Chrome: From NSS to OpenSSL

Design Doc DRAFT

Author: rsleevi@
Last edited on: 2014-01-26

Objective

Provide a consistent experience for all cryptographic applications within Chromium and Chromium OS, with reliable (and secure) performance in the common-case and hardware-accelerated performance (on all platforms) when supported. Support the variety of efforts for in-process isolation (eg: profiles / StoragePartitions), such as SSL/TLS session caches and certificate trust and distrust decisions. Reduce maintenance cost by having a single supported interface, allowing for more rapid innovations in performance and rapid responses to security issues.

Background

Currently, Chromium supports two different SSL/cryptographic backends. On Windows, OS X, iOS, Linux, and Chromium OS, Chromium uses NSS. On Android, Chromium uses OpenSSL. This split is further reflected throughout the cryptographic services exposed by Chromium - from basic hashing, the SSL/TLS implementation within net/, extensions, WebRTC's implementation of DTLS (and SRTP), the Clear Key EME Content Decryption Module, and Blink's implementation of WebCrypto, to name a few, with dual implementations of almost all core functionality. The exception to this split is certificate verification - on Windows, OS X, and Android, Chromium uses the native certificate verification functionality.

Prior to Chrome 6, Chromium supported even more implementations - supporting the native OS implementations on Windows (CryptoAPI and SChannel) and OS X (CDSA/CSSM and Secure Transport). However, Chromium transitioned away from these implementations for similar reasons as this document proposes - increased maintenance cost, decreased agility, and decreased performance.

The choice of NSS as the common desktop implementation at the time was based on the fact that NSS was already in use for the Linux port. The choice between NSS and OpenSSL for the Linux port was based on several factors; NSS was in use by the then-dominant browser (Mozilla Firefox, as Netscape originally developed NSS for Netscape Navigator), it supported smart cards using an industry standard API (PKCS#11), and had robust RFC 3280/5280 certificate path building and verification code, which is absolutely vital for being able to access public websites, due to the frequent edge cases and misconfigurations with respect to certificate chains.

Overview

The choice of NSS vs OpenSSL has been discussed several times in the past on the chromium-dev mailing list. Below is a rough summary

Pros and Cons

	NSS	OpenSSL
Pros	● RFC 5280 certificate path building and verification ● Shared with Firefox, meaning improvements can quickly benefit “the Web” ● Designed as a stand-alone module/DLL with a strong public API & ABI ● FIPS 140-2 validated on multiple desktop platforms ● PKCS#11 support ● Chromium developers are contributors or peers of NSS and NSPR (wtc@, rsleevi@)	● Heavily optimized for a variety of architectures, often by chip designers directly (Intel, ARM) ● Significantly smaller footprint - in memory and on disk - due to fewer layers of abstraction (either EVP -> function pointer or EVP -> ENGINE -> function pointer) ● Much more mature server implementation (relevant for DTLS and for extensions) ● Chromium developers are contributors to or have significant experience with (agl@, rsleevi@, Opera?) ● Used by a variety of other applications (eg: wpa_supplicant and OpenVPN on Chromium OS)
Cons	● Multiple layers of abstractions prevent compiler optimizations (CERT_* -> STAN_* -> nss3_* -> NSPR*). ● Heavy dependence on PKCS#11 means that even “simple” operations, such as parsing a certificate, may end up blocking on device I/O ● Strong ABI requirements prevent significant refactorings/cleanup. ● Certificate path building is C code designed to emulate Java code, through indirection of an average of 7 layers of macros, two platform abstraction layers, and then the remaining NSS abstraction layers (listed above) ● Heavy use of Big Global Locks to support PKCS#11 operations ● Heavy use of a variety of hidden, non-configurable static process-wide caches ● PKCS#11 support	● Lacks any reasonable form of certificate path discovery ● API has evolved over 15+ years, accumulating significant cruft and inconsistency ● Perl-based build scripts for optimized assembly ● Limited-to-no extension points for dealing with networking fetching (AIA, CRL, OCSP)

There are cons to maintaining the existing dual stack nature, such as duplicated efforts implementing features, such as those listed in the Background, as well as in fixing security issues that appear in both implementations (eg: BEAST, Lucky 13).

