GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Security Essentials (Security 401)"
at http://www.giac.org/registration/gsec

Custom Built-In Root CA Certificates for Mozilla

Gabriel L. Somlo

August 13, 2004

Abstract

SSL and TLS are heavily relied on for secure Web communications via the HTTPS protocol. Still, there is a lot of potential for abuse using man-in-the-middle attacks, which are facilitated by the lack of sophistication among the user population, and by the fact that, for cost reasons, many servers operate outside the public-key infrastructure on which the protocol's security is based. We offer an overview of the cryptographic methods underlying SSL, and propose a policy where browser-generated security warnings can always be taken seriously in a reliable way, in order to avoid security breaches caused by user confusion. By building all necessary certificate authority information into the browser, users would not be expected to make any changes on their own, and would never encounter situations where it is necessary to ignore the browser's security warnings. Examples are provided using Mozilla v.1.7.2 running on the Fedora Core 2 distribution of GNU/Linux.

© SANS Institute 2004, Author retains full rights.

1 Introduction and Motivation

Perhaps the most wildly popular application of public-key cryptography in use today is the HTTPS protocol. A vast number of people perform "secure Web" transactions on a daily basis, by shopping and banking on-line, or by managing various devices with Web-enabled management interfaces.

The security of TLS (Transport Layer Security, the protocol underlying HTTPS) [Dierks99] is based on a commercial public-key infrastructure (PKI), where the client applications (i.e., Web browsers) come with a built-in list of *trusted third parties* or authorities capable of vouching for the integrity of HTTPS servers to which they are connecting. The whole protocol becomes vulnerable when servers refer to authorities that aren't built into all popular Web browsers. Since authorities generally charge for their services, server administrators may choose to operate completely outside the established PKI. When this happens, users typically receive warning messages from their Web browser, and are asked to make a decision on whether or not to trust the integrity of the visited Web server anyway. Many large sites, including many academic institutions, instruct users to *selectively* ignore such warnings, and/or to perform a series of steps that would instruct their browsers to trust such Web servers.

In order to minimize user confusion, and to facilitate a simple policy where *all* browser warnings could reliably be taken seriously, we propose that any necessary credentials be built into the browsers on a site-wide basis, before having them deployed to the users. Users would not be expected to perform any configuration steps, and could be instructed to *always* notify their administrator if they received security warnings from their Web browser.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

We continue by reviewing the cryptographic techniques at the basis of TLS in section 2. The problems that occur when sites use certificate authorities that are not recognized by the client software are illustrated in section 3, where we also propose a simple policy to mitigate the associated risk. Instructions on how to add a new root certificate to Mozilla's built-in list are given in section 4. Finally, section 5 shows how to build an RPM package for the modified Mozilla executable.

2 Background: Cryptography, SSL/TLS, and Certificates

This section is intended as a quick refresher on cryptography, public keys, the way SSL/TLS is used on the Web (via HTTPS), and the role of certificates and certificate chains.

2.1 Symmetric Key Encryption

Encryption is the process whereby a cleartext message is transformed into an unintelligible form intended to prevent reading by unauthorized parties. Decryption is the reverse operation, by which the recipient transforms the encrypted message back into a cleartext, readable form. Encryption and decryption rely on *cryptographic algorithms*, which are mathematical functions that perform the conversion between cleartext and encrypted messages.

Currently used crypto algorithms are usually well-known functions, widely available to any interested party. The secret information that prevents unauthorized parties from decrypting messages is actually a *parameter* to the crypto function, known as a *key*. In the most straightforward type of cryptography, the same key is used both to encrypt and to decrypt the transmitted message. This is known as *symmetric key cryptography*. The sender and receiver must make prior arrangements to agree upon, and keep secret, the value of the symmetric key *before* they can begin to communicate. A naive example of this process is for the sender to bitwise XOR the cleartext message with a string of equal length consisting of repeated concatenations of the key; and for the receiver to apply the same process to the encrypted string that is received, thus obtaining back the cleartext message. This process is illustrated in Figure 1, for cleartext message `1011010`, and key `1001`.

cleartext:	1011010	
key string:	1001100	
	——-	XOR
encrypted:	0010110	
key string:	1001100	
	——-	XOR
cleartext:	1011010	

Figure 1: Naive example of symmetric key cryptography using XOR

2.2 Public Key Encryption and Signatures

The main issue with symmetric key crypto is the chicken-and-egg problem associated with communicating the secret key securely, before reliable encryption is available in the first place. This problem is addressed by *public key cryptography*.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

The idea of public key cryptography was first proposed by Diffie and Hellman [Diffie76]. Its first implementation was provided by Rivest, Shamir, and Adleman [Rivest78], known today as the RSA public key cryptosystem. RSA is based on two secret large prime numbers, *p* and *q*. Two exponents are computed, *e* and *d*, such that

$$(e \cdot d) \bmod [(p-1) \cdot (q-1)] = 1$$

A recipient publishes the product $n = p \cdot q$ and the public exponent e, but keeps secret the individual values of p, q, and the private exponent d.

