VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

EXISTUJÍCÍ ÚTOKY NA SSL/TLS

EXISTING ATTACKS ON SSL/TLS PROTOCOL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MILAN LYSONĚK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ FIEDOR**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Lysoněk Milan**
Obor: Informační technologie
Téma: **Existující útoky na SSL/TLS**
Existing Attacks on SSL/TLS Protocol
Kategorie: Bezpečnost

Pokyny:

1. Nastudujte návrh a implementaci SSL/TLS protokolu. Seznamte se s projektem tlsfuzzer a jeho možné využití pro testování SSL/TLS protokolu. Seznamte se se stávajícími útoky na SSL/TLS protokol.
2. Identifikujte existující útoky vhodné pro reprodukci v nástroji tlsfuzzer. Vytvořte pro ně testovací sadu.
3. Implementujte testovací sadu reprodukující zvolené útoky nad platformami tlsfuzzer a tlslite-neg. Rozšiřte tyto platformy pro podporu zvolených útoků.
4. Otestujte testovací sadu nad existující SSL/TLS implementací a zhodnoťte dosažené výsledky.

Literatura:

- Ristic, I. Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications. Feisty Duck, 2014. ISBN: 978-1907117046
- Domovská stránka projektu TLS Fuzzer. URL: https://github.com/tomato42/tlsfuzzer
- Domovská stránka projektu TLSLite-ng. URL: https://github.com/tomato42/tlslite-ng

Pro udělení zápočtu za první semestr je požadováno:

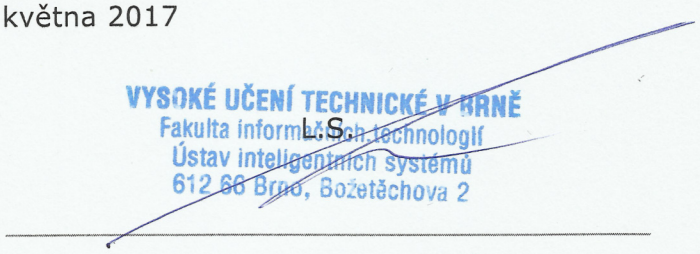
- Body 1 a 2

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Fiedor Tomáš, Ing.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017



L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

SSL/TLS je moderní kryptografický protokol, který zabezpečuje komunikaci mezi klientem a serverem. Avšak na tento protokol existují útoky, které mohou ohrozit komunikaci buď odposloucháváním nebo jejím narušením. Obrana proti těmto útokům a testování zranitelností protokolů je ale značně náročný proces. Tato práce popisuje zranitelnosti SSL/TLS protokolu a implementuje vybrané útoky v tlsfuzzeru — nástroj pro testování SSL/TLS implementací. Výsledná implementace útoků je demonstrována na třech SSL/TLS implementacích.

Abstract

SSL/TLS is a modern cryptographic protocol, which secures the communication between client and server. However, there are attacks on this protocol which can compromise communication either by eavesdropping or disruption. Defending against such attacks and testing the bulletproofness of protocols is a challenging process. This work describes attacks on SSL/TLS and implements selected attacks within tlsfuzzer — a sophisticated solution for testing SSL/TLS implementations. The resulting implementation of attacks is demonstrated on three SSL/TLS implementations.

Klíčová slova

SSL, TLS, síťový protokol, útoky, testování, CVE

Keywords

SSL, TLS, network protocol, attacks, testing, CVE

Citace

LYSONĚK, Milan. *Existující útoky na SSL/TLS*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Fiedor Tomáš.

Existující útoky na SSL/TLS

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Fiedora. Další informace mi poskytli Ing. Stanislav Židek a Hubert Kario. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Milan Lysoněk
16. května 2017

Poděkování

Rád bych poděkoval Ing. Tomáši Fiedorovi za jeho ochotu, rady a jeho odborné vedení této práce. Dále bych chtěl poděkovat Ing. Stanislavu Židkovi za jeho rady a vedení v technické stránce práce, Hubertu Kariovi za uvedení do problematiky a za cenné rady při implementaci a rodině za podporu.