There are also pros to maintaining the existing dual stack. Most notable is that when suggesting or implementing improvements to TLS - such as ChaCha20/Poly1305, Channel ID, or False Start - implementing multiple times can highlight issues in the specification or potential implementation issues that other implementers may experience.

NSS or OpenSSL?

The question is whether to switch Android to NSS or all other platforms to OpenSSL. At first glance, it would seem “less” work to switch Chromium on Android to use NSS. However, this is not entirely true. When powering WebView, Chromium on Android uses the Android system-provided OpenSSL library - something not available to applications building with the Android NDK (like Chrome for Android or other Chromium-based Android applications). This helps reduce memory usage by having a single shared library in memory. To accomplish this with NSS, NSS would have to be part of the Android base image - which would still increase memory usage, as most other Android (native) services would still use OpenSSL.

Additionally, the increased redundancy and abstraction within NSS would be detrimental to the overall performance of Android.

Switching to OpenSSL, however, has the opportunity to bring significant performance and stability advantages to iOS, Mac, Windows, and ChromeOS immediately out of the gate. Switching Linux to use OpenSSL will take longer, due to the desire to continue to support PKCS#11-based smart card authentication, which will require more work. The biggest risk/cost to such a switch is no longer being able to help Firefox benefit from these efforts, nor benefiting from Firefox’s efforts in these areas.

Transition Work (preliminary)

SSL Sockets

Windows, OS X

To transition from SSLClientSocketNSS to SSLClientSocketOpenSSL, support for using the platform-native client certificate handles (eg: via CryptoAPI and Keychain Services, respectively) is needed. This is similar to the requirement when transitioning to SSLClientSocketNSS from SSLClientSocketWin/SSLClientSocketMac.

iOS

Drop-in equivalent

ChromeOS
Support for TPM-backed client certificates is required. Currently, the TPM is exposed via chapsd and a PKCS#11 library that communicates with chapsd over DBus.

Depending on the desired timeframes, there are two possible outcomes:
- Introduce additional DBus methods via chapsd to handle basic key enumeration and signature operations, required to implement the minimal set of operations (similar to the Windows / OS X client certificate implementations)
- Introduce a PKCS#11<->OpenSSL shim layer, either using something like engine_pkcs11 or (more likely), writing one using the existing chaps/chapsd PKCS#11 implementing layers

Linux
Because Chromium has historically used NSS, Linux users may have already generated certificates or other credentials and installed them into the NSS shared DB utilized by Chromium.

Additionally, Linux users MAY be utilizing PKCS#11-based smart cards - although the extent is currently unknown.

The following steps are to be taken:
- UMA the number of times a client certificate is used, regardless of storage
- UMA the number of times a client certificate is used where the key material is stored somewhere other than the NSS DB.

Based on these numbers, it should be decided whether to support PKCS#11 smart cards on Linux and whether to migrate the NSS DB key storage to a Chromium-defined key storage.

Certificate Verification

Windows
For the foreseeable future, Windows will continue to use the platform-native certificate verification APIs.

OS X
Due to significant limitations in the underlying platform's capabilities for building certificate chains, the OS X certificate verification will be ported to a hybrid of the newly-developed, OpenSSL-backed certificate path building engine, with trust decisions continuing to be provided by Keychain Services.

However, Keychain-defined trust exceptions (where specific CDSA/CSSM error codes are whitelisted for certain certificates/hosts) are unlikely to be supported.

iOS
iOS will transition to the OpenSSL-backed certificate path building engine, using the same trust store as ChromeOS.

ChromeOS
ChromeOS will transition to the OpenSSL-backed certificate path building engine, and will continue using its existing trust store.

Trust decisions will need to be migrated from the ChromeOS users profile into a form recognized by the OpenSSL engine. However, these trust decisions are simple due to the UI exposed on ChromeOS.

Linux
Linux will transition to the OpenSSL-backed certificate path building engine.

The source of trust/distrust decisions is contingent upon PKCS#11 support, but may make use of p11-kit as the preferred trust backend.