Without loss of generality, we assume that we are sending and receiving numerical messages (any message can be converted into a numerical format as a pre-processing step). The encrypted message C is obtained from the cleartext message M using encryption function E:

$$C = E(M) = M^e \bmod n$$

The information required to perform the encryption function E (e and n) is publicly available to anyone. To decrypt the encrypted message C, the recipient must use a decryption function D:

$$M = D(C) = C^d \bmod n$$

Since the recipient keeps d secret, in order to deduce it, an attacker would first need to obtain p and q by factoring n, which is prohibitively expensive computationally, and thus impossible to do in any reasonable amount of time.

To summarize, public key crypto relies on a *pair* of functions: E for encryption, and D for decryption. E is made public, and computing D by knowing E is prohibitively expensive. Moreover, E and D are the inverse of each other. In other words,

$$D(E(M)) = M,$$

$$E(D(C)) = C,$$

for any message M or C. Not only can these methods be used for encryption, but also for *digital signatures*, using the second equation above. Once a user's public key E is well known, he can sign messages by encrypting them with his *private* key D, before sending them to the recipient. The public key E can then be used for *authentication* (i.e., to make sure the sender is who he claims to be), as well as for *non-repudiation* (i.e., to prevent the sender from later denying to have sent the message). In practice, the sender only computes an encryption of a condensed *hash* of the cleartext message, and appends it as a signature at the end of the cleartext. The recipient then decrypts the hash, and compares it to his own version obtained by hashing the cleartext directly.

2.3 HTTPS, SSL, and Certificates

On the Web, secure transactions occur over the HTTPS protocol [Rescorla00], which, in essence is HTTP [Fielding99] over TLS [Dierks99]. TLS is still very often referred to using the name of its predecessor, SSL (Secure Sockets Layer). Without getting into specifics, TLS uses public key crypto to help two communicating parties set up a shared symmetric encryption key which is then used to transfer data for the duration of a session. Public key crypto helps work around symmetric

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

key crypto's chicken-and-egg problem mentioned earlier, but since it is computationally more costly, it makes sense to only use it for setting up a symmetric key, which is much cheaper to use for transferring a high volume of data.

Even when two parties are using public key crypto, an attacker still has a window of opportunity to perform a Man In The Middle attack (MITM) the first time public keys are exchanged. Essentially, an attacker will replace the transmitted public key(s) with his own, and decrypt, modify, and re-encrypt traffic (this time using the real public key(s) intercepted earlier) without the knowledge of the communicating parties. For this reason, TLS requires public keys to be digitally signed by a member of a select list of trusted third parties. Signed keys together with added information identifying the owner of the key are known as *certificates*, and the trusted third parties are referred to as *Certificate Authorities* (or CAs). Users of TLS-based protocols (such as HTTPS) are supplied with hard-coded public keys of the CAs when the client applications (such as Web browsers) are shipped to them, and can verify signatures attached to the public keys of servers they visit. For simplicity, the CA public keys are packaged as certificates as well. Since there's no higher authority to sign a CA's public key, the CA's own private key is used to build the certificate's digital signature. Hence, the CA's certificates are referred to as *self signed* certificates.

3 Local Policy for Certificate-Based Server Authentication

Client applications usually ship with a built-in list of CA certificates chosen by the application's author. Whenever the user connects to a server using TLS, the client automatically checks whether any of its configured CAs vouches for the server's certificate. If this check fails, a warning is displayed and the user is asked to decide whether to abort the transaction, or trust the server anyway and continue. The issue is further complicated by the fact that CAs are usually commercial entities, and obtaining a server certificate from one of them costs money. For this reason, many organizations decide to use their own self signed certificates, which will obviously not be verifiable through an application's list of built-in CA certificates.