Obsah

Kapitola 1

Úvod

V dnešní době internet používají miliony lidí každý den. Pro většinu lidí se stal neodmyslitelnou součástí jejich životů, součástí, bez které by často nebyli schopni vykonávat každodenní činnosti. Používání internetu však nemusí být vždy bezpečné. Může se stát, že mezi dvěma komunikujícími zařízeními bude naslouchat někdo neznámý. Někdo, kdo chce vědět, o čem komunikace je a v mnoha případech by ji i rád využil pro vlastní prospěch. Jako příklad můžeme uvést bankovní transakce pomocí internetového bankovnictví — neradi bychom, aby někdo bez našeho vědomí věděl, na který účet byla transakce odeslána, nebo dokonce aby změnil cílový účet či velikost odeslané hotovosti. Proto, aby podobná situace nenastala, byly vytvořeny kryptografické protokoly, které zajišťují naši bezpečnost a soukromí na internetu. Ovšem tyto protokoly jsou vymyšleny a implementovány lidmi, tedy nejsou dokonalé a stále existují možnosti, jak lze naše soukromí narušit. Tyto možnosti jsou útoky na bezpečnost. Těchto útoků existuje velké množství a další se stále objevují, proto je potřeba stále vytvářet další testy, kterými bychom ověřili, jestli jsme zranitelní.

SSL/TLS je známý kryptografický protokol, jehož použití sahá od elektronické pošty, přes webové servery až po přenos hlasu pomocí VoIP protokolu. Jeho nejznámější open source variantou je OpenSSL vyvíjený komunitou od 90-tých let. Jeho bezpečnost však již byla několikrát pokořena, například v roce 2012 tzv. Heartbleed útokem [5], kdy bylo zkompromitováno přes půl miliónu webových serverů. Avšak testování útoků vůči implementacím SSL/TLS protokolu, jako je například uvedené OpenSSL, není zcela jednoduché.

Často používanou technikou testování síťových protokolů je tzv. fuzz testování — zasílání velkého množství mutovaných dat a sledování odezvy protokolu. Tato technika je často využívána pro testování bezpečnostních problémů a pro testování využívá nástroje, kterým se říká fuzzery. Tlsfuzzer 3.4 je testovací sada a open source fuzzer specifický pro testování implementací SSL/TLS protokolu. Jeho stávající implementace je však nedostačující, jelikož ani nepodporuje zmíněný Heartbleed útok. Cílem mojí práce je rozšířit tento nástroj o další funkce a testy reprodukující vybrané útoky na SSL/TLS.

Tato práce je členěna do sedmi kapitol. V kapitole 2 je popsán protokol SSL/TLS a jeho součásti. Kapitola 3 popisuje fuzz testování a tlsfuzzer (testovací sada a nástroj na fuzz testování implementací TLS protokolu). V kapitole 4 jsou popsány jednotlivé kategorie útoků na SSL/TLS protokol a vybrané zaznamenané útoky. V kapitole 5 jsou popsány útoky, které byly implementovány a jejich bližší principy. Kapitola 6 popisuje experimentální vyhodnocení implementace na třech SSL/TLS implemnetacích — OpenSSL, GnuTLS a NSS. V poslední kapitole je závěr práce.

Kapitola 2

SSL a TLS

Secure Sockets Layer (SSL) a Transport Layer Security (TLS) jsou kryptografické protokoly navržené pro poskytnutí zabezpečené komunikace na službách bez vlastního zabezpečení. Mezi tyto služby patří například WWW, elektronická pošta nebo přenos hlasu přes protokol IP (VoIP). SSL a TLS umožňuje komunikovat s libovolnou službou na internetu a zajišťuje, že obsah zasílaných zpráv si bude moci přečíst pouze cílový server, jehož identita je ověřena důvěryhodnou autoritou.

SSL je předchůdce TLS. Je založen na stejných principech a jeho poslední verze SSLv3 je velmi podobná první verzi TLS (TLS 1.0). Změna názvu proběhla kvůli neshodám s firmou Microsoft. Můžeme proto narazit na použití názvu jednoho protokolu, např. SSL, kterým jsou myšleny i nové verze, které ale správně nesou název TLS. V této práci je vždy ovšem zdůrazněno, o který protokol a verzi se jedná, jelikož většina útoků je možné provést pouze na konkrétní verze protokolu.

2.1 Vývoj SSL a TLS

SSL protokol byl vyvinut firmou Netscape. Jeho první verze, SSLv1, však nebyla nikdy zveřejněna. První zveřejněnou verzí bylo až SSLv2 roku 1994. Jelikož tato verze byla vyvinuta pouze firmou Netscape bez konzultace s jinými bezpečnostními experty, obsahovala mnoho nedostatků a vážných slabin. Velkou změnou pak byla verze SSLv3 v roce 1995, což byl zcela nově navržený protokol, který se již podobal protokolu, jaký známe dnes [16].

