FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Podpora CryptoAPI Next Generation v OpenSSL
Student:	Jan Pešek
Vedoucí:	Ing. Josef Kokeš
Studijní program:	Informatika
Studijní obor:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Seznamte se s knihovnou OpenSSL a její strukturou.
Proveďte rešerši rozhraní Microsoft CryptoAPI a CryptoAPI: Next Generation, popište rozdíly mezi nimi.
Prozkoumejte, jak je v OpenSSL implementován modul e_capi, který zajišťuje podporu kryptografických funkcí pomocí CryptoAPI.
Na základě předchozích zjištění vytvořte pro OpenSSL nový modul e_cng, který bude pro kryptografické operace používat CryptoAPI: Next Generation.
Otestujte funkcionalitu svého modulu při použití certifikátu v protokolech SSL/TLS.
Diskutujte své výsledky.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Pavel Tvrdík, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 13. února 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Podpora CryptoAPI Next Generation v OpenSSL

Jan Pešek

Katedra počítačových systémů
Vedoucí práce: Ing. Josef Kokeš

16. srpna 2020

Poděkování

Poděkování patří vedoucímu této práce Ing. Josefu Kokešovi za pomoc při vypracování, za odborné konzultace a zvláště pak za úctyhodnou rychlost v emailové komunikaci.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. srpna 2020 . . . . . .

Odkaz na tuto práci

Pešek, Jan. *Podpora CryptoAPI Next Generation v OpenSSL.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato práce demonstruje tvorbu kryptografického modulu pro OpenSSL s podporou kryptografického rozhraní dodávaného jako součást operačních systémů Windows. Za tím účelem je z počátku popsána architekturu současné LTS verze OpenSSL 1.1.1. Mimořádnou pozornost si zasluhuje rozhraní Engine API, na kterém tvorba modulu stojí. Součástí práce je také rešerše kryptografických rozhraní Microsoft CryptoAPI a Cryptography API: Next Generation. Výstupem práce je navíc funkční a otestovaný kryptografický modul, který zprostředkovává podporu Cryptography API: Next Generation pro OpenSSL.

Klíčová slova OpenSSL, Tvorba kryptografického modulu, Microsoft CryptoAPI, Next Generation

Abstract

The main object of this bachelor thesis is to create an OpenSSL engine supporting Microsoft Cryptography API: Next Generation. The thesis contains an analysis of the OpenSSL architecture, focusing on the current LTS version 1.1.1. As Engine API is an important component for engine development, one section is dedicated to properly describing the interface. Thesis also provides a description of Microsoft CryptoAPI and Cryptography API: Next Generation supplied with Windows OS. Working and tested OpenSSL engine supporting Microsoft Cryptography API in OpenSSL is delivered.

Keywords OpenSSL, Creating an Engine, Microsoft CryptoAPI, Next Generation

Obsah

Seznam obrázků

Seznam výpisů kódu

Úvod

Bezpečnost komunikace v počítačové síti závisí na použitých kryptografických algoritmech. Se vzrůstajícím výpočetním výkonem snadno dostupných zařízení se návrh bezpečného algoritmu stává velmi náročným úkolem a vývojářům není doporučeno vynakládat snahu na vytváření vlastního řešení. Lepší možností je spolehnout se na ověřené implementace, které respektují pravidelně aktualizovaná doporučení známá jako Request for Comments (RFC).

Prvním cílem této práce je popsat strukturu OpenSSL, velmi využívané knihovny pro zprostředkování šifrované komunikace. Implementuje protokoly SSL/TLS a s tím spojené kryptografické operace. OpenSSL navíc poskytuje Engine API, rozhraní pro kompatibilitu s knihovnami třetích stran. Umožňuje tak rozšířit nebo nahradit defaultní implementaci funkcí.

Druhým cílem práce je provést rešerši kryptografických rozhraní, které Microsoft využívá ve svých operačních systémech. Rozhraní Microsoft CryptoAPI je již považováno za zastaralé a v budoucnu bude zcela nahrazeno jeho nástupcem Cryptography API: Next Generation. Kapitolu uzavírá popis rozdílů mezi nimi.

Dalším cílem je prozkoumat rozhraní Engine API a implementaci již existujícího OpenSSL modulu pro podporu CryptoAPI. Na základě těchto zjištění bude vytvořen zcela nový modul, který bude pro kryptografické operace využívat Cryptography API: Next Generation. Jeho implementace bude součástí práce. Funkcionalita modulu bude na závěr otestována při použití certifikátu v protokolech SSL/TLS.

Tato práce vznikla za použití Long Term Support verze OpenSSL 1.1.1. Počínaje verzí 3.0, která je v současné době ve fázi develop, bude docházet k výrazným změnám v celém konceptu. Verze 3.0 bude stále podporovat stejné rozhraní pro kryptografické moduly, na kterém tato bakalářská práce stojí. Dovršení změny ve struktuře se očekává nejdříve s příchodem verze 4.0.

První kapitola představuje projekt OpenSSL a popisuje jeho strukturu. Odhaluje, že OpenSSL je poskytovatelem dvou knihoven, které se dále rozdě-

lují do několika komponent. Komponenty, jež jsou využívány k tvorbě kryptografického modulu nebo jen pro jeho testování, jsou diskutovány podrobněji.

Druhá kapitola se věnuje kryptografickým službám, které jsou dodávány ve výchozím nastavení operačních systémů Windows. Prvním popisovaným konstruktem je certifikační úložiště. To představuje zajímavé řešení pro jednotnou organizaci certifikátů. Dále je uvedena architektura kryptografického rozhraní CryptoAPI. Kromě kryptografických primitiv zprostředkovává práci s úložištěm a navíc dokáže bezpečně manipulovat se soukromými klíči. Část tohoto rozhraní byla označena za zastaralou, a proto se kapitola dále věnuje jeho nástupci Cryptography API: Next Generation. Celá tato kapitola si neklade za cíl popsat celou architekturu kryptografických služeb vyčerpávajícím způsobem, jako spíše poukázat na ty rozdíly mezi rozhraními, které je nutné znát při migraci aplikace na novější rozhraní.

Třetí kapitola se detailně zabývá komponentou engine z OpenSSL knihovny libcrypto. Prezentuje strukturu kryptografického modulu a nabídku možností rozhraní Engine API. Poslední kapitola předvádí funkčnost nově vytvořeného kryptografického modulu.

Kapitola 1

OpenSSL

Projekt OpenSSL je balík open source nástrojů pro podporu TLS a SSL protokolů a s tím spojené kryptografické operace. Produkt podléhá licenci Apache a lze jej tedy plně využívat i pro komerční účely. V projektu převládá programovací jazyk C. Implementace se striktně drží standardů popsaných v RFC.

První verze byla vytvořena v roce 1998 skupinou vývojářů [1]. Nyní existuje osmnáct přispěvatelů s právem přímo upravovat projekt. V roce 2017 vstoupila v platnost interní směrnice [2], která jasně definuje hierarchii přispěvatelů. Management Committee definuje požadavky na produkt a rozhoduje o strategických směrech, členové Technical Committee zastávají funkci softwarových architektů. Členství v těchto kruzích se ze značné části překrývá.

OpenSSL je poskytovatelem knihoven libssl a libcrypto, které lze využívat pomocí API. Libssl dokáže zprostředkovat šifrovanou komunikaci mezi serverem a klientem pomocí protokolů SSL nebo TLS. Libcrypto nabízí široké spektrum kryptografických operací, které je navíc rozšiřitelné pomocí kryptografických modulů. Bližšímu popisu architektury se věnuje kapitola 1.2.

Vedle knihoven do balíku nástrojů zapadá program [3] ovládaný přes příkazovou řádku, s jehož pomocí lze přímo vykonávat následující kryptografické operace:

- Tvorba a správa soukromých a veřejných klíčů.
- Vykonávání operací asymetrické kryptografie.
- Vytváření certifikátů, CSR a CRL.
- Vytváření heše zprávy.
- Šifrování a dešifrování zprávy.
- Testování spojení serveru a klienta.
- Manipulace s emaily chráněnými protokolem S/MIME.

1.1 Alternativy

Implementace protokolů SSL a TLS není výsadou OpenSSL. Vedle této knihovny existují jak další open source projekty, tak proprietární software. V době vzniku této práce většina velkých projektů disponovala podporou TLS 1.3, výjimku tvoří například SChannel využívaný v systémech Windows. Níže je stručný popis knihoven, které se často uvádějí jako vhodné alternativy k OpenSSL.

LibreSSL vznikl v roce 2014 jako odnož OpenSSL krátce poté, co byla odhalena vážná bezpečnostní hrozba Heartbleed. Software vzniká pod záštitou projektu OpenBSD a jeho hlavním cílem [4] je zpřehlednit zdrojové kódy, zbavit se zastaralých nebo nefunkčních částí a zmodernizovat celou metodiku vývoje. Hned v první verzi byl z důvodu závislosti na nesvobodném software odstraněn modul pro podporu Microsoft CryptoAPI.

WolfSSL rovněž vznikl kvůli nespokojenosti s již existujícím OpenSSL. Tvůrci dávají důraz na rychlost, malou velikost a přenositelnost. Knihovna wolfSSL je údajně [5] dvacetkrát menší než implementace od OpenSSL. Obdobnou strategii minimalistického řešení volí i MatrixSSL [6]. Díky své malé náročnosti na paměť najdou knihovny využití ve vestavěných systémech a IoT zařízeních. Přechod z OpenSSL ulehčují nadstavbou pro snazší portaci.

BoringSSL je oddělenou větví od OpenSSL, je určena čistě pro potřeby produktů Google a neklade si za cíl proniknout do širší veřejnosti. Hned v prvním odstavci dokumentace [7] nalezneme varování vývojářů, že nelze garantovat stabilitu při používání jejich API. Vývojáři totiž nevěnují pozornost zpětné kompatibilitě. Značná část struktury je stále s OpenSSL shodná.

GnuTLS v přehledu [8] sděluje, že se jedná o jednoduchou implementaci TLS protokolů v jazyce C tak, aby se dokázala vmísit mezi již existující linuxové knihovny. Dělí se do tří nezávislých částí. Jedna část je samotná implementace TLS protokolu, ta tvoří hlavní jádro projektu. Kryptografický backend a část pro práci s certifikáty pak využívá jiné knihovny GNU projektu.

Existují i knihovny, které nejsou vytvořeny v programovacím jazyce C. Rustls, jak již název napovídá, je vytvořen v Rust. Zajímavostí je, že ačkoliv celý projekt udržuje jediný vývojář [9], je projekt stále aktuální. Dalším příkladem budiž Java Secure Socket Extension pod dohledem firmy Oracle.

Apache HTTP Server a Nginx jsou nejvyužívanějším softwarem pro provoz webových serverů, spolu tvoří přibližně 70 % webových serverů [10]. Z dokumentace Nginx [11] vyplývá, že ke konfiguraci HTTPS serveru je potřeba OpenSSL. Apache nabízí možnost výběru mezi OpenSSL a WolfSSL. Nová verze WolfSSL 4.2.0 [12] totiž v lednu 2020 přišla s podporou Apache HTTP Serveru.

Evidentně existuje několik zajímavých alternativ, některé z nich poukazují na nedostatky OpenSSL a přichází s odlehčenější strukturou. Žádná z nich nemá tak bohatou komunitu, jako je tomu u OpenSSL. Tato práce je zaměřená

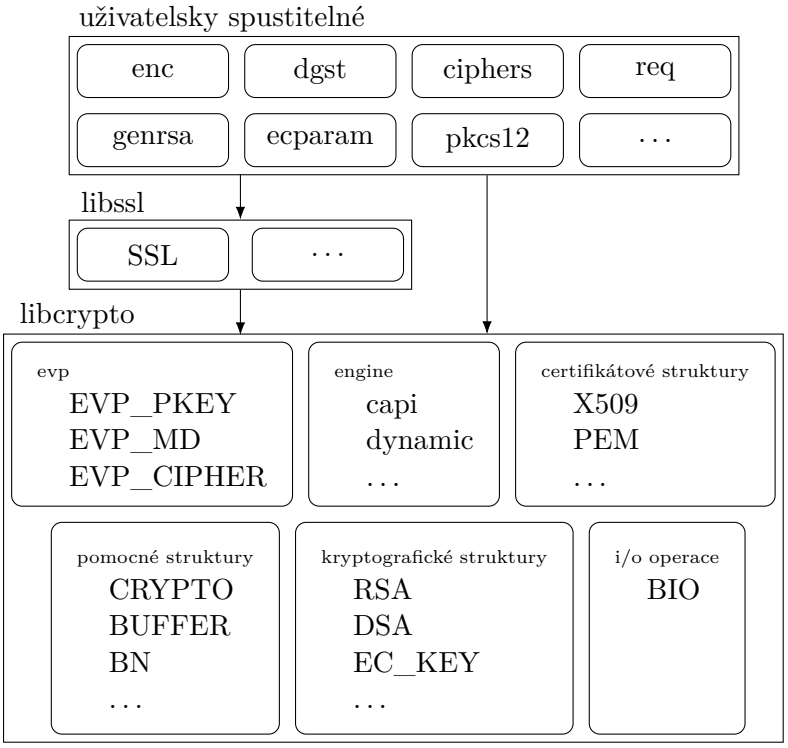
na vytvoření modulu pro zapojení Crypto API Next Generation do implementace TLS protokolu a k tomu je přizpůsobena právě knihovna OpenSSL.

1.2 Architektura

OpenSSL verze 1.1.1 je strukturovaná do čtyř vrstev [13]. Za jádro projektu lze považovat kryptografickou vrstvu, tu reprezentuje knihovna libcrypto. Obsahuje implementace kryptografických algoritmů a může být volitelně rozšířena o další funkcionalitu pomocí doplňkového modulu, oficiální dokumentace ho nazývá jako engine a považuje jej z hlediska architektury za zvláštní komponentu.

Další část utváří TLS komponenta, ta popisuje roli knihovny libssl. Implmentuje SSL, TLS a DTLS protokoly dle aktuálně definovaných standardů a je závislá na funkcích knihovny libcrypto. Nejvyšší vrstvu představuje rozhraní pro aplikace spustitelné v příkazovém řádku, ke své činnosti potřebuje obě zmiňované knihovny.

Diagram na obrázku 1.1 zachycuje architekturu OpenSSL a vnitřní komponenty knihoven, zredukované na prvky, které byly využity při tvorbě kryptografického modulu.



Obrázek 1.1: Zjednodušený diagram struktury OpenSSL

1.3 Knihovna libcrypto

Knihovna libcrypto přináší implementace kryptografických operací. Jak je naznačeno na obrázku 1.1, knihovna samotná se dělí do několika komponent. Je pravidlem, že každá z komponent poskytuje své rozhraní, které je definované v hlavičkových souborech v adresáři `include/openssl/`. Implementace jednotlivých struktur nebo funkcí jsou v projektu umístěny pod adresářem `crypto/`. Následující kapitoly se věnují komponentám, které byly přímo využity při tvorbě kryptografického modulu a jejichž funkcionalita by měla být vysvětlena.

1.3.1 Kryptografické struktury

Každý kryptografický algoritmus implementovaný v OpenSSL má svoji nízkoúrovňovou reprezentaci. Typicky to znamená, že existuje nezávislá struktura, která zabaluje funkce nezbytné k provedení daných kryptografických operací, dodatečné parametry a případně i klíče. Přímé přístupy k těmto strukturám jsou zastíněny rozhraními, která jsou deklarovány příslušnými hlavičkovými soubory.

Struktura `RSA` se skládá z několika `BIGNUM` komponent pro uchování modulu, exponentů a dalších parametrů využívaných v RSA algoritmech. Slouží tedy pro reprezentaci soukromých a veřejných klíčů. Struktura mimo to obsahuje `RSA_METHOD`, nositele implementací RSA operací.

`RSA` strukturu lze inicializovat pomocí `RSA_new_method()`, jejíž argumentem může být ukazatel na nějaký `ENGINE`. V tom případě se při volání RSA operací využije `RSA_METHOD` implementovaná v příslušném modulu. Pokud je argumentem `NULL`, bude namísto toho zvolena defaultní implementace. Dosadit lze samozřejmě i instanci `RSA_METHOD` přímo, v tom případě musí následovat přenastavení až po inicializaci pomocí `RSA_set_method()`, která zapříčiní uvolnění dříve přiděleného modulu.

Struktura `RSA_METHOD` [14] předepisuje funkce pro RSA operace. Úkolem `RSA_public_encrypt()` a `RSA_private_decrypt()` je šifrování veřejným klíčem a dešifrování soukromým klíčem. `RSA_private_encrypt()` má dle dokumentace [15] za úkol zvládat vytvoření digitálního podpisu předem zahešovaných dat a `RSA_public_decrypt()` ověřování podpisu. Všechny tyto funkce spojují vstupní argumenty. Očekávají data ke zpracování, jejich velikost, výstupní paměťový blok s dostatečnou velikostí, strukturu `RSA` a použitý padding. Od toho se odlišují funkce `RSA_sign()` a `RSA_verify()`, které jsou obohacené informací o hešovacím algoritmu použitém před samotným podepsání. Naproti tomu jsou funkce ochuzené o možnost specifikovat padding. To v současné době, kdy se používají dvě různá podpisová schémata RSA, nemusí být vyhovující řešení. Avšak pro obejití nedostatku lze využít právě `RSA_private_encrypt()` nebo `RSA_public_decrypt()`. Poslední

funkce `rsa_mod_exp()` a `bn_mod_exp()` jsou využívány interně pro výpočet mocniny v konečném tělese.

Pro DSA operace je připravená struktura `DSA`. Je rovněž složena ze sady `BIGNUM` komponent pro reprezentaci klíčů a odkazu na `DSA_METHOD`. Systém inicializace je stejný, ale formát kryptografických funkcí se značně liší. Funkce `DSA_sign_setup()` pro výpočet k^{-1} a r způsobí jen zpomalení výpočtů, existuje už jen kvůli zpětné kompatibilitě a neměla by být používána [16]. Funkce `DSA_do_sign()` vrátí digitální podpis předaných dat jako strukturu `DSA_SIG`, což reprezentuje dvojici `(r, s)`. Je to jediná funkce ze všech výše zmiňovaných, která nemá návratový typ celé číslo pro ověření správnosti běhu.

V případě protokolu digitálního podpisu s využitím eliptických křivek nalezneme odlišností více. Struktura `EC_GROUP` definuje eliptickou křivku. Body na křivce reprezentuje struktura `EC_POINT`. `EC_KEY` pak slouží k uchování klíčů, soukromý klíč je definován pomocí `BIGNUM`, veřejný pomocí `EC_POINT`. Implementace jednotlivých operací zastřešuje struktura `EC_METHOD`.

`EC_KEY` se inicializuje funkcí `EC_KEY_new_by_curve_name()`, které se předá číselný identifikátor příslušné křivky. Seznam podporovaných křivek v instanci OpenSSL lze získat příkazem `openssl ecparam -list_curves` v příkazovém řádku. Uvedená inicializační funkce jen obaluje zavolání posloupnosti dvou funkcí pro alokaci paměti a pro přidělení křivky.

K vytvoření digitálního podpisu se nabízí několik [17] kryptografických funkcí. `ECDSA_do_sign_ex()` očekává vstupní data, již alokovaný výstupní buffer, strukturu `EC_KEY` a volitelně i k^{-1} a r pro urychlení výpočtů. Obdobně funguje i `ECDSA_sign_ex()` s tím rozdílem, že podpis bude vrácen strukturou `ECDSA_SIG` jako návratová hodnota. Funkce `ECDSA_sign_setup()` slouží ke spočítání dvou výše zmíněných hodnot pro urychlení procesu podepisování.

Nízkoúrovňové struktury pro symetrickou kryptografii a hešování jsou reprezentovány na podobném principu. Každý algoritmus a každý mód má svoji strukturu, tedy balík funkcí a parametrů nezbytné k vykonávání příslušných operací v příslušném hlavičkovém souboru `\include\openssl\`. Pro všechny takové struktury platí, že by v aplikaci neměly být využívány, a namísto toho kryptografické operace provádět prostřednictvím EVP API.

1.3.2 EVP

Envelope, zkráceně EVP, je vysokoúrovňové rozhraní pro kryptografické operace jako je šifrování a dešifrování, podepisování a ověřování nebo hešování. Z pohledu architektury leží nad strukturami z předchozí kapitoly a rozdělují se do tří hlavních objektů [18].

Struktura `EVP_CIPHER` představuje jeden konkrétní mód nebo algoritmus pro symetrické šifrování. Obsahuje identifikátor šifry, výchozí velikost klíče, bloku nebo inicializačního vektoru. Dále už následují funkční ukazatele, jimž je přiřazena implementace dané operace. Jádrem `EVP_CIPHER` jsou funkce namapované na `*do_cipher` a `*init`, které typicky vyústí ve využití metod

příslušné nízkoúrovňové kryptografické struktury. K `EVP_CIPHER` aplikace nepřistupuje přímo, ale prostřednictvím kontextové struktury `EVP_CIPHER_CTX`, která se inicializuje pomocí `EVP_EncryptInit()`. V ten moment je kontextové struktuře přiřazena `EVP_CIPHER`, popřípadě další data jako klíč nebo inicializační vektor. Volitelně lze určit i konkrétní engine, který bude pro výpočty využit. Pro následující šifrovací operace, využívá aplikace pouze kontextovou strukturu. V případě EVP API je dokumentace [19] velice sdílná, nabízí ukázku šifrování zprávy pomocí pomocí EVP API.