WebRTC
No special considerations are known to exist for WebRTC, as it supports using both NSS and OpenSSL as cryptographic engines.

RemotingIt will be necessary to finish implementing SSLServerSocketOpenSSL. Beyond that, Remoting uses the interfaces provided by crypto/ and net/, for which NSS and OpenSSL are interoperable and consistent.

Pepper/PPAPI
Certain Pepper APIs expose the ability to initiate SSL/TLS connections. However, these interfaces are minimal and expected to be equivalent between NSS and OpenSSL.

There may be existing Pepper plugins that make use of src/crypto or NSS, from within the Pepper sandbox. Such plugins are shipping their *own* copy of NSS, and can continue to do so, much in the way that “any” code can be run within the Pepper sandbox. However, they’re encouraged to use OpenSSL.

On Linux/ChromeOS, there exists special considerations to permit using a “system” NSS from within the sandbox, by preloading the NSS shared objects. When Linux/ChromeOS transitions away from NSS, so too will this warmup code.
As sandboxed plugins should not be depending on any state from the ‘host’ environment, this is arguably a “bug” today that will need to be resolved when transitioning to OpenSSL.

The primary purpose of such pre-warming is:
1) Load any necessary shared objects needed by NSS

With OpenSSL, this is not needed, as Chrome will not be loading a “shared” OpenSSL. Pepper applications are expected to link in their dependencies - including OpenSSL, if they have such a dependency.

2) Load /dev/urandom within the sandbox for use as an entropy source.

With Pepper, applications can request random data from the host using supported Pepper APIs. Modifications to OpenSSL to supply an alternate “system” entropy source (as opposed to directly open /dev/urandom) will be included in src/crypto - for Pepper plugins that bundle Chromium’s crypto

3) Load certificate trust databases

This is again a “bug”, in that it is not a “supported” Pepper API and not meant to be utilized by Pepper plugins.

EME / Content Decryption Module
No special considerations are known to exist for the Clear Key CDM included as part of Chromium. The Clear Key CDM makes use of the crypto/ interfaces, which are interoperable with the NSS implementations.

WebCrypto
WebCrypto development is currently focused on NSS support, and will continue to be for the immediate future based on plans. However, because the WebCrypto API does not define specific requirements with respect to Key Storage, Blink should be able to seamlessly transition from the NSS implementation to the OpenSSL implementation.

Note that the OpenSSL implementation is being pursued in parallel, but is slightly less feature complete than NSS at this time.

Required New Development (preliminary)

The following items require new code to be developed before the OpenSSL equivalent classes from src/crypto and src/net can be transitioned to.

SSLServerSocketOpenSSL

The remoting host makes use of TLS server sockets, and thus it's necessary to implement SSLServerSocketOpenSSL.

This is a relatively simple, straight forward implementation.

EVP interface for CryptoAPI/CNG

To support using client certificates on Windows, the relevant EVP methods for supporting RSA/ECDSA operations using CryptoAPI will need to be developed.

The OS integration code already exists for NSS, this is merely adopting it to the OpenSSL API.

EVP interface for Keychain Services/CDSA

To support using client certificates on OS X, the relevant EVP methods for supporting RSA/ECDSA operations using Keychain Services/CDSA will need to be developed.

The OS integration code already exists for NSS, this is merely adopting it to the OpenSSL API.

Key migration from NSS DB to Chromium-defined DB

On Linux, uses of the <keygen> tag and the application/x-x509-user-cert mime type cause keys and certificates to be imported into the NSS shared DB (~/.pki/nssdb). Depending on the user metrics related to client certificate authentication , it may be necessary to export keys/certificates from the common store into a Chromium-defined database for use with OpenSSL.

This is similar to the existing Chromium-defined database for Channel ID storage.

Alternatively, support for PKCS#11 may mean that Chromium will load the NSS “softoken” PKCS#11 module to continue reading/using the existing database, while generating/storing all new keys into its own database.

OpenSSL support for asynchronous certificate validation

OpenSSL currently provides a single callback for certificate verification, and expects that callback to complete synchronously. However, callbacks for retrieving a client certificate and for retrieving a Channel ID are both allowed to be completed asynchronously.