V roce 1999 bylo zveřejněno TLS 1.0. Tento protokol byl vytvořen firmou IETF (The Internet Engineering Task Force) se změněným názvem kvůli neshodám mezi firmami Netscape a Microsoft, jelikož Microsoft vydal svůj PCT (Private Communications Technology) protokol jako náhradu za SSLv2. Komunita nechtěla, aby byly protokoly rozděleny, proto se na společné schůzi dohodli o vývoji v IETF a tedy i na změně názvu [7]. TLS 1.0 však nepřinesla žádné výrazné změny oproti SSLv3. Roku 2006 byla vydána verze TLS 1.1, jež přidala podporu pro TLS rozšíření, umožňující přidávat funkcionalitu beze změny samotného protokolu. Verze TLS 1.2 byla zveřejněna roku 2008 a přidala podporu pro ověřené šifrování (authenticated encryption) a odstranila ze specifikace bezpečnostní primitiva (security primitives, např. hašovací funkce nebo generátor pseudonáhodných čísel), které v ní byly přímo zakódovány, aby byl protokol plně přizpůsobitelný. V roce 2016 byl vytvořen návrh na TLS 1.3, ve kterém by měly být odstraněny podpory pro nezabezpečené nebo zastaralé funkce jako je komprese dat, renegociace či Change Cipher Spec protokol [15].

Většina změn mezi jednotlivými protokoly od verze SSLv3 jsou přidání podpory pro nové šifrovací sady (cipher suites), odebrání podpory pro šifrovací sady, které již nejsou bezpečné, přidání nových TLS rozšíření nebo přidání využití pseudonáhodných funkcí do dalších částí protokolu [16].

2.2 Popis SSL/TLS

Hlavním cílem SSL/TLS protokolu je bezpečnost, která zahrnuje důvěrnost (zachování tajemství), pravost (ověření totožnosti) a integrita (zajištění bezpečného přenosu). Mezi stanovené cíle SSL/TLS mimo jiné patří [16]:

Kryptografická bezpečnost
: Úlohou tohoto cíle je umožnit bezpečnou komunikaci mezi dvěma stranami. Mezi nimi však může být strana třetí, útočník, který celou komunikaci odposlouchává. Pro něj by se proto měla komunikace jevit jako řada nicneříkajících zpráv a ani by neměl být schopen znovu použít odposlechnuté zprávy.

Interoperabilita
: Programátoři by měli být schopni nezávisle na sobě vyvíjet programy a knihovny, které spolu budou komunikovat pomocí společných kryptografických parametrů.

Rozšířitelnost
: Možnost jednoduchého rozšíření protokolu bez potřeby měnit aktuálně použité kryptografické metody.

Efektivita
: Vhodný poměr mezi náklady na výkon a funkčností celého protokolu.

Využití SSL/TLS protokolu tedy nevyžaduje znalost jeho implementace (interoperabilita). Pokud je nalezen nedostatek v protokolu, pak jeho oprava či rozšíření neznamená změnu základů (rozšiřitelnost). A náklady na výkon jsou sníženy pomocí ukládání relací do cache, díky čemuž není potřeba při každé jednotlivé komunikaci vytvářet nové parametry pro zabezpečení (efektivita).

V protokolu SSL/TLS je používána symetrická i asymetrická kryptografie [4]:

- **Symetrická kryptografie** — kryptografické algoritmy, ve kterých se pro šifrování a dešifrování používá stejný *klíč*. S pomocí této kryptografie jsou šifrovány např. aplikační data v SSL/TLS.
- **Asymetrická kryptografie** — kryptografické algoritmy, ve kterých se pro šifrování a dešifrování používají odlišné klíče. Pro šifrování je použit *veřejný klíč* a pro dešifrování *privátní klíč*. Pomocí této kryptografie je vyměněn *klíč*, který je využit v SSL/TLS protokolu pro symetrickou kryptografii.

Protokol je rozdělen do čtyř podprotokolů [16]:

- **Handshake protokol** — tímto protokolem jsou mezi stranami dohodnuty parametry pro připojení a je ověřena autenticita vystupujících stran,
- **Change Cipher Spec protokol** — definuje zprávu, kterou se strany navzájem informují o tom, že začínají komunikovat šifrovaně,

- **Application data protokol** — umožňuje zpracování a odesílání zpráv aplikací,
- **Alert protokol** — zajišťuje jednoduchý oznamovací mechanismus pro oznámení výjimečných okolností druhé straně.