`EVP_MD` je určena pro reprezentaci hešovacích funkcí. Obdobně, jako v předchozím případě, existuje příslušná kontextová struktura `EVP_MD_CTX` a funkce z rozhraní EVP API s prefixem `EVP_Digest`. Jejich aplikační použití a architektonická stavba je shodná s šifrovacími strukturami [18], a proto nebude podrobněji rozebírána.

S asymetrickou kryptografií přichází změna v názvosloví. Schránku funkcí netvoří struktura `EVP_PKEY`, nýbrž `EVP_PKEY_METHOD`. Ve té jsou připraveny ukazatele na operace spojené s asymetrickou kryptografií. Pro vytvoření digitálního podpisu jsou připraveny hned dvě funkce. Rozdílem je, že `*digestsign` oproti `*sign` neočekává na vstupu již zahešovaná data. Opět existuje kontextová struktura `EVP_PKEY_CTX`, jež plní stejný účel jako u výše jmenovaných. Obsahuje `EVP_PKEY_METHOD` a `EVP_PKEY` spolu s dalšími informacemi a daty potřebnými k uskutečnění požadovaných operací.

Struktura `EVP_PKEY` uchovává soukromý nebo veřejný klíč. Jeho položka `type` určuje identifikátor algoritmu, s kterým je spojen klíč `pkey`, reprezentovaný formou příslušné struktury z předchozí kapitoly, tedy `RSA`, `DSA`, `EC_KEY` a podobně [20]. Jedná se tedy opět o abstrakci nízkoúrovňových struktur.

OpenSSL poskytuje asi osmnáct [21] různých `EVP_PKEY_METHOD`, například RSA, DSA nebo ECDSA algoritmů. Vnitřně jsou tyto struktury uchovávány v poli a lze do něj přidat další specifické implementace.

EVP je komplexní rozhraní, zastiňuje používání kryptografických struktur představené v předchozí kapitole a ulehčuje tak použití celé knihovny. Pomocí EVP API lze snadno přistupovat ke všem výše popsaným strukturám. V aplikacích je hojně používané a dokumentací podrobně popisované.

1.3.3 Pomocné struktury

Knihovna libcrypto implementuje mimo jiné algoritmy, jejichž bezpečnost je založena na aritmetice velkých čísel. Pro tento účel je vhodné použít strukturu `BIGNUM`, která k uchovávání čísla využívá dynamické alokace a tudíž není limit pro velikost [22]. Při práci s touto strukturou je však nutné věnovat zvýšenou opatrnost, zda nedošlo k chybě při alokování prostředků. `BIGNUM` poskytuje rozhraní, s jehož pomocí lze provádět standardní aritmetické operace. Implementační část práce přímo využívá tuto strukturu pro reprezentaci klíčů. Funkce `BN_bin2bn()` převede kladné celé číslo v big-endian podobě do nově alokované `BIGNUM`.

`CRYPTO` je asi nejrozsáhlejší pomocnou strukturou. Její hlavní síla je v implementaci funkcí pro správu paměti a vláken, také provádí inicializace dalších struktur. Z této nabídky se při tvorbě kryptografického modulu uplatnily pouze funkce `OPENSSL_malloc()` a `OPENSSL_free()`, jež jsou abstrakcemi pro známé funkce z jazyka C.

Vybraným OpenSSL strukturám je umožněno připnout k sobě libovolná data, která by se mohla hodit v daném případě užití. Možnost je otevřena nízkoúrovňovým kryptografickým strukturám jako je `RSA`, `DSA`, `EC_KEY` a komponentě `ENGINE`. Dokumentace tato přídavná data označuje jako `exdata`.

`Exdata` jsou jednoznačně identifikována indexem, který je výstupem volání `CRYPTO_get_ex_new_index()` na začátku aplikace [23]. Pro tuto obecnou funkci jsou vytvořená makra ke každé struktuře, která ulehčí její využívání. Jako argument lze předat callback tří funkcí, které budou vyvolány při vzniku, uvolnění nebo kopírování původní struktury. Získaný index se typicky uchová ve formě globální proměnné. Pro zápis nebo čtení z `exdata` je potřeba využít funkce poskytované konkrétní strukturou, uvolnění dat proběhne nastavením položky `exdata` na `NULL`.

```
RSA *rsa = RSA_new();
int rsa_ex_idx;

rsa_ex_idx = RSA_get_ex_new_index(0, NULL, NULL, NULL, NULL);
RSA_set_ex_data(rsa, rsa_ex_idx, "Save this string.");
RSA_get_ex_data(rsa, rsa_ex_idx);
RSA_set_ex_data(rsa, rsa_ex_idx, NULL);
```

Výpis kódu 1.1: Příklad využití exdat bez callback funkcí

1.3.4 Vstupní a výstupní operace

OpenSSL poskytuje obecnou strukturu pro zpracování vstupních a výstupních operací. `BIO`, jak je struktura pojmenovaná, není omezená na standardní file stream operace, dokáže zpracovat i komunikaci mezi serverem a klientem [24]. Jednotlivé struktury lze totiž řetězit, což otevírá možnost provádění transformací nad protékajícími daty.

`BIO` je velmi silnou abstrakcí, která odstiňuje všechny ostatní komponenty knihovny. Nešifrovanou komunikaci serveru a klienta je možné snadno navázat pomocí `BIO_new_connect()`, jejíž vstupem je jen adresa a port. Číst a psát do streamu umožňují funkce `BIO_read()` a `BIO_write()`. Uskutečnění šifrované spojení vyžaduje zavolání několika funkcí navíc a strukturu `SSL`, té je potřeba specifikovat verzi protokolu TLS a cestu k certifikátům.

1.3.5 Engine

Engine, nebo také kryptografický modul, umožňuje rozšířit funkcionalitu libcrypto přes rozhraní Engine API. Typickým užitím je nahrazení defaultní implementace kryptografických funkcí. Pokud tedy máme k dispozici knihovnu třetí strany, můžeme její výhody přivést do OpenSSL bez zásahu do již existujících řešení. Mějme například hardware, který dokáže urychlit kryptografické operace. Výrobce samozřejmě k zařízení poskytne nezbytné ovladače pro správnou funkci v běžném užití, ale nebude distribuovat instanci OpenSSL. Ani ze strany OpenSSL nepřijde přímá podpora pro toto specifické zařízení uvnitř libcrypto. Engine API nám však pomůže namapovat volání OpenSSL na funkce našeho zařízení, rozhraní můžeme chápat jako kompatibilní vrstvu pro přechod libovolné knihovny do libcrypto.

V roce 2000 existovalo OpenSSL 0.9.6 ve dvou verzích, jedna standardní a druhá s podporou engine. Až verze 0.9.7 přišla se sjednocením a Engine API se tak stalo nedílnou součástí projektu. V současné verzi je v OpenSSL implementováno devět kryptografických modulů, v projektu [25] jsou rozděleny do dvou adresářů.

V adresáři `crypto/engine/` nalezneme čtyři moduly. Devcrypto podporuje kryptografické pseudozařízení `/dev/crypto` operačního systému OpenBSD. Modul rdrand vytváří kompatibilní vrstvu pro hardwarový generátor náhodných čísel od Intelu, respektive jeho instrukci pro získání náhodného čísla. Engine openssl je v prvním komentáři zdrojového kódu označen jako „testing gunk“, implementuje RC4 a ostatní operace jsou delegovány na defaultní implementace, pravděpodobně tedy vznikl za účelem testování Engine API. Posledním je dynamic engine, který se od všech ostatních velmi odlišuje. Neplní klasickou funkci kryptografického modulu, jeho účel bude vysvětlen později.

Druhý adresář `engines/` vznikl až v pozdější verzi a skrývá zbylých pět modulů. Engine afalg byl vytvořen pro podporu linuxového crypto API. Capi engine byl vytvořen pro podporu Microsoft CryptoAPI, podrobnějšímu popisu se věnuje tato práce v kapitole 3.4. Modul padlock umožňuje kompatibilitu CPU s hardwarovou podporou kryptografických funkcí od firmy VIA Technologies. Engine dasync, zkratka pro sousloví Dummy Async, prezentuje asynchronní volání operací. Všechny jeho kryptografické funkce jsou však delegovány na defaultní implementace, je určen pouze k testování. Poslední modul ossltest má záměrně oslabené šifrovací funkce a je určen pouze pro testovací účely.

Není pravidlem, že se při kompilaci vytvoří všechny dostupné moduly. Záleží na konfiguraci prostředí, pro které se knihovna kompiluje a na parametrech makefile. Moduly, které byly staticky linkované s knihovnou libcrypto, lze vypsat v příkazovém řádku příkazem `openssl engine`. Engine API umožňuje využívat i moduly, které v době kompilace neexistovaly, a to pomocí speciálního modulu dynamic.

Možnost využívat moduly, které nejsou součástí OpenSSL, přináší řadu výhod. Například v situaci, kdy je potřeba upravit již existující s OpenSSL propojenou implementaci modulu, není nutné znovu kompilovat celou knihovnu. Stejně tak se dynamické linkování hodí, když OpenSSL neposkytuje engine ke specifickému zařízení. Můžeme pak využít engine poskytnutý výrobcem nebo si vytvořit vlastní. Dynamické linkování nám také díky svojí menší náročností na paměť pomůže s vytvořením méně nákladného řešení. Jak tyto struktury fungují, vysvětluje kapitola 3.

1.4 Knihovna libssl

Knihovna libssl implementuje protokoly SSL, TLS a DTLS. Poskytuje rozhraní SSL API, které je definované v hlavičkovém souboru `openssl/ssl.h`. Vnitřně využívá konstrukty definované v knihovně libcrypto, například rozhraní EVP. Podobně jako je tomu u libcrypto, i tato knihovna se dělí do několika komponent reprezentovaných jako struktury s vlastním rozhraním.

První strukturou, kterou je nutné vytvořit pro zahájení bezpečné komunikace, je `SSL_CTX`. Tato struktura se využívá na straně klienta i serveru, existuje po celou dobu života aplikace a specifikuje budoucí spojení. Inicializační funkce `SSL_CTX_new()` vytvoří novou strukturu a přidělí jí seznam šifer, callback funkce a výchozí nastavení parametrů. Funkci se jako argument musí předat `SSL_METHOD`, která specifikuje implementace funkcí spojené s požadovanou verzí protokolu.

S OpenSSL je dodáno několik implementací struktury `SSL_METHOD` a SSL rozhraní nabízí dva způsoby, jak je využívat. První možností je získání konkrétní verze protokolu, k tomu slouží například funkce `TLSv1_1_method()`. Druhou a doporučovanou [26] možností je využít metodu `TLS_method()`, ta vybere nejvyšší společnou verzi mezi serverem a klientem. Automatizovaná volba jde omezit zdola i shora upravením parametrů kontextové struktury.

Vytvořené spojení klienta se serverem je reprezentované strukturou `SSL`. Inicializaci lze provést tak, že se vytvoří nová `SSL` dle dříve definovaného kontextu, následně jí bude přidělen socket file descriptor, přes který po úspěšném TLS handshake probíhá komunikace [27]. Celý tento proces zastiňuje komponenta `BIO`. Konkrétní příklady užití obou způsobů jsou součástí poslední kapitoly. Implementační detaily nejsou pro tvorbu kryptografického modulu podstatné, rozhraní SSL API je však využito při jeho testování.

Kapitola 2

Kryptografické služby operačního systému Windows

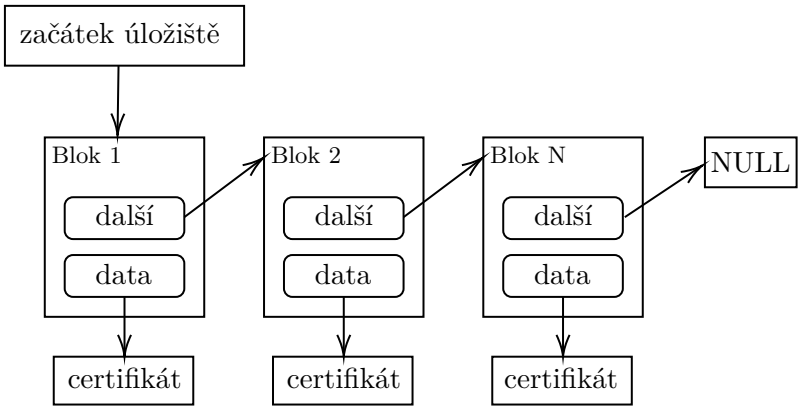
Práce se zaměřuje na analýzu a tvorbu kryptografických modulů pro OpenSSL, které zajišťují podporu kryptografických rozhraní operačních systémů Windows. S příchodem Windows Vista vzniklo nové rozhraní, jehož cílem je nahradit svého předchůdce. Vedle toho Windows nabízí úložiště certifikátů, které přináší výhody při organizaci certifikátů v různých fyzických úložištích. Tato kapitola popisuje stavbu úložišť a poukazuje na rozdíly mezi rozhraními.

2.1 Úložiště certifikátů

Důležitou součástí navázání bezpečné síťové komunikace je autentizace. Každá strana by měla být schopna doložit svoji identitu a následně ji i ověřit. Typickým prostředkem pro identifikaci subjektu na internetu je digitální certifikát. Formát certifikátu je definován standardem X.509, obsahuje informace o vlastníkovi, vystavovateli, platnosti a veřejný klíč [28].

Takových certifikátů se může na jednom zařízení nashromáždit značné množství. Pro usnadnění manipulace s těmito datovými objekty přichází CryptoAPI s nástroji, které umožňují přechovávat, mazat, vypisovat nebo ověřovat certifikáty. Jádrem celé této funkcionality je úložiště certifikátů, které dokáže trvale uchovávat certifikáty.

Úložiště certifikátů lze rozdělit do dvou druhů — logické a fyzické. Fyzické úložiště je spojový seznam certifikátů se sjednocenou strukturou. Zastiňuje formu, jakým jsou certifikáty skutečně implementovány. Takové úložiště může být například uloženo v registrech lokálního nebo vzdáleného počítače, v souboru na disku nebo na čipové kartě.



Obrázek 2.1: Reprezentace fyzického úložiště certifikátů

Obrázek výše znázorňuje, že úložiště začíná ukazatelem na první blok daného úložiště. Každý blok má ukazatel na vlastní data certifikátu v režimu pouze pro čtení a na další blok, pokud není poslední [29].

Pokud je potřeba manipulovat s velkým množství certifikátů, může jejich organizace být komplikovaná. Existence několika nezávislých fyzických úložišť způsobí, že je nebude možné snadno prohledávat. Proto byl vytvořen konstrukt logických úložišť, ten virtuálně seskupuje jednotlivá fyzická úložiště a v rámci aplikace bude vystupovat jako jeden celek. Jedno volání funkce tak může být uplatněno na několik fyzických částí. Navíc je umožněno, aby jedno fyzické úložiště bylo zahrnuto v několika skupinách, což ještě více ulehčí organizační strukturu. Při snaze přidání nového certifikátu do logické skupiny je nutné specifikovat kterému fyzickému úložišti má být připsán.

V operačním systému Windows se certifikáty dělí do skupin podle jejich využití. Jednu skupinu tvoří certifikáty určené pouze konkrétnímu uživateli, další skupina je určená pro větší množinu uživatelů, poslední je využívaná celým počítačem. Certifikáty určené pro počítač jsou mapované a zpřístupněné každému uživateli. Každá ze skupin má v základu přidělené čtyři logická úložiště certifikátů `MY`, `ROOT`, `TRUST` a `CA` [30]. Tato systémová úložiště jsou uložená v registrech. Uživatelská data skrývá registrový subklíč `HKEY_CURRENT_USER\SOFTWARE\Microsoft\SystemCertificates\`.

Úložiště tedy sjednocuje strukturu evidovaných certifikátů, bez ohledu na to, zda je certifikát ve skutečnosti datový blok na disku nebo je uložen na čipové kartě. Kromě abstrakce digitálních certifikátů přináší úložiště cerifikátů další výhodu v podobě možnosti spárovat certifikát se soukromým klíčem. Při importování nové položky do úložiště lze využít PKCS #12 formát certifikátu, který kromě standardní X.509 struktury navíc poskytuje možnost zabalení soukromého klíče. Když tento balík přidáme do úložiště, soukromý klíč bude bezpečně uložen a přiřazen k danému certifikátu.

Úložiště certifikátů je jen poskytovatel abstrakce různých způsobů imple-

mentace certifikátů a nijak neřídí práva přístupu. Veškerá záruka bezpečnosti leží na operačním systému.

2.2 CryptoAPI

Microsoft CryptoAPI, rozhraní pro zprostředkování kryptografických operací, je dodáváno do operačních systémů od firmy Microsoft v podobě knihovny Crypt32.dll. Používání CryptoAPI odstiňuje vývojáře od znalosti konkretních implementací. Implementace algoritmů zajišťuje Cryptographic Service Provider (CSP) vystupující jako nezávislý balík kryptografických funkcí.

K samotnému CSP lze přistupovat pouze prostřednictví rozhraní, nikoliv přímo. Kvůli uzavřenosti CSP tak nelze specifikovat konkrétní implementace, lze jen nařídit jaká operace se má vykonat, výběr funkce záleží na CSP. Tím by mělo být zaručeno, že vývojář aplikace nezvolí slabé nebo nevhodné řešení. CSP je tvořen dynamickou knihovnou a jedním souborem s digitálním podpisem, jenž je pravidelně kontrolován pro ujištění, že nedošlo k nějakému neoprávněnému zásahu do struktury [31].

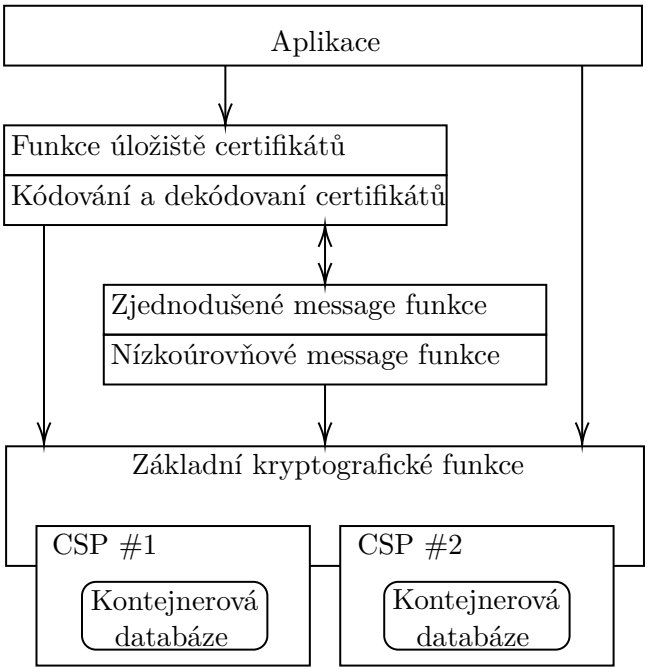
V současné době existuje mnoho různých standardů nebo protokolů, proto i CSP jsou v tomto vzoru rozdělovány do typů. Když se k aplikaci napojí CSP a specifikuje se určitý typ, lze od CryptoAPI očekávat, že volané funkce budou respektovat požadovaný standard. Součástí operačního systému jsou již předdefinované CSP a jejich typy. Například typ `PROV_RSA_FULL` podporuje šifrování dat a tvorbu digitálních podpisů s využitím RSA. Podobnou funkci plní `PROV_RSA_AES`, rozdílem je, že zahrnuje více hešovacích algoritmů a pro šifrování navíc poskytuje AES [32].

S příchodem Windows Vista se sada CSP, které jsou distribuovány společně s operačním systémem, obměnila. Například Microsoft Enhanced RSA and AES Cryptographic Provider implementuje algoritmy pro podepisování, šifrování a hešování [33].

Uvnitř každého CSP dále existuje databáze takzvaných kontejnerů pro klíč. Každý kontejner představuje dostupný bezpečně uložený klíč, například v registrech nebo v souborovém systému. Kontejner je v databázi jednoznačně identifikovatelný svým jménem, čímž se lépe organizuje množina klíčů.

Rozhraní CryptoAPI dělí svoji funkcionalitu do pěti hlavních oblastí. Základní kryptografické funkce přímo komunikují s CSP, provádějí výběr specifického CSP a tvoří přechodovou vrstvu pro funkcionalitu z jiných oblastí. Dalšími oblastmi jsou zjednodušené message funkce a nízkoúrovňové message funkce, ty zprostředkovávají kryptografické operace jako je šifrování a dešifrování dat, tvorba a ověřování digitálního podpisu nebo hešování. Rozdílem mezi nimi je, že nízkoúrovňové funkce jsou více flexibilní na úkor jednoduchosti jejich volání. Výše zmiňované funkce jsou považovány za zastaralé, protože byly nahrazeny novějším rozhraním. Zbylé dvě oblasti jsou věnovány práci s certifikáty, zahrnují právě funkce s předponou `Cert`, tedy funkce pro manipulaci

s úložištěm a certifikáty samotnými. Nové rozhraní je nijak neupravuje a jsou stále aktuální [34].



