CS Senior Honors Thesis:

Visual Validation of SSL Certificates in the Mozilla Browser using Hash Images

Hongxian Evelyn Tay
het@andrew.cmu.edu

School of Computer Science
Carnegie Mellon University

Advisor: Professor Adrian Perrig

Electrical & Computer Engineering
Engineering & Public Policy
School of Computer Science
Carnegie Mellon University

Monday, May 03, 2004

Abstract

Many internet transactions nowadays require some form of authentication from the server for security purposes. Most browsers are presented with a certificate coming from the other end of the connection, which is then validated against root certificates installed in the browser, thus establishing the server identity in a secure connection. However, an adversary can install his own root certificate in the browser and fool the client into thinking that he is connected to the correct server. Unless the client checks the certificate public key or fingerprint, he would never know if he is connected to a malicious server. These alphanumeric strings are hard to read and verify against, so most people do not take extra precautions to check.

My thesis is to implement an additional process in server authentication on a browser, using human recognizable images. The process, Hash Visualization, produces unique images that are easily distinguishable and validated. Using a hash algorithm, a unique image is generated using the fingerprint of the certificate. Images are easily recognizable and the user can identify the unique image normally seen during a secure AND accurate connection. By making a visual comparison, the origin of the root certificate is known.

1. Introduction: The Problem

1.1 SSL Security

The SSL (Secure Sockets Layer) Protocol has improved the state of web security in many Internet transactions, but its complexity and neglect of human factors has exposed several loopholes in security systems that use it. Most of the meta-data in security systems, such as certificate public keys are in seemingly meaningless strings that do not convey much of the state of security. Coupled with the inherent weakness of humans in validating strings, the ability to ascertain the trust level of a server can be easily compromised.

Communication over the Internet is prone to all sorts of vulnerabilities including eavesdropping, spoofing and tampering. The SSL protocol is designed to provide for authentication and encrypted communication between clients and servers. It is unfortunately, very complicated. Server authentication, whereby the server presents a CA (certificate authority) signed public key-certificate during an SSL handshake, is carried out by the Web browser. The human user rarely gets a clear picture (unintended pun) of what goes on beneath the authentication phase, what piece of information is getting validated, or whether it is done correctly. Some results of the process have to be conveyed to the end-user.

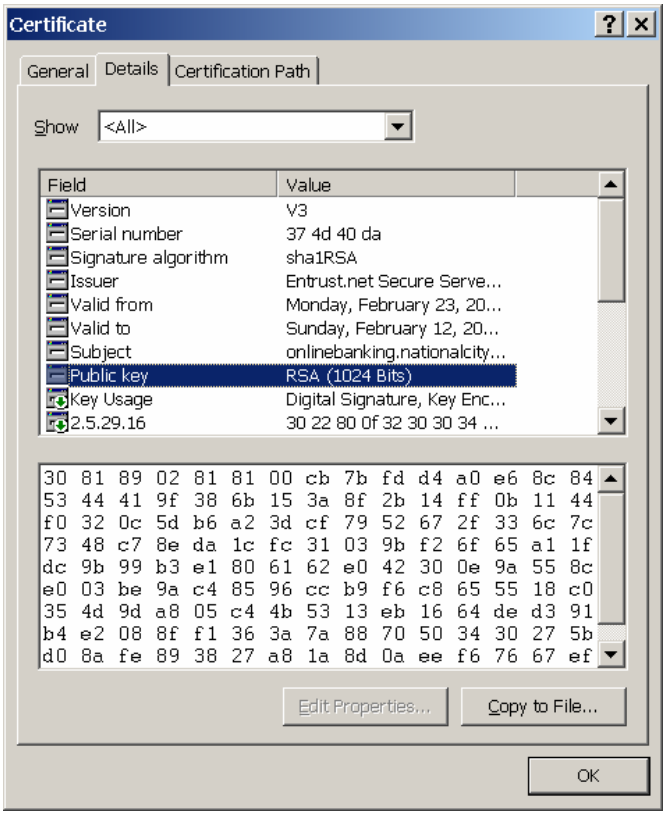
The most convenient method (but not infallible) for Netscape, Mozilla, and IE users is to click on the lock icon that shows up in the status bar of the browser to view the server certificate that comes through when a SSL session is set up. The user can then see various certificate information, such as the issuer (CA), or the fingerprint, presented in human readable form, and can then ascertain if the server can be trusted (see Figures 1 and 2 next page). However, most of this information can be forged or tampered with, with the exception of the signature, perhaps. What is shown on the browser might not guarantee that the session is secure. For example, it is easy to change the lock icon in Mozilla that shows up during a perceived SSL authentication phase. Thus, a user who relies on the icon for indication of a secured connection might be deceived if an adversary were to change the browser code to show a locked icon (instead of a broken one) even when authentication fails. This can be easily done by manipulating the browser internal CSS and javascript code, as described in a Section 4 of this paper. Furthermore, the certificate can be created by an adversary, and then implanted into the certificate database of the client computer, so we need to verify it against a reliable source.

