MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Bakalářská práce

Grafické rozhraní pro OpenSSL

Petr Lefner

Brno, 2006

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Poděkování

Na tomto místě bych rád vyjádřil své poděkování dr. Zdeňku Říhovi za poskytnuté rady a vstřícný postoj v projednávaných otázkách řešení této práce.

Vedoucí práce: Ing. Mgr. Zdeněk Říha, Ph.D.

Shrnutí

Tento dokument se snaží motivovat k pokročilejší práci s knihovnou OpenSSL jejím úvodním představením z hlediska celkové struktury a významu. V následujících dvou kapitolách stručný průnik do poskytovaného API doplňuje řada příkladů a postupů, jimiž je napomoženo programátorovi k seznámení se s klíčovými úkoly implementace zabezpečené síťové komunikace v OpenSSL.

Přiloženo je také CD obsahující zdrojový kód aplikace pro Microsoft Visual Studio .NET, která prostřednictvím grafického uživatelského rozhraní prezentuje OpenSSL v praxi. Svou funkčností je postavena na zdrojových kódech aplikace `openssl`, dodávaných společně s balíkem zdrojových kódů knihovny.

Klíčová slova

OpenSSL, tutoriál, SSLeay, crypto, API, příklady, šifra, certifikát, SSL/TLS spojení, bezpečnost

Keywords

OpenSSL, tutorial, SSLeay, crypto, API, examples, cipher, certificate, SSL/TLS connection, security

Obsah

1 Úvod

1.1 Anotace

Ochrana abstraktních hodnot v moderní informační době dosáhla tak vlivného postavení, že jako bezpečnostní cíl ji nelze stavět na nižší příčku, než požadavek na ochranu hodnot materiálních. Ve sféře Internetu a počítačové komunikace jako takové se cesty k jejímu naplnění rozvinuly do podoby obsáhlé škály bezpečnostních protokolů, složitých výpočtů a standardů, aby se sešly do série schopných projektů různého charakteru. Jedním z nich je právě OpenSSL se svou nespočetnou základnou uživatelů a mnoha příklady reálného nasazení na malých klientských stanicích i velkých podnikových serverech mnoha platforem.

Tato práce je strukturována jako dokumentační analýza knihovny OpenSSL, současně ovšem nápadně připomíná tutoriál čili výukový dokument. V první fázi totiž představuje knihovnu z hlediska modularity, kdy metodicky vychází z koncepce programů jazyka C, v kterém je knihovna napsána, a letmo seznamuje čtenáře s dílčími knihovnami balíku OpenSSL distribuovaného ve verzi 0.9.8a.

V druhé fázi přechází text k samotnému programování a analýze API historicky nejstarší části OpenSSL, totiž knihovny SSLeay. Zde výklad sleduje funkce a datové struktury tak, jakoby čtenář začínal s psaním klientského nebo serverovského programu schopného vytvářet bezpečná síťová spojení podle variant protokolu SSL.

V poslední fázi práce je podobným způsobem zdokumentována knihovna crypto, zajišťující „background“ hlavní knihovně SSLeay. Vzhledem k jejímu nesmírnému rozsahu se ale zaměřuje pouze na některé její důležitější partie – knihovny ERR, BIO a X509.

Praktickou součástí práce je aplikace prezentující schopnosti konzolového programu `openssl`; její podrobnější dokumentace je zařazena v příloze.

1.2 Cíl práce

Text dokumentu je koncipován jako analýza struktury knihovny OpenSSL i jejího rozhraní. Svou formou se však snaží ulehčit orientaci v široké škále exportovaných funkcí či datových struktur. Zjednodušeným výkladem principů jejich použití a uvedením názorných příkladů tak navíc druhotně usiluje o motivaci neobeznámeného čtenáře k hlubšímu studiu projektu OpenSSL.

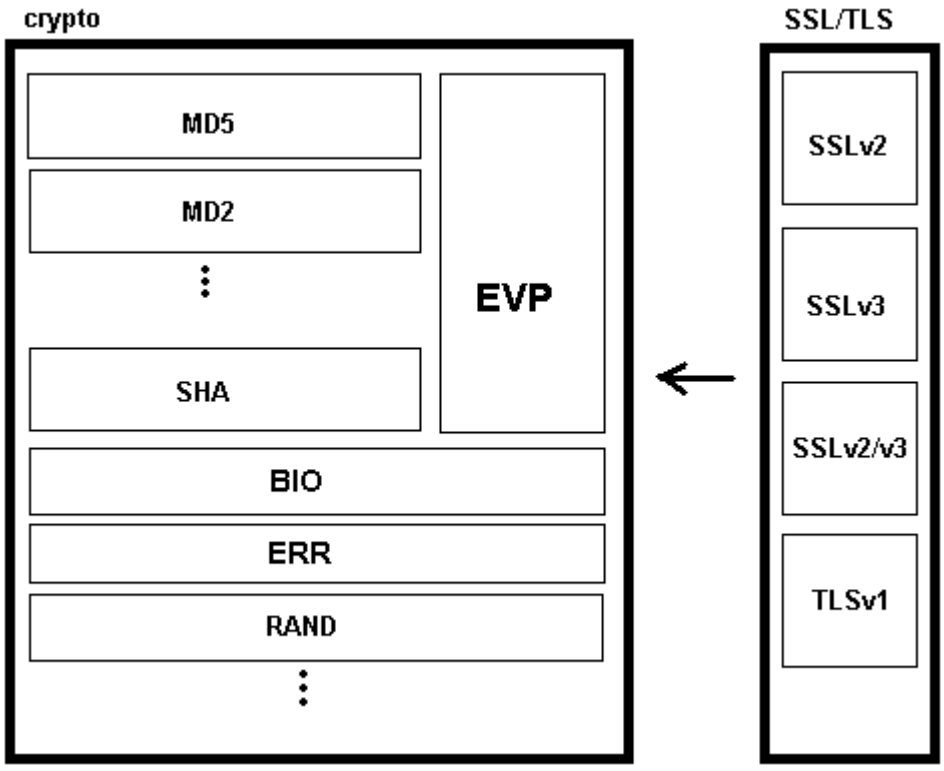
2 Struktura knihovny

V této kapitole bude přiblížena modulární struktura knihovny OpenSSL. V krátkých popisech se snaží nastínit hierarchii a zamýšlený význam hlavních modulů.

2.1 Moduly a hierarchie

Knihovna OpenSSL je sloučením tří oddělených komponent. Historický základ představují knihovny **crypto** a **SSLeay (SSL/TLS)**. S nárůstem složitosti se později izoluje ještě balík aplikací pod označením **openssl**.

Schéma knihovny OpenSSL



2.1.1 Knihovna SSLeay

Implementuje specifikace kryptografických protokolů Secure Sockets Layer (SSL v2.0 a SSL v3.0) a Transport Layer Security (TLS 1.0, RFC 2246) pro

zabezpečenou komunikaci v síti Internet. Je klíčovým pilířem celého projektu, ačkoli poskytované API nemusí být jediným použitelným rozhraním v OpenSSL (nižší rozhraní nejsou omezeny třídami viditelnosti). V hierarchii ovšem vystupuje nejvýše. Funkce a datové struktury tohoto API se exportují s prefixem „SSL_“, podrobněji budou popsány v kapitole „SSLeay API“. SSLeay pro realizaci svého účelu interně používá funkce a datové struktury z rozhraní knihovny **crypto**, popsané v kapitole 2.1.2.

2.1.2 Knihovna crypto

Implementuje jednotlivé kryptografické algoritmy používané knihovnou SSLeay. Každá taková implementace je separovatelná, lze také říci, že crypto je sestavena z řady volitelně vkládaných kryptografických podknihoven. Součást API dotváří několik podpůrných knihoven, vlastní zpracování vstupně-výstupního rozhraní a další spíše interně používané funkce.

2.1.2.1 Kryptografické knihovny

Jednotlivé knihovny implementují algoritmy symetrické, asymetrické a hašovací kryptografie, a funkce pro práci s certifikáty. Jejich konkrétní seznam je uveden v kapitolách „Moduly crypto“. Následující tabulky udávají prefixy funkcí a datových struktur exportovaných do API knihovny crypto pro jednotlivé podknihovny.

Tabulka 1: Knihovny symetrické kryptografie

Knihovna	***Prefix***
blowfish	BF_
cast	CAST_
des	DES_
idea	idea_
rc2	RC2_
rc4	RC4_
rc5	RC5_

Tabulka 2: Knihovny asymetrické kryptografie

Knihovna	***Prefix***
dsa	DSA_
dh	DH_
rsa	RSA_

Tabulka 3: Knihovny pro práci s certifikáty

Knihovna	***Prefix***
x509	X509_
x509v3	X509V3_

Tabulka 4: Knihovny hašovacích funkcí

Knihovna	***Prefix***	***Knihovna***	***Prefix***
hmac	HMAC_	mdc2	MDC2_
md2	MD2_	ripemd	RIPEMD160_
md4	MD4_	sha	SHA_, SHA1_
md5	MD5_		

2.1.2.2 Podpůrné knihovny

Knihovny **Err** pro záznam a hlášení chyb, **Rand** pro generování náhodných čísel a knihovna **Threads** pro podporu OpenSSL v aplikacích pracujících ve vláknech. Do API exportují funkce a datové struktury s odpovídajícími prefixy podle tabulky.

Tabulka 5: Podpůrné knihovny

Knihovna	***Prefix***
err	ERR_
rand	RAND_
threads	CRYPTO_

2.1.2.3 Knihovny vstupu a výstupu

Pro dosažení přenositelnosti mezi platformami je zavedeno vlastní abstrakcí I/O rozhraní **BIO**. Vzhledem k nasazení OpenSSL především do sféry internetové komunikace je BIO doplněno dalšími knihovnami pro kódování dat – **PEM** zavádí funkce pro práci s daty ve formátu PEM (zde ve smyslu „kódováno v *base64* a ohraničeno hlavičkami“) podle specifikace *Privacy Enhanced Mail*. Podobně **ASN1** implementuje standard *Abstract Syntax Notation One* (ASN.1, X.680:1995). **PKCS7** a **PKCS12** implementují standardy *Cryptographic Message Syntax Standard* a *Personal Information Exchange Syntax Standard* z rodiny standardů PKCS (*Public Key Cryptography Standards*). Knihovna **EVP** (Digital Envelope Library) je wrapper k dříve zmiňovaným kryptografickým funkcím z crypto API. Autoři doporučují jeho použití namísto přímého volání funkcí z API podknihoven. Odpovídající prefixy funkcí a datových struktur exportovaných do API jsou v tabulce.

Tabulka 6: Knihovny vstupu a výstupu, kódovaní

Knihovna	***Prefix***	***Knihovna***	***Prefix***
asn1	ASN1_	pem	PEM_
bio	BIO_	pkcs7	PKCS7_
evp	EVP_	pkcs12	PKCS12_

2.1.2.4 Interní knihovny

Šifrovací algoritmy často vyžadují celočíselné výrazy o vysokých řádech, sadu operací k tomu poskytuje knihovna **Bignum** (BN). **Lhash** implementuje datovou strukturu hašovací tabulka. Nejen pro potřeby knihovny BIO je zavedena knihovna **Buffer**, která implementuje jednoduchý znakový buffer. Odpovídající prefixy jsou uvedeny v tabulce.

Tabulka 7: Interní knihovny

Knihovna	***Prefix***
bignum	BN_
lhash	LHASH_, lh_
buffer	BUF_MEM_

2.1.3 Balík aplikací openssl

Vyčlenil se jako souhrn binárních spustitelných souborů a jejich zdrojových kódů reprezentujících knihovnu OpenSSL programově. Zdrojové kódy programů jsou sestaveny tak, že je možné kompilovat každý program zvlášť do jednotlivých binárních souborů nebo vytvořit jediný spustitelný soubor ze zvolených aplikací. Tento soubor se spouští ve třech režimech. Pokud je jeho název (v prostředí DOS bez přípony) „openssl“, spustí program uvedený svým názvem v prvním parametru. Bez uvedení parametru spustí svůj interpret a očekává příkazy. Nakonec je také možné soubor přejmenovat na název zvolené aplikace a efekt je stejný jako v prvním případě. Stručný popis programů je uveden v dalších kapitolách.

2.2 Moduly SSLeay

SSLeay jako hlavní knihovna svými moduly zpřístupní i API knihovny crypto.

Ze série inkluzí postačí ve většině případů použít pouze globální hlavičkový soubor `ssl.h`.

2.2.1 ssl.h

Hlavní hlavičkový soubor SSLeay API sám připojuje i níže uvedené soubory knihovny a všechny potřebné hlavičkové soubory z rozhraní knihovny **crypto**.

2.2.2 ssl2.h

Implementace protokolu SSL 2.0. Vkládáno v ssl.h.

2.2.3 ssl3.h

Implementace protokolu SSL 3.0. Vkládáno v ssl.h.

2.2.4 ssl23.h

Kombinace protokolů SSL 2.0 a SSL 3.0 Vkládáno v ssl.h.

2.2.5 tls1.h

Implementace protokolu TLS 1.0. Vkládáno v ssl.h.

2.2.6 kssl.h

Podpora mechanismu Kerberos podle RFC 2712. Implicitně vkládáno v ssl.h.

2.3 Moduly crypto – symetrické šifrování

2.3.1 blowfish.h

Bloková šifra užívající klíče z roku 1993, která se v současné době řadí mezi nejrychlejší šifry ve své kategorii. Viz [S1] (stránky autora algoritmu).

2.3.2 cast.h

Další bloková šifra používající klíče z roku 1996. Existuje ve dvou variantách, CAST-128 a CAST-256. Knihovna crypto v diskutované verzi OpenSSL implementuje pouze první z uvedených variant, v níž je velikost bloku nastavena na 64 bitů a největší délka klíče může být 128 bitů. Viz RFC 2144.