Obrázek 2.2: Zjednodušená architektura CryptoAPI

2.2.1 Manipulace s úložišti

Rozhraní CryptoAPI poskytuje funkce pro práci s úložišti certifikátů a jejich certifikáty, vyznačují se prefixem `Cert` v jejich názvu.

Práce s úložištěm začíná jeho otevřením. Kromě jeho jména je nutné specifikovat o jaký zdroj se jedná, zda je to systémové úložiště, dočasný paměťový blok, nebo soubor na disku. Navíc lze specifikovat další kritéria, například vynucení otevření úložiště v režimu jen pro čtení, zpřístupnění pouze uživatelských certifikátů, a tak podobně. Pokud se otevírá virtuální úložiště, otevřou se zároveň všechna fyzická úložiště zahrnutá v tomto seskupení.

```
HCERTSTORE certStore;
certStore = CertOpenStore(CERT_STORE_PROV_SYSTEM_A,
                          X509_ASN_ENCODING, NULL,
                          CERT_STORE_READONLY_FLAG, "MY");
```

Výpis kódu 2.1: Otevření úložiště certifikátů

Ukázka 2.1 představuje otevření systémového úložiště certifikátů `MY` v režimu jen pro čtení. Po získání reference `HCERTSTORE` lze pracovat s uloženými

certifikáty. První kroky mohou vést k získání seznamu dostupných certifikátů, k čemuž je určena enum funkce.

```
PCCERT_CONTEXT cert = NULL;
while(1) {
    cert = CertEnumCertificatesInStore(certStore, cert);
    if (!cert)
        break;
}
```

Výpis kódu 2.2: Listování úložištěm certifikátů

Připravená enum funkce s každým zavoláním vrátí buď odkaz na jeden certifikát, nebo `NULL`. Proto je nutné volat ji ve smyčce, dokud existuje další certifikát. Pokud argumentem, vedle odkazu na úložiště, není `NULL`, vrátí se certifikát následující po tomto specifikovaném.

CryptoAPI dále poskytuje velmi variabilní funkci pro vyhledávání certifikátu. Nabízí velké množství vyhledávacích kritérií [35].

```
PCCERT_CONTEXT cert = NULL;
CertFindCertificateInStore(certStore,
                           X509_ASN_ENCODING, 0,
                           CERT_FIND_SUBJECT_STR_A,
                           "VeriSign", NULL);
```

Výpis kódu 2.3: Vyhledávání certifikátu v úložišti

Výpis 2.3 prezentuje možný způsob vyhledávání certifikátu v úložišti. Čtvrtý vstupní argument určuje metodu použitou pro hledání, od toho se odvíjí následující argument, který slouží jako vyhledávací klíč. V ukázkovém příkladu se pokusí funkce nalézt certifikát vystavený pro subjekt, jehož common name obsahuje podřetězec VeriSign. V případě, že takovému kritériu nějaká položka úložiště vyhovuje, vrátí se reference prvního nalezeného. Pro získání dalšího je nutné předat jako poslední vstupní argument funkce referenci certifikátu získanou předchozím voláním.

Struktura `CERT_CONTEXT` reprezentuje certifikát v rozhraní CryptoAPI. Nabízí jak zakódovaná, tak strukturovaná data typická pro certifikát. Kromě toho na sebe struktura váže další vlastnosti dosažitelné specializovanou funkcí.

```
DWORD len;
LPWSTR fname;

CertGetCertificateContextProperty(cert,
                                  CERT_FRIENDLY_NAME_PROP_ID,
                                  fname, &len);
```

Výpis kódu 2.4: Získání kontextových dat certifikátu

Protože friendly name není standardní součást formátu X.509, není přímo dostupná z reference `CERT_CONTEXT`. Výpis kódu 2.4 ukazuje možný způsob získání položky friendly name ze struktury `cert`. Funkci lze využít i k získání jiných dat. Například struktury, která identifikuje kontejner pro klíč uvnitř CSP spojený s tímto certifikátem. Výstup funkce se řídí druhým parametrem, jejichž výčet nabízí dokumentace [36]. Pokud funkce vrací nějakou strukturu, je vhodné ji volat dvakrát. Poprvé kvůli získání velikosti, kterou je potřeba alokovat, a podruhé kvůli získání požadovaných dat. Výpis však první volání a následnou alokaci z důvodu čitelnosti neznázorňuje. Datový typ `LPWSTR` je win32 značení pro wide char řetězec, respektive řetězec unicode znaků.

Každá reference `CERT_CONTEXT` by měla být správně uvolněna voláním `CertFreeCertificateContext()` a úložiště by mělo být uzavřeno pomocí `CertCloseStore()`

2.2.2 Aplikace CryptoAPI

Aplikace, jejíž úmyslem je využívat CryptoAPI, musí nejdříve získat referenci na CSP, který má příkazy plnit.

```
HCRYPTPROV csp;
CryptAcquireContextA(&csp, container_name,
                     csp_name, csp_type,
                     flags);
```

Výpis kódu 2.5: Získání CSP reference

Ve výpisu 2.5 byly záměrně pojmenovány vstupní parametry, aby bylo možné se na ně nyní odkazovat. Typ, respektive identifikace balíku algoritmů, musí být nutně předána. Jméno CSP a jméno kontejneru je volitelné, ale je potřeba je použít pokud je úmyslem získat specifický kontejner konkrétního CSP. Pokud je argumentem předán jen typ, vybere se k němu příslušný výchozí CSP. Jak se funkce vypořádá s chybějícím jménem kontejneru, závisí na konkrétním CSP. Posledním argumentem lze specifikovat, zda aplikace vyžaduje uživatelovu sadu klíčů nebo klíče spojené s lokálním zařízením, nebo jestli je vůbec potřeba touto referencí ke klíčům přistupovat [37].

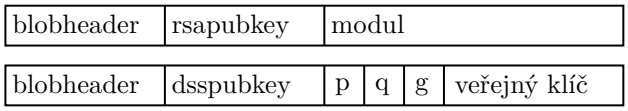
Pokud bylo funkci `CryptAcquireContext()` předáno jméno kontejneru, aplikace pravděpodobně plánuje získat přímou referenci na klíč a s ním dále pracovat.

```
HCRYPTKEY key;
CryptGetUserKey(csp, AT_KEYEXCHANGE, &key);
```

Výpis kódu 2.6: Získání klíče z CSP

Protože všechna data potřebná k identifikaci klíče shromažďuje `HCRYPTPROV`, je získání klíče velmi snadné. Stačí navíc určit takzvaný keyspec, který může nabývat dvou stavů. Klíč s označením `AT_KEYEXCHANGE` může být použit pro podepisování i šifrování, zatímco `AT_SIGNATURE` slouží jen k podepisování.

Reference `HCRYPTKEY` slouží jen jako identifikátor v rámci CSP a neumožňuje přímý přístup ke klíči. Pro získání dat je nutné exportovat klíč z kontejneru. K tomu je určena funkce `CryptExportKey()`, která dokáže serializovat klíč do datového bloku, označovaného jako blob. Stuktura takového blobu je závislá na tom, zda byl exportován veřejný nebo soukromý klíč, a zda je určen k DSA nebo RSA operacím [38]. Obrázek 2.3 naznačuje rozložení dat pro exportovaný veřejný RSA a DSA klíč.



Obrázek 2.3: Rozložení dat v publickeyblob pro RSA a DSA

První bajty každého typu blobu vytváří strukturu `BLOBHEADER`. V ní je vyjádřeno, o jaký typ blobu se jedná a jaký algoritmus reprezentuje. V případě RSA klíče následuje struktura `RSAPUBKEY`, z které lze vyčíst exponent a počet bitů v modulu. Pak už zbývá jen samotný modul v little-endian formě. V případě DSA klíče přímo za hlavičkou leží struktura `DSSPUBKEY`, jejíž obsahem je počet bitů tvořící modul `p`. Dále následují zbylé parametry opět v little-endian formě.

Jako vhodná ukázka použití kryptografických algoritmů v rámci CryptoAPI může být vytvoření digitálního podpisu.

```
HCRYPTHASH hash_obj;

CryptCreateHash(csp, CALG_MD5, 0, 0, &hash_obj);
CryptSetHashParam(hash_obj, HP_HASHVAL, digest, 0);
CryptSignHash(hash_obj, AT_KEYEXCHANGE, NULL, 0, sig, len);
```

Výpis kódu 2.7: Vytvoření digitálního podpisu pomocí CryptoAPI

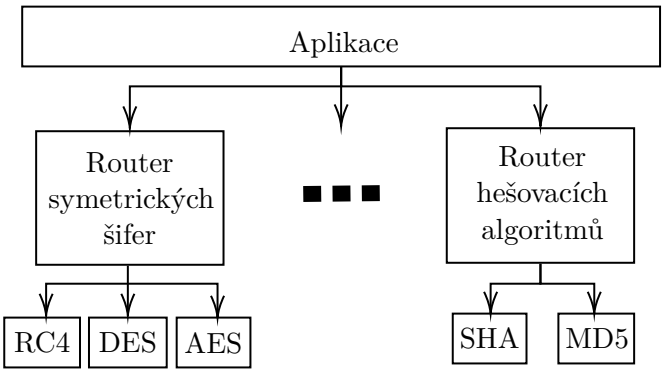
Pro jednoduchost je považována varianta, kdy již došlo k vytvoření heše zprávy pomocí algoritmu MD5 a je předán řetězcem `digest`. Dále nechť existuje buffer `sig` s alokovanou velikostí `len`. Rozhraní CryptoAPI neumožňuje předat hash zprávy v podobě řetězce rovnou k podepsání. Zpráva musí být předána prostřednictvím hash objektu, k jehož vytvoření je potřeba znát jakým algoritmem je zpráva zahešovaná. Hash zprávy se předá inicializované referenci `HCRYPTHASH` a až následně je možné provést podepsání. Funkce pro výpočet digitálního podpisu již neočekává referenci na používaného CSP, protože potřebnými informacemi disponuje již hash objekt. Jen podobně jako v případě `CryptGetUserKey()` je nutné určit keyspec. Po dokončení posledního řádku výpisu 2.7 je digitální podpis v little-endian formě uložen do bufferu `sig`.

2.3 Cryptography API: Next Generation

Rozhraní Microsoft Cryptography API: Next Generation (CNG) vzniklo s operačním systémem Windows Vista jako náhrada CryptoAPI. Současně existují obě rozhraní vedle sebe. Architektura CNG je ve svém dělení mnohem radikálnější, odděluje funkce označovány jako kryptografická primitiva od funkcí, jejichž funkcionalita vyžaduje perzistentní kryptografické klíče. Každé je dedikována i vlastní knihovna.

2.3.1 BCrypt

Funkce s prefixem `BCrypt` jsou určeny k hešování a symetrickému šifrování. Algoritmy asymetrické kryptografie implementuje knihovna také, ale dokáže pracovat pouze s dočasnými klíči. Jednotlivé kryptografické algoritmy jsou tříděny do tříd reprezentovaných takzvaným routerem, který má přehled o implementacích různých poskytovatelů. Aplikace využívající rozhraní CNG nemíří volání na konkrétní implementaci, komunikuje právě přes routery, které vyberou vhodný modul vybraného poskytovatele k vykonání požadované operace [39], jak naznačuje obrázek 2.4.

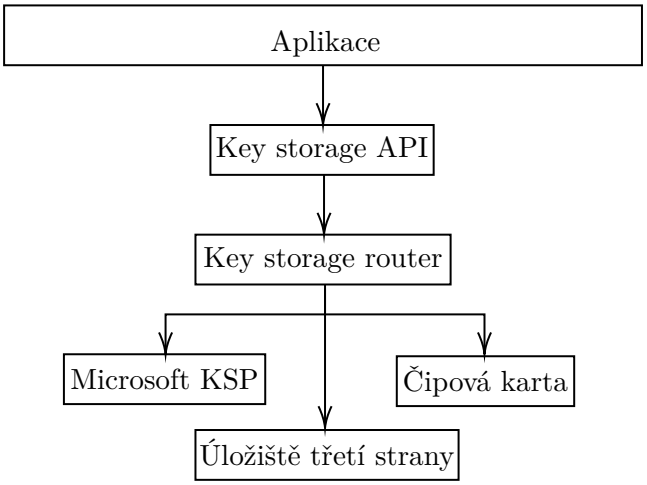


Obrázek 2.4: Zjednodušená architektura BCrypt

Pro vykonávání operací prostřednictvím BCrypt je nutné zvolit jednoho poskytovatele implementace. Volba závisí na kryptografickém algoritmu, který bude v následujících krocích vykonáván. Poskytovatele lze blíže určit pomocí jeho jména. V případě, že jméno specifikováno nebude, vybere se první dostupný poskytovatel pro daný algoritmus. Seznam všech poskytovatelů a algoritmů je dostupný z dokumentace [40], lze ho získat i prostřednictvím enum funkcí. Poté, co má aplikace k dispozici referenci na poskytovatele, může začít využívat kryptografické funkcionality.

2.3.2 NCrypt

CNG přichází s vlastní metodikou pro ukládání soukromých klíčů. NCrypt implementuje rozhraní, které odstiňuje aplikaci od izolačního procesu pro uložení klíče zvané Key storage API. Úložiště samotné je spravováno konstruktem označovaným jako Key Storage Provider (KSP). Uvnitř Key Storage rozhraní, je router, který eviduje všechny dostupné KSP a přesměrovává na ně operace podle přání aplikace. Z pohledu aplikace tedy nezáleží, s jakým úložištěm klíčů se zrovna pracuje. Obrázek 2.5 hrubě naznačuje strukturu NCrypt [41].



Obrázek 2.5: Zjednodušená architektura NCrypt

Microsoft Software KSP je defaultně součástí operačních systémů Windows. Pro získání všech dostupných poskytovatelů je v NCrypt připravena enum funkce `NCryptEnumStorageProviders()`, které se předá adresa pole struktur `NCryptProviderName`. Funkce si sama alokuje dostatek paměti a za každý dostupný KSP dosadí jednu strukturu do pole, z které lze následně získat jméno poskytovatele v podobě unicode řetězce.

Aplikace, jejíž úmyslem je využívat funkcionalitu knihovny NCrypt, musí nejdříve získat referenci na KSP.

```
NCRYPT_PROV_HANDLE ksp;

NCryptOpenStorageProvider(&ksp,
                          MS_KEY_STORAGE_PROVIDER, 0);
```

Výpis kódu 2.8: Získání KSP reference

Jediné čím se KSP identifikuje, je unicode řetězec se jménem. Součástí knihovny NCrypt jsou předdefinované konstanty zastupující úplný název KSP. Výpis 2.8

získá referenci poskytovatele jménem Microsoft Software Key Storage Provider.

Pokud aplikace využívá knihovnu NCrypt, pravděpodobně plánuje manipulovat s trvalými klíči. Po získání reference konkretního KSP lze získat seznam všech dostupných klíčů. K tomu je opět připravena enum funkce.

```
NCryptKeyName *key;
PVOID state = NULL;
SECURITY_STATUS status;

status = NCryptEnumKeys(ksp, NULL, &key, &state, 0);
while (status == ERROR_SUCCESS) {
    /* prostor pro práci s klíčem */
    NCryptFreeBuffer(key);
    status = NCryptEnumKeys(ksp, NULL, &key, &state, 0);
}
```

Výpis kódu 2.9: Iterace přes klíče v daném KSP

Funkci `NCryptEnumKeys()` je nutné volat v cyklu, protože jedno její zavolání získá nejvýše jeden klíč. Je potřeba ji specifikovat s jakým KSP se aktuálně pracuje. Třetí parametr je adresa ukazatele na strukturu, která přijme informace o klíči. Kromě jména klíče je možné získat i jméno kryptografického algoritmu, pro který je klíč určen a hodnotu keyspec, jejíž význam je totožný s CryptoAPI. Proměnná `state` funguje jako mezipaměť pro orientaci, které klíče již byly získány. Posledním argumentem je flag, který rozhoduje, zda bude iterováno přes uživatelské klíče nebo přes klíče lokálního stroje. Pro NCrypt funkce je typické, že mají číselnou návratovou hodnotu, již je možné porovnávat s předdefinovanými konstantami a získat podrobnější odezvu o běhu funkce. V případě úspěchu je vrácena hodnota `ERROR_SUCCESS`. Referenci na konkrétní klíč lze získat na základě jeho jména.

```
NCRYPT_KEY_HANDLE key;
NCryptOpenKey(ksp, &key, key_name, AT_KEYEXCHANGE, 0);
```

Výpis kódu 2.10: Získání reference klíče

Kromě jména klíče je nutné funkci `NCryptOpenKey()` specifikovat keypec a jako poslední argument informaci, zda se jedná o klíč pro zařízení nebo pro uživatele. Pokud návratovou hodnotou funkce je `NTE_BAD_KEYSET`, znamená to, že hledaný klíč v daném KSP neexistuje.

Instance `NCRYPT_KEY_HANDLE` ani v případě NCrypt neposkytuje přímý přístup k datům klíče a opět je potřeba klíč exportovat pomocí blobu. Oproti CryptoAPI funkci pro export klíče, nabízí `NCryptExportKey()` možnost zašifrovat blob pro koncového uživatele pomocí jeho klíče. Vhodným postupem,

jak získat veřejná data z reference klíče, je vyžádat si obecný blob klíče označovaný jako `BCRYPT_PUBLIC_KEY_BLOB`, z kterého lze vyčíst číslo `magic`. To jednoznačně identifikuje, zda se jedná o veřejná nebo soukromá data a jakému algoritmu klíč náleží. Pak lze obecný datový blob přetransformovat do specializovaného, a tím strukturovaně k datům přistupovat [42].

Obrázek 2.6: Rozložení dat blobu veřejného klíče pro RSA a ECDSA

Na obrázku 2.6 je znázorněné rozložení dat v blobu pro veřejný RSA klíč a pro veřejný ECDSA klíč. Struktury na začátku blobu obsahují velikosti následujících dvou parametrů. NCrypt exportuje data do bloků v big-endian formě.

Kromě manipulace s klíči nabízí funkce s předponou `NCrypt` vykonávání asymetrických kryptografických operací, jako je tvorba digitálního podpisu. Podobně jako u prezentace CryptoAPI, i zde předpokládejme již zahešovanou zprávu `digest` algoritmem MD5 a připravený buffer `sig`.

```
BCRYPT_PKCS1_PADDING_INFO PKCS1PaddingInfo = {NID_md5};
NCryptSignHash(key,
               &PKCS1PaddingInfo,
               digest, digest_len,
               sig, sig_len, &sig_len,
               BCRYPT_PAD_PKCS1);
```

Výpis kódu 2.11: Vytvoření digitálního podpisu pomocí NCrypt

Jako první parametr se funkci pro vytvoření digitálního podpisu předává reference na soukromý klíč. Druhý argument se odvíjí od posledního. NCrypt totiž dokáže pracovat se dvěma podpisovými schématy — PKCS1 a PSS. Pokud se pro podpis využívá PSS, je potřeba přes druhý argument specifikovat navíc délku soli. Dále se funkci předá už jen hash a její délka, buffer, do kterého bude předán podpis v big-endian formě a délka bufferu, jejíž hodnota bude přepsána skutečnou velikostí vytvořeného podpisu. Podobným stylem, avšak bez druhého a posledního parametru, lze vytvořit podpis i pomocí jiných algoritmů, například ECDSA.

2.4 Porovnání

Obě kryptografická rozhraní poskytují stejnou sadu algoritmů. CryptoAPI zatěžuje CSP pro jakoukoliv kryptografickou operaci. Cryptography API: Next Generation umožňuje větší flexibilitu díky svému jasnému rozdělení do dvou

knihoven a modularitě. Co se architektury týče, je zde znatelný posun k modernějším principům.

Změnil se i způsob ukládání klíčů a jejich umístění v souborovém systému. Zatímco starší CryptoAPI využívá pro přechovávání soukromých klíčů adresář `%APPDATA%\Microsoft\Crypto\RSA\User-SID\` nebo podobně pro DSA, rozhraní CNG klíče nerozděluje a nevytváří podadresář pojmenovaný podle uživatelova SID. Což nezapříčiní nedostupnost klíčů v případě změny domény. Soukromé klíče ukládá do `%APPDATA%\Microsoft\Crypto\Keys\`.

V ohledu kompatibility klíčů je novější rozhraní vstřícné, umožňuje číst klíče uložené pomocí CAPI. Pokud si aplikace vyžádá uložený klíč, CNG se ho nejdříve pokusí vyhledat ve svém adresáři a v případě neúspěchu se jej pokusí načíst z CAPI adresářů. Obrácená kompatibilita z CAPI do CNG ale možná není. Z toho důvodu umožňuje CNG při ukládání klíče vytvořit dva soubory, jeden pro své použití a druhý pro starší rozhraní [41].