These scenarios are some examples that can create illusions of SSL sessions, and are not easy to detect. Since human factors play a huge role in the real world security system, it is possible to take advantage of the user's mindset to create an impression of security. For all the privacy the SSL protocol provides, it does not prevent the end-user from authenticating and connecting to a fraud, albeit securely.

Prior work carried out by Adrian Perrig, Dawn Song [2], and Rachna Dhamija [1] investigated the idea or using images for authentication purposes. Images are far easier to distinguish than alphanumeric strings, and image based recognition as a psychological factor is effective for establishing identity.

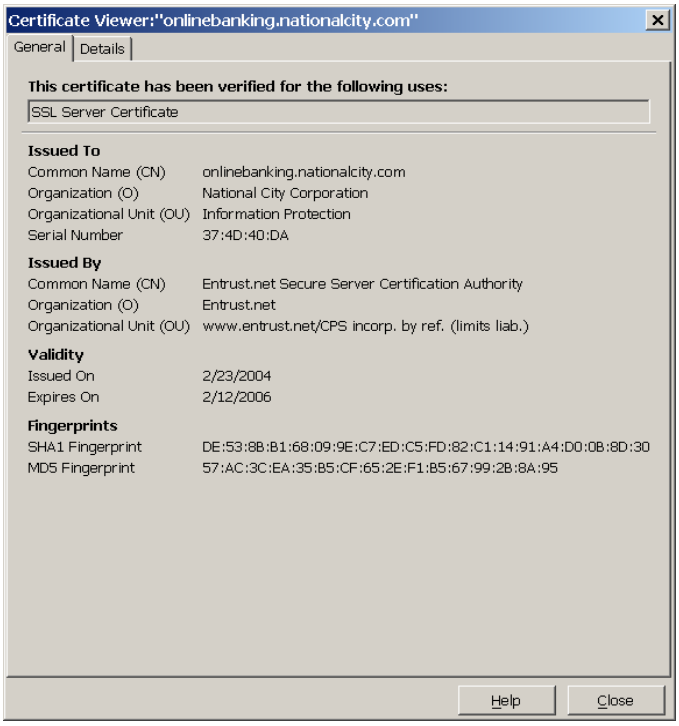
In this paper, I will discuss about my experience in designing, implementing, and evaluating Hash Visualization for the browser SSL session in the real world server authentication. In the next section, I will talk about server authentication and certificates.

(a) "General" information

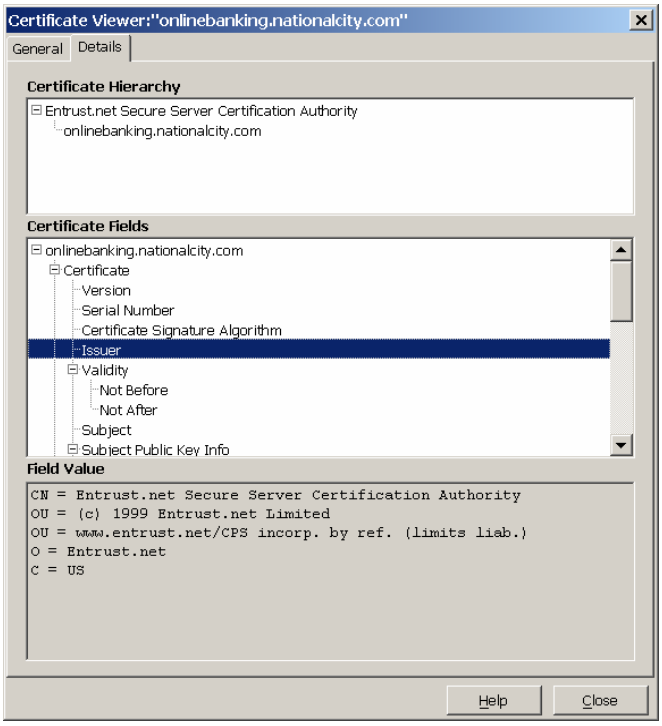


(b) "Details"

Figure 1: Certificate in presented GUI (Internet Explorer v.6)



(a) "General" information



(b) "Details"

Figure 2: Certificate in presented GUI (Mozilla 1.6b)

2. Background

2.1 SSL Handshake & Man in the middle attacks

An SSL session always begins with an exchange of messages called the SSL handshake, which allows the server to authenticate itself to the client using PKI (public key infrastructure) techniques, and also for the creation and exchange of symmetric keys used for rapid encryption, decryption, and tamper detection during the session that follows. A detailed description of the handshake can be found on the Netscape website [8]. As part of the protocol, a certificate has to be presented by the server during server authentication. This is usually the case in many web sites where clients log into the secure server of a bank, for example.

Before the SSL session is established, however, communication is open to a possible "man in the middle" attack. The latter is a rogue program that intercepts keys passed back and forth during the SSL handshake, substitutes its own, and makes it appear to the client that it is the server, and to the server that it is the client. Therefore, the encrypted information exchanged at the beginning of the SSL handshake is actually encrypted with the rogue program's public key or private key, rather than the client's or server's real keys, allowing it to read all the data exchanged without being detected. It is useful to further establish the identity of the server certificate. One way is to display its unique meta-data in a recognizable image.