2.3.3 des.h

Data Encryption Standard (DES), známá šifra z počátku 70. let minulého století. Nedoporučuje se její použití vzhledem k malé délce klíče. Od verze 0.9.7 knihovny OpenSSL je k dispozici také zpětně kompatibilní modul vázaný se starší verzí implementace DES – des_old.h. V současné verzi 0.9.8a obsahuje 15 módů/variant šifry (včetně Tripple-DES, DESX a unixovského `crypt`) tak, že vyhovuje ANSI X3.106 (módy operací pro DES); implementován je i nástupce známý jako Advanced Encryption Standard (AES)[1].

Tabulka 8: Symetrické šifry OpenSSL [3]

Algoritmus	***Typ***	***Délka klíče (implicitně)***	***Rychlost k DES***	***Podporované módy***
Blowfish	blokový	(128bit)	3x	ecb, cbc, 3ecb, 3cbc, cfb, ofb
CAST	blokový	128bit	1.5x	ecb, cbc, cfb, ofb
DES	blokový	8bit	1x	ecb, cbc, cfb, ofb, pcbc
AES	blokový	128bit	>3x	ecb, cbc, cfb, ofb
IDEA	blokový	(128bit)	1x	ecb, cbc, cfb, ofb
RC2	blokový	(128bit)	1x	ecb, cbc, cfb, ofb
RC4	proudový	(128bit)	4x	ecb
RC5	blokový	(128bit)	2x	ecb, cbc, cfb, ofb

2.3.4 idea.h

International Data Encryption Algorithm (IDEA), bloková šifra z roku 1991 nahrazující DES. Před jejím nasazením v komerční sféře je třeba vzít v potaz důsledky plynoucí z patentové ochrany, která platí pro vybrané evropské země do r. 2010 a pro USA/Japonsko do r. 2011.

2.3.5 rc2.h

Bloková šifra z roku 1987, slabší předchůdce RC šifer. Původní implementace v knihovně se inspiruje příspěvkem z Usenet fóra sci.crypt z roku 1996, kterým byla odhalena specifikace této šifry. Současná verze již vyhovuje originální specifikaci (viz RFC 2268).

1 Implementace AES ovšem není dosud oficiálně zdokumentována, tabulka tak uvádí odvozená fakta.

2.3.6 rc4.h

ARCFOUR, stále ještě prakticky použitelná a používaná proudová šifra. Knihovna implementuje neoficiální verzi specifikace.

2.3.7 rc5.h

Bloková šifra z roku 1994, v crypto je podporována pouze kombinace 64bitový blok, 8, 12 nebo 16 cyklů, a nejvýše 255 bitů dlouhý klíč. Algoritmus je patentován v USA, datum expirace není stanoveno.

2.4 Moduly crypto – asymetrické šifrování

2.4.1 dsa.h

Digital Signature Algorithm (DSA), implementace vyhovuje standardům FIPS 186 pro DSS a ANSI X9.30.

2.4.2 dh.h

Diffie-Hellmanův protokol výměny klíčů podle RFC 2631 a PKCS #3.

2.4.3 rsa.h

RSA public-key encryption algorithm, implementace podle specifikace SSL a standardu PKCS #1 v2.0. Umí generovat nové páry veřejných a soukromých klíčů.

2.5 Moduly crypto – hašování

2.5.1 hmac.h

Keyed-hash message authentication code (HMAC), podle RFC 2104.

2.5.2 md2.h

Message Digest Algorithm 2 (MD2), hašovací funkce podle RFC 1319.

2.5.3 md4.h

MD4, hašovací funkce podle RFC 1320.

2.5.4 md5.h

MD5, hašovací funkce podle RFC 1321. Vzhledem k nálezům několika příkladů kolizních posloupností již není považována za dostatečně dokonalou a

od verze 0.9.8 (z června 2005) autoři doporučují v aplikacích používat k podpisu certifikátů apod. raději RIPEMD-160 nebo SHA-1 a vyšší.

2.5.5 mdc2.h

Modification Detection code (MDC2), hašovací funkce podle ISO/IEC 10118-2 s DES. Originální specifikace je pod patentovou ochranou, která má vypršet až v roce 2007. Z tohoto důvodu je implicitně pro překlad vynechána a není k dispozici ani např. v OpenSSL dodávaném s RedHat v7.1 v USA.

2.5.6 ripemd.h

RACE Integrity Primitives Evaluation Message Digest (RIPEMD-160), hašovací algoritmus podle ISO/IEC 10118-3:2003.

2.5.7 sha.h

SHA-1, *Secure Hash Algorithm*. Implementace vyhovuje FIPS PUB 180-1 (Secure Hash Standard) a ANSI.X9.30. Pro možnost zpětné kompatibility je definována i předchozí verze – SHA (SHA-0) (podle FIPS PUB 180). Od verze 0.9.8 sem spadá také implementace algoritmů SHA-224, SHA-256, SHA-384 a SHA-512 (podle FIPS 180-2).

2.6 Moduly crypto – certifikáty

2.6.1 x509.h

Funkce pro práci s certifikáty a CRL (Certificate Revocation List) podle RFC 3280 resp. ITU-T Recommendation X.509 (1997 E), a také související Certificate Request podle RFC2896. Obsahuje i rutiny pro převod mezi datovými formáty PEM a DER a podporu pro systém SPKI provozovaný v Netscape. K realizaci zabezpečení modul používá šifru RSA, DH nebo DSA z knihoven asymetrické kryptografie dh.h, rsa.h a dsa.h.

2.6.2 x509v3.h

Zavádí podporu pro používání rozšíření X.509 v3 a CRL v2 podle RFC 3280.

2.7 Moduly crypto – podpůrné

2.7.1 err.h

Chybový systém pro OpenSSL používaný řadou modulů. Chyby se ukládají do fronty asociované se svým procesním vláknem, lze jim definovat řetězce pro chybová hlášení s přesným názvem knihovny a funkce, v níž k chybě došlo. Protože je koncipován nezávisle na OpenSSL, je možné tuto knihovnu použít i ve vlastních programech.

2.7.2 threads.h

Podpora knihovny OpenSSL v multithreadových programech – zamykání sdílených dat a dynamické zámky. Přestože některé partie kódu dynamické zámky údajně potřebují pro lepší výkonnost, nejsou zatím v diskutované verzi aktivně používány.

2.7.3 rand.h

Generátor pseudonáhodných čísel nutný především pro generování klíčů. Iniciální hodnoty pro generátor musí dodat aplikace např. z pohybu myši, knihovna ale nabízí jejich ukládání do souboru a odtud je i číst.

2.7.4 opensslv.h

Definuje verzi knihovny.

2.7.5 ui.h

Představuje interface pro uživatelské rozhraní nad zbytkem knihovny, aby bylo možné ji ovládat v nejrůznějších podobách.

2.8 Moduly crypto – vstup/výstup, kódování

2.8.1 asn1.h

Abstract Syntax Notation One (ASN.1). Standard neurčuje kódování popisované informace, v OpenSSL se ovšem pracuje s formátem DER (Distinguished Encoding Rules). DER je specializovaný model formátu BER (Basic Encoding Rules), knihovna **asn1** je tedy schopná přistupovat i k datům v základním formátu. Zavedené funkce jsou nutné pro X509 (patří mezi ně rutiny pro podepisování struktur asociovaných s X.509), PKCS7 a PKCS12.

2.8.2 bio.h

Wrapper (nebo přesněji abstrakce) nad vstupem a výstupem s ohledem na platformní odlišnosti. Vedle práce se soubory přináší možnost transparentně ustavovat a jinak spravovat nekryptovaná síťová spojení i kryptovaná SSL spojení. Koncepce rozlišuje dva typy jednotek – zdrojové/zakončovací jednotky, jimiž mohou být různé I/O kanály (soubor, soket, paměťový buffer aj.), a filtry připojované k jednotkám prvního typu. Řazení jednotek do řetězu zajišťuje transparentnost operací, což přináší více pohodlí do kódu programu. BIO je primárním prvkem pro veškeré I/O v OpenSSL a podobně jako ERR je schopná samostatného využití.

2.8.3 evp.h

Digital Envelope Library, wrapper ke kryptografickým knihovnám pro jednodušší a přehlednější kód. Nabízí také algoritmus Base64.

2.8.4 pem.h

Rutiny ke kódování digitálních dat podle Privacy Enhanced Mail.

2.8.5 pkcs7.h

Rutiny ke kódování dat podle PKCS #7, používající ASN.1 popisy.

2.8.6 pkcs12.h

Rutiny ke kódování dat podle PKCS #12, používající ASN.1 popisy.

2.9 Moduly crypto – interní funkce

2.9.1 bn.h

Knihovna pro aritmetiku s čísly teoreticky neomezené svým bitovým rozsahem. Obsahuje také generátor prvočísel a rutinu k testování na prvočíslo. Samostatná knihovna, v OpenSSL nutná např. pro knihovny RSA, SHA a DH.

2.9.2 buffer.h

Implementace jednoduchého bufferu znaků.

2.9.3 lhash.h

Lhash Table Library, dynamická hašovací tabulka. Knihovna ji používá

především pro spravování SSL/TLS relací.

2.9.4 object.h

Pomocná knihovna pro ASN1. Jedná se o sadu funkcí překládajících objektové identifikátory na konkrétní datové typy.

2.9.5 stack.h

Implementace datového typu „zásobník“.

2.9.6 txt_db.h

Knihovna pro spravování jednoduché textové databáze udržované v paměti. Vznikla v rámci účelu odzkoušet správu certifikátů (aplikace openssl).

2.9.7 conf.h

Není uvedena jako podpůrná knihovna, přesto jako podpůrná vznikla. Jde o jednoduchý systém pro spravování konfigurace programů nad OpenSSL prostřednictvím souboru nebo proměnných prostředí. Jako většina popsaných modulů, je i CONF nezávisle využitelná.

2.10 Aplikace openssl

Aplikace openssl je program příkazové řádky reprezentující OpenSSL. Lze jím

- vytvářet parametry klíčů pro RSA, DH a DSA,
- zpracovávat a vytvářet certifikáty X.509, Certificate Signing Request a Certificate Revocation List
- počítat hash zprávy (Message Digest)
- šifrovat a dešifrovat pomocí běžných algoritmů symetrické kryptografie
- provozovat zkušební SSL/TLS server se zkušebním SSL/TLS klientem
- šifrovat/dešifrovat, ověřovat a podepisovat S/MIME e-mailové zprávy

3 SSLeay API (výběr)

Tato kapitola představuje knihovnu SSLeay z pohledu její užitečnosti. Přibližuje stěžejní příklady použití v několika jmenovaných kategoriích, jež se snaží naznačit eventuální cesty k řešení hledaného problému. Přestože tedy nese název vhodný spíše k řádnému výčtu funkcí a datových struktur, smysl by měl být (alespoň částečně) podobný; pouze ve vybraných případech tento zvolený způsob prezentace knihovny prolíná do synopse a popisu poskytovaných funkcí a datových struktur.

3.1 Funkce volby metody protokolu SSL_METHOD

Funkce vztahující se k objektu SSL_METHOD

V knihovně OpenSSL každé SSL/TLS spojení staví svoje parametry na tzv. SSL kontextu. Funkce volby metody protokolu se uplatňují při přípravě tohoto kontextu, tj. při vytváření struktury `SSL_CTX`. Mají za úkol vymezit interní okruh operací tak, aby byl implementován příslušný protokol SSLv2, SSLv3 nebo (D)TLSv1.

Pro každý případ protokolu je zavedena trojice funkcí – dvě jsou dedikovány pro kontext použitelný v režimu serveru respektive klienta, jedna pro případ kontextu univerzálního. Dohromady je tedy zavedeno dvanáct funkcí.

Příklad: Vytvoření klientského kontextu protokolu typu SSL 3.0 pomocí SSLv3_client_method(), jaký se může objevit např. v rámci úkolu „vytvoření SSLv3 spojení klient-server“:

```
SSL_METHOD* meth = SSLv3_client_method();           // set-up the
method
SSL_CTX *ctx = SSL_CTX_new(meth);                   // set-up SSL
context
SSL *connection = SSL_new(ctx);                     // set-up the
connection
...
```

Ilustrace 1: SSLv3_client_method()

Viz také: `SSL_CTX_new()`, `SSL_CTX_set_ssl_version()`, `SSL_METHOD` *in 3.6*

3.2 Funkce šifer, SSL_CIPHER_*

Funkce zaměřené na manipulaci s šiframi (objekty SSL_CIPHER)

V této partii rozhraní SSLeay exportuje čtveřici funkcí informujících o atributech objektu `SSL_CIPHER`. Jejich sémantický dopad na vlastní proces není nijak přímo čitelný; svou roli hrají v dohodě klienta a serveru na metodě, jíž budou následující spojení zabezpečena. K vráceným informacím náleží slovní popis (`SSL_CIPHER_get_description()`), název (`SSL_CIPHER_get_name()`), bitový rozsah klíče (`SSL_CIPHER_get_bits()`) a verze protokolu v němž je šifra registrována (`SSL_CIPHER_get_version()`).
Viz také: `SSL_CIPHER` *in 3.7*

3.3 Funkce kontextu, SSL_CTX_*

Funkce zaměřené na manipulaci s SSL kontextem (objekt SSL_CTX)

Okolo sedmdesátky funkcí vesměs simuluje zapouzdřenost objektu `SSL_CTX`, tj. přiřazuje nebo získává hodnoty interních atributů, jimiž se knihovna řídí při zakládání SSL/TLS spojení, vzájemné autentizaci obou stran či šifrování jejich komunikace – stručněji řečeno, při implementaci ostatních cílů plynoucích ze specifikace toho kterého protokolu.