Odlišný je i způsob reference klíče z aplikace. V CAPI je klíč definovaný trojicí jméno CSP, typ CSP a jméno kontejneru. Již inicializovanému `HCRYPTPROV` nelze měnit aktivní kontejner a je nutné provést novou inicializaci. CNG je přizpůsobeno k používání několika klíčů v rámci jednoho poskytovatele, každý klíč je definován svým jménem v předem existující KSP referenci.

K žádné změně nedošlo v rozhraní pro práci s úložišti certifikátů. NCrypt s takovou funkcionalitou nepřichází a tak jediná možnost je i nadále CryptoAPI. Některé z funkcí však byly upraveny, aby umožnily například exportovat klíč i do CNG reprezentace.

Logika, že nositelem implementací jsou balíky funkcí, mezi kterými lze volit, je zachována. Nabízená sada poskytovatelů je ale obměněná. Každé kryptografické funkci existující v CAPI je nabídnut protějšek v CNG rozhraní. Zacházení s nimi je odlišné, například právě při tvorbě digitálního podpisu. Předání dat funkci pro spočítání podpisu se ve starším rozhraní provádělo prostřednictvím hash objektu, nové rozhraní dokáže zprávu přijmout přímo a navíc rozlišuje i podpisová schémata. Zásadní změnou je změna formy uchování dat. Zatímco starší CryptoAPI dodržovalo little-endian, novější CNG přechovává data v big-endian podobě.

Kapitola 3

Kryptografický modul pro OpenSSL

Zatímco kapitola 1.3.5 popisovala kryptografické moduly velice povrchně, tato část práce nabídne podrobnější popis implementace Engine API. Tyto znalosti budou následně obohaceny o vysvětlení funkčnosti modulu dynamic. Kapitola bude zakončená analýzou modulu e_capi a průvodcem tvorby modulu pro podporu Microsoft Cryptography API: Next Generation.

3.1 Struktura modulu

Samotný engine je implementován jako struktura, do které se přiřadí konkrétní implementace funkcí pomocí ukazatelů na funkce nebo na kryptografické struktury známé z první kapitoly. Každý engine je identifikovatelný svým id, lze mu také přiřadit jméno, které slouží jako stručný popis.

3.1.1 Inicializační funkce

Pro inicializaci modulu jsou určeny funkce `bind()`, která se volá implicitně při načtení modulu do vnitřní struktury, a `ENGINE_init()`, již musí aplikace volat explicitně až po načtení. Posláním `bind()` je připravit strukturu `ENGINE`, tedy pomocí `ENGINE_set_*` funkcí namapovat vytvořené funkce na `ENGINE` strukturu tak, aby se s nimi dalo manipulovat prostřednictvím Engine API. Tím se mimo jiné nadefinuje právě `ENGINE_init()`, která musí zajistit funkcionalitu kryptografických implementací. Typicky to znamená, že se přidělí implementace kryptografickým strukturám, čímž se nahradí defaultní chování libcrypto.

Podobně jsou k uvolnění prostředků připraveny také dvě funkce. Záměrem `ENGINE_destroy()` je uvolnit prostředky vytvořené v `bind()`, kdežto funkce `ENGINE_finish()` uklízí po funkci `ENGINE_init()` [43].

3.1.2 Podpora dynamického načítání

Kryptografický modul, který je součástí projektu OpenSSL, je možné načíst interně již při inicializaci libcrypto pomocí `CRYPTO_init()`. Od modulu se v tomto případě očekává funkce, která vykoná `bind()` a přidá engine do vnitřního spojového seznamu.

V případě dynamického načítání je situace mírně odlišná [44]. Nejdříve dojde k ujištění, zda kryptografický modul vůbec podporuje kompatibilní verzi dynamického načítání. Tato kontrola probíhá porovnáním konstanty `OSSL_DYNAMIC_VERSION` modulu s `OSSL_DYNAMIC_OLDEST` definovaným rozhraním Engine API. Dále, pokud aplikace žádá sdílení statických dat s modulem, je potřeba předat funkce pro správu paměti. Následně je nutné definovat funkci, která provede porovnání id hledaného modulu s id načítaného modulu. V případě shody se zavolá funkce `bind()`. Pro celý tento proces jsou připravena následující makra, která stačí vložit do zdrojového kódu kryptografického modulu.

```
IMPLEMENT_DYNAMIC_CHECK_FN()
IMPLEMENT_DYNAMIC_BIND_FN(bind_helper)
```

Výpis kódu 3.1: Makra pro dynamické načtení

3.1.3 Kryptografická funkcionalita

Hlavní motivací pro tvorbu kryptografického modulu je nahrazení výchozí funkcionality části knihovny libcrypto za účelem kompatibility OpenSSL s nějakým externím zařízením nebo knihovnou. Engine ve své připravené struktuře umožňuje definovat nízkoúrovňové kryptografické moduly i volání EVP API.

Pokud je cílem modulu pouze upravit vybrané kryptografické algoritmy a není nutné pozměňovat způsob jejich volání z vyšších vrstev, pravděpodobně stačí vytvořit nositele implementací `*_METHOD` pro příslušný kryptografický algoritmus. Pokud například engine implementuje funkci `rsa_sign_fn()` určenou pro vytvoření digitálního podpisu algoritmem RSA, přiřadíme ji kryptografickému modulu následovně.

```
RSA_METHOD* my_rsa = RSA_meth_new("MyRSA", 0);
ENGINE_set_RSA(engine, my_rsa);
RSA_meth_set_sign(my_rsa, rsa_sign_fn);
```

Výpis kódu 3.2: Přidělení nové RSA_METHOD struktuře ENGINE

Způsobem naznačeným výpisem 3.2 je možné postupně vyčerpat celou strukturu `RSA_METHOD`. Pokud by se pro ostatní RSA operace měly využívat výchozí implementace, lze to vyřešit prostřednictvím `RSA_PKCS1_OpenSSL()`, jak naznačuje výpis 3.3 . Výstupem tohoto volání je defaultní `RSA_METHOD`, ze které se získají ukazatele na jednotlivé funkce a ty se přidělí nové struktuře.

```
RSA_METHOD *default_rsa = RSA_PKCS1_OpenSSL();
RSA_meth_set_pub_enc(my_rsa, RSA_meth_get_pub_enc(default_rsa));
```

Výpis kódu 3.3: Využití defaultní implementace

Zkratkou pro řešení, při kterém je struktura `*_METHOD` zastoupená převážně defaultními funkcemi, může být vytvoření kopie již existující struktury a následné nahrazení jen vybraných funkcí.

```
RSA_METHOD *dup_rsa = RSA_meth_dup(RSA_PKCS1_OpenSSL());
RSA_meth_set_sign(my_rsa, rsa_sign_fn);
```

Výpis kódu 3.4: Kopírování defaultní RSA_METHOD

Tento postup, při kterém se modulu přiřadí pouze obměněná nízkoúrovňová kryptografická struktura, je umožněn pouze strukturám pro asymetrickou kryptografii [18]. Pokud engine usiluje o vlastní implementaci hešovacích algoritmů nebo symetrických šifer, je nutný zásah do EVP struktur. V tom případě už není nahrazení výchozí implementace tak přímočaré, protože vedle definice struktury je nutné navíc vytvořit callback funkci.

Jako příklad je uveden engine, jehož cílem je poskytnou implementaci algoritmů MD5 a SHA1.

```
EVP_MD evp_md5  = {...};
EVP_MD evp_sha1 = {...};
int my_digest_nids[] = {NID_md5, NID_sha1, 0};

int my_digests(ENGINE *e, const EVP_MD **digest,
               const int **nids, int nid) {...}
```

Výpis kódu 3.5: Přiřazení EVP struktur

Výpis kódu 3.5 zobrazuje situaci, kdy uvnitř kryptografického modulu existují dvě `EVP_MD` struktury. Obě na sebe mají namapované funkce nezbytné pro vykonání operací, ke kterým jsou určeny. Jejich napojení do struktury `ENGINE` probíhá prostřednictvím callback funkce `my_digest`, jež plní dvojí úlohu. Pokud je na vstupu jako druhý argument předán `NULL`, volající očekává, že mu bude přes `**nids` dodán seznam identifikátorů všech algoritmů, které jsou k dispozici a návratovou hodnotou bude jejich počet. Ve druhém případě volající předá číselný identifikátor `nid`, kterým lze jednoznačně určit jakou EVP

strukturu přiřadit ukazateli `digest`. Funkci `my_digest()` je nutné přiřadit modulu pomocí `ENGINE_set_digests()`.

Stejný princip se dodržuje i v případě `EVP_CIPHER` a `EVP_PKEY`. Vždy je nutné vytvořit pro každý algoritmus právě jednu strukturu a ty organizovat prostřednictvím speciální callback funkce, která je buď volána s argumentem `nid`, nebo s připraveným ukazatelem na seznam dostupných algoritmů.

Asymetrické kryptografii jsou ve struktuře `ENGINE` dále vyhrazeny funkce `ENGINE_load_private_key()` a `ENGINE_load_public_key()`, jejichž úlohou je převést soukromý nebo veřejný klíč do OpenSSL reprezentace `EVP_PKEY`. Pro bližší určení klíče je vyhrazen argument datového typu `char*`. Rozdílnou úlohu plní `ENGINE_load_ssl_client_cert()`, jež nalezne využití na straně klienta v případě, že po něm server vyžaduje autorizaci formou zprávy Certificate Request. Součástí žádosti je typicky jmenný seznam certifikačních autorit, který je server ochoten akceptovat. Funkce v modulu tento seznam příjme a na jeho základě vyhledá soukromý klíč a certifikát, který byl nějakou dodanou autoritou vystaven. Implementace těchto tří funkcí má smysl, pokud engine využívá externí úložiště certifikátů nebo klíčů, a tak je nutné převádět cizí reprezentaci do OpenSSL `EVP_PKEY` nebo `X509` struktur. Tyto funkce naleznou využití především při konfiguraci `SSL_CTX` před zahájením komunikace. Protože všechny konají na základě jmenného identifikátoru, může se stát, že vyhledávacímu kritériu vyhoví více položek. Aby bylo možné přenechat konkrétní volbu na uživateli, vstupuje do všech funkcí struktura `UI_METHOD`, přes kterou lze zprostředkovat interakci v podobě dialogových oken.

3.1.4 Další funkcionalita

Pokud by engine měl podporovat další interaktivní funkcionalitu nad rámec připravené předlohy, je možné uvnitř modulu nadefinovat rozšiřující sadu příkazů.

Nejprve je nutné vytvořit pole struktur `ENGINE_CMD_DEFN`, přičemž každé z nich je přiřazen celočíselný identifikátor, název, popis a flag. Na identifikátor je kladena podmínka, že musí mít hodnotu alespoň `ENGINE_CMD_BASE` definovanou v `engine.h` jako 200. Identifikátory s nižší hodnotou jsou rezervovány pro vnitřní příkazy nebo příkazy sdílené napříč všemi moduly. Od výsledného pole se očekává, že bude seřazeno ve vzestupném pořadí podle hodnoty. Název může být využit k identifikaci příkazu při jeho volání, popis má pouze informativní hodnotu. Flag napovídá, jaký typ vstupu příkaz očekává. Na výběr je pouze mezi celým číslem, řetězcem a žádným parametrem. `ENGINE_CMD_FLAG_INTERNAL` je čtvrtou možností a znamená, že příkaz nebude zahrnut ve výpisu dostupné funkcionality a ani nebude dostupný běžným způsobem, pouze přímým voláním `ENGINE_ctrl()` [44]. Dalším krokem je vytvořit funkci, která bude reagovat na požadavky spojené s příkazy z definovaného pole.

```
#define CMD_LIST (ENGINE_CMD_BASE)
#define CMD_SET  (ENGINE_CMD_BASE + 1)

ENGINE_CMD_DEFN cmd_defns[] = {
    {0, CMD_LIST, "list", ENGINE_CMD_FLAG_NO_INPUT},
    {0, CMD_SET, "set", ENGINE_CMD_FLAG_STRING},
    {0, NULL, NULL, 0}
}
int ctrl(ENGINE *e, int cmd,
         long i, void *p, void (*f) (void)) {...}
```

Výpis kódu 3.6: Reprezentace nadstandardních příkazů

Výpis kódu 3.6 naznačuje typický přístup k vytvoření nadstandardní funkcionality modulu. Nadefinované pole `cmd_defns[]` je nutné předat struktuře `ENGINE` pomocí `ENGINE_set_cmd_defns()`. Funkce `ctrl()` je využita pokaždé, když si aplikace vyžádá provedení nadstandardního příkazu. První argument `cmd` slouží jako identifikátor příkazu, podle kterého funkce rozhodne o dalším postupu. Následující argumenty jsou určeny k předání parametru danému příkazu, typicky je aktivní jen jeden z nich. I funkci `ctrl()` je nutné přiřadit struktuře `ENGINE`, tentokrát přes `ENGINE_set_ctrl_function()`.

3.2 Engine API

Engine API je komplexní rozhraní, které umožňuje rozšířit nebo upravit výchozí implementaci knihovny libcrypto. API je deklarováno v hlavičkovém souboru `openssl/engine.h`. Implementace funkcí a struktur jsou uloženy v adresáři `crypto/engine/`.

3.2.1 Vytvoření instance

Aby bylo možné začít kryptografické moduly využívat, musí se nejdříve načíst do aplikace. Všechny moduly, které jsou staticky slinkovány s projektem OpenSSL, lze zpřístupnit voláním `ENGINE_load_builtin_engines()`. Tím se načtou do vnitřního spojového seznamu a budou zabírat část paměti. Další možností je využít dynamické načítání, čímž se značně sleví v náročnosti na paměť, protože se vždy využívá jen explicitně načtený modul [45]. Poté, co se modul stane součástí aplikace, lze se na něj odkazovat strukturou `ENGINE`.

`ENGINE` eviduje dva druhy referencí — strukturální a funkcionální. Strukturální reference vyjadřuje existenci ukazatele na konkrétní `ENGINE`, tedy kryptografický modul je načten jako část aplikace a funkce `bind()` skončila úspěchem. Tohoto stavu lze docílit například funkcí `ENGINE_by_id()`. Každá reference by měla být po ukončení práce uvolněna voláním `ENGINE_free()`. Tím do-

jde k dekrementaci počítadla strukturálních referencí a po uvolnění poslední reference dojde ke skutečnému dealokování prostředků, respektive k zavolání funkce namapované v modulu na `ENGINE_destroy()` [46].

Funkce rozhraní Engine API, které přijímají strukturální referenci jako argument, si vždycky vytvoří svoji vlastní kopii. Volající funkce by tedy každopádně měla zajistit uvolnění vytvořené reference. Pouze v případě, že uvnitř volané funkce reference zanikne, se o uvolnění postará tato volaná funkce.

Pomocí strukturální reference je možné číst informace o modulu, kupříkladu id, jméno, získat seznam podporovaných kryptografických operací nebo seznam nadstandardních příkazů. Obecně platí, že pomocí strukturální reference není možné využívat kryptografickou funkcionalitu modulu. K tomu účelu je potřeba získat funkcionální referenci. Ta teprve zaručuje připravenost modulu k vykonávání implementovaných operací. Existence funkcionální reference tedy implikuje existenci strukturální reference, nikoliv však naopak. Z pohledu implementace se na tyto dvě položky nahlíží nezávisle [43].

Funkcionální referenci lze získat více způsoby, ten nejvyužívanější je prostřednictvím funkce `ENGINE_init()`. Pokud návratovou hodnotou není 0, ze struktury `ENGINE`, jejíž ukazatel byl funkci předán, se stane plnohodnotný kryptografický modul. I každá funkcionální reference by měla být poctivě uvolněna, a to pomocí `ENGINE_finish()`, což uvnitř struktury vyvolá stejně se nazývající funkci. Druhým způsobem, jak začít využívat kryptografický modul, je získat jeho ukazatel dotazem na výchozí `ENGINE`, který implementuje operace pro danou kryptografickou strukturu. `ENGINE_get_default_RSA()` budiž příkladem pro získání modulu, který implementuje RSA operace.

3.2.2 Výchozí modul

Každý engine poskytuje init funkci pro jeho načtení do vnitřní reprezentace tak, aby bylo možné ho využívat za běhu aplikace. Všechny kryptografické moduly, které si aplikace vyžádá, vytvoří na pozadí spojový seznam tvořený `ENGINE` strukturami.

Aby kryptografická struktura ke svým činnostem využívala nějaký z dostupných modulů, je potřeba tuto informaci explicitně uvést. To se může provést ihned při inicializaci nové struktury předáním ukazatele na `ENGINE`. Další možností je využít globální předurčení defaultního modulu za pomocí rozhraní Engine API. Příkladem budiž funkce `ENGINE_set_default_RSA()`, která jako jediný argument očekává ukazatel na strukturu `ENGINE`, jehož implementace RSA operací bude nadále využívána.

Za běhu aplikace se tak evidentně může nashromáždit značné množství kryptografických modulů. Aby bylo vždy jednoznačné, jaký právě zvolit, implementuje OpenSSL strukturu nazývanou jako `ENGINE_TABLE`.

`ENGINE_TABLE` má za úkol setřídit moduly podle poskytovaných operací a určit, jaký engine je pro konkrétní algoritmus považován za výchozí. Klíčem tabulky je identifikátor kryptografického algoritmu `nid`. Dalším sloupcem je

seznam všech modulů, které implementují evidovaný algoritmus, následuje ukazatel na jeden výchozí modul a flag, jenž ověřuje aktuálnost výchozího modulu, respektive je 0, pokud ukazatel na výchozí modul existuje déle, než byl naposledy upraven seznam modulů. [47]. Pokud není jasné, jaký modul je defaultní, průchodem vnitřním spojovým seznamem se za defaultní označí první `ENGINE` s existující funkcionální referencí [48].

Každá kryptografická struktura využívá svoji jednu `ENGINE_TABLE`, řádek tabulky pro `EVP_CIPHER` představuje šifru nebo mód a příslušné moduly, které danou operaci podporují. I nízkoúrovňové struktury jsou definovány ve své tabulce, tam je ale situace odlišná. Zatímco šifrování lze provádět několika odlišnými algoritmy, RSA nanejvýš změní způsob implementace. Proto struktury jako jsou `RSA`, `DSA` nebo `EC_KEY` mají buď prázdnou `ENGINE_TABLE`, nebo mají právě jednu řádku.

Rozhraní Engine API [44] umožňuje i přímou manipulaci s tabulkou. Například `ENGINE_register_RSA()` přidá `ENGINE` do seznamu implementací do tabulky přiřazené k `RSA`, neoznačí ho však za defaultní. Obdobně existuje i funkce, která naopak vyřadí konkrétní `ENGINE` z tabulky.

3.2.3 Nadstandardní příkazy

V kapitole 3.1.4 byl představen konstrukt, který umožňuje definovat nové funkce nad rámec předlohy struktury. Jedním z častých příkladů jejich využití je k parametrizaci modulu, jedná se totiž o jedinou možnost jak interaktivně s modulem pracovat.

Engine API poskytuje tři funkce, které umožňují využívat nabízenou sadu příkazů. Nejčistší podobu má `ENGINE_ctrl()`, volá příkaz pomocí jeho identifikátoru, argumenty má totožné s protější funkcí uvnitř struktury `ENGINE`. Situaci ulehčuje `ENGINE_ctrl_cmd()`, od předchozí se liší v tom, že místo identifikátoru se požadovaný příkaz vyhledává podle jména. Navíc přibývá argument `cmd_optional`, pokud je nastaven, funkce bude úspěšně ukončená i v případě, že požadované jméno příkazu pro daný engine neexistuje. Funkce `ENGINE_ctrl_cmd_string()` zastiňuje ještě více. Argumenty pro příkaz již nejsou rozděleny do několika datových typů. Před předáním argumentu se zkontroluje flag o datovém typu v definici příkazu, a podle toho se argument upraví [44].

Příkazy definované modulem lze snadno volat párem klíč a hodnota, přesto současně jediná možnost, jak tento koncept využívat, je pouze přes funkce z výpisu 3.7 až po načtení modulu. Každý příkaz vyžaduje své vlastní volání. Vývojáři však pracují [45] na způsobu načtení konfigurace ze souboru, obdobně jako pro aplikační vrstvu. Konfigurační soubor by tedy vypadal jako výčet dvojic příkaz a argument. Případná parametrizace by tak nemusela být součástí zdrojového kódu.

```
int ENGINE_ctrl(ENGINE *e, int cmd,
                long i, void *p, void (*f) (void));
int ENGINE_ctrl_cmd(ENGINE *e, const char *cmd_name,
                    long i, void *p, void (*f) (void),
                    int cmd_optional);
int ENGINE_ctrl_cmd_string(ENGINE *e, const char *cmd_name,
                           const char *arg,
                           int cmd_optional);
```

Výpis kódu 3.7: Funkce pro využití nadstandardních příkazů

3.3 Modul dynamic

Tento speciální engine neposkytuje vůbec žádnou kryptografickou funkcionalitu a každý pokus o jeho inicializaci skončí chybou. Podporuje jen sadu nadstandardních příkazů, přes které se provádí načítání externích modulů ze sdílené knihovny. Po úspěšném načtení se reference dynamic modulu přetransformuje do požadovaného externího modulu [45]. Funkcionalita modulu dynamic je přístupná pouze ze sedmi nadstandardních příkazů.