2.2. Certificates & Certification Systems

A certificate is an electronic document containing a public-key value and information that unambiguously identifies the certificate subject, that is, the entity that holds the matching private key. The certificate is digitally signed by the CA's signing private key, whose corresponding public key is distributed to anyone who cares to keep a copy of it. If that CA is a trusted source, its public key might be stored in a web browser's key store or its certificate stored in the database for validating other CA's certificates. For example, in the Mozilla browser, cert7.db stores root certificates recognized by the browser as well as those the user has decided to trust. These databases are password protected, and are vulnerable to tampering or theft.

Furthermore, the Certificate Database tools (certutil) allow for the creation of certificates. A skilled adversary could manufacture and plant his own certificate into the user's root key database and authenticate himself later on. The certificate validation process does not safeguard against that possibility, as shown in the next section.

Thus, one idea is to compare the hash image generated during a successful SSL handshake against that seen from another secure channel (e.g. the Internet, newspaper)

2.3 Certificate Validation

The Mozilla and Netscape browsers validate all certificates by checking their expiry dates, signatures, domain names (for SSL sessions) and chain of issuer certificates, until they come across a trusted root CA. They also check against Certificate Revocation Lists (CRLs). Usually, the user has no idea what root certificate is being used to validate the server certificate, unless he meticulously goes through all the root certificates in his browser database and identifies them one by one. The browser happily assumes all root keys in its database are from trusted sources. The problem is worsened if certificate revocation information is not available or not updated quickly enough to reflect the current status of certificates.

The most widely recognized standard public key certificate format is the X.509 standard and include the following fields: serial number, signature, issuer, validity, subject, subject public-key information, among others. All these may be tampered with or falsely generated, so the fingerprint, a unique number generated by applying an algorithm (eg. SHA, or RSA) to the contents of the certificate, is useful in verifying its integrity. Certificates and

their contents supported by Netscape and Mozilla, as well as many other browsers are organized according to the X.509 v3 specification.

The public key of the certificate is a human unfriendly DER-encoded string, which is long and hard to recognize even when converted to an ASCII string. In fact, X.509 certificates are not human readable at all, and the user cannot easily see what is being accepted during authentication. He has to trust that the GUI (graphical user interface) version of the certificate presented by the browser is correct and no information is thwarted without his knowledge.

The fingerprint is the hash of the whole certificate computed using one or more algorithms. In Mozilla, the algorithms used are SHA1 and MD5. It is not possible to modify the fingerprint without invalidating the signature.

2.4 Hash Visualization using Random Art

To generate images from strings, I used *Random Art* [6] developed by Andrej Bauer. The idea behind the algorithm is to use a binary string *s* as a seed for a random number generator, constructing a random expression tree which describes a function *F,* which in turn, defines an image. *F* maps each pixel *(x,y)* to a RGB (a set of red, green, blue intensities) value. As an illustration, the expression *F(x,y) = (x,x,x)* produces a horizontal gray shade. The grammar used in *Random Art* include sin, cos, square root, division, etc.

3. Implementation

I implemented hash visualization in the 1.6b branch of Mozilla, an open source browser.

3.1 Mozilla User Interface and Architecture

Mozilla has a user interface, called chrome, made up of XUL and CSS (Cascading Style Sheets) files. XUL is an XML-based user interface language that defines how each browser element should present itself. It is in a tree format, like html, and its code designs the GUI. CSS files are used to add to the definition of how the XUL element looks like. The chrome files are also collectively called a *skin.* For example, Mozilla ships with the Classic and Modern skin as default skin packages. The presentation of GUI elements is configurable and not hardwired in its application layer. The latter is written in C and C++, and its behavior, dictated by *XPCOM* [15] *objects* which are written using module-based programming, can be accessed by the chrome through browser internal javascript, the latter acting as a communication path or "glue" between the presentation and application layers.

The *XPCOM* code is compiled into dynamic link libraries (dll on windows) and accessed by the browser instance during run time. It also has a mechanism to detect and register new *XPCOM* objects. Most of the Mozilla code bas is written in *XPCOM*, which is how applications are hooked onto the main browser instance. *XPCOM* also serves as a wrapper for the underlying C code base. Certain *XPCOM* functions are made available as *Interfaces* (*IDL* files) so that application programmers who are not concerned with the internals of *XPCOM* object implementation can use them. These interfaces usually start with "*nsI*".

Mozilla extensions are stored in *jar* archives in the chrome directory of the Mozilla root. For example, some of the cryptographic features for the GUI (including the PSM, or Personal Security Manager, a security module built on top of NSS) reside in *pippki.jar* and *comm.jar*. To facilitate development, I uncompressed the necessary archives into its hierarchy of directories and files so that these files are easily modified. Also, I modified "*installed-chromed.txt*" to read from the uncompressed files (instead of the *jar* files) when the browser starts up. That is also when the browser detects and registers new components. Essentially, this is how Mozilla and

Netscape browsers have extensions installed, except that when downloaded over a network (see section 3.3) the installation script (*install.js*) takes care of all the manual work just mentioned.