3.3.1 Vytvoření kontextu: `SSL_CTX_new()`

Synopse: `SSL_CTX *SSL_CTX_new(SSL_METHOD *method)`

Nový SSL kontext (strukturu `SSL_CTX`, viz kapitola 3.8) pro daný protokol vytvoří `SSL_CTX_new()`; tento „konstruktor“ (či přesněji „simulace konstruktoru v jazyce C“) přiřadí atributům struktury určitou implicitní hodnotu, takže je připraven pro tvorbu objektů `SSL`, které v OpenSSL popisují vlastní SSL/TLS spojení. Alokovaný objekt pak přizpůsobí stanoveným požadavkům jeho následná inicializace. Její průběh, popsaný kapitolou 3.8, doplňují některými detaily následující kapitoly.
Příklady: Ilustrace 1
Viz také: `SSL_CTX` in 3.8, `SSL_new()`

3.3.2 Přiřazení soukromého klíče: `SSL_CTX_use_Private_Key()`

Jedním z mnoha úkolů pro implementaci některého z SSL protokolů je zajištění důvěryhodnosti mezi komunikujícími stranami prostředky asymetrické

kryptografie. V OpenSSL je toho dosaženo inicializací kontextu a provedením *handshake* podle specifikace nastaveného protokolu (viz kapitola 3.8).

V rámci *autentizace mezi klientem a serverem* je tak použito soukromého klíče. V knihovně jsou k dispozici dvě trojice funkcí, kterými je ke kontextu připojen konkrétní klíč; funkce se liší očekávaným typem klíče (RSA nebo jiný) a zdrojem (paměť, soubor, struktura `EVP_PKEY`).[2]
Příklady: Ilustrace 2, Ilustrace 3

3.3.3 Heslo šifrovaného PEM souboru: `SSL_CTX_set_default_password_cb()`

Synopse:
```
void SSL_CTX_set_default_passwd_cb(SSL_CTX *ctx, pem_passwd_cb *cb)
  void SSL_CTX_get_default_passwd_cb_userdata(SSL_CTX *ctx, void* u)
    int pem_passwd_cb(char *buf, int size, int rwflag, void *userdata)
```

Soukromý klíč vystupující v autentizaci bývá chráněn svým heslem. Knihovna jej od aplikace získává voláním callback funkce `pem_passwd_cb()`[3] pokaždé před uložením nebo načtením klíče. Tento callback nastavuje kontextu právě `SSL_CTX_set_default_password_cb()`.
Příklady: Ilustrace 2

3.3.4 Připojení certifikátu: `SSL_CTX_use_certificate()`

Poslední nevyhnutelný krok v autentizaci mezi klientem a serverem představuje použití certifikátu připojeného ke kontextu. V inicializaci je proto k dispozici trojice funkcí načítajících certifikát z různých zdrojů (paměť, soubor, struktura `X_509`) a pro přípravu k autentizaci kompletním řetězem certifikátů funkce `SSL_CTX_add_extra_chain_cert()` nahrávající řetěz po jednotlivých souborech.[2] Příklad č. 2 připojuje celý strom certifikátů uložený v jednom souboru pouze pomocí `SSL_CTX_use_certificate_chain_file()`.
Příklady: Ilustrace 2, Ilustrace 3

---

2 Tyto funkce mají také svou obdobu mezi funkcemi pro strukturu SSL – jejich název se liší prefixem (SSL_* místo SSL_CTX_*). Sémanticky se nejedná o redundanci, protože změny kontextu se aplikují na spojení pouze při `SSL_new()`.

3 Parametr **rwflag** je roven 0, pokud se bude klíč dešifrovat.

Ilustrace 2: Inicializace SSL kontextu pro připojení klienta k autentizovanému serveru

```
#define CERTFILE          „client_cert.pem“
#define KEYFILE           „client_pkey.pem“

static char* PASSWORD =   „password“;
int pem_passwd_cb(char *buf, int size, int rwflag, void *userdata);

/* Initialize given SSL_CTX */
void _init_ssl_ctx(SSL_CTX* ctx)
{
      /* enable SSLv2 or SSLv3 */
      SSL_METHOD meth_sslv23 = SSLv23_method();
      SSL_CTX_set_ssl_version(ctx, meth_sslv23);

      /* load certificate chain */
      SSL_CTX_use_certificate_chain_file(ctx, CERTFILE);
      /* load private key */
      SSL_CTX_use_PrivateKey_file(ctx, KEYFILE, SSL_FILETYPE_PEM);
      /* set the password callback */
      SSL_CTX_set_default_passwd_cb(ctx, password_cb);
      /* set trusted CAs */
      SSL_CTX_load_verify_locations(ctx, CA_LIST, 0);
}

/* password callback */
int pem_passwd_cb(char *buf, int size, int rwflag, void *userdata)
{
      int len = size<strlen(PASSWORD)?size:strlen(PASSWORD);
     strncpy(buf, PASSWORD,len);
     return(len);
}
```

3.3.5 Módy autentizace: `SSL_CTX_set_verify()`

Inicializaci kontextu doplňuje volitelná změna módu, v němž proběhne verifikace certifikátů v budoucích spojeních. Základní testy korektnosti posílaných certifikátů a důsledek jejich výsledků provádí každá strana spojení podle příznaků specifikovaných funkcí `SSL_CTX_set_verify()`. Pokud má neúspěšný výsledek vést k přerušení handshake, klient či server musí mít nastaven `SSL_VERIFY_PEER`. Protože autentizace klienta je pouze možností (vynutitelnou právě tímto příznakem), musí server navíc použít `SSL_VERIFY_FAIL_IF_NO_CERT`, pokud klient svůj certifikát vůbec nenabídne – jinak bude handshake pokračovat bez indikace chyby.

`SSL_set_verify()` také dovoluje projektu řešit verifikaci podle svých představ přesměrováním adresy na související callback; bližší detaily uvádí originální dokumentace k diskutované funkci.

3.3.6 Změna implicitní metody protokolu: `SSL_CTX_set_ssl_version()`

Synopse: `int SSL_CTX_set_ssl_version(SSL_CTX *ctx, SSL_METHOD *m)`

I v situaci, kdy je v paměti již zavedeno a eventuálně i otevřeno několik spojení, může kontext měnit svoje vlastnosti. Např. `SSL_CTX_set_ssl_version()` změní kontextu stávající metodu protokolu, přitom existujícím objektům SSL vytvořeným z tohoto kontextu se metoda protokolu nezmění.

Příklady: Ilustrace 2

3.3.7 Používání systému správy relací: `SSL_CTX_set_session_cache_mode()`

Pojem (s*ystém) správy relací* zavádí tento dokument jako označení pro sadu funkcí a mechanismů, která svým účelem implementuje pravidla charakterizující SSL/TLS relaci podle specifikací protokolů SSLv3 a TLSv1. Jak zmiňuje kapitola 3.9, některé postupy plynoucí z pravidel specifikace knihovna automatizuje, a to jak pro klienta, tak pro server.

`SSL_CTX_set_session_cache_mode()` usměrňuje chování knihovny v oblasti správy SSL/TLS relací – *sessions*. Ve svých parametrech očekává ukazatel na kontext a některý z příznaků ovlivňujících využití *session cache* tohoto kontextu v souvislosti s prováděním *handshake*. Z širšího úhlu pohledu význam této funkce spočívá v možnosti volby – knihovna ponechává aplikaci prostor pro vlastní řešení databáze relací, pokud se snad jeví vlastnosti knihovny **lhash** (jejími prostředky knihovna tabulku relací realizuje) jako nevýhodné nebo nedostačující. Tabulka 9 osvětluje význam jednotlivých příznaků.

Příklady: Ilustrace 7

Tabulka 9: Příznaky ovládání session cache

Příznak	***Význam***
`SSL_SESS_CACHE_OFF`	Relace nejsou automaticky kešovány.
`SSL_SESS_CACHE_CLIENT`	Kešují se relace vznikající při `SSL_connect()`.
`SSL_SESS_CACHE_SERVER`	Kešují se relace vznikající při `SSL_accept()`.
`SSL_SESS_CACHE_BOTH`	Platí oba příznaky výše.
`SSL_SESS_CACHE_NO_AUTO_CLEAR`	Session cache se implicitně vyprazdňuje po každých 255 voláních `SSL_connect()` a `SSL_server()`. Nastavením příznaku se předává kontrola nad velikostí cache aplikaci (funkce

Příznak	*Význam*
	`SSL_CTX_flush_sessions()` a `SSL_CTX_sess_set_cache_size()`).
`SSL_SESS_CACHE_NO_INTERNAL_LOOKUP`	Pouze pro server; k opětovnému použití uložené session není prohledávána session cache, ale pouze externí zdroj (byl-li kontextu nastaven příslušný callback pomocí `SSL_CTX_sess_set_get_cb()`). Poskytování relací tak zůstává pouze na aplikaci: buď čtením dat z paměti ve formátu ASN.1 (`d2i_SSL_SESSION()`), nebo ze souboru ve formátu PEM (`PEM_read_SSL_SESSION()`).
`SSL_SESS_CACHE_NO_INTERNAL_STORE`	Implicitně se do session cache ukládají relace z externího zdroje a relace vznikající při `SSL_connect()` a/nebo `SSL_accept()`. Příznak zamezí tomuto chování a ukládání do session cache zůstává pouze na aplikaci.
`SSL_SESS_CACHE_NO_INTERNAL`	Platí oba předchozí příznaky.

3.3.8 Ruční kešování relací: `SSL_CTX_add/remove_session()`

Synopse:
```
int SSL_CTX_add_session(SSL_CTX *ctx, SSL_SESSION *c);
int SSL_CTX_remove_session(SSL_CTX *ctx, SSL_SESSION *c);
```

Popis: Do tabulky relací v SSL kontextu ctx zařadí prvek **c** (pouze jedenkrát – identičnost se rozlišuje podle ukazatele a *session id*), resp. z tabulky relací kontextu **ctx** uvolní prvek **c**.

Knihovna exportuje funkci `SSL_CTX_sessions()` vracející ukazatel na *session cache* kontextu, současně ale nabízí elegantnější nástroje k manipulaci s jejími prvky a rutiny v nadpisu představují ty nejčastěji používané. Celý smysl těchto dvířek k tajným informacím o SSL/TLS spojení vyplyne v dalších kapitolách, jako součást systému relací mohou např. předcházet opětovnému použití *session*. Příklad za všechny nabízí Ilustrace 3.

3.3.9 Externí session cache callbacky: `SSL_CTX_sess_set_*_cb()`

K otázce ručního kešování patří také problém správy nějakého vlastního, externího úložiště. Kontext lze konfigurovat tak, aby k ukládání i čtení relací vůbec nepoužíval svou vlastní cache, a v tom případě je nutný nějaký mechanismus, kterým dojde k propojení mezi cílem (či zdrojem) zaváděných SSL relací a vlastním SSL spojením. K vybudování takového mechanismu má API připraveno několik callback funkcí, jež musí aplikace implementovat a s daným

SSL kontextem svázat:

- `new_session_cb()` má danou session vložit do externí cache daného spojení
- `remove_session_cb()` má danou session z cache odebrat
- `get_session_cb()` má vrátit objekt `SSL_SESSION`, jemuž odpovídá dané *session id*, nebo `NULL`, pokud takový v cache neexistuje. Tento callback používá pouze server, protože pouze server zahajuje *session resumption*.

Příklady implementací této trojice metod lze nalézt ve zdrojových kódech OpenSSL aplikací `s_client.c` a `s_server.c`.

3.3.10 Export/import relací: `i2d_SSL_SESSION()`, `d2i_SSL_SESSION()`

Pokud aplikace hodlá spravovat systém SSL relací sama nebo relace z nějakého důvodu zálohovat (např. kvůli sdílení dat v případě serveru replikujícího svůj proces pro obsluhu klientů), nevyhne se používání externích úložišť. Poněkud mimo zavedené partie rozhraní knihovny SSLeay vyčnívá dvojice funkcí zaměřených na tento problém: `i2d_SSL_SESSION()` a `d2i_SSL_SESSION()`.

`i2d_SSL_SESSION()` transformuje objekt `SSL_SESSION` na jeho reprezentaci ve formátu ASN.1 a tu uloží na určené paměťové místo, odkud už ji lze např. zapsat na disk. Druhá funkce, `d2i_SSL_SESSION()`, provádí opačný proces – pokusí se transformovat data na určeném paměťovém místě do nové SSL relace. Export/import do/z souboru ve formátu PEM provádí dvojice rutin `PEM_write_SSL_SESSION()` resp. `PEM_read_SSL_SESSION()`, které zavádí knihovna PEM.

Příklad: Server vytvoří a přijme SSLv3 spojení, v němž pro handshake nabídne *session cache* s relacemi načtenými z paměti (jako blok dat v ASN.1) a ze souboru (jako data ve formátu PEM). Poté uloží použitou *session* na standardní výstup.

Ilustrace 3: SSLv3 server a session cache

```
/* setup a context */
meth=SSLv3_method();
ctx=SSL_CTX_new(meth);
/* Load some session from binary into the cache */
session=d2i_SSL_SESSION(....)
SSL_CTX_add_session(ctx,session);

/* Lets even add a session from a file */
session=PEM_read_SSL_SESSION(....)
SSL_CTX_add_session(ctx,session);

/* Create a new SSL structure */
```

```
ssl=SSL_new(ctx);

/* At this point we want to be able to 'create' new session if required,
 * so we need a certificate and RSAkey. */
SSL_use_RSAPrivateKey_file(ssl,...)
SSL_use_certificate_file(ssl,...)

/* Now since we are a server, it makes little sense to load a session
 * against the ssl structure since SSL_accept() will either create a new
 * session or grab an existing one from the cache. */

/* grab a socket descriptor */
fd=accept(...);

/* associate it with the ssl structure */
SSL_set_fd(ssl,fd);

/* 'do' SSLv3 handshake using a certificate and a RSA key */
SSL_accept(ssl);

/* Lets print out the session details */
PEM_write_SSL_SESSION(SSL_get_session(ssl),stdout,....);
```

3.3.11 Platnost relací: `SSL_CTX_set_timeout()`, `SSL_CTX_get_timeout()`

Použitelnost SSL relací je protokolárně omezena na dobu *timeout*, která může uplynout od okamžiku vzniku této relace. Funkce `SSL_CTX_set_timeout()` specifikuje hodnotu *timeout* pro všechny relace, které budou přidány do *session cache* daného kontextu, stávající relace přitom změna tohoto intervalu neovlivňuje.