2.2.1 Handshake protokol [16]

	bity 0-7	8-15	16-23	24-31
0	22	Verze (MSB)	Verze (LSB)	Délka (MSB)
32	Délka (LSB)	Typ zprávy	Délka zprávy	
64	Délka zprávy	Zpráva		
...	Zpráva	Typ zprávy	Délka zprávy	
...	Délka zprávy	Zpráva		

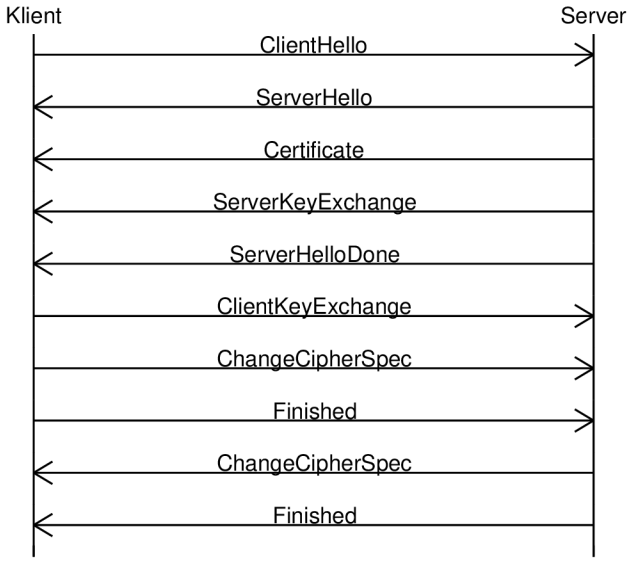
Obrázek 2.1: Záznam Handshake protokolu

Velmi důležitou částí SSL/TLS je tzv. handshake, při kterém jsou dohodnuty parametry pro komunikaci a je ověřena pravost komunikujících stran. Specifikace záznamu protokolu je zobrazena na obrázku 2.1. První byte záznamu obsahuje hodnotu 22, čímž určuje, že se jedná o handshake záznam. Dále je zde uvedena verze protokolu a celková délka záznamu. Za délkou záznamu je typ zprávy — např. *ClientHello* nebo *ServerHello*. Na bitech 48 až 71 je informace o délce první zprávy v tomto záznamu. Hned za ní následuje samotná zpráva, která má celkovou délku 32 bitů. Za touto zprávou může následovat další zpráva, díky čemuž můžeme v jednom záznamu řetězit více zpráv, např. *ServerKeyExchange* a *ServerHelloDone* [27].

Existují tři druhy handshaků, které se liší toky zpráv vyměňovaných mezi klientem a serverem:

1. Kompletní handshake s ověřením pravosti pouze serveru,
2. Zkrácený handshake pro obnovu předešlé komunikace (session resumption),
3. Handshake s ověřením pravosti klienta a serveru.

Na obrázku 2.2 je zobrazena posloupnost zpráv *kompletního handshaku*. V úvodu je uvítání od klienta, ve kterém klient informuje, jakou verzi protokolu, jaké šifrovací sady a metody pro kompresi podporuje. V rozšíření může sdělit např. jaké algoritmy pro elektronický podpis podporuje nebo o možnosti budoucí renegociaci. Server vybere nejvhodnější parametry, které mu byly nabídnuty od klienta, a ve zprávě *ServerHello* sdělí zvolené parametry. V případě, že server nepodporuje žádnou z nabídnutých verzí protokolu, nepodporuje žádnou nabídnutou šifrovací sadu nebo nezná metodu komprese, pak je spojení ukončeno. Pokud ale najde shodu, informuje klienta a zašle certifikát odpovídající dohodnutým parametrům, pokud nebyla zvolena anonymní šifrovací sada, při které se server neprokazuje certifikátem. V další zprávě *ServerKeyExchange*, která není povinná, jsou zaslány dodatečné



Obrázek 2.2: Kompletní handshake s ověřením pravosti serveru (bez parametrů)

informace potřebné pro výměnu klíčů. Zprávou *ServerHelloDone* server klientovi oznamuje, že již zaslal všechny informace a nyní čeká na klienta. Klient, ve chvíli, kdy obdrží tuto zprávu, zašle zprávu *ClientKeyExchange*, ve které je klientův podíl potřebný pro tvorbu klíčů. Obsah této zprávy záleží na zvolené šifrovací sadě. Tímto má klient i server všechny potřebné parametry pro začátek šifrované komunikace. Podprotokolem *Change Cipher Spec* jsou strany navzájem informovány, že začínají komunikovat šifrovaně. O dokončení jsou obě strany informovány zprávou *Finished*, která už je šifrována, čímž si obě strany ověří, že handshake proběhl v pořádku.