SO_PATH Definuje cestu ke sdílené knihovně s implementací modulu.

NO_VCHECK Specifikuje, zda má být ignorována případná chyba při kontrole verzí v průběhu dynamického načítání.

ID Specifikuje id modulu pro případ, že sdílená knihovna implementuje více modulů, nebo jen jako další kontrola při načítání.

LIST_ADD Rozhoduje, zda má být engine přidán do vnitřního spojového seznamu.

DIR_ADD Přidá další adresář, ve kterém se modul má vyhledávat.

DIR_LOAD Udává, zda se mají k načítání využít specifikované adresáře.

LOAD Příkaz, který provede načtení modulu dle výše popsaných specifikací.

Načítání kryptografického modulu do aplikace obvykle probíhá za pomocí funkce `ENGINE_by_id()`, jejíž parametrem je identifikátor modulu. Funkce se nejdříve pokusí najít modul ve vnitřním spojovém seznamu a v případě úspěchu bude vrácena příslušná strukturální reference. V opačném případě je postup složitější, funkce místo požadovaného modulu nejdříve načte dynamic engine a deleguje část práce na něj.

Výpis kódu 3.8 zobrazuje část zdrojového kódu funkce `ENGINE_by_id(id)`, která se provede v případě, že požadované id modulu nebude nalezeno ve vnitřním spojovém seznamu. Nejdříve dojde k načtení modulu dynamic a nad ním se provede sekvence příkazů. Jako první se specifikuje hledané id, dalšími dvěma příkazy se vynutí hledání modulu v adresáři, k němuž vede cesta

```
char* load_dir = ossl_safe_getenv("OPENSSL_ENGINES");
ENGINE* iterator = ENGINE_by_id("dynamic");

ENGINE_ctrl_cmd_string(iterator, "ID", id, 0);
ENGINE_ctrl_cmd_string(iterator, "DIR_LOAD", "2", 0);
ENGINE_ctrl_cmd_string(iterator, "DIR_ADD", load_dir, 0);
ENGINE_ctrl_cmd_string(iterator, "LIST_ADD", "1", 0);
ENGINE_ctrl_cmd_string(iterator, "LOAD", NULL, 0);
```

Výpis kódu 3.8: Sekvence volání pro dynamické načítání

definovaná v proměnné prostředí `OPENSSL_ENGINES`, jejíž výchozí hodnotu lze vypsat v příkazové řádce pomocí `openssl version -a`. Dalším parametrem je požadavek, aby byl úspěšně načtený engine přidán do vnitřního spojového seznamu. Posledním voláním se zahájí operace načítání. K tomu se využije dosud nejmenovaná komponenta `DSO`, jejíž úkolem je zprostředkovat načtení sdílené knihovny do aplikace. Začíná se tím, že se sestaví jméno hledané sdílené knihovny v závislosti na id a platformě. Například pokud `id = "capi"` a OpenSSL běží na operačním systému Windows, jméno souboru bude `OPENSSL_ENGINES\capi.dll`. Pokud bude tento soubor nalezen, převede se obsah sdílené knihovny do `DSO` struktury. Za pomocí metody této struktury se získá ukazatel na funkci `bind()` načteného modulu, jejíž volání přepíše současnou referenci dynamic modulu na referenci nově načteného modulu. Nakonec, pokud o to bylo žádáno, se engine zařadí do vnitřního spojového seznamu. `ENGINE_by_id()` vrátí strukturální referenci na dynamicky načtený modul [49]. Pokud chce aplikace načítat externí modul z vlastní cesty, je vhodné vyhnout se funkci `ENGINE_by_id()` a raději řídit dynamic modul přímo.

3.4 Modul e_capi

Kryptografický modul capi byl vytvořen pro podporu Microsoft CryptoAPI v OpenSSL. Stal se součástí projektu ve verzi 0.9.8. Ke své funkcionalitě vyžaduje knihovny Crypt32.dll a Advapi32.dll

3.4.1 Struktury

Modul e_capi definuje pro své vnitřní užití dvě struktury. Ta první, `CAPI_CTX`, uchovává parametry, které ovlivňují operace vykonávané modulem. Všechny tyto parametry jsou nastavitelné z aplikace prostřednictvím nadstandardních příkazů. Jejich definice rozhoduje například o použitém CSP, o jméně úložiště certifikátů, o použité metodě pro vyhledávání certifikátů, o obsahu výpisu dat z certifikátů, nebo o přesměrování debugovacích hlášek do souboru.

Struktura `CAPI_CTX` je vytvořena při inicializaci modulu a následně je přiřazena struktuře `ENGINE` ve formě `exdata`. Vytvoření struktury zajišťuje funkce `capi_ctx_new()`, která ji rovněž přidělí výchozí nastavení.

Druhou strukturou je `CAPI_KEY`, jejíž účelem je uchovávat reference pro práci s klíči prostřednictví CryptoAPI. Obsahuje reference na `CERT_CONTEXT`, na `HCRYPTPROV`, a `HCRYPTKEY`.

3.4.2 Komunikace s úložištěm certifikátů

Motivací pro využívání capi modulu je získání benefitu v podobě úložiště certifikátů operačního systému Windows. Pro zprostředkování této funkčnosti využívá rozhraní CryptoAPI.

Získání reference konkrétního úložiště řídí funkce `capi_open_store()`. Pokud není v `CAPI_CTX` specifikováno jiné systémové úložiště, pokusí se otevřít `MY`. Spolu se jménem vstupují do funkce `CertOpenStore()` i příznaky definované v `CAPI_CTX->store_flags`, které podmiňují vytvoření reference. Ve výchozím nastavení bude otevřeno pouze existující systémové úložiště pod registrem `HKEY_CURRENT_USER` v režimu jen pro čtení.

Pro nalezení specifického certifikátu slouží `capi_find_cert()` a capi nabízí dva způsoby vyhledávání, mezi kterými volí parametr `lookup_method`. První možností je kritérium obsahu hledaného řetězce ve jméně subjektu, respektive v common name. Druhý způsob, vyhledávání podle friendly name, je z hlediska implementace více komplikovaný, protože není přímo CryptoAPI podporován. Postupně se projde každý certifikát, získá se jeho friendly name a porovná se s hledaným řetězcem. Tentokrát je přípustná jen přesná shoda. K získání friendly name certifikátu je určena funkce `capi_cert_get_fname()`, která zabaluje volání `CertGetCertificateContextProperty()`. Řetězec získaný touto CryptoAPI funkcí je datového typu `LPWSTR`, respektive `wchar*`. Unicode řetězce nejsou dobře tisknutelné funkcemi modulu `BIO`, proto je wide char posloupnost nutné nejdříve převést na posloupnost ascii znaků, k čemuž je implementovaná funkce `wide_to_asc()`. Ta využívá Win32 API funkci `WideCharToMultiByte()` pro žádoucí konverzi.

Funkce `capi_list_certs()` je určená k výpisu certifikátů a koná podle vstupních parametrů. Pokud je na vstupním argumentu nějaký řetězec `id`, dojde k jeho předání funkci `capi_find_cert()` a vypíše se nejvýše jeden nalezený certifikát. Pokud `id` specifikováno není, vypíšou se všechny certifikáty dostupné v daném úložišti. Výpis informací z certifikátu na standardní výstup je řízen funkcí `capi_dump_cert()`. Ta nejdříve převede CryptoAPI reprezentaci certifikátu do struktury `x509`, pro niž OpenSSL poskytuje funkce pro tisk obsažených informací. Množina informací ve výpisu je řízena parametrem `dump_flags` v `CAPI_CTX`. Capi předurčuje několik možností výpisu, které lze libovolně kombinovat logickým součtem. Ve výchozím nastavení se zobrazí friendly name s informacemi o vlastníkovi a vystavovateli, navíc je možné získat informace o CSP, který doprovázel ukládání certifikátu do úložiště.

3.4.3 Získání klíče

Engine API funkce pro získání soukromého klíče vyústí uvnitř capi voláním funkce `capi_load_privkey()`, jejíž úkolem je připravit soukromý klíč pro pozdější užití do podoby struktury `EVP_PKEY`. Do této funkce mimo jiné vstupuje řetězec `key_id` pro bližší identifikaci klíče. Způsob vyhledávání klíče závisí na určené `lookup_method` v kontextové struktuře. Evidují se tři metody. První dvě nejsou v zásadě odlišné, obě využijí `key_id` k nalezení certifikátu v úložišti. Z reference `CERT_CONTEXT` jsou následně pomocí CryptoAPI funkce získány údaje potřebné k identifikaci kontejneru pro klíč uvnitř konkrétního CSP. Třetí metoda předpokládá, že struktura `CAPI_CTX` má správně nadefinované jméno a typ CSP, `key_id` je totiž přímo využito jako jméno kontejneru.

Všechny způsoby tedy vedou k trojici dat, které jednoznačně identifikují kontejner v konkrétním CSP. Po otevření CSP je získání odkazu na daný klíč snadné a reference `HCRYPTPROV` je společně s `HCRYPTKEY` zabalena do vnitřní struktury `CAPI_KEY`. Proces přípravy klíče je považován za dokončený až po transformaci CryptoAPI reprezentace klíče do OpenSSL reprezentace, struktury `EVP_PKEY`.

`HCRYPTKEY` se nejdříve serializuje do datového blobu, z něhož jsou přečteny veřejné informace o klíči. Protože v blobu jsou data uložena v little-endian podobě, ale OpenSSL pracuje s big-endian formou, je nutné data do struktur `BIGNUM` zapsat v opačném pořadí.

```
RSA *key;
BIGNUM *e, *n;
EVP_PKEY *pkey;

/*vynechaná inicializace e, n*/
key = RSA_new_method(engine);
RSA_set0_key(key, n, e, NULL);
RSA_set_ex_data(key, rsa_idx, capi_key);
EVP_PKEY_assign_RSA(pkey, key);
```

Výpis kódu 3.9: Načtení RSA klíče kryptografickým modulem

Výpis kódu 3.9 zobrazuje část `capi_get_pkey()`, při kterém probíhá přiřazení získaných dat z blobu do struktury `RSA`. Vstupem této funkce je reference modulu `engine`, kterému bude klíč přidělen a `CAPI_KEY` obsahující CryptoAPI reprezentaci klíče. Číselný index `rsa_idx` je globální proměnná pro práci s exdaty struktury `RSA`. Nejdříve se vytvoří nová `RSA` struktura, jíž je přidělen engine a následně veřejný klíč. Přes `EVP_PKEY` bude umožněn přímý přístup pouze k veřejným datům. Soukromý klíč bude struktuře přidělen jen jako jeho exdata. Výpis vynechává inicializaci `BIGNUM` struktur, respektive vytěžování a transformaci datového blobu, protože tento postup bude podrobněji vyobrazen v kapitole o modulu cng.

K podobné situaci dochází i v případě použití DSA klíče, rozdílem jsou samozřejmě použité struktury a data získaná z blobu. Přesto, že se funkce pro přípravu klíče do volá přes Engine API funkci `ENGINE_load_private_key()`, evidentně se do `EVP_PKEY` reprezentace dostanou pouze veřejné informace. Pro DSA je to čtveřice p, q, g a veřejný klíč, pro RSA je to veřejný exponent a modulo. Soukromý klíč zůstává bezpečně ukryt na pozadí v CryptoAPI odkazu napojený formou exdat.

3.4.4 Kryptografická funkcionalita

Modul capi implementuje nízkoúrovňové struktury `RSA_METHOD` a `DSA_METHOD`, čímž přepisuje výchozí implementace vybraných RSA a DSA operací. Obě struktury existují ve formě globální proměnné. K jejich vytvoření dochází v rámci bind funkce, ale jejich pravá funkcionalita je jim přidělena až v průběhu inicializace modulu. V této funkci jsou rovněž vytvořeny indexy pro jejich exdata, které jsou taktéž globálně dostupné.

Mezi RSA operace, kterými capi přepisuje výchozí implementaci, se řadí šifrování soukromým klíčem, dešifrování soukromým klíčem a tvorba digitálního podpisu. V případě DSA je implementována pouze tvorba digitálního podpisu. Pro další operace jsou využity kopie defaultní `*_METHOD` struktury. Modul capi nijak nezasahuje do EVP, to znamená, že volání těchto RSA kryptografických operací zajišťuje výchozí `EVP_PKEY_RSA`, jejíž definice leží v `crypto/rsa/rsa_pmeth.c`. Pro DSA je to `EVP_PKEY_DSA` v souboru `crypto/dsa/dsa_pmeth.c`.

Při použití RSA algoritmu vyústí volání `EVP_sign()` z aplikace ve využití funkce `capi_rsa_sign()`. Ta jako argument očekává již zahešovanou zprávu, NID použitého hešovacího algoritmu, dostatečně velký buffer pro umístění podepsané zprávy a strukturu `RSA`, jejíž exdata obsahují `CAPI_KEY` s referencí na CSP a klíč. CryptoAPI neumožňuje podepsat hešovanou zprávu přímo, pouze prostřednictvím hash objektu. Pro vytvoření hash objektu je nutné znát hešovací algoritmus aplikovaný na zprávu, a tak je nezbytné převést OpenSSL NID identifikátor na CryptoAPI ALG_ID. CSP reference se využije ta z `RSA` exdat. Správné použití CryptoAPI pro výpočet heše bylo uvedeno ve výpisu 2.7 . Do výstupního bufferu se musí podpis zapsat v opačném pořadí kvůli rozdílné endianitě.

Funkce pro šifrování soukromým klíčem ve skutečnosti implementovaná není, její zavolání vždy skončí chybovým kódem s doprovodnou chybovou hláškou o chybějící podpoře operace. Dešifrování soukromým klíčem implementuje `capi_rsa_priv_dec()`, přijímá buffer se zašifrovanou zprávou, alokovaný buffer pro uložení dešifrované zprávy, strukturu `RSA` a identifikátor použitého RSA paddingu. Ze zdrojového kódu je evidentní podpora pouze dvou typů paddingu. Buď PKCS1, nebo žádný. Z definice struktury `EVP_PKEY_RSA` však vyplývá, že pokud je `RSA_private_decrypt()` zavolána s identifikátorem `RSA_NO_PADDING`, může to znamenat i použití OAEP padidngu ve zprávě.

EVP_PKEY_RSA totiž tento typ odchytává, a mgf1 aplikuje mimo dešifrovací funkci. O samotné dešifrování se stará CryptoAPI funkce CryptDecrypt(). Té se předá reference klíče z RSA exdat, informace o paddingu a buffer s převrácenou šifrovou zprávou, která bude nahrazena dešifrovaným textem.

Výpočet digitálního podpisu algoritmem DSA je ve smyslu využití CryptoAPI stejné. Funkce capi_dsa_do_sign() očekává na vstupu zahešovanou zprávu a strukturu DSA s CAPI_KEY v exdatech. Opět je nutné před samotným podpisem vytvořit hash objekt. V tomto případě je přípustný pouze SHA1 jako hešovací algoritmus, proto se nejdříve kontroluje že přijatý hash je 20 bajtový. Dále se hash objektu předá zahešovaná zpráva a pomocí CryptSignHash() je naplněn 40 bytový buffer s po sobě jedoucími stejně velkými hodnotami r a s. Protože se dvojice nevrací odkazem, ale jako návratová hodnota ve formě struktury DSA_SIG, je nutné strukturu alokovat a prostřednictvím BIGNUM ji předat dvojici r a s v obráceném pořadí.

K DSA_METHOD i RSA_METHOD je navíc navázána funkce finish, která uvolní exdata RSA a DSA struktur, respektive prostředky alokované pro CNG_KEY jako je reference klíče, CSP a certifikátu.

3.4.5 Načítání certifikátu klienta

Kryptografický modul capi jako jediný engine v projektu OpenSSL implementuje volání ENGINE_load_ssl_client_cert(). Do funkce vstupuje seznam jmen certifikačních autorit, které server dodal jako součást zprávy Certificate Request. Ve výstupních argumentech se očekává certifikát reprezentovaný strukturou X509, kterým se bude klient autorizovat, a soukromý klíč k vybranému certifikátu ve struktuře EVP_PKEY.

Funke capi_load_ssl_client_cert() nejdříve otevře systémové úložiště certifikátů MY. Přestože zdrojový kód nabízí parametrizaci jména přes kontextovou strukturu, neexistuje žádná možnost jak z aplikace ssl_client_store změnit. Následuje průchod celého úložiště a každý certifikát je otestován na dvě podmínky. První podmínkou je existence vystavovatele certifikátu v seznamu autorit přijatém funkcí a druhou, že účelem certifikátu je autorizace klienta, tedy má správně nastavené X.509 Extensions. K certifikátu, který splňuje obě podmínky, je nalezen klíč pomocí capi_get_cert_key() a jako pár jsou přiřazeny do dočasného souboru struktur X509. Pokud je po dokončení průchodu úložištěm soubor neprázdný, využije se k výběru certifikátu funkce definovaná strukturou CAPI_CTX.

Capi nabízí dvě funkce pro výběr certifikátu. Ve výchozím nastavení se využije cert_select_simple(), která vždy vrátí hodnotu nula. To znamená, že se jako výstupní certifikát zvolí vždy ten první. Druhý způsob volby implementuje cert_select_dialog() a pro její zpřístupnění je nutné engine kompilovat s příznakem OPENSSL_CAPIENG_DIALOG. Tato funkce využívá knihovnu User32.dll pro vykreslení dialogových oken s nabídkou nalezených certifikátů. Zvolený certifikát je přidělen výstupním argumentům a ostatní jsou uvolněny.

3.4.6 Nadstandardní příkazy

Capi modul disponuje nabídkou nadstandardních příkazů, které umožňují měnit výchozí parametry umístěné ve struktuře `CAPI_CTX` a tisknout obsah úložiště, obsah kontejnerové databáze nebo dostupné CSP. Všechny dostupné příkazy včetně jejich popisu a očekávaného vstupního parametru lze vypsat v aplikaci příkazového řádku pomocí `openssl engine -vvvv capi`.

První zajímavou možností je zpřístupnit debugovací informace. K tomu je potřeba určit `debug_file`, kam budou hlášky přesměrovány a `debug_level` musí být nastavený na hodnotu `2`.

Dále capi modul poskytuje šest příkazů pro manipulaci s CSP. Příkazem `list_csps` lze vypsat všechny dostupné CSP, což je ve skutečnosti jen zabalená posloupnost volání CryptoAPI funkce `CryptEnumProviders()`, která získá jméno prvního nebo dalšího dostupného CSP. Vybrat CSP, jenž bude za běhu aplikace využit, je možné podle indexu, nebo jména. Výběr podle indexu je uměle vytvořený modulem capi. Ve skutečnosti to znamená, že pokud je vyžádán CSP s indexem n, využije se CSP, který je vrácen po n-tém volání `CryptEnumProviders()`. Ve výchozím nastavení se používá Microsoft Enhanced RSA and AES Cryptographic Provider typu PROV_RSA_AES. Seznam kontejnerů lze získat příkazem `list_containers`. Tuto funkcionalitu zajišťuje cyklické volání `CryptGetProvParam()` s parametrem `PP_ENUMCONTAINERS` dokud návratová hodnota není `ERROR_NO_MORE_ITEMS`.

Procházení certifikačního úložiště je navrženo tak, že funkce představené v podkapitole 3.4.2 jsou přímo zpřístupněny přes volání příkazů `list_certs` nebo `lookup_cert`. Použité úložiště se volí prostřednictvím `store_name`. Parametr, který vstupuje do funkce `CertOpenStore()` jako `dwFlags` lze libovolně upravovat pomocí příkazu `store_flags`. Pro změnu metody vyhledávání je určen příkaz `lookup_method` a formát výpisu certifikátu se řídí přes `list_options`.

3.4.7 Nedostatky

Existující capi engine implementuje kryptografické funkce, které se uplatní při zahájení šifrované komunikace mezi serverem a klientem. Vhodným protokolem, který takovou komunikaci definuje je TLS.

TLS Handshake začíná v momentě, kdy klient kontaktuje server. Tato zpráva se nazývá `Client Hello` a mimo jiné obsahuje upřednostňovanou verzi TLS a výčet šifrových sad, pomocí kterých je klient ochoten komunikovat. Server po přijetí této zprávy vytvoří průnik nabízených a vlastních šifrových sad a zvolí tu nejvíce bezpečnou. Obdobně vybere i verzi TLS a obě tyto volby spolu se svým certifikátem, pokud to je nutné, odešle klientovi zprávou zvanou jako `Server Hello`. Další dění je ovlivněno použitou šifrovou sadou. Typicky následuje kontrola certifikátu proti certifikační autoritě a bezpečná výměna klíčů pro použití v následující šifrované komunikaci [50].

RFC vedle definicí standardů kryptografických algoritmů používané v síťové komunikaci vydává i osvědčené postupy a upozorňuje na zranitelnosti ve starších verzí protokolů. Současně je vydán koncept [51], který označuje TLS 1.0 a 1.1 za zastaralé a měly by se přestat zcela používat. Právě tyto verze jsou jediné, při kterých capi engine funguje spolehlivě.