Viz také: `SSL_SESSION_set_timeout()`, `SSL_SESSION_set_time()`

3.3.12 Setřásání tabulky relací SSL kontextu: `SSL_CTX_flush_sessions()`

Jistě je samozřejmé, že systém správy relací musí ošetřovat paměťová omezení stanovená tabulce relací každého SSL kontextu a případných externích úložišť. Ukládané relace mají svou časovou platnost, ale protože ji lze měnit, není přesně jasné, kdy má knihovna relace z tabulek odebírat. Zároveň pouze specifikace programované aplikace podává nějaká fakta o paměťovém zatížení podle předpokládaného počtu vzniklých spojení, takže je právě na ní, aby setřásání tabulek obstarala.

`SSL_CTX_flush_sessions()` projde *session cache* určeného kontextu a odstraní záznamy neplatných relací s vypršeným *timeout*. Rozhodující podmínkou pro odstranění záznamu je mezní čas vyjádřený parametrem **tm** –

obvykle je za něj dosazeno volání `time()`.

Pokud specifikace projektu nevypovídá nic o intenzitách zátěže nebo tento problém zkrátka aplikace zcela opomíjí, provádí knihovna setřásání tabulky po každém 255. volání `SSL_connect()` resp. `SSL_accept()`, ale pouze je-li automatické správě relací kontextu nastaven příznak `SSL_SESS_CACHE_NO_AUTO_CLEAR` (viz `SSL_CTX_set_session_cache_mode()`).

3.4 Funkce SSL/TLS relace, SSL_SESSION_*

Funkce zaměřené na manipulaci s SSL relací (objekt SSL_SESSION)

3.4.1 Nastavení časové známky: `SSL_SESSION_set_time()`

Když knihovna alokuje nový objekt `SSL_SESSION` (viz kapitola 3.9), ukládá do něj také údaj vypovídající o čase, v kterém struktura vznikla. `SSL_SESSION_set_time()` tuto časovou známku mění na hodnotu parametru odpovídající počtu sekund „since the Epoch“ – tak jak se typicky rozumí v prostředí UNIX/Linux. `SSL_SESSION_get_time()` vrací časovou známku relace odkazované v parametru.

3.4.2 Nastavení timeout: `SSL_SESSION_set_timeout()`

Hodnota *timeout* (v sekundách) spolu s časovou známkou vymezují dobu platnosti SSL relace ve vztahu k jejím opětovným použitím pro rychlý handshake. Při vzniku objektu `SSL_SESSION` je diskutovaný interval nastaven na protokolárně specifickou hodnotu, kterou vrací `SSL_get_defaut_timeout()`.

3.4.3 Uvolňování objektů `SSL_SESSION`: `SSL_SESSION_free()`

`SSL_SESSION_free()` dekrementuje počet referencí na blok dat reprezentujících `SSL_SESSION`, aby mohl být uvolněn jakmile počet odkazů dosáhne nuly.

Aplikace by měla funkci volat po každé explicitní inkrementaci počtu referencí na alokovanou strukturu. K tomu dochází při importu SSL relace prostřednictvím `d2i_SSL_SESSION()`, pokud se neodehrává v callback funkci nastavené `SSL_CTX_sess_set_get_cb()` – tehdy referenci přebírá knihovna. Explicitně je inkrementován počet referencí také při `SSL_get1_session()`.

Viz také: `SSL_SESSION`, `SSL_get_session()`

3.5 Funkce SSL/TLS spojení, SSL_*

Funkce zaměřené na manipulaci se spojením (objekt SSL)

3.5.1 Inicializace knihovny, `SSL_library_init()`, `SSL_load_error_strings()`

Synopse:
```
int SSL_library_init(void)
void SSL_load_error_strings(void)
```

Dříve než je vytvořen SSL kontext a proběhne jeho inicializace, měla by aplikace provést také inicializaci knihovny OpenSSL. `SSL_library_init()` zastupuje rodinu příbuzných funkcí podřízených tomuto účelu. Ve skutečnosti pouze registruje názvy všech kryptografických algoritmů do implicitního seznamu šifer, proto by se tedy její volání mělo vyskytovat na začátku. K explicitnímu použití libovolné šifry (jakým je např. volání `EVP_sha1()`) není tato inicializace nutná.

Příklad: Obvykle program začíná načítáním řetězců chybových hlášení knihovny, registrací kryptografických algoritmů a inicializací generátoru pseudonáhodných čísel.

Ilustrace 4: Inicializace knihovny (podle originální dokumentace [D2])

```
SSL_load_error_strings();                /* readable error messages */
SSL_library_init();                      /* initialize library */
actions_to_seed_PRNG();
...
```

3.5.2 Vytvoření SSL spojení: `SSL_new()`

Synopse:
```
SSL *SSL_new(SSL_CTX *ctx)
```

Tato funkce vytvoří nové spojení podle kontextu **ctx** (neboli asociuje spojení s kontextem, tzn. že vlastnosti kontextu se kopírují na vzniklé spojení). Ke každému `SSL_new()` by mělo být uzavírající `SSL_free(SSL *s)`.

Příklady: Ilustrace 1

Viz také: `SSL`, `SSL_CTX`

3.5.3 Propojení objektu SSL s I/O kanály: `SSL_set_bio()`, `SSL_set_fd()`

Spojení vytvořené voláním `SSL_new()` neovládá komunikaci koncových entit, dokud není propojeno s vstupním a výstupním BIO. Jeho typ je libovolný; často se jedná o socket, existují ovšem i programy testující funkcionalitu SSL/TLS v OpenSSL na paměťových bufferech, které poskytuje knihovna BIO. Je namístě uvědomit si, že charakter BIO určuje chování `SSL_connect()`/`SSL_accept()` – je-li blokující, dotyčná funkce vyčká dokud nebude dokončen handshake nebo dokud nenastane chyba; s neblokujícím BIO-propojením se vrací funkce okamžitě.

Příklad: Spojení klient-server pomocí soketů. Vytvoříme nadstavbovou vrstvu SSL/TLS protokolu propojenou deskriptorem TCP/IP soketu.

Ilustrace 5: SSL_set_fd(). SSL/TLS nad existujícím protokolem, klient-server.

```
SSL *con = SSL_new(a_ssl_context);
int s;

/*TODO: that usual socket(), [bind()] connect() stuff here */
/* Having socket 's', connect it with SSL/TLS mechanism */
SSL_set_fd(con,s);
SSL_connect(con);

/*TODO: SSL_read(), SSL_write() stuff here */

SSL_shutdown(con);
SSL_free(con);
```

Podobným způsobem je implementován SSL/TLS nad server-klient TCP/IP spojením v ilustraci 6.

Ilustrace 6: SSL_set_fd(), SSL_use_certificate(). SSL/TLS nad existujícím protokolem, server-klient

```
SSL* con = SSL_new(a_ssl_context);
int s;

- do normal socket(), bind(), listen(), accept()

SSL_set_fd(con, s);

/* specify private key */
SSL_use_RSAPrivateKey_file(con, "server_rsa.pem", SSL_FILETYPE_PEM);

/* specify certificate */
SSL_use_certificate_file(con, "server_cert.pem", SSL_FILETYPE_PEM);
SSL_accept(con);

- then use SSL_read() and SSL_write() rather than
  read() and write()

e.g.
  SSL_read(con,buf,1024);
    for the client example this will return the 4 bytes
    written "bye\n" (possibly in multiple packets)
```

Viz také: `SSL_connect()`

3.5.1 Navázání SSL/TLS spojení, iniciace handshake se serverem: `SSL_connect()`

Funkce `SSL_connect()` zahájí *SSL/TLS handshake* klienta se serverem a podle charakteru BIO-propojení vrací výsledek okamžitě, respektive až při výskytu chyby nebo úspěšném dokončení. Návratová hodnota `SSL_connect()` rovná číslu 1 pak indikuje úspěšný *handshake*, číslu 0 protokolárně odůvodněnou chybu, a záporné číslo znamená jinou chybu. Přesnější informaci o chybě poskytne funkce `SSL_get_error()`.

Po zdařeném *handshake* už lze na objektu `SSL` provádět I/O operace pomocí `SSL_write()` a `SSL_read()`, přesto v této chvíli nemusí být navázané spojení ze strany serveru bezpečné. Klientská aplikace by ještě měla dokončit autentizaci serveru postupem uvedeným v kapitole 3.5.7.[4]

Příklady.: Ilustrace 5

Viz také: `SSL_set_bio()`, `SSL_accept()`

4 Podle specifikací SSL/TLS protokolů jsou autentizace serveru i autentizace klienta součástí handshake. OpenSSL však nedokáže implementovat veškeré kontroly automaticky (jsou aplikačně závislé), proto nabízí řadu funkcí k jejich provedení ze strany aplikace.

3.5.2 Navázání SSL/TLS spojení, čekání serveru na handshake: `SSL_accept()`

Funkce čeká na iniciaci *handshake* ze strany klienta a podle charakteru nastaveného komunikačního BIO-kanálu vrací výsledek. Návratová hodnota má význam podobný `SSL_connect()`.

Po dokončení `SSL_accept()` už lze na spojení provést veškeré I/O operace nebo v případě potřeby dokončit handshake autentizací klienta – např. podle postupu popsaného v kapitole 2.5.8.

Příklady: Ilustrace 6

Viz také: `SSL_set_bio()`, `SSL_accept()`

3.5.3 Uzavření SSL/TLS spojení: `SSL_shutdown()`

Specifikace SSLv2 a TLSv1 protokolů definují dva způsoby, jak uzavřít existující spojení; v OpenSSL jsou oba implementovány funkcí `SSL_shutdown()`, skutečné přerušení svého konce I/O kanálu pak knihovna ponechává na aplikaci – stejně jako jeho otevření.

Spojení pracující podle protokolu SSLv2 ukončí aplikace voláním `SSL_shutdown()` a následným uzavřením zápisového konce svého I/O kanálu. Druhá entita ve spojení obdrží zprávu *close_notify* a reaguje stejně. Podle protokolu TLSv1 musí klient i server vědět, že jeden z nich chce spojení ukončit, aby se předešlo narušení bezpečnosti třetí stranou. Entita uzavírající spojení jako první by tedy měla uvedením dalšího volání `SSL_shutdown()` počkat na doručení *close_notify* od svého partnera. Na druhou stranu není dodržení tohoto postupu nutné (po odeslání *close_notify* je SSL relace uzavřená a tudíž platná po zbytek svého *timeout*), takže program může ušetřit čas pro jiné úkoly.

3.5.4 Autentizace serveru po `SSL_connect()`:

V rámci *handshake* knihovna provádí základní kroky k ověření bezpečnosti spojení se serverem, vyšší úrovně bezpečnosti však zajišťují aplikací prováděné kontroly řetězce certifikátů, pod nímž server v SSL/TLS spojení vystupuje.

Nejdůležitějším z těchto testů, které by měly následovat ihned po SSL_connect(), je kontrola identity serveru podle údajů v jeho certifikátu. Dokument RFC2818 protokolu HTTPS uvádí dva způsoby ověření identity – polem `commonName` a rozšířením `subjectAltName` typu `dNSName`. Níže jsou uvedeny příklady obou způsobů, jimiž lze kontrolu v OpenSSL provést.

Příklad: Kontrola identity serveru polem Common Name

Klient se ujistí o validitě certifikátu serveru použitého v *handshake* z výsledku `SSL_get_verify_result()` a jeho kopii alokuje do objektu typu X509 funkcí `SSL_get_peer_certificate()`. Pole `commonName` z objektu do znakového bufferu extrahuje `X509_NAME_get_text_by_NID()`[5]. Získaný řetězec musí být shodný s adresou HOST, na kterou se klient připojoval. Ukázkový kód podává příklad č. 7.

Příklad: Kontrola identity X.509 rozšířením dNSName/iPAddress

Klient získá podobně jako v předchozí variantě serverův certifikát, tentokrát ale potřebuje extrahovat X.509 rozšíření `subjectAltName` a zjistit hodnotu pole `dNSName` resp. `iPAdress`, pokud je server identifikován IP adresou místo DNS adresou. Kód k extrakci požadovaných data je poněkud složitější a protože používanými funkcemi souvisí spíše s knihovnou **X509**, vše podstatné obsahuje demonstrační příklad uvedený jako (kapitola č. , strana).

Ve většině případů se aplikuje kontrola polem Common Name, ale pokud je zavedeno, má rozšíření `dNSName` přednost.

5 Symbol `NID_commonName` i ostatní symboly, které označují standardní pole certifikátu podle RFC 3280, definuje hlavičkový soubor **objects.h**.

Ilustrace 7: SSL_get_peer_certificate(), SSL_get_verify_result: Kontrola identity serveru polem Common Name v certifikátu

```
SSL_connect(ssl);
X509* server_cert;
/* NULL indicates no certificate is set or it couldn't be sent */
if (server_cert = SSL_get_peer_certificate(ssl))
{
    if (SSL_get_verify_result(ssl) == X509_V_OK)
    {
        char buf[256];
        /* obtain Common Name into buf */
        X509_NAME_get_text_by_NID (X509_get_subject_name(server_cert),
                                   NID_commonName, server_cert, 256);
        /* compare values */
        if (strcasecmp(buf, HOST))
            BIO_puts(bio_err, „Server identity not trusted!“);
    }
    /* free-up X509 at the end ! */
    X509_free(server_cert);
}
```

3.5.5 Autentizace klienta po `SSL_accept()`:

Oba protokoly SSL i protokol TLSv1 charakterizují autentizaci klienta jako volitelnou operaci. Nezajistí-li klient svůj certifikát, jehož údaje server prověří stejným způsobem, kterým klient autentizuje server, rozhoduje o dalším postupu specifikace projektu pro stranu serveru.