A patch to OpenSSL to support asynchronous verification will be needed. Patching OpenSSL with Chromium is fine, as we will distribute OpenSSL together with Chromium on all platforms (eg: there will be no reliance on a “System OpenSSL”). The exception is Chromium WebView for Android, which links against Android’s OpenSSL, but WebView will always be shipped simultaneously with the platform, and thus patches can be integrated with AOSP upstream.

Certificate Path Building

A directed, cyclic graph traversal engine, using the strategies described in RFC 4158, will need to be developed, since OpenSSL lacks one, while NSS has one (via libpkix).

To ensure both the correctness of the implementation and to avoid a “flag day” for Linux, the path building should be agnostic to the underlying cryptographic library. This is to permit comparing the library (and performance) using NSS against libpkix and comparing the NSS implementation against the OpenSSL implementation, to provide a smooth transition.

Such a path building engine minimally needs to support:

- RFC 5280 verification
- Support for application-defined policies
    - SSL/TLS
        - Verification of EKUs
        - RFC 6125 verification of hostname matching
        - [Optional] Verification callbacks for using pinning information
        - [Optional] Verification callbacks for using CRLSets
        - [Optional] Verification callbacks for cryptographic policy checks (eg: no MD5, no weak keys in Baseline Requirements conformant certificates)
    - [Optional, ChromeOS] EAP-TLS
        - Likely consistent with other EAP-TLS implementations (eg: Microsoft)
    - [Optional, ChromeOS] VPN certificate requirements
        - Likely consistent with other implementations (eg: RFC 4809)
    - [Optional] Other, embedder-defined requirements
- Support for stapled OCSP responses to be included
- Fetching of Authority Info Access when no valid issuers can be located
- Fetching of CRL and OCSP information when required by Enterprise Policy
- Sorting and returning of “bad paths” when no path can be found
    - eg: To distinguish “Chains to a trust anchor but is expired” from “Chains to a trust anchor but violates name constraints” and “Unknown trust anchor, expired, violates name constraints”

- Disabling specific errors during path building (eg: “Ignore expired, but do not expore any more serious errors”)

PKCS#11 integration library

To support smart cards on Linux, a bridge between OpenSSL’s core abstractions (eg: EVP_*) and PKCS#11’s abstractions (C_Sign/C_Verify/*) is needed.

- Sign the CertificateVerify using PKCS#11
  - Note that while some PKCS#11 smart cards require that all keys related to SSL be kept “on device” (eg: MAC keys, bulk encryption keys, master secret derivation), support for such smart cards (really, SSL “de”cellerators) is NOT planned
- Discover certificates and keys using PKCS#11 (for client certificate selection)
- Respond to PKCS#11 events (such as token insertion/removal)
- Extract trust information from p11-kit, which can interface with the smart cards

Pepper/Plugin Cleanup

Some internal Pepper plugins are known to be relying on side-effects of NSS being initialized the way it currently is by Chromium. These are design issues with the plugins and implementation details not meant to be exposed to Pepper.

- Audit all “official” Pepper plugins for dependencies on NSS
- Convert such code to not rely on these side-effects

Schedule / Timing

M35

- SSLServerSocketOpenSSL
- SSLClientSocketOpenSSL for Windows, OS X
  - OpenSSL support for asynchronous certificate validation
  - EVP interface for CryptoAPI/CNG
  - EVP interface for Keychain Services/CDSA
- Gather metrics on Linux smart card usage
- Work with internal teams on determining scope/effect Pepper/Plugin Cleanup

M36-M37

- Certificate Path Building (NSS)
  - Corpus tests to ensure libpkix-or-better path building
  - Performance tests
  - Switch from libpkix to this library on iOS, ChromeOS, and Linux
- PKCS#11 Integration Library (OpenSSL)
- Continued Pepper/Plugin Cleanup

M37

- Certificate Path Building (OpenSSL)
- Switch to OpenSSL on iOS, ChromeOS
- Switch to dual-stack implementation of NSS & OpenSSL on Linux
    - OpenSSL default, NSS available via command-line flag
    - Goal is to identify and work out issues with smart cards

M38

- Remove NSS from Linux