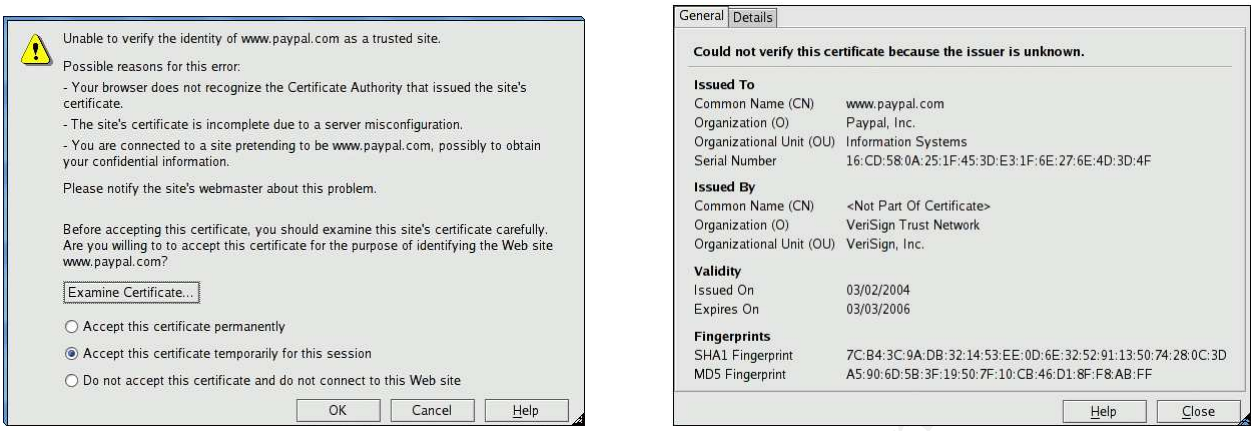
3.1 The Problem: When Certificate Warnings Pop Up

The browser is unable to distinguish between public keys that can't be verified because their certificates are self-issued by a server's administrator, and those that have fake certificates built by a MITM attack tool such as `ettercap`. The use of `ettercap` for MITM attacks against, among other things, HTTPS transactions, is documented in [Norton04]. Assuming the victim's IP address is `192.168.1.234/24`, an attacker could use the following command line to obtain cleartext of all HTTPS traffic between the victim and its default gateway, `192.168.1.1`:

```
$ ettercap -T -M arp:remote /192.168.1.1/ /192.168.1.234/
```

The victim would notice a warning from the browser, shown in Figure 2 (a), stating that the authenticity of the server's key could not be verified. If the user chooses to examine the certificate, information will be displayed similar to what is shown in Figure 2 (b). It is not immediately apparent what might be wrong with the certificate in question. Savvy users would immediately pick up on the fact that something is fishy, at least because a real Verisign certificate would be verified automatically, and wouldn't generate any warnings to begin with. However, naive users, especially if

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.



a) b)

Figure 2: a) MITM attack against a user visiting paypal.com; b) Fake Verisign certificate generated by Ettercap

de-sensitized to this type of warning due to extensive use of self-signed, locally generated server certificates, would be very likely to just click OK on the first window and try to get on with their work.

The ideal solution would be an attempt to educate users, allowing them to distinguish between several scenarios:

- Test server on the intranet: the admin probably generated a local self-signed certificate, and it's OK to go ahead, especially if we don't intend to supply any sensitive information.
- Some mailing list archive on the Web: they chose to use HTTPS for reasons unknown, and we aren't expected to type in sensitive information (actually, we don't expect to type in anything at all); it's OK to continue.
- We are trying to connect to a Web-mail interface, or to our banking site. We expect to type in an account name/number, a password, and all sorts of other information that is sensitive in nature. This is what HTTPS was invented for in the first place. If this type of server has a certificate that can't be verified, we need to abort the transaction right away, and start making angry/worried phone calls to people in charge.