3.5.1 Přiřazení relace pro klientské spojení: `SSL_set_session()`

Klient v TLSv1 spojení může rychleji dokončit *handshake* se serverem, pokud si uloží relaci získanou z prvního spojení. Funkce `SSL_set_session()` nabídne tuto relaci pro příští `SSL_connect()` a některé fáze handshake tak budou „vynechány“. Nabízená *session* musí být přítomná v *session cache* serveru a mít platný *timeout*. Pokud pochází z *session cache* určitého SSL kontextu klienta, nelze ji nabídnout pro spojení asociované s jiným SSL kontextem.[6]

Příklad: Po otevření SSL/TLS spojení je možné získat všechny informace nutné k autentizaci. Nepodaří se je ale zneužít pro cizí SSL kontext, protože objekty *session cache* se váží i s kontextem, do něhož cache patří. Otevřít druhé spojení se

6 Klientská aplikace zjistí reakci serveru na nabídku z výsledku `SSL_session_reused()`. Funkce vrací `1` v kladném případě, `0` pokud musela být vytvořena nová relace.

nepodaří také pokud předtím nedošlo k uzavření prvního spojení (nelze použít tutéž relaci na dvě otevřená spojení)[7].

Ilustrace 8: SSL_set_session(), SSL_get_session(): Rychlá autentizace klienta bez handshake

```
/* Let OpenSSL store newly created client-sessions into session cache */
SSL_CTX_set_session_cache_mode(con, SSL_SESS_CACHE_CLIENT);
/* Create initial connection 'con' created from context 'ctx' */
SSL* con = SSL_new(ctx);
SSL_set_fd(con, socket);
/* Establish the connection; newly created session will be stored into
 * cache */
SSL_connect(con);
SSL_shutdown(con);
/* Get opened session */
SSL_SESSION* sess = SSL_get1_session(con);
/* Create secondary connection from the context 'ctx2'. */
SSL* con2 = SSL_new(ctx2);
SSL_set_fd(con2, socket);
/* Re-use session from the first connection to avoid handshake */
SSL_set_session(con2, sess);
SSL_connect(con2); // will FAIL, use 'ctx' instead of 'ctx2'
SSL_shutdown(con2);
/* Decrement reference count */
SSL_SESSION_free(sess);
...
```

3.5.10 Získání relace z otevřeného SSL/TLS spojení: `SSL_get_session()`

Synopse:
```
SSL_SESSION *SSL_get_session(const SSL* ssl)
SSL_SESSION *SSL_get0_session(const SSL* ssl)
SSL_SESSION *SSL_get1_session(SSL* ssl)
```

Vrací ukazatel respektive konstantní ukazatel na SSL relaci charakterizující otevřené SSL/TLS spojení. Třetí funkce vylučuje neplatnost ukazatele inkrementací počtu knihovnou kontrolovaných referencí, proto by mělo na konci dojít k dekrementaci explicitním voláním `SSL_SESSION_free()`.
Příklady: Ilustrace 8

3.6 Datová struktura SSL_METHOD

metoda SSL protokolu

Objekt `SSL_CTX` používá strukturu `SSL_METHOD` (též *metoda SSL protokolu*) jako metodu spojení.. Existují tři typy takových metod – první typ je univerzální

7 Chce-li aplikace použít stejné prostředky k otevření druhého spojení, stačí kopírovat *session* prvního spojení na druhé ve funkci `SSL_copy_session_id()`.

pro server i klienta, druhý pouze pro server a třetí typ pouze pro klienta. Viz ilustrace č.1.

3.7 Datová struktura SSL_CIPHER

SSL šifra

Tato struktura (též *SSL šifra)* obecně popisuje algoritmus konkrétní šifry (viz funkce). Objekt `SSL_CTX` spravuje seznam objektů `SSL_CIPHER`, z něhož jsou později vybírány šifry k zabezpečené komunikaci v SSL/TLS spojení. Výběr skutečně používané instance probíhá na základě její priority (vyplývá z pozice v seznamu) a dohody obou zapojených entit během handshake.

Seznam šifer podporovaných příštími SSL/TLS spojeními nastavuje danému SSL kontextu `SSL_CTX_set_cipher_list()`.

Ilustrace 9: SSL_CTX_set_cipher_list()

```
SSL_CTX_set_cipher_list(ctx,SSL3_TXT_RSA_RC4_40_MD5);
```

3.8 Datová struktura SSL_CTX

SSL kontext

Význam

Struktura SSL_CTX (též *SSL kontext*) určuje vlastnosti každého nového síťového spojení (viz `SSL` in 2.10), které je s ní asociováno. Asociací se zde rozumí přechod informací (atributů) kontextu na spojení (objekt `SSL`) při jeho vytvoření (`SSL_new()`). Kontext tak ovlivňuje průběh navazování spojení (*handshake*) i čtení nebo zápis dat v tomto spojení. Sdílením kontextu mezi více spojeními je docíleno efektivnějšího kódu.

Zmíněnými informacemi v objektu `SSL_CTX` jsou obecně

- metoda protokolu `SSL_METHOD`
- seznam šifer (používané k zabezpečení posílaných dat, nastavován též implicitně),
- tabulka objektů `SSL_SESSION` (orig. *session cache,* implicitně prázdná)
- adresy funkcí zpětného volání (orig. *callbacks*, některé mají implicitní handlery),
- soukromé/veřejné klíče, klientské certifikáty a seznam certifikačních autorit

(nutné pro autentizaci),

- další vlastnosti pro SSL/TLS spojení (příznaky, nastavovány též implicitně).

Pro ustanovení autentizovaného SSL/TLS spojení je třeba kontextu připravit potřebné informace o autentizačních prostředcích, tj. o certifikátech, certifikačních autoritách, soukromých klíčích a heslech. K těmto úlohám se vztahují základní inicializační rutiny `SSL_CTX_use_PrivateKey()`, `SSL_CTX_use_certificate()`, `SSL_CTX_set_default_passwd_cb()` a jejich varianty.

Inicializace kontextu SSL_CTX

Spojení SSL musí být autentizováno pomocí páru veřejného/soukromého klíče a certifikátu – to platí pro SSL server i pro SSL klienta (pokud je/má být autentizace schopen). Určením zdroje certifikátu, klíče a callback funkce k získání hesla v kontextu ovšem jeho inicializace nutně nekončí. Kontextu lze pozměnit i další vlastnosti, včetně metody spojení – `SSL_CTX_set_ssl_version()` v ilustraci č.2 dodatečně připustí spojení SSLv3 nebo SSLv2 namísto původního nastavení. V některých případech se kontextu mění také implicitně nastavený *callback* pro verifikaci certifikátu a různé příznaky.

Souhrn kroků:

- ✔ Nastavit cestu k certifikátu pomocí `SSL_CTX_use_certificate_chain_file()`. S kontextem jsou svázány i související certifikační autority (CA) tvořící řetězec, jež připojují příbuzné funkce API popsané kapitolou 3.3. Samotný průběh verifikace certifikátů vychází z módu autentizace, pozměnitelného funkcí `SSL_CTX_set_verify()`.
- ✔ Nastavit cestu k soukromému klíči, SSL_CTX_use_PrivateKey_file().
- ✔ Určit metodu pro získání hesla, je-li soukromý klíč zašifrován – nastavení callback funkce, SSL_CTX_set_default_password_cb().
- ✔ Specifikovat důvěryhodné servery – příklad demonstruje SSL klienta, který se autentizuje jen určitým serverům, a to podle seznamu důvěryhodných certifikačních autorit (CA). Jejich seznam stanoví volání `SSL_CTX_load_verify_locations()`.

3.9 Datová struktura SSL_SESSION

SSL relace

Význam

Objekt SSL_SESSION (*též „SSL relace“*) spravuje tajné informace o existujícím SSL/TLS spojení; obvykle k nim patří šifra zvolená k zabezpečení dat a použité autentizační prostředky, tak jak definují specifikace SSLv3 a TLSv1. Vzniká na straně serveru i klienta jako atribut objektu SSL poté, co se dvojice entit v SSLv3/TLSv1 spojení vzájemně uspokojivě autentizuje při handshake podle podmínek plynoucích z dohodnutého protokolu.

Reference na nově vytvořený objekt knihovna ukládá do hašovací tabulky – *session cache*. Tabulka vzniká pro každý nově vytvořený kontext a je sdílena každým objektem `SSL`, který je s tímto kontextem asociován. Server v ní při `SSL_accept()` hledá relace pro další použití; klient iniciuje toto hledání na straně serveru při `SSL_connect()`. Předtím ovšem klientská aplikace musí explicitně určit relaci, která má být na serveru nalezena, pomocí `SSL_set_session()`.

Proces kešování SSL/TLS relací lze provozovat aplikačně nebo automaticky nastavením patřičných příznaků pomocí `SSL_CTX_set_session_cache_mode()`, přičemž obě řešení mohou prolínat.

Opětovné použití SSL/TLS relace

Opětovné použití relace (viz *session resumption* ve specifikaci) je pravděpodobně nejčastější úlohou spojenou s kešováním `SSL_SESSION`. Následující podkapitoly uvádí souhrn kroků uzpůsobujících serverovskou i klientskou aplikaci k této schopnosti pro oba přístupy v kešování.

Automatizované řešení session resupmtion

Následující výčet kroků napovídá postup, kterým server i klient plně využijí automatizované správy relací v API knihovny. Oproti tomu následující podkapitola načrtne postup, který se snaží zavést vlastní řešení správy relací a automatice knihovny se spíše vyhnout. Volba mezi oběma možnostmi může padnout i na určitou tolerantní úroveň, v níž se řešení bude pohybovat někde mezi oběma protipóly. Mezi rozhodující argumenty pro ruční kontrolu patří např. nutnost zavést určitou hierarchii mezi ukládanými relacemi, nebo

požadavek organizovat SSL relace v nějakém databázovém manažeru podle souvisejícího protokolu jako např. HTTP, SMTP aj.

➔ **Konfigurace na straně serveru**

- ✔ Zajistit kešování relací, tzn. nastavit SSL kontextu příznak `SSL_SESS_CACHE_SERVER`, a nenastavovat příznak `SSL_CACHE_NO_INTERNAL_STORE`.
- ✔ Zajistit výběr relací k opětovnému použití – nekonfigurovat kontext k zákazu vyhledávání v interní cache příznakem `SSL_SESS_CACHE_NO_INTERNAL_LOOKUP`.
- ✔ Voláním `SSL_CTX_set_session_id_context()` kontextu spojení přiřadit libovolnou posloupnost bajtů, která se bude připojovat k session id relací z něj vytvářených. Tato data, označovaná jako *session id context*, musí být jedinečná pro každý objekt `SSL_CTX`, jinak může dojít k chybě (klient si bude při *session resumption* korektnost hodnot ověřovat).
- ✔ Nechat knihovnu kontrolovat velikost session cache tím, že se kontextu nenastaví příznak `SSL_SESS_CACHE_NO_AUTO_CLEAR`. Podle potřeby ale může aplikace upravit kontextu nejvyšší velikost tabulky relací z implicitní hodnoty (1024*20 bajtů) funkcí `SSL_CTX_sess_set_cache_size()`.

➔ **Konfigurace na straně klienta**

Lze říci, že opětovnému použití SSL relace dává smysl pouze server; na druhou stranu bez iniciativy klienta podle následujících kroků k ní pro něj vůbec nemusí dojít:

- ✔ Nastavit SSL kontextu příznak `SSL_SESS_CACHE_CLIENT`.
- ✔ Z nového SSL spojení získat vytvořený objekt `SSL_SESSION` voláním funkce `SSL_get_session()`. Ten může aplikace zálohovat do paměti jako ASN.1 popis prostřednictvím `i2d_SSL_SESSION()`, nebo zapsat na disk jako soubor ve formátu PEM pomocí `PEM_write_session()`.
- ✔ Po korektním uzavření prvního spojení (`SSL_shutdown()`) a vytvoření druhého spojení získanou relaci nabídnout serveru pro příští *handshake* voláním `SSL_set_session()`. Objekt `SSL_SESSION` může aplikace extrahovat z ASN.1 popisu transformací `d2i_SSL_SESSION()`, nebo ji načíst z PEM souboru použitím makra `PEM_read_SSL_SESSION()`.

Aplikační řešení session resumption

Projekt konstruující vlastní systém správy relací touto cestou také nesmí zapomenout na bezpečnost dat, která `SSL_SESSION` schraňuje. Pro interní tabulku relací by to neměl být problém, vše je uchováváno v paměti procesu; externích paměť už ale může představovat riziko. Příkladem z reálného světa je projekt Postfix – zde jsou relace ukládány na disk, přístup k nim má ale pouze uživatel patřící ke skupině „postfix" a pro jistotu je obvykle platnost relací omezena na dobu 300s pro HTTP/SSL resp. 3600s pro SMTP/SSL.