Při použití TLS 1.2 nebo 1.3 nedokáže engine zprostředkovat handshake, protože skončí s následující chybou.

```
capi_rsa_priv_enc:function not supported:engines\e_capi.c:814
```

To oznamuje, že na řádku 814 zdrojového souboru modulu capi byla vyvolána chyba s hláškou o chybějící implementaci funkce `capi_rsa_priv_enc()`. K diagnostice problému byl odchycen handshake pomocí programu Wireshark. Ve verzích TLS starších než 1.2 se běžně pro hešování dat používají algoritmy SHA a MD5 za sebou a následně se použije RSA-PKCS schéma pro podpis. Ukázalo se, že od TLS 1.2 přibylo ve zprávě `Client Hello` rozšíření s nabídkou podpisových algoritmů, při kterém nejvyšší prioritu získá schéma RSA-PSS.

Capi engine neupravuje abstrakci `EVP_PKEY_METHOD`, ale pouze nízkoúrovňové kryptografické moduly `RSA_METHOD` a `DSA_METHOD` V průběhu konání TLS Handshake tak volání `EVP_DigestSign()` vyústí ve využití defaultní implementace `EVP_PKEY_METHOD`. Ta v případě použití RSA-PKCS vyvolá funkci `RSA_sign()`, ale pro RSA-PSS zavolá `RSA_private_encrypt()` [52].

Modul lze nadále používat, pokud se smíříme s použitím starší a méně bezpečné konfigurace. Je možné zakázat použití TLS 1.2 a 1.3 nebo omezit nabídku podpisových schémat a vynutit tak RSA-PKCS. Tato omezení na bezpečnosti mohou mít za následek, že se množina možných šifrových sad zmenší natolik, že znemožní navázání spojení s moderními a bezpečnými zařízeními.

Dalším nedostatkem řešení modulu capi je použití kryptografického rozhraní Microsoft CryptoAPI, které je autory označeno za zastaralé a mělo by být nahrazeno novějším Cryptography API: Next Generation. Oba popsané nedostatky se snaží napravit modul e_cng.

3.5 Modul e_cng

Kryptografický modul cng vznikl jako součást této bakalářské práce. Vytváří podporu Cryptography API: Next Generation v OpenSSL. Pro svou činnost vyžaduje OpenSSL alespoň verze 1.1, protože v EVP rozhraní, především pak v součinnosti s eliptickými křivkami, nastaly s touto verzí změny. Tento problém by bylo možné vyřešit podmíněným překladem, avšak OpenSSL 1.0 není nadále udržovanou verzí.

Obdobně jako modul capi vyžaduje cng knihovnu Crypt32.dll, nové Cryptography API totiž nepřichází s novým řešením pro komunikaci s certifikačním

úložištěm, a tak je nutné využívat funkce z CryptoAPI. Pro ostatní kryptografické operace jsou využity knihovny BCrypt.dll a NCrypt.dll z novějšího API.

Řešení nového kryptografického modulu je inspirováno capi modulem, jehož funkcionalita byla představena již v předchozí kapitole. Proto tato kapitola je více mířená jako demonstrace budování nového kryptografického modulu, než analýza již existujícího produktu.

3.5.1 Základní modul

První řádky zdrojového kódu jsou zpravidla věnovány začlenění knihoven. Pro zpřístupnění rozhraní Engine API, tedy zpřístupnění i engine strukutry, je potřeba hlavičkový soubor `<openssl/engine.h>`. Jádrem enginu je funkce `bind()`, která připraví strukturu `ENGINE` a umožní aplikaci s modulem manipulovat.

Na počátku modulu tedy stojí funkce `bind_cng()`, která přiřazuje struktuře `ENGINE` její id, jméno a init funkci. Projekt OpenSSL se řídí pravidlem, že návratová hodnota 1 je považována za úspěch, záporné hodnoty a 0 vyjadřují chybu. Většina funkcí má proto návratovou hotnotu `int` a jiné výstupy se předávají jako vstupně-výstupní ukazatel.

Protože nově vytvořený engine není součástí projektu OpenSSL, bude načítán dynamicky. Pro ten účel je nutné doplnit příslušná makra, první je pro kontrolu verze dynamického načítání, druhé pro specifikaci bind funkce. V případě dynamického načtení modulu cng je jako první zavolána funkce `bind_helper()`, která nejdříve zkontroluje, zda hledané id odpovídá implmentovanému modulu a v případě shody zavolá skutečný `bind_cng()`. Je-li předpokladem, že engine by někdy mohl být součástí projektu, je vhodné připravit i možnost statického načítání. Tomu odpovídá `engine_load_cng_int()`, jejíž volání by se muselo stát součástí inicializačního procesu knihovny libcrypto. Tato funkce vytvoří strukturální referenci modulu a přidá ji do vnitřního spojového seznamu.

Tento základní nic nedělající modul je možné zkompilovat a dynamicky načíst aplikací. Toho lze docílit buď přímou komunikací s dynamic modulem, nebo DLL soubor s modulem umístit do adresáře uvnitř projektu OpenSSL. Ukázka načítání modulu je součástí poslední kapitoly.

3.5.2 Parametrizace

Již v další části tvorby modulu se ukáže, že Cryptography API funkce jsou ve svém užití variabilní a bylo by vhodné umožnit aplikaci nějaké prvky modifikovat. Vhodným konstruktem pro parametrizaci jsou nadstandardní příkazy.

Nejdříve se vytvoří konstanty s prefixem `CNG_CMD_`, které označují číselný identifikátor příkazu. Následuje definice pole `cng_cmd_defns[]` složeného ze struktur `ENGINE_CMD_DEFN` a definice funkce `cng_ctrl()`, jejíž účelem je rea-

govat na žádosti o vykonání nadstandardního příkazu dle jejich číselné hodnoty. Vytvořené pole i funkce musí být přiděleny `ENGINE` struktuře uvnitř `bind_cng()`.

Pro uchování nastavení je určena struktura `CNG_CTX`, jejíž alokaci a přiřazení výchozích hodnot zajišťuje `cng_ctx_new()`. Tato funkce je volána v průběhu inicializace modulu. Nově vytvořený kontext je přidělen struktuře `ENGINE` jako exdata.

3.5.3 Komunikace s úložištěm certifikátů

Implementace funkcí pro manipulaci s úložištěm certifikátů a certifikáty samotnými jsou až na drobné detaily shodné s řešením v capi modulu, většinou tvoří jen obal pro rozhraní CryptoAPI.

Logika celého řešení je s capi identická. Byla vytvořena funkce, která na základě parametrů určených ve struktuře `CNG_CTX` otevře úložiště certifikátů, respektive získá referenci `HCERTSTORE`. Dále existuje stejný systém funkcí pro vypsání všech certifikátů na standardní výstup nebo pro nalezení jednoho konkrétního certifikátu na základě vyhledávacích kritérií.

Žádná z těchto funkcí není přímou součástí struktury `ENGINE`, a tak se v této části nijak neupravovala ani `bind_cng()`, ani `cng_init()`. Došlo však k významnému obohacení nadstandardních příkazů a položek v kontextové struktuře.

3.5.4 Komunikace s KSP

Key Storage Provider je správce soukromých klíčů. Tato služba přišla nově s Cryptography API: Next Generation, avšak její aplikační využití se podobá CSP a klíčovým kontejnerům.

Pro seznámení s dostupnými KSP je k dispozici příkaz `list_ksps` implementovaný funkcí `cng_list_ksps()`. Ta využije NCrypt API pro získání seznamu všech dostupných KSP. Každá položka tohoto seznamu je reprezentována strukturou, která se skládá ze jména a komentáře. Obě tyto informace jsou do výpisu na standardní výstup zahrnuty.

Další funkcí provádějící výpis je `cng_list_keys()`, taktéž ji je možné volat pomocí nadstandardního příkazu. Funkce nejdříve získá referenci na KSP podle jména definovaného v `CNG_CTX->ksp_name`. Ve výchozím nastavení se jedná o Microsoft Software Key Storage Provider. Poté už je prostřednictvím NCrypt rozhraní možné postupně získávat klíče dostupné pro daného KSP a pro skupinu klíčů danou proměnnou `keyflag` v `CNG_CTX`. Ve výchozím nastavení je `keyflag` nastaven na skupinu uživatelských klíčů, druhou volbou jsou klíče sdílené napříč počítačem. Získaný klíč je reprezentován strukturou `NCryptKeyName`, ze které je na standardní výstup odesláno jméno, použitý algoritmus a keyspec.

Poslední funkcí, která KSP využívá, je `cng_key_from_ksp()`. Ta přijímá jako argument řetězec `id`, který slouží jako kritérium pro nalezení klíče podle jeho jména. Nejdříve je opět získána reference na KSP, až poté následuje využití funkce `NCryptOpenKey()`. Do ní, pro identifikaci klíče, navíc vstupuje `keyflag` a `keyspec` z kontextové struktury. Z možných chybových stavů je odchytáván pouze `NTE_BAD_KEYSET` oznamující neexistenci hledaného klíče. Pokud funkce úspěšně skončí, bude dostupná reference CNG klíče, která je po zabalení do struktury `CNG_KEY` výstupem funkce `cng_key_from_ksp()`.

Ani v této kapitole neproběhlo přímé obohacení kryptografického modulu, spíše vytvoření pomocných funkcí pro další použití. Opět se ale významně rozšířila struktura `CNG_CTX` a s tím i množství nadstandardních příkazů.

3.5.5 Získání klíče

Nyní bude konečně objasněn význam předchozí přípravy. V modulu cng je pro načtení klíče připravena funkce `cng_load_privkey()`, do které vstupuje řetězec `id`. Logika tohoto procesu je podobná s capi modulem, metoda vyhledávání se odvíjí od definice `lookup_method`. Pokud je tento parametr přenastaven na hodnotu 4, předpokládá se, že `id` značí jméno klíče a dojde k volání funkce `cng_key_from_ksp()` z minulé podkapitoly. Pro ostatní metody platí, že `id` je identifikátorem certifikátu a jeho nalezení je delegováno na funkci `cng_find_cert()`. Pokud existuje certifikát splňující zadaná kritéria, funkce vrátí `CERT_CONTEXT`, ze které lze získat referenci na spárovaný soukromý klíč. Protože rozhraní CNG neposkytuje žádné prostředky pro manipulaci s certifikáty, je potřeba použít CryptoAPI rozhraní.

```
BOOL callerFree;
DWORD keySpec;

CryptAcquireCertificatePrivateKey(
        cert, CRYPT_ACQUIRE_ONLY_NCRYPT_KEY_FLAG, NULL,
        &key, &keySpec, &callerFree);
```

Výpis kódu 3.10: Získání klíče z reference certifikátu

Protože funkce využívaná capi modulem je již označená za zastaralou, cng modul využívá funkci představenou ve výpisu 3.10. Na vstupu očekává referenci certifikátu, konfigurační příznaky a adresy pro předání výstupu. Příznak použitý v ukázce znamená, že klíč bude získán pomocí CNG, nikoliv CryptoAPI. Což je v pořádku, protože CNG dokáže načítat klíče uložené prostřednictvím staršího rozhraní. Následující parametr je využit jen s vybranými příznaky. Zvolený druhý argument také zaručuje, že do `key` bude uložena reference `NCRYPT_KEY_HANDLE` a do `keySpec` konstanta `CERT_NCRYPT_KEY_SPEC`. Poslední výstupní argument předává informaci, zda za uvolnění reference klíče

zodpovídá volající, nebo k uvolnění dojde samo po poslední free operaci daného certifikátu. Získaná reference klíče je zabalena do struktury `CNG_KEY` a vrácena.

V tomto momentě by `cng_load_privkey()` měla mít k dispozici inicializovanou strukturu `CNG_KEY` bez ohledu na použitou vyhledávací metodu. Pokud však neobsahuje referenci `NCRYPT_KEY_HANDLE`, znamená to, že zadaným kritériím neodpovídá žádný klíč. Nyní je potřeba převést CNG reprezentaci klíče na `EVP_PKEY`. Proto vznikla funkce `cng_get_pkey()`, očekává referenci `ENGINE` a inicializovanou strukturu `CNG_KEY`.

```
unsigned char *blob;
NCryptExportKey(key, 0, BCRYPT_PUBLIC_KEY_BLOB, NULL,
                blob, len, &len, 0);

BCRYPT_KEY_BLOB *keyBlob = (BCRYPT_KEY_BLOB*)blob;
if (keyBlob->Magic == BCRYPT_RSAPUBLIC_MAGIC) {
    BCRYPT_RSAKEY_BLOB *rsaBlob = (BCRYPT_RSAKEY_BLOB*)keyBlob;
    BIGNUM *e, *n;

    ULONG offset = sizeof(BCRYPT_RSAKEY_BLOB);
    e = BN_bin2bn(blob + offset, rsaBlob->cbPublicExp, NULL);
    offset += rsaBlob->cbPublicExp;
    n = BN_bin2bn(blob + offset, rsaBlob->cbModulus, NULL);
    /*Definice EVP_PKEY*/
} else if ((curveNid = is_ecc(keyBlob->Magic)) != 0) {...}}
```

Výpis kódu 3.11: Parsování RSA blobu

Z výpisu 3.11 je vynecháno první volání exportovací funkce pro získání očekávané délky blobu pro alokaci paměti. Nedříve je z CNG reference klíče vytvořen obecný datový blob pro veřejný klíč. Tento kus paměti je reprezentovaný strukturou `BCRYPT_KEY_BLOB` obsahující právě jednu položku Magic. Tato hodnota jednoznačně identifikuje algoritmus, ke kterému je klíč určen a zda se jedná o veřejný nebo soukromý klíč. Pokud magic hodnota odhaluje veřejný RSA klíč, lze obecný blob transformovat do RSA blobu obsahující délku veřejného exponentu a modulu. Protože CNG pracuje ve formě big-endian, není potřeba měnit pořadí v bufferu, jak tomu bylo v capi modulu. Funkce `BN_bin2bn()` převede část paměti do reprezentace `BIGNUM`. Přiřazení do `EVP_PKEY` probíhá stejně jako v případě capi modulu. Nově vytvořené `RSA` struktuře je přiřazen veřejný klíč a jako exdata jí je přidělena `CNG_KEY`. Pomocí `EVP_PKEY_assign_RSA()` následně dojde k definici návratové struktury.

Identifikátory klíčů pro algoritmy využívající eliptické křivky nemají souhrnnou magic hodnotu pro všechny křivky a porovnání hned s několika konstantami je nezbytné. Pro ten účel byla vytvořena funkce `is_ecc()`, která

vrátí NID identifikátor křivky používaného v OpenSSL, nebo nulu, pokud magic hodnota neodpovídá žádné BCrypt konstantě. V případě, že se potvrdí shoda, lze obecný blob reprezentovat strukturou `BCRYPT_ECCKEY_BLOB`, ze které jsou podobně jako v případě RSA vyčteny veřejné informace, v daném případě tedy body x, y na křivce.

```
EC_KEY *ec = EC_KEY_new_by_curve_name(curveNid);
EVP_PKEY pkey = EVP_PKEY_new();

EC_KEY_set_public_key_affine_coordinates(ec, x, y);
EC_KEY_set_ex_data(ec, ec_idx, cng_key);
EVP_PKEY_assign_EC_KEY(pkey, ec);
```

Výpis kódu 3.12: Inicializace a přiřazení EC_KEY

Po vytvoření OpenSSL reprezentace ECDSA klíče dojde k přiřazení veřejných hodnot. Soukromý klíč je opět pouze na pozadí jako exdata struktury v reprezentaci CNG a celý balík je prostřednictvím `EVP_PKEY` vrácen z funkce.

Protože volání této funkce bude probíhat prostřednictvím Engine API, je nutné obohatit bind funkci o `ENGINE_set_load_privkey_function()`. Nezapomenout se nesmí ani na vytvoření indexů pro exdata uvnitř `cng_init()`. Funkce `cng_load_ssl_client_cert()` se od řešení poskytnutém v capi až na drobné implementační detaily neliší. Nebude proto znovu analyzována. Je ji však nutné přidat do evidence ve funkci bind.

3.5.6 RSA funkcionalita

Protože modul již má k dispozici klíče, je možné začít s implementací kryptografických RSA operací. Modul cng implementuje vlastní šifrování soukromým klíčem, dešifrování soukromým klíčem a vytváření digitálního podpisu.

Z diskuze nedostatků capi modulu vyplynulo, že současné architektonické řešení nízkoúrovňové struktury `RSA_METHOD` nedovoluje funkci pro tvorbu podpisu přijímat použitý padding, což působí potíže při výběru podpisového schématu. Výchozí EVP rozhraní pro RSA funkcionalitu to řeší tak, že pro vytvoření podpisu PSS schématem zavolá funkci `RSA_private_encrypt()` s argumentem `RSA_NO_PADDING`. Jako první řešení by se nabízelo skutečně tuto funkci implementovat pro získání podpisu. Přišel by však nový problém s identifikací použitého hešovacího algoritmu, který musí vstoupit do NCrypt funkce. Žádný prostředník mezi nízkoúrovňovou `METHOD` strukturou a vysokoúrovňovou `EVP` nestojí. K tomuto problému se vyjadřuje jeden z přispěvatelů [53] OpenSSL projektu a doporučuje, že kryptografické moduly, které usilují o podporu RSA-PSS, by měly funkcionalitu vytvořit prostřednictvím `EVP_PKEY_METHOD`, tedy upravit EVP volání.

Začne se vytvořením funkce `cng_rsa_sign()`, jejíž vstupní parametry odpovídají `sign` funkci struktry `EVP_PKEY_METHOD`. Vedle zahešované zprávy, její

délky, a adres přes které proběhne předání podpisu a jeho délky, vstupuje do funkce i `EVP_PKEY_CTX`. Tato struktura obsahuje veškeré informace o probíhající operci. Pro získání `RSA` struktury není připravena žádná get funkce, a tak je nutné její referenci získat přímým vstupem do EVP kontextové struktury. V exdatech `RSA` je uložena `CNG_KEY`, čímž je vyřešen přístup ke klíči. Dalším krokem je zjistit, jakým algoritmem je zpráva zahešována. Zde už je EVP API přívětivější, poskytuje get funkci, která z kontextu vrátí `EVP_MD`, jejíž typem je NID použitého hešovacího algoritmu. Při konverzi OpenSSL identifikátorů na CNG identifikátor ale přichází problém v podobě neexistujícího protějšku pro `NID_md5_sha1`, který je často využíván TLS protokoly verze 1.0 a 1.1. Pro získání paddingu je také připravena get funkce vracející identifikátor. Modul cng poskytuje podporu pro dva druhy RSA paddingu — PKCS1 a PSS.

V tomto stavu má `cng_rsa_sign()` k dispozici soukromý RSA klíč, CNG identifikátor použitého hešovacího algoritmu a informaci o tom, jaké podpisové schéma použít. Pokud je vyžadováno RSA-PKCS1, stačí už jen využít CNG API k podpisu tak, jak bylo představeno ve výpisu 2.11. Protože CNG pracuje se stejnou endianitou, funkci `NCryptSignHash()` se předají přímo výstupní argumenty.

Pokud je pro podpis vyžádnáno RSA-PSS, předchází samotnému výpočtu podpisu ještě několik operací. Přesto, že to není doporučováno, je z hlediska definice standardu RSA [54] přípustné, aby funkce generující masku a hešovací funkce byla rozdílná. CNG API tu možnost však vůbec nepřipouští, a tak je potřeba ověřit, že tato situace nenastala. Pro získání mgf1 je připravena get funkce s výstupem reprezentovaným jako `EVP_MD`, tedy stejně jako hešovací funkce. Porovnání dvou `EVP_MD` typů je už triviální. Pokud `NCryptSignHash()` má podepsat zprávu pomocí RSA-PSS, musí se navíc specifikovat délka soli. Pro zjištění délky soli je rovněž připravena get funkce, která tuto informaci zjistí z `EVP_PKEY_CTX`. Návratová hodnota může být i záporná, v tom případě má speciální význam. Konkrétně -1 vyjadřuje délku soli rovnou délce heše, -2 a -3 znamená maximální délku heše. Typicky je sůl alespoň stejné délky jako hash, její horní limit je závislý na délce modulu a heše. Platí totiž, že `max_délka_soli = velikost_modula/8 - délka_heše - 2`. Tím se dospělo do stavu, kdy jsou funkci známé všechny informace k vytvoření digitálního podpisu, a tak stačí už jen využít CNG API.