3.2 Design Considerations

In essence, there are four main steps to look into in the implementation of hash visualization. Firstly, the image should be display appropriately and noticeably on the browser GUI. In my opinion, the *navigator toolbar* is a good place to display the hash image. Located beside the navigation buttons (*back, forward, reload,* and *stop*), it is minimally intrusive with the user's surfing experience, and yet positioned significantly near the center and top of the browser. The idea came from the throbber [a] element displaying an animated gif whenever the browser is in the midst of a web document request. It seems only natural that the XUL element containing the image should be a button because its useful attributes and behaviors can add to the functionality of the visual hash element. XUL elements in the mozilla chrome are assigned an id for easy accessibility by other elements. The id assigned to this button is *hashViz*.

The second step is to access the information on the SSL certificate received from the server. Then, I would use a unique piece of information from the certificate generate the image. Image generation is carried out using *Random Art* written in C. Finally, I would have to notify the browser object to display the hash image.

A special characteristic Mozilla has is tabbed browsing. Each tab shows different web documents. Thus, it is important that the *hashviz* shows the correct image (or none) between two or more tabbed documents that are encrypted. The "active" document currently being viewed may be different within the same browser GUI.

3.3 Considered Approaches

The plan was to write the hash visualization implementation as a Mozilla extension.

Initially, I attempted to compile the *Random Art* code (for image generation) as an *XPCOM* object using cygwin (a Unix wrapper for Windows), since I was using a Windows platform. However, that was unsuccessful because of compilation difference between the source code for *Random Art, ImageMagick* (see later section), and the *XPCOM* code. The source code for *ImageMagick* proved too bulky for it included a lot of their own modules, which were not required for the purposes of this project.

A simpler and more optimized solution is to separately compile the different programs and install them where the system can locate and execute them. The process is relatively fast and despite some time lag between generating the image and rendering it, at its slowest, the hash visualization process does not take longer than waiting for the reply of a high-traffic server. In the future, however, it would be better to modularize the code in order to make it platform independent and deployable through installation mechanism like distributing *XPI*s (mozilla XPInstall [14] zip files) over the web. Using *XPI*, the files can be packaged into *XPI* and then installed through the browser, by opening the *XPI* document like any normal file or through a web link. Mozilla recognizes the *XPI* and will be able to install it.

As for getting hold of certificate data, one possible solution is to trace the certificate all the way to the root issuer and run hash visualization on its one of its unique data. The Mozilla browser does this as it verifies the integrity of the certificate chain. However, this can be taken advantage of by the certificate subject. Considering that certificates can be bought, generated, or even stolen, even if the certificate were issued by a trusted CA, the subject might not be trustworthy. Thus, what is important is to reflect the state of the current SSL session. In other words, the user should be notified of the data from the server certificate.

Because hashes are designed to be as collision-free as possible, I decided to use the unique fingerprint hashes in mapping strings to images as an added safeguard for collision free image generation. Mozilla generates fingerprint

hashes by using the content of the certificate, its length, and two hashing algorithms: SHA1 and RSA (unlike Internet Explorer 6 that use only SHA1). Since the hash is already calculated during signature verification anyway, it is unlikely to affect the efficiency of image generation. I also tried using other certificate meta-data like the public keys of the certificates, but aesthetically speaking, the fingerprint hashes give the best results.

3.4 GUI Implementation

The button *hashViz* is added to the main browser XUL file by adding (overlaying) *navigatorOverlay.xul* with *hashVizOverlay.xul.* Overlays are XUL GUI files that contain pieces of GUI that is dynamically added to the existing GUI. Like some of the browser elements, it can be hidden by clicking on the toggle button under View-> Show/Hide. I added this feature for flexibility.

hashViz takes on two different attributes according to the security state of the web document. These attributes are specified in the CSS file *hashViz.css*:

```
#hashViz[displayCert="yes"] {

      list-style-image: url("chrome://hashViz/skin/display_small.jpeg");

}

#hashViz[displayCert="no"] {

      list-style-image: url("chrome://hashViz/skin/default.jpeg");

}
```

The image displayed on *hashViz* is updated by every time there is a security level change in the browser content. I used an *XPCOM* listener object (*nsIWebProgressListener*) that updates the image every time the browser receives notification that a certificate is present from a SSL session handshake. Behaviors such as this are written into browser-internal JavaScript code that is read by the XUL file where the button GUI code resides. I also modified the run time code in *nsBrowerStatusHandler.js* to change the attribute of *hashViz.* When there is a certificate to be display, hash Visualization is called.

If the web document request is of "https", the browser will report the certificate by displaying the lock icon on the status bar. The browser content will be encrypted, but Mozilla carries out a series of checks to discriminate between four states: `STATE_SECURE_LOW, STATE_SECURE_HIGH, STATE_IS_BROKEN,` and `STATE_IS_INSECURE`. The "`STATE_IS_BROKEN`" state occurs when the browser contains encrypted and unencrypted data at the same time. In any case, the certificate is presented and it is only reasonable to display a visual hash representation of its data to the user. It is then up to the user to decide if he wishes to carry on with browsing.

nsISSLStatus keeps track if there is a certificate object received. With the aid of methods available in *nsIX509Cert,* I was able to obtain its fingerprint hash if the certificate (*siteCert*) is not null, i.e., if there is a server certificate present coming from the other end of the connection. I then run the modified *Random Art* algorithm (that I installed in on the system path) on the fingerprint to obtain a PPM file. Since Mozilla is currently unable to render PPM image files, I used *ImageMagick*'s image conversion utilities [7] to convert the image to JPEG. A screen shot of the end result is shown in Figure 3 on the next page.