➔ **Konfigurace na straně serveru**

- ✔ Zajistit kešování relací nastavením příznaku `SSL_SESS_CACHE_SERVER` a ukládat novou relaci do cache kontextu pomocí `SSL_CTX_add_session()`. Požaduje-li specifikace projektu vlastní řešení úložiště, ukládání bude probíhat v implementaci callback funkce `new_session_cb()`, kterou s kontextem sváže `SSL_CTX_sess_set_new_cb()`. Pak může být správa relací upravena také příznakem `SSL_CACHE_NO_INTERNAL_STORE`.
- ✔ Zajistit výběr relací k opětovnému použití – pokud chce aplikace přejít k výhradně vlastnímu řešení úložiště (v referenční dokumentaci označovanému jako externí kešování), nastaví kontextu příznak `SSL_SESS_NO_INTERNAL_LOOKUP`, implementuje callback funkci `get_session_cb()` a pomocí `SSL_CTX_sess_set_get_cb()` nastaví kontextu její adresu.
- ✔ Kontextu spojení přiřadit *session id* postupem uvedeným v příbuzném oddíle pro automatizované řešení. Bude-li server chtít nějak automaticky generovat klíče *session id* připojované k *session id context*, může toto provádět v implementaci callbacku `GEN_SESSION_CB()` svázaného s kontextem rutinou `SSL_CTX_set_generate_session_id()`. Nabízená hodnota *session id* by měla být před přiřazením zkontrolována na unikátnost podle návratové hodnoty z funkce `SSL_has_matching_session_id()`; ta ale pracuje pouze s interní *session cache*, při externím kešování proto musí kontrolu svého úložiště provést aplikace sama.
- ✔ Volitelně kontrolovat velikost session cache: z interní tabulky relací aplikace může odebírat sama funkcemi `SSL_CTX_rem_session()` a `SSL_CTX_flush_sessions()` při nastaveném příznaku `SSL_SESS_CACHE_NO_AUTO_CLEAR`; používá-li projekt ruční kešování callbackem `new_session_cb()`, měla by také implementovat callback

funkci `remove_session_cb()` pro odebírání relace z vybraného externího úložiště – její adresu kontextu nastaví `SSL_CTX_sess_set_remove_cb()`.

➔ **Konfigurace na straně klienta**

✔ Ručně kešovat relace podle možností popsaných pro server.

✔ Z nového SSL spojení získat vytvořený objekt `SSL_SESSION` voláním funkce `SSL_get_session()`. Objekt může aplikace zálohovat do paměti jako ASN.1 popis funkcí `i2d_SSL_SESSION()`, zapsat na disk ve formátu PEM funkcí `PEM_write_session()`, nebo zkrátka využít svého wrapperu pro řešení externího kešování.

✔ Po korektním uzavření prvního spojení (`SSL_shutdown()`) a vytvoření druhého spojení získanou relaci nabídnout serveru pro příští *handshake* pomocí `SSL_set_session()`. Objekt `SSL_SESSION` může aplikace také extrahovat z ASN.1 popisu transformací `d2i_SSL_SESSION()`, nebo ji načíst z PEM souboru použitím `PEM_read_SSL_SESSION()`.

3.10 Datová struktura SSL

SSL spojení

Význam

Objekt struktury SSL spravuje informace o existujícím SSL/TLS spojení. Tyto informace na objekt přecházejí z kontextu SSL_CTX při svém vytvoření pomocí `SSL_new()`. Každé SSL spojení tak vstupuje do *handshake* s těmito údaji:

- certifikát (udává identitu entity)
- soukromý klíč
- dva páry šifer zabezpečujících komunikaci
- adresy *callback* funkcí k ověření certifikátu partnera
- BIO objekt spojující SSL/TLS nadstavbu s komunikačními I/O kanály
- *session id* předchozí SSL relace
- další vlastnosti SSL/TLS spojení (nepřevzaté příznaky jsou nastavovány implicitně)

Objekt typu `SSL` zaniká voláním `SSL_free()`.

Inicializace spojení SSL

Struktura v rámci své alokace přebírá většinu nastavení z asociovaného

kontextu, takže další série opatření tohoto charakteru nejsou nutná. Před zahájením handshake potřebuje objekt `SSL` znát pouze cestu, jíž budou protékat data a kterou dojde k propojení celého SSL/TLS mechanismu s nižší úrovní budovaného modelu – zde se většinou jedná o přiřazení nějaké koncové BIO entity nebo souborového deskriptoru/soketu pomocí `SSL_set_bio()` resp. `SSL_set_fd()`. Ale je-li to nutné, kopírované atributy může aplikace objektu upravit podobnými funkcemi, které se používají pro atributy SSL kontextu – v API je lze snadno rozpoznat podle shodující se minuskulové části názvu.

Příklad navázání klientského/serverovského SSL spojení s autentizací

Souhrn kroků:

- ✔ Inicializace knihovny – včlenění kryptografických algoritmů do seznamu povolených, `SSL_library_init()`
- ✔ Vytvořit klíčovou strukturu `SSL_CTX` – kontext spojení – a inicializovat ji (viz příklad v kapitole 2.8).
- ✔ K realizaci I/O chystané spojení svázat s párem vstupního a výstupního BIO. Na jeho charakteru bude záviset, zda funkce pro navázání spojení bude čekající nebo nečekající (orig. *blocking BIO/non-blocking BIO*). Viz `SSL_set_bio()`.
- ✔ Iniciace *handshake* – `SSL_connect()`/`SSL_accept()`.

4 crypto API (výběr)

Tato kapitola se snaží čtenáře seznámit s knihovnou „crypto“. Činí tak podobnou cestou, jako v kapitole 3. Zaobírá se opět pouze užším výběrem ze všech modulů, které knihovna obsahuje, a stejně stručně podává jejich logickou stavbu a princip použití přítomných funkcí.

4.1 BIO

I/O abstrakce

4.1.1 Vytvoření BIO entity: `BIO_new()`

Knihovna BIO realizuje I/O operace prostřednictvím univerzální datové struktury `BIO`. Zvláštní okruh schopností a vlastností jí vymezuje parameter typu `BIO_METHOD` v konstruktoru `BIO_new()`: finální objekt se tak stává zástupcem jednoho z podporovaných druhů entit (soubor, soket, …). Některé typy není nutné po vytvoření inicializovat, ale po dokončení práce by měly být vždy uvolněny funkcí `BIO_free()` nebo `BIO_vfree()`.

Následující odstavce představí nejdříve základní funkce implementující operace zápisu a čtení nad objektem `BIO`, poté přijdou na řadu konstruktory operačních metod, vázaných k jednotlivým entitám. Spolu s nimi budou také zmíněna související omezení na množině exportovaných funkcí. Příklady ke konkrétním metodám lze nalézt mezi ilustracemi v tomto dokumentu, více jich však nabízí originální dokumentace.

Ilustrace 10: "Hello world" (z originální dokumentace [D2])

```
BIO *out;
 out = BIO_new_fd(fileno(stdout), BIO_NOCLOSE);
 BIO_printf(out, "Hello World\n");
 BIO_free(out);
```

4.1.2 Zápis a čtení dat: `BIO_write/read/puts/gets()`

Rodina funkcí ke čtení a zápisu v podstatě vychází z tradičních funkcí standardní knihovny C – syntaxí i sémantikou se jedná o jakési přetížení (s ohledem na typ zdrojové/koncové entity). V tomto případě ovšem existuje

samozřejmý rozdíl např. mezi `BIO_read()` a klasickým `read()`, protože `BIO_read()` ke čtení používá právě `read()` (v jednom volání to může být dokonce několikrát).

4.1.3 Blokující a neblokující režim BIO: `BIO_should_retry()`

Charakter I/O kanálů vázaných na BIO nadstavbu způsobuje blokující resp. neblokující chování všech rutin v OpenSSL používajících vstupně-výstupních funkcí BIO abstrakce. I/O rutiny `BIO_*()` vracejí 0 nebo -1 v případě nějaké chyby, ale pro neblokující datové kanály se může jednat nikoliv o chybu během dané operace, ale prostě o žádost operaci opakovat. Zde se nedoporučuje řešit situaci systémovými voláními `select()` či `poll`() přímo na vázaný kanál, ale raději ověřit rozlišit platnost požadavku opakování od nějaké chyby pomocí knihovních maker v čele s `BIO_should_retry()` po každém neblokujícím volání.

4.1.4 Koncové entity BIO

Těmito typy entit se rozumí datové kanály podporované místní platformou, klasicky jde o souborový deskriptor, socket-deskriptor a ukazatel na paměťový buffer. Datové kanály lze pojímat také podobně jako roury, tedy na ryze vstupní a ryze výstupní část. K těmto všem entitám implementuje OpenSSL svou I/O nadstavbu BIO propojením v konstruktoru `BIO_new()` za použití příslušné BIO-metody, alokované podle metod uvedených v tabulce 10.

Tabulka 10: Koncové entity BIO a konstruktory jejich objektů `BIO_METHOD` pro `BIO_new()`

Metoda	***Datová entita***
`BIO_s_bio()`	Dvojice ze vstupní a výstupní BIO entity
`BIO_s_fd()`	Deskriptor souboru
`BIO_s_file()`	Soubor
`BIO_s_mem()`	Vyrovnávací paměť
`BIO_s_socket()`	Soket
`BIO_s_null()`	Unixovské `/dev/null`
`BIO_s_connect()`	TCP/IP spojení

Deskriptor k souboru v klasickém slova smyslu by se neměl používat pro sokety, především protože na platformách Windows* není soubor a soket totéž.

Umístění cílového souboru specifikuje objektu `BIO` funkce `BIO_set_fd()` v rámci inicializace. Pro systémové implicitní proudy `stdout` apod. je vhodné nastavit příznak BIO_NOCLOSE (např. v `BIO_new_fp()`), aby nedocházelo k pokusům o jejich uzavření při uvolňování objektu `BIO`.

Zvláštní pozornost bude tato kapitola věnovat pseudoentitě TCP/IP spojení, pro kterou rozhraní implementuje funkce umožňující vytvořit TCP/IP spojení bez odkazu do SSLeay API. Podrobněji tento problém zachycuje příklad níže.

Příklad: **TCP/IP spojení k HTTP serveru pomocí transparentní BIO abstrakce**

Následující souhrn kroků se vztahuje k ilustraci 11, v které je vytvořeno spojení k místnímu HTTP serveru a po zaslání požadavku o stránku je vypsána odpověď na standardní výstup. Vytvoření spojení i veškerou komunikaci přitom vykonává pouze wrapper integrovaný do knihovny BIO jako entita metody `BIO_s_connect()` a sada rutin získávajících nebo nastavujících DNS/IP adresu a port hostitele:

- ✔ Spojení vytvoří `BIO_new()` pro metodu `BIO_s_connect()` a inicializuje nastavení hostitelské adresy pomocí `BIO_set_conn_hostname/ip/port()`. Skloubení obou těchto dílčích kroků představuje `BIO_new_connect()`.
- ✔ Navázání spojení s hostitelem zahájí `BIO_do_connect()`, přitom je na místě pamatovat na charakter datových kanálů a v případě neblokujícího režimu rozlišit chybu od požadavku na opakování nebo dokončení operace spojení.
- ✔ HTTP požadavek odešle serveru `BIO_puts()`, která samozřejmě není dedikovaná pouze k tomuto účelu, takže řetězec požadavku musí sestavit aplikace explicitně. Příjem odpovědí provádí `BIO_read()` dokud bude délka dat nenulová. Entita spojení nepodporuje `BIO_gets()`, je ale možné vytvořit řetěz BIO entit připojením vyrovnávací paměti, jež tuto funkcí podporuje.
- ✔ Spojení uzavře prosté uvolnění entity v `BIO_free()`.

Ilustrace 11: `BIO_connect()`: TCP/IP pomocí BIO (z orig. dokumentace)

```
BIO *cbio, *out;
int len;
char tmpbuf[1024];
 ERR_load_crypto_strings();
 cbio = BIO_new_connect("localhost:http");
 out = BIO_new_fp(stdout, BIO_NOCLOSE);
 if(BIO_do_connect(cbio) <= 0) {
        fprintf(stderr, "Error connecting to server\n");
        ERR_print_errors_fp(stderr);
        /* whatever ... */
        }
 BIO_puts(cbio, "GET / HTTP/1.0\n\n");
 for(;;) {
        len = BIO_read(cbio, tmpbuf, 1024);
        if(len <= 0) break;
        BIO_write(out, tmpbuf, len);
 }
BIO_free(cbio);
BIO_free(out);
```

4.1.1 Řetězení entit, filtrující BIO: `BIO_push()`,`BIO_pop()`

Každou BIO entitu si lze jasně představit jako nějaký objekt interně určený svou operační metodou a jinými atributy, jeho externí reprezentace je však neméně významná. Zatímco vnitřně se mohou od sebe objekty lišit, z vnějšku se jeví jako invariantní entity se vstupním a/nebo výstupním datovým kanálem. To v důsledku přináší možnost entity zapojovat svými konci do řady bloků plnících určitou funkci.

Obecný řetěz tvoří nejméně jedna koncová entita a jeden BIO-filtr, plnící nějaké transformace na datech, jež mu posílá koncová entita. K filtru lze připojit také další filtr nebo koncovou entitu uzavírající řetěz z druhé strany. Implementačně připojení zajistí `BIO_push()` pro připojovanou entitu v prvním a nosnou entitu v druhém parametru. Odpojení provede `BIO_pop()`, uvolnění celého řetězce pak `BIO_free_all()`.

Ilustrace 12: Řetězení BIO entit, BIO_f_buffer()

```
BIO* file_bio = BIO_new_file(„./tmp.dat“, „rw“);    /* open a file */
BIO_set_close(file_bio, BIO_CLOSE);       /* be sure to close it when
freeing*/
BIO* bbio = BIO_new(BIO_f_buffer());                /* prepare buffering
BIO */
file_bio = BIO_push(bbio, file_bio);           /* make all file I/O
buffered */
…
BIO_flush(file_bio);         /* flush buffer */
BIO_free_all(file_bio);      /* free whole BIO chain */
```

Tabulka č. 11 vyjmenovává jednotlivé typy filtrujících entit a konstruktory jejich operačních metod pro `BIO_new()`. Všechny filtry i sadu relevantních rutin popisuje originální dokumentace, tento dokument je přiblíží pouze v ilustračních příkladech.