Klient i server si uchovávají bezpečnostní parametry připojení po určitou dobu. Pokud se chce klient připojit k serveru a oba si pamatují parametry z jejich spojení, pak proběhne tzv. *zkrácený handshake*. Úvod tohoto handshaku je podobny jako u ostatních, tedy je posláno *ClientHello* a *ServerHello*, ve kterých je uvedeno ID relace (session ID), čímž se strany informují o obnovení dřívějšího spojení. Server vygeneruje novou sadu klíčů pomocí *master secret*, což je hodnota, ze které se s pomocí šifrovací sady generují klíče sloužící pro šifrování a dešifrování komunikace. *Master secret* je vytvořen při *kompletním handshaku*. Dále jsou odeslány zprávy *ChangeCipherSpec* a *Finished* od obou stran, čímž začne být komunikace šifrována a handshake proběhl v pořádku.

Handshake s ověřením pravosti klienta a serveru je podobný kompletnímu handshaku. Začátek je stejný, ale u tohoto handshaku server po zprávě *ServerKeyExchange* zašle *CertificateRequest*, čímž požádá klienta, aby se prokázal platným certifikátem. Po *ServerHelloDone* je první odpovědí od klienta certifikát. Za ním následuje *ClientKeyExchange* a *CertificateVerify*. Zprávou *CertificateVerify* klient prokáže své vlastnictví privátního klíče, který souvisí s veřejným klíčem, který již zaslal v certifikátu. Toto prokáže tím, že ve zprávě zašle podpis, který je vytvořen s pomocí podpisů všech zpráv handshaku, které byly do této chvíle zaslány. Po tomto ověření je handshake oběma stranami ukončen a potvrzen zprávami *ChangeCipherSpec* a *Finished*.

2.2.2 Change Cipher Spec protokol [16]

	bity 0-7	8-15	16-23	24-31
0	20	Verze (MSB)	Verze (LSB)	0
32	1	1		

Obrázek 2.3: Záznam Change Ciper Spec protokolu

Change Cipher Spec protokol definuje zprávu, kterou se strany při handshaku informují o tom, že mají dostatek informací na vytvoření bezpečnostních parametrů připojení, vygenerovali klíče a začínají komunikovat šifrovaně. Na obrázku 2.3 je uvedena specifikace záznamu tohoto podprotokolu. V prvním bytu je definován protokol *Change Ciper Spec* hodnotou 20. Dále je zde uvedena verze a délka, která je 1, protože v tomto záznamu je pouze na bitech 40 až 47 určen *Change Cipher Spec* typ protokolu [27]. V návrhu TLS 1.3 je *Change Cipher Spec protokol* odstraněn, protože byl zařazen mezi vlastnosti, které nejsou bezpečné nebo jsou již zastaralé.

2.2.3 Application data protokol

	bity 0-7	8-15	16-23	24-31
0	23	Verze (MSB)	Verze (LSB)	Délka (MSB)
32	Délka (LSB)	Aplikační data		
...	Aplikační data			

Obrázek 2.4: Záznam Application data protokolu

Tento protokol obstarává zasílání zpráv aplikací. Podle definice jsou zde data před odesláním roztříděny, zmenšeny pomocí komprese dat a poté zašifrovány. V dnešní době se ale komprese dat v SSL/TLS protokolu nepoužívá, protože byly objeveny útoky jako např. CRIME a TIME [17], díky kterým lze komprese dat zneužít. Na obrázku 2.4 je zobrazena specifikace záznamu tohoto podprotokolu. První byte obsahuje hodnotu 23, která určuje, že se jedná o záznam typu *Aplication data protokolu*, dále je zde určena verze protokolu a délka záznamu. Za délkou záznamu jsou aplikační data [27], jejichž velikost je omezena pouze maximální velikostí záznamu, která je u TLS 16 kB [2].