Realistically, however, a simpler approach is always preferable to complex rules with motivations that seem incomprehensible to the user. For instance, an alternative to the above list would be: *"If a window pops up, warning you about certificates, it is no longer safe to enter any sensitive information. Please notify your administrator immediately."*

3.2 Importing Certificates (The User Perspective)

What to do if a resource-strapped site still needs to use locally generated, self-signed certificates for sensitive information? Also, many applications and appliances (e.g., Cisco 3000 VPN concentrators) are shipped with self-signed SSL certificates for their HTTPS management interfaces. Many universities require their users to download these certificates from the local intranet, and *import* them into their browser profiles. The process is simple: certificates are ASCII-encoded (the format

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

is known as PEM, for Privacy Enhanced Mail) and placed on an intranet Web server. Most modern Web browsers will recognize the format, and when the link is clicked, open a window that guides the user through the certificate import process, asking if the user trusts the authority issuing the certificate for various purposes such as secure HTTP, secure mail, etc. The downside of this solution is twofold. First, it requires (potentially naive) users to make decisions for which they don't have enough information. Secondly, the process is itself vulnerable to MITM attacks.

3.3 Built-In Certificates (The Administrator Perspective)

In a centrally managed environment (such as a lab, or a company), certificate authorities could be added to the built-in list that ships with the application. This has the advantage that users would no longer be required to import certificates on their own. The downside of this solution is that it's restricted to centrally managed environments, and does not address the problem on user-managed equipment, such as personally owned laptops or home PCs.

The rest of this paper shows the technical details required to add a built-in root CA certificate to the Mozilla Web browser. We use ipsCA as an example, since they offer free certificates to educational institutions, and are not already included in Mozilla's list of built-in root CAs.

The problem of distributing such changes to user-owned equipment that is not centrally managed can be worked around by building an easy-to-install package of the Mozilla browser that could be written to a CD and passed around to users for easy installation. We show how to build a modified Mozilla package for RPM-based Linux distributions (such as Fedora Core 2), but the principle should apply to any other operating system and/or package manager.

4 Adding a Built-in Root CA Certificate to Mozilla

Adding built-in root CA certificates to the Mozilla browser is not a difficult task, once the relevant details and idiosyncrasies are known. This section offers detailed instructions on how to add an extra built-in root certificate to the Mozilla executable.

4.1 Where Mozilla keeps its built-in certificates

After unpacking the `mozilla-source-1.7.2.tar.bz2` tarball, and performing some exploratory searches for files containing the string `cert` in their names, we find that Mozilla's built-in root certificates are kept in the `libnssckbi.so` dynamic library, whose source code can be found under `mozilla/security/nss/lib/ckfw/builtins` within the Mozilla source tree.

The `README` file inside that directory offers really helpful advice on how to add or remove root certificates, but starts out by stating that, in order to do so, one must first build the `addbuiltin` tool, which is "not built by default" as part of the Mozilla build process.

4.2 Building `addbuiltin`

As it turns out, trying to build the `addbuiltin` tool from within the Mozilla tree is no easy feat. After attempting all sorts of Makefile hacks, and an extensive search on the Web, it became apparent that `addbuiltin` is best built from the standalone NSS module, which is available at `ftp://ftp.`

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

mozilla.org/pub/mozilla.org/security/nss/releases/NSS_3_9_2_RTM/src/nss-3.9.2.tar.gz. First, we build the NSS main package by following the instructions available at http://www.mozilla.org/projects/security/pki/nss/nss-3.7.1/nss-3.7.1-build.html; next, we change into addbuiltin's directory, and finish building the tool as shown below:

```
$ ls
nss-3.9.2.tar.gz
$ tar xfz nss-3.9.2.tar.gz
$ cd nss-3.9.2/mozilla/security/nss/
$ make nss_build_all
...  <lots of output from make> ...
$ cd cmd/addbuiltin
$ make
...  <some more output from make> ...
$ ls Linux*
addbuiltin addbuiltin.o
```

We now place the compiled addbuiltin binary into a directory that is part of the $PATH variable (such as ~/bin), in order to have it readily available for execution during the next step. For completeness, we attempted the same process from Mozilla's full source tree, but the build failed with make reporting errors during the make nss_build_all stage.

4.3 Patching the list of built-in root CA certificates

In our example, we use ipsCA as the certificate authority (CA), since they offer free certificates to educational institutions such as ours, and are not by default included in Mozilla's list of built-in CAs. The same technique applies for any other root CA certificate (which can be generated using the instructions provided in Apache's SSL FAQ, at http://httpd.apache.org/docs-2.0/ssl/ssl_faq.html#aboutcerts. The ipsCA root certificate files (IPSCACLASEA1.crt and IPSServidores.crt) are available in x509 ASCII-encoded PEM format. We convert both certificates into the binary DER (Distinguished Encoding Rules) format, which is required by the addbuiltin program:

```
$ ls
IPSCACLASEA1.crt IPSServidores.crt
$ openssl x509 -inform PEM -outform DER -in IPSServidores.crt
-out IPSServidores.der
$ openssl x509 -inform PEM -outform DER -in IPSCACLASEA1.crt
-out IPSCACLASEA1.der
$ ls
IPSCACLASEA1.crt IPSCACLASEA1.der IPSServidores.crt IPSServidores.der
```