Tabulka 11: Filtrující entity BIO a konstruktory jejich BIO_METHOD objektů pro BIO_new()

Metoda	***Filtr***
`BIO_f_buffer()`	Vyrovnávací paměť
`BIO_f_cipher()`	Šifrování/dešifrování dat
`BIO_f_base64()`	Kódování/dekódovaní do/z base64
`BIO_f_md()`	Message digest na sadě dat
`BIO_f_ssl()`	SSL/TLS wrapper

Příklad: **SSL/TLS spojení pomocí transparentní BIO abstrakce**

Systém koncových entit a filtrů v řetězu nakonec svými schopnostmi spěje až k tomu, že není nijak náročné vytvořit a spravovat zabezpečené SSL/TLS spojení pomocí API knihovny BIO. Pro implementaci klienta stačí vytvořit z připraveného kontextu SSL filtr pomocí `BIO_new_ssl_connect()`, nastavit hostitelskou adresu, navázat spojení (stejně jako pro `BIO_s_connect()`) a poté ještě nechat funkci `BIO_do_handshake()` iniciovat handshake. Ke čtení a zápisu opět poslouží `BIO_puts()` a jí podobné. Implementace serveru obnáší více kroků:

- ✔ Z daného kontextu připravit nové spojení konstruktorem `BIO_new_ssl()`.

```
BIO* sbio = BIO_new_ssl(a_ssl_context, 0);
```

- ✔ Server potřebuje znát porty, na nichž má poslouchat – pro základní serverovskou entitu z prvního kroku je specifikuje `BIO_set_accept_bios()` jako zvláštní BIO entity vytvořené funkcí `BIO_new_accept()`. Jejím parametrem je číselný řetězec udávající právě číslo portu.

```
BIO* acptbio = BIO_new_accept(„80“);
BIO_set_accept_bios(acptbio, sbio);
```

- ✔ Dvojím voláním `BIO_do_accept()` inicializovat a spustit `accept()` smyčku (nekladná návratová hodnota znamená chybu).

```
if (BIO_do_accept(sbio)<=0) {/* err setting-up accept-bios */}
if (BIO_do_accept(sbio)<=0) {/* err accepting*/}
```

- ✔ Po navázání spojení přijde na řadu iniciace handshake a vlastní komunikace, a pokud má být přijato pouze jedno spojení (proces se nebude `fork()`ovat) , z *accept-BIO* se zde toto spojení vypouští a zbytek řetězce se uvolňuje. Nakonec spojení uzavře `BIO_free()`.

```
sbio = BIO_pop(acptbio);      /* obtain connection ssl-bio */
BIO_free_all(acpt);           /* won't accept more, free-up */
if (BIO_do_handshake(sbio)<=0) { /* err handshake */ }
BIO_puts(sbio, ...);
...
BIO_free();
```

4.2 ERR

Knihovna správy chyb

OpenSSL se odkazuje na API tohoto modulu v případě, že se během operací vyvolaných voláními z rozhraní knihoven SSLeay a crypto vyskytne nějaká chyba. Chybové kódy se zavádí ze strany programátora a teprve kompilací jsou sesbírány do interní organizace, formované s ohledem na přednostně informativní schopnosti knihovny ERR.

4.2.1 Inicializace systému ERR pro OpenSSL aplikaci

V rámci inicializace celé knihovny OpenSSL aplikace často konfigurují chybový systém, aby tak uživateli nebo autorům dovolily vysledovat příčiny a původ případných chyb. Konzolovým programům stačí voláním `SSL_load_error_strings()` nahrát řetězce chybových hlášení ze zdrojových

kódů OpenSSL[8], a později pouze vypsat asociované hlášení.

4.2.2 Interpretace chyb: `ERR_error_string()`

Každá chyba vyvolaná v některé z rutin API je knihovnou OpenSSL zaznamenána do fronty chyb, z které je odebíráno funkcí `ERR_get_error()`. Číslo, které vrátí, pak v parametru přebírá `ERR_error_string()` a pokud bylo dříve asociováno s nějakým řetězcem (pomocí `ERR_load_strings()`), bude tento nakopírován v druhém parametru.

Popis chyby v získaném řetězci má formát *error:[error code]:[library name]:[function name]:[reason string]*, z něhož vyplývá související chybový kód, název knihovny i název funkce, a vše je doplněno slovním vysvětlením. V praxi se často tato hlášení na obrazovce konzolové aplikace openssl kumulují tak, jak je jednotlivé moduly do globální fronty chyb zapisovaly[9].

Rozhraní exportuje kromě `ERR_error_string()` také další funkce, které obsah fronty chyb vyprazdňují nebo kopírují po jednom či všech záznamech. Cílem může být znakový buffer, BIO entita i souborový deskriptor.

4.3 X.509

Management certifikátů

4.3.1 X509 Certifikát

Certifikát jako datový objekt definuje po významové i obsahové stránce jeho specifikace (např. RFC3280), v OpenSSL implementovaná strukturou `X509` a mnoha souvisejícími funkcemi. V následujících řádcích budou tyto přestaveny podle charakteru své činnosti. Co vše tedy lze s certifikátem (objektem `X509`) dělat:

- vytvořit nový, starý uvolnit, duplikovat – `X509_new()`, `X509_free()`, `X509_dup()`,
- konvertovat z/do binárního formátu DER makry `d2i_X509()`, `i2d_X509()`,
- exportovat/importovat jako soubor v textovém, DER nebo PEM formátu – `X509_print[_fp](),d2i_X509_fp/bio(),i2d_X509_fp/bio()`,

8 Řetězce pouze pro crypto registruje volání `ERR_load_crypto_strings()`.

9 V některých místech ovšem aplikace openssl místo toho používá standardní `perror()` apod. Takové chyby nejsou tím pádem ve frontě zaneseny.

`PEM_write/read[_bio]()`,

- verifikovat, podepsat a vytvořit otisk makry `X509_verify()`, `X509_sign()` resp. `X509_digest()`.

4.3.2 Standardní a rozšiřující pole

Specifikace jednotlivých verzí X509v1/v2/v3 také popisují povinná a rozšiřující pole certifikátu; OpenSSL víceméně respektuje jejich identifikátory, jde tedy především o seznámení se s širokým spektrem obslužných rutin počínaje implementací ASN.1 a X509v3 konče:

- Hodnoty povinných polí získávají, nastavují a porovnávají funkce pojmenované ve stylu `X509_get_issuer_name()`, `X509_set_issuer_name()` a `X509_issuer_name_cmp()`. Kromě informačního významu mohou být hodnoty polí získávány např. za účelem porovnání s důvěryhodnými údaji, nastavovány např. v rámci programování vlastní certifikační autority nebo v aplikaci generující žádosti o certifikát.
  S myšlenkou na organizaci certifikátů v nějaké databázi nabízí digitální otisky některých těchto polí funkce jako `X509_issuer_name_hash()`. Vrácené údaje pak mohou posloužit jako klíč v tabulce nebo název souboru ve složce, když se vyhledává shodná hash z pole *subject name* při verifikaci certifikátů.
- Skutečný počet rozšíření a jejich pozici v zásobníku podle jednotlivých typů identifikátorů nebo vyžadovanosti obstarají `X509_get_ext_count()`, `X509_get_ext_by_NID()`, `X509_get_ext_by_OBJ()` respektive `X509_get_ext_by_critical()`. Blíže vysvětlují získání hodnot následující odstavce, jejich mazání či přidávání provádí `X509_delete_ext()` resp. `X509_add_ext()`.
  K certifikátu podle X509v3 volitelně náleží také definice a zavádění vlastních rozšíření, což je ale mimo rámec tohoto dokumentu[10].

4.3.3 Extrakce (hodnoty z) rozšiřujícího pole

Existuje více způsobů, jak získat konkrétní rozšíření z daného certifikátu, na všech se ale musí zapojit hned několik velkých modulů z knihovny crypto. Ilustrace č. na str. dosahuje extrakce pole `dNSName` s pomocí knihoven STACK (některá rozšíření jsou interpretována jako blok unionů), ASN (veškeré hodnoty jsou reprezentovány notací ASN.1) a samozřejmě X509V3. Jinak řečeno, program musí získat rozšíření *subjectAltName* jako zásobník `GENERAL_NAMES` (pomocí

10 Viz např. *X509V3 Extension code: programmers guide ([D3], soubor doc/openssl.txt)*

parsovací funkce `X509_get_ext_d2i()`) a v něm podle typu atributu `type` každé položky hledat kýžené pole. Jiné typy rozšíření implementují jiné typy struktur, ale stejně jako u *GeneralName*, i jejich názvy vyplývají z ASN.1 specifikace v RFC3280.

Jiný způsob extrakce rozšíření ukazuje ilustrace č. 13. V certifikátu jej vyhledá `X509_get_ext_by_NID()` podle NID-identifikátoru, `X509_get_ext()` podle získané pozice a jeho hodnotu nakonec vytiskne pomocí `X509V3_EXT_print()`. Oproti prvnímu způsobu je takto ale získána hodnota celého rozšíření, nikoli pouze jeho vybraného pole.

Třetí cesta začíná parsováním objektu `X509_EXTENSION` funkcí `X509V3_EXT_d2i()`, jíž lze získat požadovaný objekt, např. zmíněný `GENERAL_NAMES`.

Všeobecný chaos souvisejících API a také matoucí pojmenování polí a rozšíření pro konfiguraci je nepříjemným znakem specifikace X.509, a tedy i její implementace v OpenSSL. Při hledání konkrétních identifikátorů tak nezbývá než prohlédnout dokumentací a zdrojové kódy: pojmenování typu NID i ASN.1 knihovna definuje v `objects.h`, struktury implementující pole a standardní rozšíření podle specifikace deklaruje v `x509v3.h`.

4.3.4 Verifikace certifikátu

Klíčovou partií v procesu *handshake* u SSL spojení je ověření platnosti certifikátu, kterým se prezentuje druhá strana. Knihovna ve své široké přizpůsobivosti dokáže tento úkol zvládnout sama buďto z úrovně knihovny SSLeay (na základě cesty k CA souborům, např. podle `SSL_CTX_load_verify_locations()`), nebo z úrovně knihoven okolo X.509. V prvním případě se certifikát ověřuje implicitním postupem a interně se volá funkce `X509_verify_cert()`; v tom druhém případě se otevírá možnost standardní postup nahradit vlastním řešením podle explicitně stanovených podmínek (vlastní způsob vyhledávání certifikačních autorit, vlastní validační metody...).

Nestandardní verifikace se implementuje jako `verify_callback()` callback po implicitním ověření (nastavený SSL kontextu/spojení pomocí `SSL_[CTX]_set_verify()`) a stává se tak spíše pokynem k dodatečnému ověření resp. úpravou validačních kritérií. Má-li se aplikace chopit verifikace zcela sama, implementuje tento proces v rutině nastavené úložišti CA certifikátů

použitého SSL kontextu makrem `X509_STORE_set_verify_func()`. Bližší charakteristiku možných přístupů včetně příkladů uvádí [D3] v oddíle X.509 Routines, kapitoly *X509 verification overview* a následující. Dokumentace se sice vztahuje k první generaci v historii OpenSSL, stále ale ve spojení se zdrojovými kódy současné verze OpenSSL nese svůj význam.

4.3.5 Příklad verifikace certifikátu

Někdy se vyskytne úkol, kdy je požadováno verifikovat certifikát bez nutnosti používat k tomu objekty SSL kontextu a SSL spojení. Standardní implementace ověření certifikátu, exportovaná jako funkce `X509_verify_cert()`, požaduje specifikovat kontext úložiště důvěryhodných CA, které program běžně získá buďto explicitně z SSL kontextu nebo v parametru **ctx** jeho `verify_callback()` rutiny. V tomto případě si jej aplikace musí připravit, inspirovat se lze následujícím příkladem:

- ✔ Program načte ze souboru certifikát, jež má být ověřen, do objektu **cert**:

```
X509 *cert = PEM_read_X509(fopen(...), &cert, 0, 0);
```

- ✔ Platnost má být zaručena proti CA certifikátu, který je k dispozici v paměti **mem**. Odtud jej program načte do **root_cert**.

```
unsigned char *buf = mem;
X509 *root_cert = d2i_X509(NULL,&buf,sizeof(mem));
```

- ✔ Následuje vytvoření úložiště a jeho kontextu. Následuje vložení důvěryhodného CA certifikátu:

```
X509_STORE *store = X509_STORE_new();
X509_STORE_CTX *store_ctx = X509_STORE_CTX_new();
X509_STORE_add_cert(store,root);
```

- ✔ Důvěryhodnost kontrolovaného certifikátu mají umožnit i další autority, které jsou k dispozici v PEM souboru `root2.pem` a řadou dalších autorit v adresáři **dir**:

```
char *dir = ...
X509_STORE_load_locations(store,"root2.pem", dir);
```

✔ Vytvoření kontextu úložiště dokončí jeho inicializace, pak už je na řadě samotné ověření a k dokončení práce uvolnění paměti:

```
X509_STORE_CTX_init(store_ctx,store,x509,NULL);
/* Verify: */
int valid = X509_verify_cert(store_ctx);
if (i) { /* Certificate is valid. */ }
else { /* Cert. Is invalid, revoked, or expired */ }
/* clean-up */
X509_STORE_CTX_cleanup(store_ctx);
X509_STORE_CTX_free(store_ctx);
X509_STORE_free(store);
X509_free(root_cert);
```

Ilustrace 13: X509_get_ext_by_NID(), X509_get_ext(), X509V3_EXT_print(): Extrakce rozšíření z certifikátu

```
int pos = (-1);
pos = X509_get_ext_by_NID(cert, NID_subject_alt_name, pos);
if (pos>=0)
{
   /* The extension stays at <pos>th cell in the STACK
OF(X509_EXTENSION)
      stack of the X509 structure. So let's get it: */
   ext = X509_get_ext(cert, pos);
   if (ext)
   {
      /* print-out extension values using a memory BIO*/
      BIO* bio_buf = BIO_new(BIO_s_mem());
      X509V3_EXT_print(bio_buf, ext, 0, 0);
      /* move them to ASCII buffer and print to stdout*/
      BIO_gets(bio_buf, values, 255);
      fprintf(stdout, „subjectAltName value is %s\n", values);

      BIO_vfree(bio_buf);
   }
}
```