2.2.4 Alert protokol

Alert protokol definuje upozornění, které se využívají jako oznamovací mechanismus, pokud nastala nečekaná událost, o které musí být informována druhá strana. Specifikace záznamu protokolu je zobrazena na obrázku 2.5. Prvním bytem je určen typ záznamu. Tento protokol je identifikován hodnotou 21. Dále je zde verze a délka záznamu bez hlavičky, která má

	bity 0-7	8-15	16-23	24-31
0	21	Verze (MSB)	Verze (LSB)	0
32	2	Závažnost	Popis	

Obrázek 2.5: Záznam Alert protokolu

hodnotu 2. Za velikostí je prvním bytem určena závažnost upozornění a na dalším bytu je popis upozornění [27].

Všechna upozornění jsou vytvořeny při chybě, kromě `close_notify`, který je použit pro ukončení spojení. Závažnost může být *Warning* nebo *Fatal*.

V případě, že strana obdrží upozornění závažnosti *Warning*, pak standard nevyžaduje konkrétní chování. Komunikace tedy může dále probíhat. V případě, že strana, která obdržela tento typ upozornění, ho vyhodnotí jako závažný, zašle druhé straně *Fatal alert*. Mezi *Warning alerty* patří např. `certificate_unknown` (vznikly nespecifikované potíže při práci s certifikátem), `certificate_expired` (certifikát je prošlý nebo není v současné době platný) a `unsupported_certificate` (certifikát je nepodporovaného typu) [8].

Pokud jedna ze stran obdrží *Fatal alert*, spojení se ukončí a zruší se i platnost celého spojení, což znamená, že nelze toto spojení obnovit pomoci *ID relace* a *zkráceného handshaku*. Mezi tyto alerty patří např. `unexpected_message` (byla obdržena neočekávaná zpráva), `illegal_parameter` (parametr v handshaku je nekonzistentní s ostatními) `unsupported_extension` (odesláno klientem, pokud obdrží ve zprávě a *ServerHello* rozšíření, o které nežádal v *ClientHello*) [8].

Upozorněním `close_notify` oznamuje jedna strana druhé, že ukončuje spojení. Jako jediné upozornění není vytvořeno při chybě. Po obdržení tohoto upozornění jsou všechny další doručené zprávy ignorovány [8].

2.2.5 Rozšíření TLS [16]

Důležitou součástí TLS od verze 1.1 jsou TLS rozšíření umožňující protokolu rozšířit funkcionalitu bez změny celého protokolu. Mezi nejpoužívanější rozšíření patří:

- **Označení jména serveru (Server Name Indication)** — TLS nenabízí mechanismus, jak by mohl klient při navazování komunikace sdělit konkrétní jméno serveru, který kontaktuje, což je žádoucí pokud komunikujeme s virtuálními webovými servery, tj. více doménových jmen umístěných na jedné IP adrese. Tímto rozšířením lze specifikovat konkrétní jméno serveru, na který se chceme připojit.
- **OCSP[1] stapling** — rozšíření, také známé jako *TLS žádost o stav certifikátu*, umožňuje, aby server odeslal klientovi stav certifikátu. Tento stav je vystaven s časovou značkou a podepsán certifikační autoritou, tudíž klient nemusí kontaktovat certifikační autoritu pro ověření platnosti certifikátu.
- **Algoritmy pro digitální podpisy** — tímto rozšířením může klient sdělit, jaké algoritmy pro digitální podpis a jaké hašovací funkce podporuje.

[1] Online Certificate Status Protocol

- **Heartbeat** — rozšíření, pomocí kterého lze přidat podporu pro funkcionalitu, která umožňuje jedné straně zjišťovat, jestli je druhá strana stále dostupná, a pro zjištění maximální přenosové jednotky (MTU). Toto rozšíření je zaměřeno na DTLS (Datagram Transport Layer Security), což je TLS nad nespolehlivými protokoly, jako je UDP. Heartbeat je ale podporováno i u TLS. Heartbeat rozšíření má 2 módy — lze přijímat Heartbeat požadavky nebo ne. Klient svůj Heartbeat mód specifikuje v ClientHello a server v ServerHello. S Heartbeat rozšířením souvisí Heartbeat protokol, který definuje zprávy, které se využívají pro výše zmíněné účely. Tyto zprávy obsahují typ zprávy (požadavek nebo odpověď na požadavek), velikost tzv. payloadu, payload (data, odeslané v požadavku a očekávané v odpovědi, pomocí nichž se pozná, která odpověď patří ke kterému požadavku) a vycpávka (padding, což jsou náhodné data délky minimálně 16 B, které musí protistrana ignorovat).