For a more detailed inspection of the image, the user can click on *hashViz* to bring up a larger version. A screen shot example of this is shown on Figure 4 on the page 9.

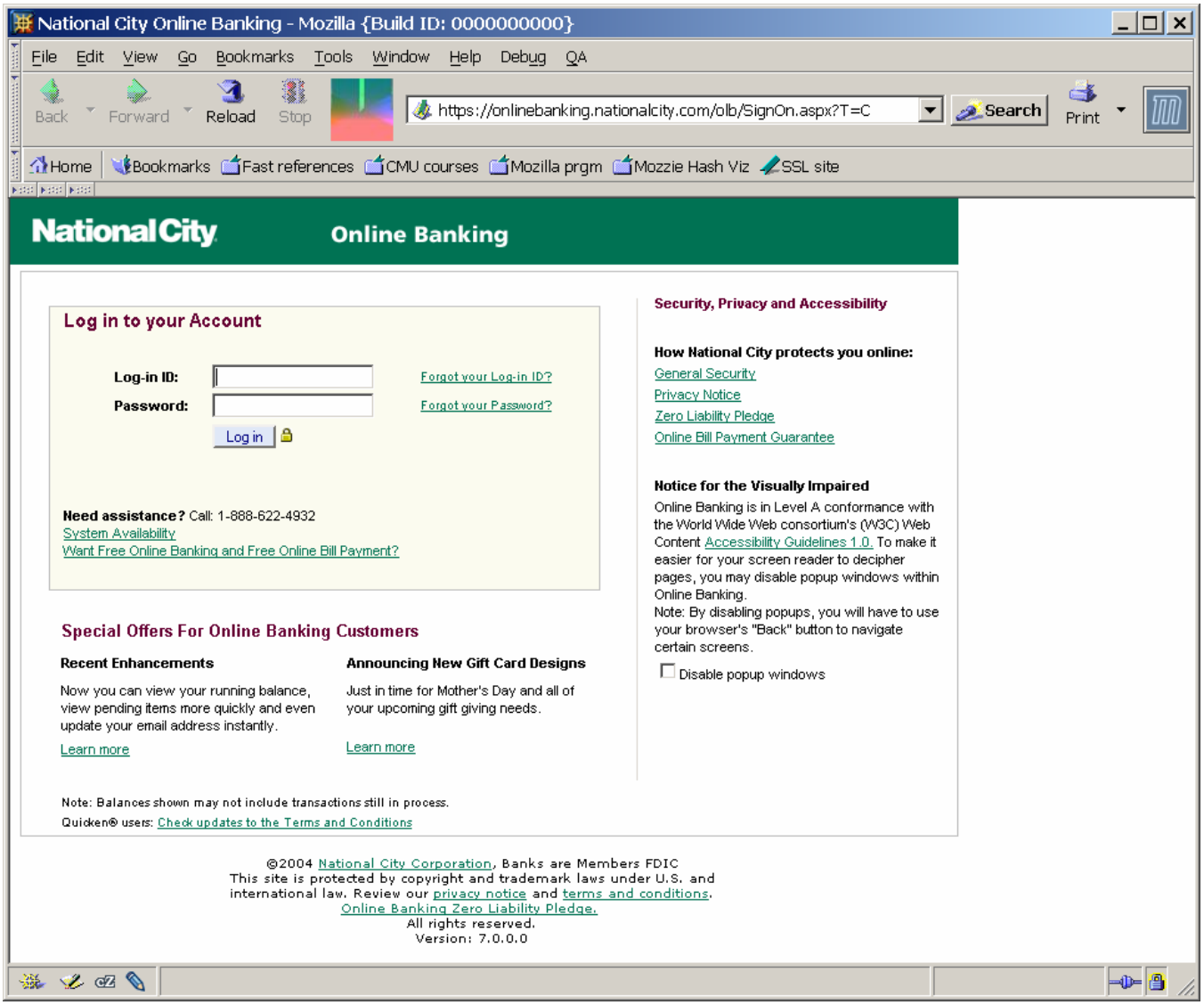


Figure 3: Mozilla browser shown with hashViz image (beside the url bar) on the navigator toolbar. The image is generated using the fingerprint calculated by MD5.

Two images (differing only in terms of dimension and resolution) are generated from the fingerprint and stored in a special folder installed in the chrome directory: one large (256x256 pixels) and one small (50x50 pixels). The small image is named `display_small.jpeg` and is displayed on *hashViz* button, while the larger one is displayed when the user clicks on *hashViz*. This works because only one certificate is reported at any point in time. If there is no certificate, no image is displayed. Alternatively, the user can add his favorite image (name as `default.jpeg`) to be displayed.