Funkci `cng_rsa_sign()` je nyní nutné předat do správy modulu cng. Do bind funkce se přidá alokace nové, globálně přístupné `EVP_PKEY_METHOD`, kterou je nutné v init funkci definovat. Protože motivací je upravit pouze funkci pro výpočet podpisu, je snazší použít existující EVP strukturu.

```
EVP_PKEY_METHOD *ossl_pkey = EVP_PKEY_meth_find(EVP_PKEY_RSA);
EVP_PKEY_meth_copy(cng_pkey, ossl_pkey);
EVP_PKEY_meth_set_sign(cng_pkey, NULL, cng_rsa_sign);
```

Výpis kódu 3.13: Úprava výchozí EVP_PKEY_METHOD

Výpis kódu 3.13 ukazuje způsob úpravy existující EVP struktury. V prvním řádku je získána výchozí `EVP_PKEY_METHOD` pro RSA operace, která je vzápětí zduplikována do té nově vytvořené. Při přiřazování nové funkce pro podepisování je možné specifikovat i `signinit()` funkci. Ve výpisu tato možnost z důvodu čitelnosti není využita. K takto připravené EVP struktuře je nutné vytvořit callback funkci podobně, jako bylo vysvětleno výpisem 3.5 . V cng modulu tuto funkcionalitu zajišťuje `cng_pkey_meths()`, kterou je nutné předat struktuře `ENGINE` v bind pomocí `ENGINE_set_pkey_meths()`.

Implementace šifrování a dešifrování soukromým klíčem je řešena přes strukturu `RSA_METHOD`, nepotřebují totiž zásah do způsobu volání těchto funkcí. Nejdříve byla vytvořena funkce `cng_rsa_priv_enc()`, jejíž vstupní parametry jsou dány předpisem v `RSA_METHOD` struktuře. Dodaný identifikátor paddingu je nutné nejdříve převést na CNG identifikátor. Přesto, že je ze zdrojového kódu patrná podpora jen PKCS1 nebo žádného paddingu, dochází ke stejné situaci jako v případě capi modulu. Tedy, pokud je na vstupu `RSA_NO_PADDING`, může to v konečném důsledku znamenat OAEP padding. Soukromý klíč je opět získán ze vstupní `RSA`. Pak už se jen zavolá `NCryptEncrypt()`, která data zašifruje do výstupního bufferu. Návratovou hodnotou je počet zapsaných bajtů. Dešifrovací funkce `cng_rsa_priv_dec()` postupuje identicky. Rozdíl je pouze ve využití jiné NCrypt funkce, konkrétně `NCryptDecrypt()`.

Aby funkce pro šifrování a dešifrování byly evidované kryptografickým modulem, je nutné v bind vytvořit novou strukturu `RSA_METHOD`, která se přidělí prostřednictvím `ENGINE_set_RSA()`. Do init funkce se přidá blok kódu, ve kterém se nově vytvořená struktura inicializuje, respektive se do ni přiřadí funkce. Nad rámec dvou implementovaných funkcí byla vytvořena `cng_rsa_free()`, jejíž úkolem je uvolnit exdata `RSA` struktury.

3.5.7 ECDSA funkcionalita

S příchodem OpenSSL 1.1 nejsou šifrové sady založeny na DSA běžnou součástí projektu. Aby byly dostupné, musí se OpenSSL kompilovat s příznakem `enable-weak-ssl-ciphers`. Z toho důvodu cng modul neimplementuje DSA, ale ECDSA. Pro podepisování za pomocí algoritmu ECDSA byla pro modul cng vytvořena funkce `cng_ecdsa_sign_sig()`, jejíž vstupní argumenty odpovídají metodě `*do_sign` ze struktury `EC_KEY_METHOD`. Znatelným rozdílem je, že se výsledný podpis předává strukturou `ECDSA_SIG`, nikoliv přes vstupně-výstupní buffer.

Opět je na vstupu struktura reprezentující klíč `EC_KEY`, jejíž exdata obsahují CNG referenci soukromého klíče. Funkce `NCryptSignHash()` je volána dvakrát. Jednou pro získání očekávané velikosti podpisu, aby se mohl připravit dostatečně velký přechodný buffer. Po druhém zavolání se do bufferu zapíše dvojice (r, s) představující ECDSA podpis. Oba prvky jsou stejně velké, a tak jsou dvě poloviny bufferu převedeny na dvě `BIGNUM` struktury. Následuje vytvoření nové návratové struktury a přiřazení hodnot `r` a `s`.

Práci s ECDSA poznamenalo v OpenSSL 1.1 několik změn oproti starší verzi a dokumentace se o EC API vůbec nezmiňuje. Uvnitř `ecdsa.h` je zapsána pouze jedna řádka, která provádí include `ec.h`. Jediná funkce, která nabízí vytvoření nové metodové struktury je `EC_KEY_METHOD_new()`, jež přijímá ukazatel na `EC_KEY_METHOD`. Ze zdrojového kódu je patrné, že pokud argument není `NULL`, vytvoří se kopie ze specifikované reference. Nová metodová strukutra je tedy vytvořena jako duplikát výchozí a pouze se její `*sign_sig` funkce předefinuje na `cng_ecdsa_sign_sig()`. Funkcionalita je do evidence `ENGINE` přidána v bind pomocí `ENGINE_set_EC()`.

Kapitola 4

Testování

Poslední kapitola se věnuje přípravě testovacího prostředí. Nejdříve bude připravena aplikace client, která k síťové komunikaci používá komponentu `BIO`. Kryptografický modul využívá pouze v případě, že součástí TLS Handshake je zpráva Certificate Request. Server je připraven jiným způsobem a kryptografický modul potřebuje pro svoji funkčnost vždy. Protože jsou tyto dvě aplikace určeny k uzavření bezpečného síťového spojení, potřebují mít k dispozici sadu digitálních certifikátů, proto tato kapitola představuje, jak vygenerovat testovací certifikáty. Poslední aplikace je izolovaná od předchozích a pouze prezentuje nadstandardní příkazy. Závěr kapitoly se věnuje diskuzi o výsledku testování.

4.1 Příprava klienta

Pokud je ambicí aplikace pracovat s knihovnou libssl, je nutné ji nejdříve inicializovat. Starší verze OpenSSL vyžadovaly zvláštní volání pro načtení chybových hlášek, současná verze celý inicializační proces zabaluje do volání `OPENSSL_init_ssl()`. Tato funkce do paměti načte kryptografické EVP struktury pro šifrování a hešování. Pokud si je tedy vývojář aplikace jistý, které algoritmy se budou za běhu využívat, může ušetřit paměť a načíst jen potřebné struktury.

V dalším kroku je potřeba vytvořit a nastavit `SSL_CTX`, který udržuje parametry pro budoucí spojení. Při vytváření nové kontextové struktury je nutné specifikovat verzi TLS protokolu. Dokumentace [26] doporučuje vytvořit strukturu podporující všechny verze TLS a v případě potřeby verzi zdola nebo shora omezit při konfiguraci existující reference. V kontextu lze dále specifikovat jaké konkrétní šifrové sady mohou být použity k navázání spojení. Tím je možné vynutit, aby se v první části TLS Handshake využil pouze RSA algoritmus, nebo ECDSA algoritmus. Bez bližšího určení klient nabídne serveru prioritní seznam šifrových sad, které jsou knihovnou libssl pokládány

za bezpečné. Podobně lze specifikovat i podpisové algoritmy pro TLS verze alespoň 1.2. Dalším důležitým krokem při konfiguraci je určit certifikáty autority, které rozhodnou o důvěryhodnosti certifikátu serveru.

```
SSL_CTX* ctx = SSL_CTX_new(TLS_method());
SSL_CTX_set_max_proto_version(ctx, TLS1_2_VERSION);
SSL_CTX_set_cipher_list(ctx, "DHE-RSA-AES256-SHA");
SSL_CTX_set1_sigalgs_list(ctx, "RSA+SHA256");
SSL_CTX_load_verify_locations(ctx, "/path/to/ca.pem", NULL);
```

Výpis kódu 4.1: Vytvoření a konfigurace struktury SSL_CTX na straně klienta

Výpis kódu 4.1 ukazuje vytvoření nové struktury `SSL_CTX` s omezenou konfigurací. Nejdříve je nastaveno, že nejvyšší možnou verzí TLS protokolu je 1.2. Následně je vynucená jedna konkrétní šifrová sada a jeden podpisový algoritmus, který znemožní využití RSA-PSS podpisového schématu. Na závěr je určen certifikát certifikační autority.

Pokud existuje více autorit, proti kterým se má certifikát serveru validovat, není nutné volat funkcí z příkladu pro každý soubor zvlášť. Prvním ulehčením je předat funkci soubor, ve kterém jsou certifikáty zřetězené. Druhým řešením je místo souboru specifikovat celý adresář, ze kterého se mají certifikáty načíst. V tom případě je ale nutné, aby tyto soubory byly pojmenovány jako hash jména autority.

Posledním krokem konfigurace kontextu je připravit klienta na Certificate Request, tedy žádost serveru o autenizaci klienta v průběhu TLS Handshake. Toho lze docílit více způsoby. Prvním je předem určit soukromý klíč a certifikát, čímž dojde k jejich načtení bez závislosti na tom, zda budou skutečně potřeba. Druhou a tímto klientem využívanou možností, je specifikovat pouze callback funkci, která se zavolá po přijetí žádosti o předložení certifikátu klienta. Do takové funkce vstupuje struktura `SSL` reprezentující probíhající spojení, ze které lze vyčíst serverem akceptovatelná jména certifikačních autorit. Na výstupu se očekává certifikát vydaný jednou z těchto autorit v reprezentaci strukturou `X509` a příslušný soukromý klíč v `EVP_PKEY`. Obě tyto položky jsou automaticky přiřazeny aktuálnímu spojení.

```
STACK_OF(X509_NAME) *ca_list = SSL_get0_peer_CA_list(ssl);
ENGINE *engine = load_engine("/path/to/engine.dll");
ENGINE_load_ssl_client_cert(engine, ssl, ca_list,
                            &x509, &pkey, NULL,
                            NULL, NULL);
```

Výpis kódu 4.2: Callback funkce pro vyřízení Certificate Requestu

Výpis kódu 4.2 zobrazuje část implementace callback funkce pro výběr certifikátu a klíče, nalezení vhodného páru je delegováno na engine. Nejdříve je

získán seznam akceptovatelných certifikačních autorit. Funkce `load_engine()` na základě specifikované cesty ke sdílené knihovně vytvoří funkcionální referenci modulu a označí jeho implementace za defaultní. Poslední příkaz využívá Engine API pro získání vhodného certifikátu a soukromého klíče. Poslední tři argumenty nevyužívá ani capi modul, ani cng. Nakonec je celá callback funkce přiřazena ssl kontextu prostřednictvím `SSL_CTX_set_client_cert_cb()`.

Až poté, co je správně inicializován `SSL_CTX`, může nastat spojení se serverem. Pro implementaci klienta jsou použity vysokoúrovňová volání modulu `BIO`, který kromě standardních I/O operací umí zprostředkovat šifrovanou komunikaci po síti. Nové `BIO` struktuře stačí specifikovat SSL kontext a adresu serveru, následně může proběhnout samotné spojení. Po provedení čtyř kroků z výpisu 4.3 lze zasílat a číst zprávy pomocí `BIO_read()` a `BIO_puts()`.

```
BIO *web = BIO_new_ssl_connect(ctx);
BIO_set_conn_hostname(web, "127.0.0.1:27015");
BIO_do_connect(web);
BIO_do_handshake(web);
```

Výpis kódu 4.3: Zahájení spojení pomocí modulu BIO

Pokud aplikace chce získat podrobnosti o spojení, je nutné si z `BIO` vyžádat `SSL` strukturu. Ta mimo jiné nese informace o serverovém certifikátu, o dohodnuté šifrové sadě a o dohodnuté verzi TLS protokolu.

4.2 Příprava serveru

Pro přípravu serveru bylo zvoleno jiné řešení než prostřednictvím modulu `BIO`. Protože server taktéž využívá knihovnu libssl, začíná zdrojový kód inicializací knihovny. V dalším kroku se pro server připraví socket pomocí rozhraní Winsock.

Tato aplikace také prezentuje možné využití kryptografického modulu, proto následuje jeho načtení a konfigurace. První možností jak získat strukturální referenci je voláním `ENGINE_by_id()`, jež deleguje načtení na dynamic engine. V tom případě se vyhledávání modulu omezí na adresář definovaný v proměnné `ENGINESDIR`, což je typicky v podadresáři OpenSSL projektu `.\lib\engines-1_1\`. Pro větší flexibilitu je výhodnější komunikovat s dynamic modulem přímo. Následně je nutné přenastavit `ENGINE_TABLE` tak, aby se využívaly implementace algoritmů poskytované právě načteným modulem.

```
ENGINE_load_dynamic();
ENGINE *e = ENGINE_by_id("dynamic");
ENGINE_ctrl_cmd_string(e, "SO_PATH", "/path/to/dll", 0);
ENGINE_ctrl_cmd_string(e, "LOAD", NULL, 0);
ENGINE_set_default(e, ENGINE_METHOD_ALL);
```

Výpis kódu 4.4: Načtení kryptografického modulu

Výpis kódu 4.4 ukazuje dynamické načítání externího modulu. Nejdříve je nutné získat dynamic engine do interního spojového seznamu. Druhý řádek výpisu získá strukturální referenci modulu, který zprostředkuje načtení externího kryptografického modulu. Dynamic modul se řídí pomocí nadstandardních příkazů. `SO_PATH` specifikuje cestu ke sdílené knihovně, která obsahuje požadovaný engine a příkazem `LOAD` dojde k načtení. Do původní reference se dostane strukturální reference hledaného modulu. Poslední řádek výpisu zaručí, že implementace všech algoritmů, které modul poskytuje, budou považovány za výchozí.

Pro server se také musí vytvořit `SSL_CTX`, nositele konfigurace pro příchozí spojení. Tentokrát je potřeba kontextové struktuře přidělit certifikát jakým se bude server prokazovat a klíč pro asymetrickou kryptografii. Server pro uchování certifikátu využívá úložiště operačního systému Windows, proto implementuje funkci `get_cert_from_winstore()`, která vyhledá v systémovém úložišti `MY` certifikát podle jména subjektu. Jedná se pouze o zjednodušenou funkci `find_cert()` známé z capi modulu, jen na závěr je certifikát převeden do struktury `X509`, aby byl čitelný OpenSSL knihovnou.

```
SSL_CTX *ctx = SSL_CTX_new(TLS_method());
ENGINE_init(e);

X509* cert = get_cert_from_winstore("localhost");
SSL_CTX_use_certificate(ctx, cert);

EVP_PKEY *key = ENGINE_load_private_key(e, "localhost",
                                        NULL, NULL);
SSL_CTX_use_PrivateKey(ctx, key);
```

Výpis kódu 4.5: Vytvoření a konfigurace SSL_CTX na straně serveru

Výpis kódu 4.5 zobrazuje průběh přípravy SSL kontextové struktury. Po vytvoření nové `SSL_CTX` se získá funkcionální reference modulu. Následně je v úložišti vyhledán certifikát, jehož jméno subjektu obsahuje řetězec localhost a v `X509` reprezentaci je předán kontextové struktuře. Pro získání klíče k příslušnému certifikátu se již využívá řešení vytvořené modulem. Připravená `EVP_PKEY` struktura je předána kontextu. Do kontextové struktury lze při-

pojit více certifikátů a klíčů. Jaký pár se k zahájení spojení nakonec využije záleží na dohodnuté šifrové sadě.

Pokud má server po klientovi vyžadovat autorizaci, je nutné konfiguraci ještě obohatit. Nejdříve se musí, podobně jako v případě klienta, specifikovat certifikáty certifikačních autorit a následně vytvořit jejich jmenný seznam.

```
SSL_CTX_load_verify_locations(ctx, "ca.pem", NULL);
STACK_OF(X509_NAME) *certs = SSL_load_client_CA_file("ca.pem");
SSL_CTX_set_client_CA_list(ctx, certs);
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER |
                        SSL_VERIFY_FAIL_IF_NO_PEER_CERT, NULL);
```

Výpis kódu 4.6: Vynucení autorizace klienta

Výpis 4.6 zobrazuje konfiguraci serverového SSL kontextu tak, aby byla vždy zaslána zpráva Certificate Request. Nejdříve dojde k určení certifikátu autority, který bude validovat klienta. Tento příkaz nijak neovlivní obsah zprávy Certificate Request, proto je nutné vzápětí certifikát znovu načíst a zařadit ho do jmenného seznamu, čímž se už stane součástí zprávy. Nakonec je nařízeno, že pokud ověření certifikátu klienta selže, nepodaří se navázat spojení.

Po inicializaci struktury může server přistoupit k utváření spojení s klienty. Nejdříve server vyčkává, dokud nepřijde žádost ke spojení, poté vytvoří novou SSL strukturu, která spojení reprezentuje. Po přesměrování komunikačního kanálu na příslušný socket vyčkává dokud, klient nezahájí handshake.

```
int socket = accept(); /*funkce z WinSock2.h*/
ssl = SSL_new(ctx);
SSL_set_fd(ssl, client);
SSL_accept(ssl);
SSL_write(ssl, "Hello there", 11);
SSL_shutdown(ssl);
```

Výpis kódu 4.7: Vytvoření spojení s klientem

Výpis kódu 4.7 je průvodcem připojením nového klienta k serveru. První řádek pracuje s rozhraním WinSock, povolí klientovi přístup a vrátí číslo socketu, na který server následně přesměruje I/O operace. Na čtvrtém řádku server reaguje na žádost o provedení TLS Handshake a po jeho úspěšném dokončení je klientovi poslána krátká zpráva. Aniž by server čekal na jakoukoliv reakci klienta, ukončuje navázané spojení.

4.3 Příprava certifikátů

Každý server na internetu, který umožňuje komunikaci pomocí protokolu HTTPS, vystupuje pod svým certifikátem. Digitální certifikát je množina dat, která identifikuje daný subjekt. Vytvoření takového certifikátu není náročné a lze mu přiřadit lživá data, sám o sobě tedy důvěryhodnost nepřináší. Aby byla potvrzena jeho pravost, musí se za něj zaručit certifikační autorita.

Certifikační autorita disponuje soukromým klíčem a certifikátem, jehož obsahem je standardně veřejný klíč a informace o subjektu. Pokud nový subjekt usiluje o získání ověřeného certifikátu, předá autoritě CSR s informaci na jejichž základě bude vytvořen ověřený certifikát. Certifikát certifikační autority je takzvaný self-signed, tedy ověřený sám sebou.

Zahájení bezpečné komunikace mezi serverem a klientem předchází TLS Handshake, v rámci kterého dojde k předání certifikátů a dohodnutí na parametrech spojení. Pro ověření důvěryhodnosti certifikátu je potřeba mít k dispozici certifikát autority, respektive její veřejný klíč. Typicky to probíhá tak, že certifikát je podepsán světově známou certifikační autoritou jako je například Verisign, jejíž certifikát je součástí webových prohlížečů, tedy klientů, kteří usilují o připojení k serverům.

I pro výše popsané testovací aplikace je potřeba dodržet tuto infrastrukturu certifikátů. Protože se jedná jen o testování, je možné vytvořit si lokální certifikační autoritu. K tomu účelu lze využít aplikační vrstvu projektu OpenSSL.

```
openssl genrsa -out CA.key
openssl req -x509 -new -key CA.key -out CA.pem
```

Výpis kódu 4.8: Vytvoření self-signed certifikační aurtority

Výpis 4.8 vytváří lokální certifikační autoritu. Nejdříve vytvoří soukromý klíč do `CA.key` a následně certifikát do `CA.pem`. Při vykonávání druhého příkazu se otevře možnost interaktivního doplnění informací o certifikátu, tomu lze předejít předáním dat přes přepínač `subj`.

Pro vytvoření ověřeného certifikátu pro server se postupuje následovně.

```
openssl genrsa -out server.key
openssl req -new -key server.key -out server.csr
openssl x509 -req -in server.csr
                  -CA CA.pem -CAkey CA.key
                  -CAcreateserial -out server.pem
```

Výpis kódu 4.9: Vytvoření ověřeného serverového certifikátu

Nejdříve se vytvoří nový soukromý klíč do souboru `server.key`. Následně se vytvoří CSR, do jehož tvorby se opět vloží interaktivní zadávání informací

o serveru. Opět tomu lze předejít stejně jako v předchozím případě. Posledním příkazem se pomocí dříve vytvořené certifikační autority akceptuje CSR a vytvoří certifikát `server.pem`. Stejným stylem se vytváří i certifikát klienta.

Těmito pěti příkazy byla vytvořena infrastruktura pro síťovou autorizaci pomocí RSA. Pro vygenerování ECDSA klíče je nutné specifikovat identifikátor eliptické křivky.

```
openssl ecparam -name prime256v1 -genkey -out ecdsa.key
```

Výpis kódu 4.10: Vytvoření soukromého ECDSA klíče

Generování CSR a jeho schválení je identické s předchozími příklady. Vytvořeným serverovým certifikátům bude samozřejmě důvěřovat jen klient, který důvěřuje těmto lokálním certifikačním autoritám a má k dispozici jejich certifikáty. Aby se využilo schopností capi a cng modulů, je nyní potřeba umístit certifikáty do úložiště certifikátu ve Windows.