Ilustrace 14: X509_get_ext_d2i(), sk_GENERAL_NAME_*(), ASN1_STRING_data(): Extrakce X.509v3 rozšíření z certifikátu

```
X509 *cert; STACK_OF(GENERAL_NAME) *gens;
char *value; int i;

/* prepare structures */
cert = X509_new();

/* Load certificate prom PEM-formatted file. Note that a password
callback
 * will be needed as the 3d parameter if the certificate was encrypted.
*/
PEM_read_X509( fopen(„mycert.pem“, „r“), &cert, NULL, NULL);

/* obtain the field by its NID name (see objects.h) - extract
subjectAltName,
 * dNSName print separately */
/* First obtain whole the stack of that extension's unions */
gens = (STACK_OF(GENERAL_NAME)*)X509_get_ext_d2i (cert,
NID_subject_alt_name,
  if (gens)
  {
    for (i=0; i < sk_GENERAL_NAME_num( gens ); i++)
    {
      gen = sk_GENERAL_NAME_value( gens, i);
      /* check gen to have dNSName subfield, if so, extract it */
      if (gen->type == GEN_DNS)
      {
        fprintf (stdout, "\t Found subjectAltName:dNSName: ");
        buf = ASN1_STRING_data(gen->d.dNSName);
        if (buf)
        {
          fprintf (stdout, "%s\n", buf);
          /* Freeing of 'buf' is recommended! */
          free (buf);
          buf = NULL;
        }
} else {
   /* Extension not present */
}

/* free-up ! */
X509_free(cert);
fclose(fp);
```

5 Závěr

Složitost a rozsah knihovny OpenSSL je ve skutečnosti mnohem vetší, než zachytil tento dokument. Výsledná práce specifickým způsobem analyzuje knihovnu na problému, jak s její pomocí vytvořit spojení podle protokolu SSL 2.0/3.0 nebo TLS 1.0 a jak toto spojení zabezpečit pomocí certifikátů. Z takového úkolu se odvíjí řada dílčích problémů, proto text (alespoň částečně) seznamuje s knihovnami BIO, ERR a X509.

Samotná implementace uvedeného problému nevyžaduje tak široce rozvedený návod, ovšem pouze za předpokladu, že programátor nepotřebuje více proniknout do teoretické stránky projektu OpenSSL. Tímto směrem nebyla práce koncipována, na druhou stranu tak bohužel její omezený rozsah nedovolil seznámit s okolnostmi instalace, konfigurace a kompilace knihovny, ani s dalšími stěžejními významy OpenSSL. Prostoru se nedostalo všem článkům sofistikovaného mechanismu knihovny **crypto** ve spojení s problematikou managementu certifikátů – např. konfigurace serveru pracujícího jako entita PKI infrastruktury, v níž lze vydávat a autorizovat certifikáty.

Zcela mimo spektrum zájmu práce je uvedení OpenSSL jako toolkit produktu, navzdory frekventovanosti takového využití: na Internetu jsou k nalezení návody konfigurace známého webového serveru Apache, kterou je povýšen na CA uzel v síti PKI; prostředkem je zde právě OpenSSL a jeho balík aplikačních nástrojů.

I přes malý úhel záběru práce je tato výchozím bodem k dalšímu seznámení se s OpenSSL. Motivací budiž její univerzálnost, díky které se lze setkat s jejím obsazením do role nadstavby protokolům HTTP pro zabezpečené webové servery (projekt modSSL i jeho rodičovská alternativa ApacheSSL tvořící rozhraní k Apache serveru), a protokolům SMTP, POP a IMAP pro komunikaci elektronickou poštou (MTA/MDA program pine).

6 Literatura

Stručné, přehledné informace

- Transport Layer Security, Wikipedia, `http://en.wikipedia.org/wiki/Ssl` (březen 2006)

Dokumentace

- [D1] YOUNG, E., aj.: *OpenSSL documents*, The OpenSSL Project, říjen 2005. Dostupné na URL `http://www.openssl.org/docs/` (březen 2006)
- [D2] GLENN, A.T.: *SSLeay documentation*, Columbia University, leden 1999. Dostupné na URL *`http://www.columbia.edu/~ariel/ssleay/ssleay.html`* (březen 2006)
- [D3] YOUNG, E., aj.: *OpenSSL 0.9.8a*, The OpenSSL Project, říjen 2005. Dostupné na URL `http://www.openssl.org/source/openssl-0.9.8a.tar.gz` (březen 2006)
- [D4] HUDSON,T.J., YOUNG, E.: *SSLeay Programmer Reference*, School of Psychology, The University Of Queensland, leden 1996. Dostupné na URL `http://www2.psy.uq.edu.au/~ftp/Crypto/ssl.html` (březen 2006)

Příklady

- [EX1] RESCORLA, E.: *An introduction to OpenSSL programming (Part I)*, RTFM, Inc., říjen 2001. Dostupné na URL `http://www.rtfm.com/openssl-examples/part1.pdf` (březen 2006)
- [EX2] RESCORLA, E.: *An introduction to OpenSSL programming (Part II)*, RTFM, Inc., leden 2002. Dostupné na URL `http://www.rtfm.com/openssl-examples/part2.pdf` (březen 2006)

Specifikace

- [S1] SCHNEIER, B.: *The Blowfish Encryption Algorithm*. Dostupné na URL `http://www.schneier.com/blowfish.html` (duben 2006)
- [S2] *SSL 2.0 specification*, Netscape, únor 1995. Dostupné na URL `http://wp.netscape.com/eng/security/SSL_2.html` (březen 2006)
- [S3] *SSL 3.0 specification*, Netscape, listopad 1996. Dostupné na URL

http://wp.netscape.com/eng/ssl3/index.html (březen 2006)

- [S4] DIERKS, T., ALLEN, C.: *The TLS Protocol Version 1.0*, IETF, leden 1999.Dostupné na URL http://www.ietf.org/rfc/rfc2246.txt (březen 2006)
- [S5] RESCORLA, E.: *HTTP Over TLS*, RFC 2818, květen 2000. Dostupné na URL http://www.ietf.org/rfc/rfc2818.txt (březen 2006)

Příloha A: Specifikace WinSSL

Cíl

Aplikace poskytuje běžnou funkčnost nástroje openssl knihovny OpenSSL ve formě grafického uživatelského rozhraní.

Charakteristika

Aplikace WinSSL disponuje většinou funkcí nabízených nástrojem openssl jako příkazy interpretu, konkrétně podle následujícího seznamu:

- ✔ asn1parse
- ✔ ca
- ✔ ciphers
- ✔ crl
- ✔ crl2pkcs7
- ✔ dgst
- ✔ dhparam
- ✔ dsa
- ✔ dsaparam
- ✔ enc
- ✔ gendh
- ✔ dendsa
- ✔ genrsa
- ✔ ocsp
- ✔ passwd
- ✔ pkcs12
- ✔ pkcs7
- ✔ pkcs8
- ✔ rand
- ✔ req
- ✔ rsa
- ✔ rsautl
- ✔ smime
- ✔ verify
- ✔ x509

Prostřednictvím grafického rozhraní uživatel může zvolit vybraný příkaz, specifikuje parametry převzaté ze souvisejícího příkazu nástroje openssl a následně nechat aplikaci příkaz provést. Po jeho dokončení program informuje o výsledku.

GUI

Grafické rozhraní tvoří soustava nabídek v menu, uspořádaná podle charakteru na dialogy a průvodce. Dialogy zastupují příkazy s malým počet parametrů, průvodci naopak příkazy s tolika nastaveními (parametry), že by bylo nepřehledné je uspořádat pouze na jednom formuláři.

Koncepce řešení

Vývojovým prostředím je Microsoft Visual Studio .NET Professional 2003, za programovací jazyk je zvoleno C++ s kompilátorem dodávaným společně s vývojovým prostředím.

Wrapper k openssl aplikacím

Přístup ke knihovným funkcím, jež plní své dílčí úkoly v implementaci každého z příkazů, zajišťuje wrapper **openssl_win**. Je implementován jako modifikace

existujícího modulu `openssl.c` (tedy zdrojového kódu nástroje `openssl`), upraveného pro potřeby WinSSL.

Příkazy jsou vykonávány voláním funkce `main()` z wrapperu. Její deklarace se shoduje s klasickou deklarací funkce `main()` v programu jazyka C, pracujícího s parametry příkazové řádky*. WinSSL modifikuje původní zdrojový kód nástroje `openssl` tak, aby mohl simulovat jeho spouštění z příkazové řádky: v parametrech předává vektor argumentů a jeho velikost. Ve funkci `main()` dojde k jejich zpracování a provedení příslušného příkazu.

O výsledku příkazu informuje návratová hodnota funkce `main()`: nenulová hodnota znamená chybu. Bližší popis chyby získá z fronty chyb knihovny OpenSSL funkce `ossl_wrapper_get_last_err_string()`, exportovaná v rozhraní wrapperu.

Třída `OSSL_OPS`

`OSSL_OPS` je abstraktní třídou uchovávající nastavení příkazu. Jelikož nastavení pro jednotlivé příkazy se liší, implementují se potomci této třídy pro jednotlivé příkazy, a to ve vlastním jmenném prostoru. Pro formální oddělení používaných parametrů jsou symbolické názvy parametrů definovány ve jmenném prostoru „Options“ v rámci jmenného prostoru příkazu.
Existence objektu třídy – viz níže.

Třída CCmdBase

Pro příkazy je definována abstraktní třída `CCmdBase`, jejíž potomci používají jako atribut relevantního potomka třídy `OSSL_OPS`. Třída spouští funkci `main()` z *wrapperu k aplikacím* `openssl`, udržuje nastavení. Existence objektu třídy – viz níže.

Třída `CwizardBase`

Třída definuje dialogového průvodce, který uživatel vyvolá z menu aplikace. Opět je abstraktní; potomci, kteří ji implementují, představují specifické průvodce jednotlivých příkazů.

Pro spuštění průvodce se použije metoda `CwizardBase::Start()`, která postupně vybírá objekty typu `System::Windows::Forms::Form` (formuláře)

* Důvodem je snaha co nejméně modifikovat originální zdrojový kód nástroje `openssl`. Ten je dedikován pro práci v textovém režimu a pomocí argumentů příkazové řádky ovládá další příkazy (aplikace openssl).

z atributu `actions` a vkládá je do implicitního dialogu průvodce. Tento dialog obsahuje tlačítka „Cancel“, „Back“ a „Next“ (resp. „Finish“, je-li vkládaný formulář posledním prvkem ve vektoru formulářů v `actions`).

Atribut `actions` typu `ACTMAP` představuje mapu akcí. Akce vymezuje svým významem učitou množinu parametrů příkazu, s kterými může být spojena (pro většinu příkazů je pouze jedna akce použitelná se všemi parametry, ale např. příkaz `ocsp` definuje tři akce, které mají svůj okruh parametrů), určená svým jménem (jako hašovací klíč) a vektorem formulářů, které nastavují související parametry (pro většinu příkazů pouze jeden formulář). Dialogu příkazu v charakteristice GUI odpovídá akce s jednoprvkovým vektorem, průvodci pak akce s víceprvkovým vektorem formulářů.

Ilustrace A.1: Ze zdrojového kódu: implementace akcí a sad formulářů, které jsou s akcemi spojeny

```
public __gc struct SContext {
       System::Collections::ArrayList *     options;
       System::Windows::Forms::Form *   form;
};
typedef SContext    PAGE;
typedef ArrayList   PAGES;        // collection of PAGE*
typedef ArrayList __gc *  PPAGES;       // pointer to PAGES
typedef SortedList  ACTMAP;             // map of <const ACTION_NAME,
PPAGES> pairs
```

Stisknutím tlačítka „Next“ nebo „Finish“ se vyvolá událost, kterou zpracuje handler `CwizardBase::Step()` – ten uloží dostupná nastavení podle údajů z formuláře. Stiskem „Finish“ se navíc vytvoří instance z potomka třídy `CcmdBase` (jako atribut `CwizardBase`), do níž se zkopírují nastavení z formuláře, a zavolá se metoda `CcmdBase::Run()`, která použije funkce `main()` z wrapperu **openssl_win** k provedení příkazu. Po dokončení metody objekt implementující `CCmdBase` (potomek) zanikne a aplikace informuje o výsledku. V případě chyby nabídne akci opakovat, v případě úspěchu nebo odmítnutí dalšího pokusu celý objekt vzniklý z potomoka `CWizardBase` sám sebe uvolní.

Průvodci

WinSSL poskytuje příkazy `ca`, `crl`, `ocsp`, `x509` a `enc` ve formě průvodců. Následující tabulka spojuje každý z těchto příkazů s implementovanou akcí.

Tabulka A.1: Implementované příkazy

Příkaz	***Název příkazu***	***Akce***
req	X.509 CSR	Create certificate request

Příkaz	*Název příkazu*	*Akce*
	Management	Create self-signed certificate
ca	Certificate Authority management	Sign certificate request
		Sign self-signed certificate
		Sign Netscape SPKAC
x509	X.509 certificate data management	Sign to create self-signed certificate
		Sign using CA
		Certificate conversions
ocsp	Online Certificate Status Protocol utility	Create OCSP request
		Parse OCSP response
enc	Encoding with ciphers	Encryption/decryption with symmetric ciphers

Dialogy

Příkazy ze seznamu v úvodu tohoto textu, které nezmiňuje tabulka výše, aplikace WinSSL implementuje jako dialogy, tj. potomky třídy CWizardBase, definující jedinou akci s jedním formulářem.

Omezení

- Platforma: Win32 pro OS Windows XP libovolné verze
- Paměťová náročnost: min. 10MB
- Závislosti: OpenSSL v0.9.8a, NET Framework v1.1+