet Explorer and the Web-Browser Controls: An Array of Options. Microsoft Corporation. MSDN Library*. http://msdn.microsoft.com/library/default.asp?URL=/library/ens/dnWebgen/html/reusebovw.asp. Visited 10 May 2003.

[2] Mozilla (1998) *Mozilla Layout Engine*. The Mozilla Organization. http://www.mozilla.org/newlayout. Visited 10 May 2003.

[3] Dierks, T., Allen, C. (1999) *The TLS Protocol Version 1.0*. Network Working Group RFC 2246.

[4] Kristol, D., Montullu, L. (2000) *HTTP State Management Mechanism*. Network Working Group. RFC 2965.

[5] Ylonen, T. (1995) *Secure Shell Agent man page*.

[6] Project Minotaur (1997) *Computing Services Carnegie Mellon University*. http://asg.Web.cmu.edu/minotsaur/index.html. Visited 10 May 2003.

[7] Mills, D. (1992) *Network Time Protocol (Version 3) Specification, Implementation and Analysis*. Network Working Group RFC 1305.