Pokud je požadováno předat certifikát úložišti a mít k němu navázán soukromý klíč, je vhodné převést pár do formátu pkcs12, jak ukazuje příkaz 4.11.

```
openssl pkcs12 -export -inkey private_key.key
                       -in certificate.pem
                       -out key_cert_bundle.pfx
```

Výpis kódu 4.11: Transformace certifikátu a klíče do formátu PKCS12

Existuje více způsobů jak pfx soubor předat úložišti. První možností je využít přímo CryptoAPI rozhraní prostřednictvím aplikace v jazyce C nebo C#. Rychlejší variantou je využít příkaz `certutil -importpfx`, kterému lze specifikovat konkrétní CSP, nebo PowerShell příkaz `Import-PfxCertificate`, který ale využívá CNG, a tak uložené klíče nebudou z CryptoAPI dostupné. Možností, která nejlépe představí certifikační úložiště, je využít grafické rozhraní označováno jako Microsoft Management Console. Aplikaci lze spustit příkazem `certmgr` v příkazovém řádku. Ve výchozím nastavení zobrazí pouze certifikáty přístupné aktuálnímu uživateli. Import certifikátu se provede pomocí `Action > All Tasks > Import...`, což spustí interaktivního průvodce. Po určení cesty k pfx souboru je možné vybrat, do kterého úložiště soubor přiřadit. Certifikát serveru a klienta by se měl umístit do Personal, certifikát autority by se měl importovat do Trusted Root Certification Authorities.

4.4 Ukázka dalších funkcí

Pro prezentaci práce s nadstandardními příkazy modulu byla vytvořena aplikace `eng_load.c`. Ta očekává předání cesty ke sdílené knihovně s kryptografickým modulem jako první argument. K jeho načtení je použit dynamic engine, stejně jako tomu je u předchozích aplikací. Aplikace nejdříve vypíše celou nabídku nadstandardních příkazů a vybrané z nich vykoná.

Kromě příkazů, které si engine definuje sám, existuje i sada obecných příkazů. Protože je nutné tyto příkazy volat přímo přes jejich číselný kód, je ze tří rozhraním nabízených funkcí přípustná pouze `ENGINE_ctrl()`. Aplikace implementuje funkci `print_ctrls()`, která využívá obecné funkce pro vypsání všech dostupných specifických příkazů.

Aplikace dále prezentuje volání příkazů modulu pomocí jejich jmenného identifikátoru. K tomu Engine API vyhrazuje funkci `ENGINE_ctrl_cmd()` nebo `ENGINE_ctrl_cmd_string()`. Ta druhá přináší vyšší míru abstrakce, protože není třeba znát datový typ, který příkaz očekává. Funkce to sama ověří a pokud je to nutné, datový typ převede. Poslední argument těchto funkcí rozhoduje, jestli má být ignorována případná neexistence požadovaného příkazu. Následující ukázka prezentuje použití obou funkcí pro přesměrování debugovacích hlášek.

```
/*e je reference struktury ENGINE*/
ENGINE_ctrl_cmd_string(e, "debug_file", "trace.log", 0);
ENGINE_ctrl_cmd(e, "debug_level", 2, NULL, NULL, 0);
```

Výpis kódu 4.12: Volání nadstandardních příkazů

4.5 Pozorování

Pro otestování funkčnosti kryptografického modulu byl využit testovací server a klient za použití certifikátů a klíčů vygenerovaných metodou, která byla popsána v kapitole 4.3. Server i klient je nastavitelný pomocí konfiguračního souboru vstupujícího jako první argument při spouštění aplikace. Postup pro přípravu testovacího prostředí a způsob jak testovací aplikace spouštět a konfigurovat je popsán v souboru README.txt na přiloženém datovém úložišti.

Testovací sada byla nejdříve vyzkoušena na ověřeném capi modulu za použití RSA certifikátů. Nejdříve byl klient omezen tak, aby umožnil spojení pouze pomocí protokolu TLS 1.0. V tom případě se spojení uskutečnilo a klient zobrazil přijatou zprávu. Stejný výsledek mělo i spojení řízené protokolem TLS 1.1. Na těchto dvou verzích v pořádku proběhla i autorizace klienta. Protože byl capi modul zkompilován s příznakem `OPENSSL_CAPIENG_DIALOG`, po obdržení Certificate Requestu se zobrazilo dialogové okno pro výběr certifikátu klienta. Při použití novějších protokolů se zobrazila chybová hláška o chybějící

implementaci funkce pro šifrování soukromým klíčem, jak bylo popsáno v kapitole 3.4.7. Tuto chybu lze obejít tak, že nastavením spojení bude vynuceno použití podpisového algoritmu, který nepracuje se schématem PSS, například `RSA+SHA256`. Poté bylo možné spojení uzavřít i s použitím aktuálních protokolů a přijmout zprávu ze strany serveru.

Podobně byla otestována i funkčnost cng modulu. Server byl spuštěný tak, aby využíval kryptografický modul cng. Pokud je vedle toho spuštěn klient bez jakéhokoliv omezení, dohodnuté spojení dodržuje protokol TLS 1.3 s šifrovou sadou `TLS_AES_256_GCM_SHA384`. Kromě obdržené zprávy jsou dále vypsány informace o serverovém certifikátu, v tomto případě se jedná o certifikát podepsaný RSA klíčem. Pokud klient omezí podmínky komunikace na protokol TLS 1.2, uzavře se spojení s šifrovou sadou `ECDHE-ECDSA-AES256-GCM-SHA384`, využije se tedy ECDSA algoritmus implementovaný modulem. Upravením klientem nabízené šifrové sady lze vynutit použití RSA algoritmu, jenž taktéž úspěšně skončí vypsáním zprávy od serveru. Nic na tom nezmění ani zákaz RSA-PSS. Klientova reakce na zprávu Certificate Request je stejná jako v případě capi. Pokud je modul kompilován s příznakem `OPENSSL_CNGENG_DIALOG`, zobrazí se dialogové okno, přes které lze interaktivně zvolit certifikát, který bude zaslán serveru. Stejný postup testování byl aplikován i při využití aktuálně vyvíjené verze OpenSSL 3.0, výsledky byly identické.

Pro úplnost byla otestována i zpětná kompatibilita modulu cng. Při využití algoritmu RSA společně se zastaralými protokoly TLS 1.0 nebo 1.1 nelze uzavřít spojení. Je to způsobeno tím, že tyto verze si pro hešování vyžádají algoritmus SHA+MD5, tedy SHA následovaný MD5, což Cryptography API: Next Generation ve svém základu nepodporuje. Pomocí algoritmu ECDSA lze ale zahájit komunikaci ve všech verzích protokolu TLS.

Při pokusu zkompilovat modul pomocí knihoven OpenSSL 1.0 nastane několik chyb. To je způsobeno tím, že s OpenSSL 1.1 bylo vytvořeno nové rozhraní kryptografických struktur. Modul cng používá struktury `EC`, zatímco ve starší verzi OpenSSL byly pojmenovány `ECDSA`. Také přibylo několik get funkcí, které dříve neexistovaly. Tyto rozdíly by bylo možné vyřešit podmíněným překladem, avšak OpenSSL 1.0 není již dále podporovanou verzí, a proto se engine soustředí na funkčnost v aktuální LTS verzi.

Závěr

Primárním cílem této bakalářské práce bylo vytvořit kryptografický modul pro OpenSSL s podporou Microsoft Cryptography API: Next Generation.

První kapitola uvedla projekt OpenSSL s důrazem na popis architektury. V knihovně libcrypto nalezneme implementaci kryptografických algoritmů, a proto ji lze považovat za jádro celého projektu. Výchozí funkcionalita knihovny je rozšiřitelná díky rozhraní Engine API.

Druhá kapitola prezentuje kryptografická rozhraní dodávaná jako součást operačních systémů od firmy Microsoft. Prvním je CryptoAPI, které je již považováno za zastaralé. Autoři doporučují využívat novější rozhraní označené jako Cryptography API: Next Generation. Obě rozhraní poskytují kryptografická primitiva a spolupráci s úložištěm certifikátů. Výhoda tohoto úložiště tkví v jeho abstrakci. Certifikáty evidované tímto prostředím totiž vystupují pod stejnou strukturou, bez ohledu na to, zda se jedná o certifikát na čipové kartě nebo o datový blok na disku.

Třetí kapitola podrobně popisuje Engine API, rozhraní pro tvorbu kryptografického modulu pro OpenSSL. Analyzuje již existující capi engine pro podporu CryptoAPI, jehož řešení není nadále vyhovující. Dále byl demonstrován postup pro vytvoření kryptografického modulu s podporou Microsoft Cryptography API: Next Generation, který může sloužit jako náhrada za capi. V závěrečné kapitole byl otestován nově vytvořený cng engine a bylo předvedeno jeho běžné použití.

Dlužno dodat, že součástí projektu OpenSSL není dostatečně kvalitní a jednotná dokumentace. Zcela chybí popis logických celků nebo architektury jednotlivých rozhraní, využívání zdrojového kódu pro získání podstatných informací bylo běžnou součástí vzniku práce. Na zcela chybějící dokumentaci k Engine API upozorňují samotní vývojáři, žádají dobrovolníky o poskytnutí shrnující dokumentace spolu s instrukcemi, jak rozhraní používat. Tato práce se těmto tématům podrobně věnuje a po překladu do anglického jazyka by tak mohla být nápomocná celé OpenSSL komunitě.

Bibliografie

1. COX, Mark. Celebrating 20 Years of OpenSSL. In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online]. 2018 [cit. 2020-05-24]. Dostupné z: `https://www.openssl.org/blog/blog/2018/12/20/20years`.
2. THE OPENSSL PROJECT. OpenSSL Bylaws. In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-04-22]. Dostupné z: `https://www.openssl.org/policies/omc-bylaws.html`.
3. THE OPENSSL PROJECT. OpenSSL command line tool manual. In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-04-22]. Dostupné z: `https://www.openssl.org/docs/man1.1.1/man1/openssl.html`.
4. THE OPENBSD PROJECT. LibreSSL Goals. In: *LibreSSL* [online] [cit. 2020-04-22]. Dostupné z: `https://www.libressl.org/goals.html`.
5. WOLFSSL INC. wolfSSL About Us. In: *wolfSSL* [online] [cit. 2020-04-22]. Dostupné z: `https://www.wolfssl.com/about`.
6. RAMBUS INC. Inside Secure TLS Toolkit. In: *Rambus* [online] [cit. 2020-04-22]. Dostupné z: `https://www.rambus.com/security/software-protocols/secure-communication-toolkits/tls-toolkit/`.
7. GOOGLE LLC. BoringSSL. In: *BoringSSL* [online] [cit. 2020-04-22]. Dostupné z: `https://boringssl.googlesource.com/boringssl`.
8. GNUTLS CONTRIBUTORS. GnuTLS 3.6.13. In: *The GnuTLS Transport Layer Security Library* [online] [cit. 2020-04-22]. Dostupné z: `https://www.gnutls.org/manual/gnutls.html`.
9. BIRR-PIXTON, Joseph. rustls versus OpenSSL: resumption performance. In: *jpb.io* [online]. 2019 [cit. 2020-04-22]. Dostupné z: `https://jbp.io/`.

10. W3TECHS. Comparison of the usage statistics of Apache vs. Nginx for websites. In: *W3Techs — World Wide Web Technology Surveys* [online] [cit. 2020-04-30]. Dostupné z: https://w3techs.com/technologies/comparison/ws-apache,ws-nginx.

11. NGINX INC. Configuring HTTPS servers. In: *nginx* [online] [cit. 2020-04-30]. Dostupné z: http://nginx.org/en/docs/http/configuring_https_servers.html.

12. WOLFSSL. wolfSSL + Apache httpd. In: *wolfSSL* [online]. 2020 [cit. 2020-04-30]. Dostupné z: https://www.wolfssl.com/wolfssl-apache-httpd/.

13. THE OPENSSL PROJECT. OpenSSL Strategic Architecture. In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-04-22]. Dostupné z: https://www.openssl.org/docs/OpenSSLStrategicArchitecture.html.

14. THE OPENSSL PROJECT. RSA_METHOD(3). In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-05-05]. Dostupné z: https://www.openssl.org/docs/man1.1.1/man3/RSA_set_method.html.

15. THE OPENSSL PROJECT. RSA_private_encrypt(3). In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-05-05]. Dostupné z: https://www.openssl.org/docs/man1.1.1/man3/RSA_private_encrypt.html.

16. THE OPENSSL PROJECT. DSA_sign_setup(3). In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-05-05]. Dostupné z: https://www.openssl.org/docs/man1.1.1/man3/DSA_sign_setup.htmll.

17. THE OPENSSL PROJECT. ECDSA_sign(3). In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-05-05]. Dostupné z: https://www.openssl.org/docs/man1.1.0/man3/ECDSA_sign_setup.html.

18. THE OPENSSL PROJECT. openssl/evp.h. In: *GitHub, Inc.* [online] [cit. 2020-05-09]. Dostupné z: https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/include/openssl/evp.h.

19. THE OPENSSL PROJECT. EVP_CIPHER(3). In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-05-09]. Dostupné z: https://www.openssl.org/docs/man1.1.1/man3/EVP_EncryptInit.html.

20. THE OPENSSL PROJECT. crypto/evp.h. In: *GitHub, Inc.* [online] [cit. 2020-05-09]. Dostupné z: https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/include/crypto/evp.h.

21. THE OPENSSL PROJECT. pmeth_lib.c. In: *GitHub, Inc.* [online] [cit. 2020-05-09]. Dostupné z: https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/crypto/evp/pmeth_lib.c.

22. THE OPENSSL PROJECT. BN(3). In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-05-06]. Dostupné z: https://www.openssl.org/docs/man1.1.1/man3/BN_bn2bin.html.

23. THE OPENSSL PROJECT. exdata(3). In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-05-06]. Dostupné z: https://www.openssl.org/docs/man1.1.1/man3/CRYPTO_set_ex_data.html.

24. THE OPENSSL PROJECT. BIO(3). In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-05-06]. Dostupné z: https://www.openssl.org/docs/man1.1.1/man3/BIO_do_connect.html.

25. THE OPENSSL PROJECT. TLS/SSL and crypto library. In: *GitHub, Inc.* [online] [cit. 2020-05-01]. Dostupné z: https://github.com/openssl/openssl/tree/OpenSSL_1_1_1-stable.

26. THE OPENSSL PROJECT. SSL_CTX(3). In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-05-09]. Dostupné z: https://www.openssl.org/docs/man1.1.1/man3/SSL_CTX_new.html.

27. THE OPENSSL PROJECT. libssl(7). In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-05-09]. Dostupné z: https://www.openssl.org/docs/man1.1.1/man7/ssl.html.

28. HOUSLEY, R.; POLK, W.; FORD, W.; SOLO, D. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile* [Internet Requests for Comments]. RFC Editor, 2002. Dostupné také z: https://www.ietf.org/rfc/rfc3280.txt. Standards Track. RFC Editor.

29. SATRAN, Michael; BATCHELOR, Drew. Managing Certificates with Certificate Stores. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-12]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/seccrypto/managing-certificates-with-certificate-stores.

30. SATRAN, Michael; BATCHELOR, Drew. System Store Locations. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-12]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/seccrypto/system-store-locations.

31. SATRAN, Michael; BATCHELOR, Drew. CSPs and the Cryptography Process. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-12]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/seccrypto/csps-and-the-cryptography-process.

32. SATRAN, Michael; BATCHELOR, Drew. Cryptographic Provider Types. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-12]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/seccrypto/cryptographic-provider-types.

33. SATRAN, Michael; JACOBS, Mike. CryptoAPI Cryptographic Service Providers. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-12]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/seccertenroll/cryptoapi-cryptographic-service-providers.

34. SATRAN, Michael; BATCHELOR, Drew. CryptoAPI System Architecture. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-12]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/seccrypto/cryptoapi-system-architecture.

35. MICROSOFT CORPORATION. CertFindCertificateInStore function. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-12]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-certfindcertificateinstore.

36. MICROSOFT CORPORATION. CertGetCertificateContextProperty function. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-23]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-certgetcertificatecontextproperty.

37. SATRAN, Michael; BATCHELOR, Drew. Cryptographic Service Provider Contexts. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-12]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/seccrypto/cryptographic-service-provider-contexts.

38. SATRAN, Michael; JACOBS, Mike; BATCHELOR, Drew. CertGetCertificateContextProperty function. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-23]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/seccrypto/base-provider-key-blobs.

39. SATRAN, Michael. Cryptographic Primitives. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-12]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/seccng/cryptographic-primitives.

40. SATRAN, Michael; JACOBS, Mike; BATCHELOR, Drew. CNG Algorithm Identifiers. In: *Microsoft Docs* [online]. 2008 [cit. 2020-05-12]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/seccng/cng-algorithm-identifiers.

41. SATRAN, Michael; LIU, Xiaoyin. Key Storage and Retrieval. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-12]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/seccng/key-storage-and-retrieval.

42. MICROSOFT CORPORATION. NCryptExportKey function. In: *Microsoft Docs* [online]. 2018 [cit. 2020-05-23]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/api/ncrypt/nf-ncrypt-ncryptexportkey.

43. THE OPENSSL PROJECT. ENGINE(3). In: *OpenSSL — Cryptography and SSL/TLS Toolkit* [online] [cit. 2020-05-07]. Dostupné z: https://www.openssl.org/docs/man1.1.1/man3/ENGINE_init.html.

44. THE OPENSSL PROJECT. engine.h. In: *GitHub, Inc.* [online] [cit. 2020-05-07]. Dostupné z: https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/include/openssl/engine.h.

45. THE OPENSSL PROJECT. README.ENGINE. In: *GitHub, Inc.* [online] [cit. 2020-05-01]. Dostupné z: https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/README.ENGINE.

46. THE OPENSSL PROJECT. eng_lib.c. In: *GitHub, Inc.* [online] [cit. 2020-05-08]. Dostupné z: https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/crypto/engine/eng_lib.c.

47. THE OPENSSL PROJECT. eng_local.h. In: *GitHub, Inc.* [online] [cit. 2020-05-07]. Dostupné z: https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/crypto/engine/eng_local.h.

48. THE OPENSSL PROJECT. eng_table.c. In: *GitHub, Inc.* [online] [cit. 2020-05-08]. Dostupné z: https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/crypto/engine/eng_table.c.

49. THE OPENSSL PROJECT. eng_dyn.c. In: *GitHub, Inc.* [online] [cit. 2020-05-11]. Dostupné z: https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/crypto/engine/eng_dyn.c.

50. DIERKS, T.; RESCORLA, E. *The Transport Layer Security (TLS) Protocol Version 1.2* [Internet Requests for Comments]. RFC Editor, 2008. Dostupné také z: https://tools.ietf.org/html/rfc5246. RFC. RFC Editor.

51. MORIARTY, K.; FARRELL, S. *Deprecating TLSv1.0 and TLSv1.1 draft-ietf-tls-oldversions-deprecate-06* [Internet Requests for Comments]. RFC Editor, 2020. Dostupné také z: https://tools.ietf.org/html/draft-ietf-tls-oldversions-deprecate-06. Best Current Practice. RFC Editor.

52. THE OPENSSL PROJECT. rsa_pmeth.c. In: *GitHub, Inc.* [online] [cit. 2020-05-11]. Dostupné z: https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/crypto/rsa/rsa_pmeth.c.

53. LEVITTE, Richard. TLS server handshake fails when using CAPI engine. In: *GitHub Issues* [online] [cit. 2020-05-30]. Dostupné z: https://github.com/openssl/openssl/issues/8872#issuecomment-557005351.

54. MORIARTY, K.; KALISKI, B.; JONSSON, J.; RUSCH, A. *PKCS #1: RSA Cryptography Specifications Version 2.2* [Internet Requests for Comments]. RFC Editor, 2016. Dostupné také z: `https://tools.ietf.org/html/rfc8017`. Standards Track. RFC Editor.

Příloha A

Seznam použitých zkratek

API Application Programming Interface.

CAPI CryptoAPI.

CNG Cryptography API: Next Generation.

CRL Certificate Revocation List.

CSR Certificate Signing Request.

DTLS Datagram Transport Layer Security.

KSP Key Storage Provider.

MD5 MD5 message-digest algorithm.

RFC Request for Comments.

SHA Secure Hash Algorithm.

SSL Secure Sockets Layer.

TLS Transport Layer Security.

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... popis obsahu CD a manuál k testovacímu prostředí
- `impl` ..... zdrojové kódy implementací
  - `certs` ..... skripty pro přípravu certifikátů
  - `engine` ..... zdrojové kódy modulů
  - `test` ..... zdrojové kódy testovacích aplikací
  - `_run.cmd` ..... spouštěcí script
  - `example.conf` ..... ukázkový konfigurační soubor
  - `makekfile` ..... makefile pro automatizovanou přípravu prostředí
- `exe` ..... připravené testovací prostředí
  - `certs` ..... vygenerované certifikáty
  - `engine` ..... knihovny modulů
  - `test` ..... zdrojové kódy testovacích aplikací
  - `_run.cmd` ..... spouštěcí script
  - `example.conf` ..... ukázkový konfigurační soubor
  - `libcrypto-1_1.dll` ..... knihovna libcrypto
  - `libssl-1_1.dll` ..... knihovna libssl
- `text` ..... text práce
  - `BP_Pesek_Jan_2020.pdf` ..... text práce ve formátu PDF
  - `src` ..... zdrojová forma práce ve formátu LaTeX
- `archive` ..... zkomprimované adresáře text, exe a impl