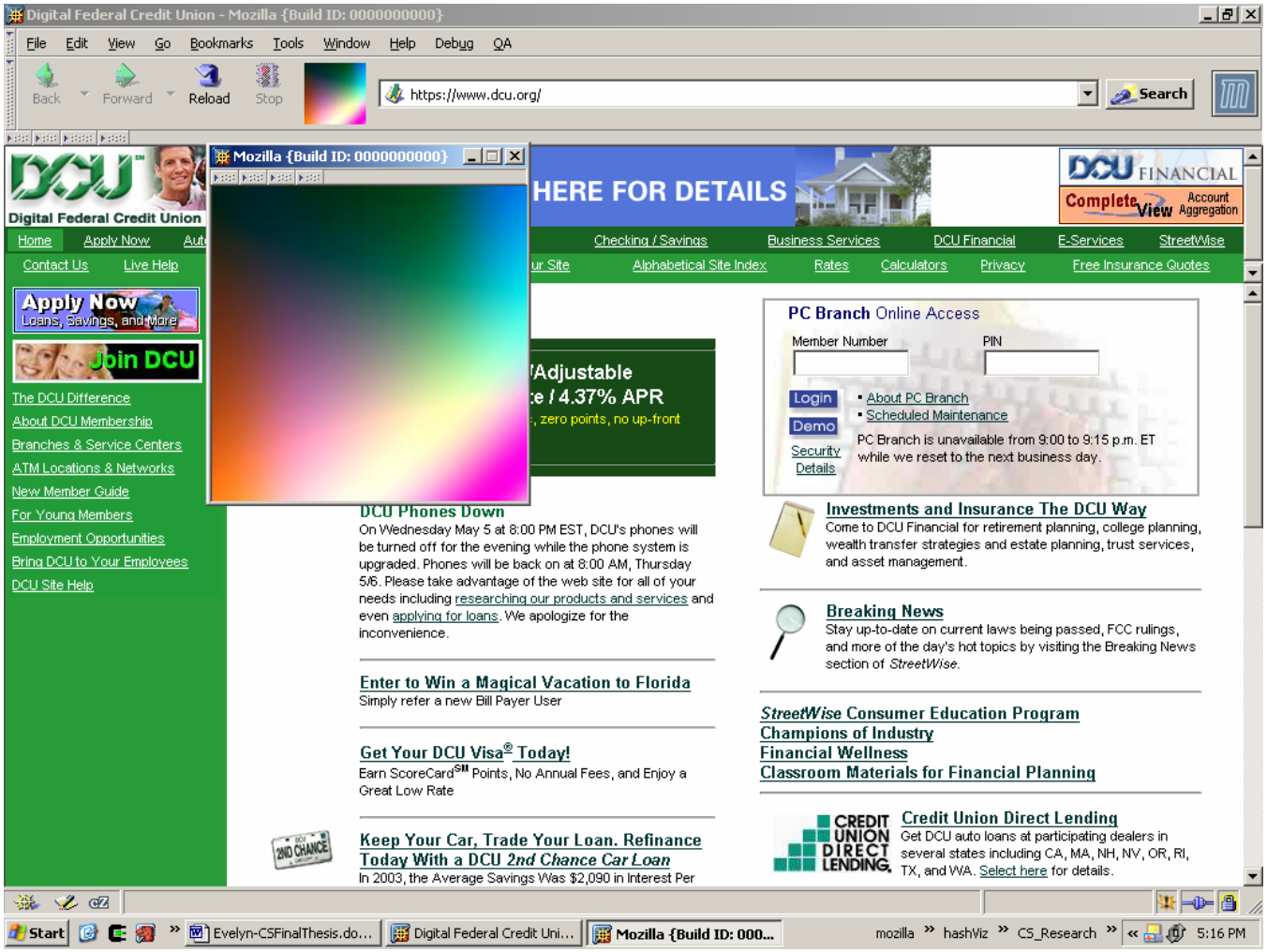


Figure 4: Another example of the Mozilla browser with the enlarged version of the image (SHA1 Fingerprint) after clicking on the button. The dimension of the larger image is 256x256 pixels.

3.5 Testing

Most of the images generated have color, geometric components, are irregular and thus easily discernable from one another, with the exception of some images (for example (xiii) shown in the image gallery is shown at **Appendix A**).

3.6 Problems Encountered

Mozilla's chrome, including its CSS (cascading style sheets), is cached and only refreshed upon shutting down and restarting the browser, which explains why that is carried out whenever new extensions are installed. This is a built-in feature necessary for speed and efficiency. This poses significant problems when visiting different SSL sites in a browsing session using the same browser instance. The browser had a tendency to use the cached image generated in the prior SSL connection and display it in the hashViz button, even though an accurate image from the current sever certificate has been generated. This was the case when there were two tabbed SSL web pages that has finished loading. A temporary solution was to open up a new tab and load the other SSL document. A new tabbed panel is instantiated with the `STATE_IS_INSECURE` and a transition to a secure state fires off a notification so that *hash_verify* gets called. Another more convenient solution is to install the Mozilla PrefBar[16], an extension and have the options checked for clearing the chrome cache.