- **Encrypt-Then-MAC** — tímto rozšířením lze nahradit výchozí ověřené šifrování *MAC-Then-Encrypt* bezpečnějším ověřeným šifrováním *Encrypt-Then-MAC*. Při výchozím MAC-Then-Encrypt je první z nezašifrovaných dat vytvořen s pomocí hašovací funkce MAC (Message authentication code), pomocí kterého si protější strana ověřuje zprávu, zda-li nebyla pozměněna. Po výpočtu MAC jsou spojeny nezašifrované data a MAC a zašifrovány jako celek. Rozšíření Encrypt-Then-MAC tento proces upravuje — první je nezašifrovaný text zašifrován, ze vzniklého zašifrovaného textu vytvořen MAC a nakonec jsou tyto dvě části spojeny a odeslány jako celek [10].

- **Vyjednání protokolu aplikační vrstvy** — umožňuje vybrat jiný protokol aplikační vrstvy pro TLS spojení. Např. server na portu 443 ve výchozím nastavení používá HTTP 1.1, ale pomocí tohoto rozšíření lze vybrat HTTP 2.0.

- **Tiket relace (session ticket)** — pomocí tohoto rozšíření může být obnovena předešlá komunikace ze strany klienta. Server data spojená s relací zašifruje a pošle je klientovi ve formě tiketu. Pomocí tohoto tiketu může klient obnovit předešlou komunikaci tak, že ho serveru předloží při handshaku. Server po kontrole a dešifrování využije informace z tiketu, aby obnovil relaci. Díky tomu nemusí být informace o relacích uloženy na straně serveru.

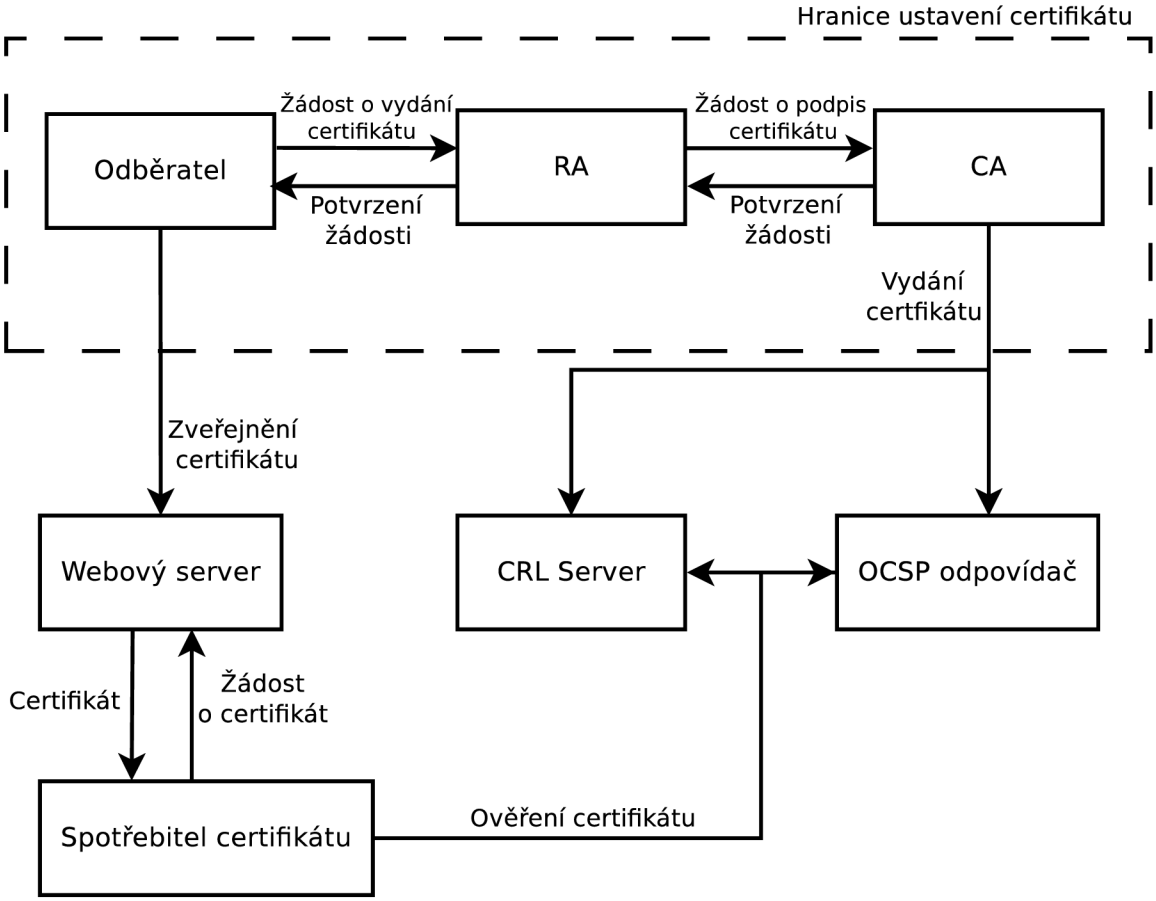
2.2.6 Infrastruktura veřejných klíčů [16]