Next, using the addbuiltin tool, we convert the DER-formatted certificates to the format expected by Mozilla/NSS:

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

```
$ touch new_certdata.txt
$ cat IPSServidores.der | addbuiltin -n "IPS SERVIDORES" -t
"C,C,C" >> new_certdata.txt
$ cat IPSCACLASEA1.der | addbuiltin -n "ipsCA CLASEA1
Certification Authority" -t "C,C,C" >> new_certdata.txt
```

The certificate name (i.e., the string used as an argument to the `-n` flag to `addbuiltin` can be extracted using the following command line (look for the contents of the field named `CN`):

```
$ openssl x509 -inform PEM -text -in IPSCACLASEA1.crt | grep
Subject
Subject:  C=ES, ST=Barcelona, L=Barcelona, O=IPS Certification
Authority s.l., O=general@ipsca.com C.I.F. B-B62210695, OU=ipsCA
CLASEA1 Certification Authority, CN=ipsCA CLASEA1 Certification
Authority/emailAddress=general@ipsca.com
```

We are now ready to patch Mozilla/NSS itself, using the `new_certdata.txt` file obtained earlier:

```
$ cd mozilla/security/nss/lib/ckfw/builtins/
$ cat ~/new_certdata.txt >> certdata.txt
$ make generate
perl certdata.perl < certdata.txt
Name "main::a" used only once:  possible typo at certdata.perl line 207.
$ find .  -newer certdata.txt
.
./certdata.c
```

Running `make generate` does give a warning (at least in Mozilla version 1.7.2), but it seems to be harmless. We notice from the output of `find` that, as a consequence of `make generate`, we have also modified the `certdata.c` file. This latter file will cause our changes to be incorporated into Mozilla's `libnssckbi.so` dynamic library, thus making our changes available to all system users by default. In order to facilitate re-applying our changes to a pristine source tree (`mozilla`), we use `diff` to generate a patch that would make the necessary changes outlined above. The modified source tree was renamed to `mozilla-ips` for this purpose.

```
$ diff -rU5 mozilla mozilla-ips > mozilla-ipsca.patch
```

5 Incorporating Changes Into an RPM Package

This section explains how to build an RPM package containing the modified Mozilla (with added built-in root CA certificates), to facilitate easy distribution to a large install base.

In addition to being a package management system, which keeps track of the location and version of the various files that belong to a set of software packages, RPM (http://www.rpm.org/) is also a *build* system, in that it allows a binary package to be built from sources according to a precise set of instructions. In essence, the traditional UNIX/Linux `configure` / `make` / `make install` process is scripted precisely, and the correct parameters are supplied at each step, in order for the resulting package to play well with the rest of the installed packages making up the operating system or distribution.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

5.1 Quick overview of `.spec` files

The scripted instructions for building a package from source come in a *specification* file (a.k.a. the `.spec` file). The complete set of sources and patches required to build a package, along with the `.spec` file, are bundled together in a source RPM package (often referred to as a `.src.rpm` or SRPM). A detailed description of the various sections of a `.spec` file is outside of the scope of this paper, but excellent documentation is provided by [Barnes99].

A `.spec` file starts out with a header section listing the name of the package, version information, all source files and patches, and dependency information such as prerequisites for building and running the application. A source RPM package may generate several binary packages, one for each independently installable component of the application. A one-paragraph `description` for each of these sub-packages follows the header in the `.spec` file. The `prep` section is next, and its job is to control the unpacking of source tarballs and application of all necessary patches to the application's source tree. Next, the `build` section controls how the application is compiled, by issuing commands such as `configure` and `make` with the appropriate parameters. The `install` section will install all application files into a mock-root directory (specified in the `$RPM_BUILD_ROOT` variable). Various pre and post (un)install sections direct the binary RPM packages to run various commands before or after being installed or removed from the system. Several `files` sections specify which of the files installed under `$RPM_BUILD_ROOT` belong to which binary sub-package. Finally, a `changelog` section keeps track of all modifications made to the `.spec` file throughout its lifespan.