Meanwhile, this introduces another issue. Given that there is only one hash image at any point in time, going from one SSL site to another and back again will result in a series of unnecessary image generation. If an SSL site has been idle with all its contents cached and the user wishes to return to it after browsing other sites (waiting for it to load perhaps), it will be helpful if *hashViz* immediately displays the image generated when that website first loaded. In other words, it may be more efficient to stored the all the images generated during a browsing session to provide for quick retrieval when needed. This is different from the cache problem because the browser displays the hash image associated with the active web site accordingly, and not simply any generated from a previous verification.

To associate each stored image generated with the website, it can be named something unique, for e.g. its serial number that was assigned by the certificate issuer. As long as the SSL certificate in question is the same one, it is fine to used stored images. This is true since certificates are cached. Since there are no protective mechanisms for the stored images, it is best to remove and re-generate them after a certain time period. Ideally, the images should be deleted when the browser shuts down, or even when the SSL session ends.

Multiple frames and content sources (from different servers) in the browser also created some difficulties. The browser updates its status every time it detects a change in its security level. If there is content from different servers, or many frames within the browser serving content by different servers or even loading at different times, the browser calls the hash visualization algorithm (implemented as *hash_verify*) multiple times too. One way I got around this was to add checks throughout the url document loading process, for example, checking to see that *hash_verify* is called right at the start when *nsIWebProgressListener* detects a certificate, and only once so that only one image generation process takes place during the browsing session using the same certificate for server authentication.

An area of greater concern is to identify multiple ssl servers' content on a single document rendered by the browser. To illustrate, suppose an SSL page from server *A* embeds content coming from server *B*. It is interesting to know which certificate the browser will report. Reports from a previous study [3] reported that Netscape and Mozilla only reported one server A's certificate in "Security Information" under such a scenario. If server *A* is trusted but not server *B*, this could pose as a problem since the user would assume that everything he sees in the browser is to be trusted. Indeed, it will be hard to decide on which certificate to use for authenticating the whole web document. Documents containing (encrypted or unencrypted) data from more than one server would have to establish trust for all their sources for they are thus vulnerable to attacks coming from another channel (e.g. server *B*). Currently, the closest capability Mozilla has with regards to this is detecting mixed encrypted and unencrypted content.

3.7 Installation

At this stage of infancy, *hashViz* can be installed on Mozilla in the following way: install *hashViz.xpi* (available on **www.andrew.cmu.edu/~het/research.html**) containing the image generation executable and the GUI features, and overwrite the existing *nsBrowserStatusHandler.js* in *comm.jar* (the original version of should be backed up). To do this, first uncompress the archive into a directory (*comm*) in the *chrome* directory using standard jar utilities. Next, image conversion utilities from ImageMagick [7] should be installed. When the browser starts up after installation, enter the location of the executable *convert.exe* (for image format conversion) into the file picker dialog which will show up if it cannot be located.

4. Possible Issues and Hacks

At its best, Hash Visualization in the Mozilla browser only provides a partial solution to tampering and impersonation problems present in Internet Security. All it does is reflect the certificate state, bringing to the user's attention the identity of the other end of the connection. There are ways to exploit the remaining vulnerabilities that hashViz cannot detect. Furthermore, since Mozilla is open-source, anyone can see the working mechanism behind the browser. The SSL code in the security module is solid as far as records show, but the chrome is accessible and can be manipulated to show content different from that of the true state.

XPInstall allows a user to download a Mozilla extension from a web site. This installs jar files (normally the case) into the chrome. An adversary can take advantage of this can have his own version of chrome files installed, especially if the user is not too particular about downloading only signed *XPI* files. This way, he can control what the browser shows. For example, he can change the behavior of the lock object to have it display a secure image (locked icon) by setting its `level` attribute to `high, low` or `broken` as he wishes.

Currently, there is no mechanism in place to protect unauthorized access to the folder where the hash images are stored, although this shouldn't pose as a problem because images are generated every SSL session and cannot be forged unless the attacker has the server certificate in his possession.

5. Future work

Future work to be considered include a having better implementation by building *Random Art* and the image conversion utilities into a dynamic link library to be distributed with *XPI* installation. This should allow for faster image creation. All the code should be in standalone files instead of existing in modified Mozilla source code. This will make it easier to package *hashViz* for people to installation as an extension.

As the reader can see, *hashViz* lacks ease of installation. To cut down on a necessary installation of a third party software (ImageMagick), Professor Perrig suggested including the capability to write the hash visualization output into PNG files, which can be displayed straightway on the browser. Also, integrating this capability into the image generation code would be ideal.

I am also looking to add features to make it more platform-portable. Some of these include a file dialog to locate executables if they cannot be found. As of this writing, the file paths are for Windows, but they will be changed to include those for other platforms. I would like to enable an automatic detection of the platform and if time permits, a search for the needed libraries and executables. For the sake of completeness, I would also like to include an option for users to choose if they would prefer a SHA1 fingerprint hash of a MD5 one, as well as input for the preferred image size.