5.2 Modifying `mozilla.spec`

We start out by unpacking the original Mozilla source RPM package, using the `rpm2cpio` tool:

```
$ ls
mozilla-1.7.2-0.2.0.src.rpm
$ rpm2cpio mozilla-1.7.2-0.2.0.src.rpm | cpio -id
```

We now have access to the official Mozilla tarball (`mozilla-source-1.7.2.tar.bz2`), the various other source files and patches specific to Fedora Core 2, and the `mozilla.spec` file.

First, we need to modify the release component of the version information from `0.2.0` to `0.2.0ipsCA`. Next, we add the file `new_certdata.txt` obtained in the previous section to the list of sources for this package. In the `prep` section, we add instructions to append the content of `new_certdata.txt` to the file `mozilla/security/nss/lib/ckfw/builtins/certdata.txt` and run `make generate` in the `builtins` directory (as shown in the previous section). We also document the changes in the `changelog` section. The modified `mozilla.spec` file can be downloaded at http://www.cs.colostate.edu/~somlo/rpms/mozilla.spec. The difference between the original and modified `.spec` file is shown in Figure 3. Finally, we build the new binary RPM package to be distributed:

```
$ rpmbuild -bs mozilla.spec
Wrote:  mozilla-1.7.2-0.2.0ipsCA.src.rpm
$ rpmbuild --rebuild mozilla-1.7.2-0.2.0ipsCA.src.rpm
```


© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

```
$ diff -U2 mozilla.spec.orig mozilla.spec
--- mozilla.spec.orig   2004-08-10 11:25:14.295795130 -0600
+++ mozilla.spec        2004-08-10 09:16:08.000000000 -0600
@@ -9,5 +9,5 @@
 Summary:     Web browser and mail reader
 Version:     1.7.2
-Release:     0.2.0
+Release:     0.2.0ipsCA
 Epoch:       37
 License:     MPL/NPL/GPL/LGPL
@@ -27,4 +27,5 @@
 Source18:    mozilla-xpcom-exclude-list
 Source19:    mozilla-redhat-default-bookmarks.html
+Source20:    new_certdata.txt
 Patch0:      mozilla-navigator-overlay-menu.patch
 Patch1:      mozilla-editor-overlay-menu.patch
@@ -208,4 +209,8 @@
 /bin/cp %{SOURCE19} $RPM_BUILD_DIR/mozilla/profile/defaults/bookmarks.html

+# add ipsCA certificates
+/bin/cat %{SOURCE20} >> $RPM_BUILD_DIR/mozilla/security/nss/lib/ckfw/builtins/certdata.txt
+make -C $RPM_BUILD_DIR/mozilla/security/nss/lib/ckfw/builtins generate
+
 %build
 # Rebuild configure to ensure that any patches to configure.in get applied
@@ -738,4 +743,7 @@

 %changelog
+* Mon Aug 06 2004 Gabriel Somlo <somlo at acns dot colostate dot edu>
+- added ipsCA certificates to nss builtins
+
 * Tue Jun 22 2004 Christopher Blizzard <blizzard@redhat.com>
 - Fix include paths in mozilla-xpcom.pc so that all the paths are
```

Figure 3: Changes to the `mozilla.spec` file

6 Conclusion

We have pointed out a policy-related security problem with HTTPS server certificates, where users tend to become de-sensitized to warning messages related to server key authentication failures as a consequence of over-use of self-signed certificates. We provide a simple and robust solution, by building the desired CA certificates directly into the binary browser code. We also provide an example of how these changes could be distributed to users more easily, by providing a replacement RPM package for the Fedora Core 2 GNU/Linux distribution.

References

[Barnes99] D. Barnes. RPM HowTo. http://www.rpm.org/RPM-HOWTO/, 1999.

[Dierks99] T. Dierks and C. Allen. The TLS Protocol. RFC 2246, 1999.

[Diffie76] W. Diffie and M.E. Hellman. New Directions in Cryptography. *IEEE Transactions in Information Theory*, 22(6):644–654, 1976.

[Fielding99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. RFC 2616, 1999.

[Norton04] D. Norton. An Ettercap Primer. *GIAC Security Essentials Certification Practical*, http://www.giac.org/practical/GSEC/Duane_Norton_GSEC.pdf, 2004.

[Rescorla00] E. Rescorla. HTTP over TLS. RFC 2818, 2000.

[Rivest78] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.