Also, more intensive testing is needed to determine what properties of the certificate fingerprints result in plain images that might look too similar to one another, and whether the human user is able to distinguish between similar looking hash images.

6. Conclusion

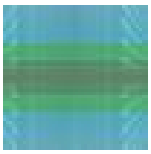
Humans are the weakest link in Internet security, a fact often exploited by malicious intruders. Although not perfect, hash visualization offers more convenient and secure ways of verifying the server certificates because it makes use of the human ability to recall images. It is a media rich method, which hopefully serves as bridge between the underlying certificate verification process and the GUI displayed. Of course, it can be extended to include other certificates like CA root certificates and S/MIME (for encrypted mail) certificates.

References:

[1] Rachna Dhamija, Adrian Perrig. Déjà vu: A User Study Using Images for Authentication.
[2] Adrian Perrig, Dawn Song. Hash Visualization: A New Technique to improve Real-World Security
[3] Zishuang Ye, Sean Smith. Trusted Paths for Browsers
[4] Ed Gerck. Overview of Certification Systems: X.509, PKIX, CA, PGP & SKIP
[5] Nigel McFarlane. Rapid Application Development with mozilla
[6] Andrej Bauer. Random Art **http://gs2.sp.cs.cmu.edu/art/random/howto/**
[7] ImageMagick Convert utilities **http://studio.imagemagick.org/www/convert.html**
[8] Netscape. Introduction to SSL **http://developer.netscape.com/docs/manuals/security/sslin/index.htm**
[9] Netscape. Introduction to Public key Cryptography **http://developer.netscape.com/docs/manuals/security/pkin/index.html**
[10] Mozilla.org. The SSL Reference **http://www.mozilla.org/projects/security/pki/nss/ref/ssl/index.html**
[11] Mozilla.org. Netscape Security Services **http://www.mozilla.org/projects/security/pki/nss/**
[12] Seamonkey Cross Reference **http://lxr.mozilla.org/seamonkey/source/**
[13] Mozilla Chrome **http://www.mozilla.org/xpfe/ConfigChromeSpec.html** **http://bdsmedberg.no-ip.org/chrome/**
[14] XPInstall. **http://www.mozilla.org/projects/xpinstall/**
[15] XPCOM: Cross Platform Component Object Module **http://www.mozilla.org/projects/xpcom/**
[16] PrefBar. **http://prefbar.mozdev.org/**
[17] NSS (Netscape Security Services) **http://www.mozilla.org/projects/security/pki/nss/**

[a] throbber element: 

Appendix A: HashViz Images Generated by Some SSL Sites


(i)


(ii)


(iii)


(iv)


(v)


(vi)


(vii)


(viii)

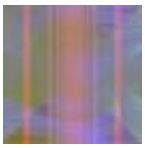

(ix)

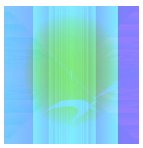

(x)


(xi)

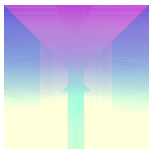

(xii)


(xiii)

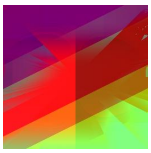

(xiv)


(xv)


(xvi)


(xvii)


(xviii)


(xix)

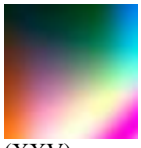

(xx)


(xxi)


(xxii)


(xxiii)


(xxiv)


(XXV)

(i) **attwireless.com** (MD5)
(ii) **attwireless.com** (SHA1)
(iii) **onlinebanking.nationalcity.com (SHA1)**
(iv) **onlinebanking.nationalcity.com (MD5)**
(v) **https://www.andrew.cmu.edu** (SHA1)
(vi) **https://www.andrew.cmu.edu** (MD5)
(vii) **www.paypal.com** (MD5)
(viii) **www.paypal.com** (SHA1)
(ix) **secure.safaribooksonline.com**.(MD5)
(x) **secure.safaribooksonline.com**.(SHA1)
(xi) **www22.verizon.com/secure/myaccount/anonymous/** (MD5)
(xii) **www22.verizon.com/secure/myaccount/anonymous/** (SHA1)
(xiii) **chaseonline.chase.com** (MD5)
(xiv) **chaseonline.chase.com** (SHA1)
(xiv) **www.verisign.com** (SHA1)
(xv) **adwords.google.com/select** (MD5)
(xvi) **adwords.google.com/select** (SHA1)
(xvii) **www.verisign.com** (MD5)
(xviii) **www.verisign.com** (MD5)
(xix) **www.hushmail.com** (SHA1)
(xx) **webiso.andrew.cmu.edu/login.cgi** (SHA1)
(xxi) **webiso.andrew.cmu.edu/login.cgi** (MD5)
(xxii) **https://www.accountlink.pncbank.com/index.html** (MD5)
(xxiii) **https://ugrant.web.cmu.edu/intro.jsp** (SHA1)
(xxiv) **https://ugrant.web.cmu.edu/intro.jsp** (MD5)
(xxv) **https://www.dcu.org/** (SHA1)