Infrastruktura veřejných klíčů, neboli PKI (Public-Key Infrastructure), umožňuje zabezpečenou komunikace mezi dvěma stranami, které se nikdy nesetkaly. Pro tuto možnost existují důvěryhodné třetí strany, které jsou nazývány jako certifikační autority, které vystavují certifikáty, kterým důvěřujeme. Certifikát je digitální dokument, který obsahuje veřejný klíč, informace o subjektu, kterému certifikát patří, a digitální podpis od certifikační autority.

Na obrázku 2.6 je znázorněn životní cyklus certifikátu. Je zde znázorněno několik účastníků, kteří se podílí na žádání, vystavování nebo používání certifikátu.

Odběratel
: Strana, která si přeje poskytovat své služby pomocí zabezpečení, které vyžaduje platný certifikát.

CA (Certifikační autorita)
: Certifikační autorita vystavuje certifikáty a zveřejňuje informace o certifikátech, jako je jejich validita či aktuálnost.



Obrázek 2.6: Životní cyklus certifikátu [16]

RA (Registrační autorita)
Registrační autorita, která vyřizuje vydávání certifikátu a ověřuje identitu žadatele o certifikát. V mnoha případech se může jednat o certifikační autoritu.

Webový server
Server, pro který je certifikát vystaven.

CRL[2] server
Server, který poskytuje seznam zrušených certifikátů. Entitám, které se prezentují certifikáty, které jsou na tomto seznamu, by nemělo být důvěřováno.

OCSP odpovídač (OCSP responder)
Alternativa k CRL serveru. Poskytuje méně informací než CRL server, tudíž menší zátěž pro síť a pro klienta.

Spotřebitel certifikátu
Strana, která si přeje připojit se zabezpečeně k serveru a ověřit si platnost certifikátu. Jedná se většinou o webový prohlížeč, program nebo operační systém, který provádí ověření certifikátu.

[2]Certificate Revocation List

2.3 Implementace SSL/TLS

Knihoven, které implementují SSL/TLS protokol existuje několik. Většina z nich je napsána v jazyce C a jsou open source. Existuje pár výjimek, které se tímto vyznačují od ostatních, např. Bouncy Castle[3] je implementace SSL/TLS napsána v jazycích Java a C# a SChannel[4], která je vyvíjena firmou Microsoft, není open source softwarem [20]. V této kapitole jsou popsáni někteří zástupci implementací.

2.3.1 OpenSSL

OpenSSL[5] je projekt s otevřeným zdrojovým kódem. Implementuje SSL, TLS a DTLS protokoly. Je založeno na SSLeay, což je open source implementace SSL protokolu, jejíž vývoj neoficiálně skončil v roce 1998. Vývojový tým OpenSSL se aktuálně skládá z 11 lidí, kdy pouze jeden z nich je zaměstnanec na plný úvazek, ostatní jsou dobrovolníci. Rozpočet tohoto projektu je necelý 1 milion dolarů ročně a musí se proto částečně spoléhat na dary z nadací [25].

Hlavní knihovna tohoto projektu, která je napsána v jazyce C a jazyku symbolických adres, implementuje základní kryptografické funkce. OpenSSL však lze pomocí dedikovaných rozhraní použít i v jiných programovacích jazycích. Kromě knihovny je OpenSSL i nástroj pro příkazovou řádku, pomocí kterého lze vytvořit certifikáty standardu X.509, CSR[6] (žádost o vystavení SSL certifikátu), šifrovat a dešifrovat s pomocí vybrané šifry, spustit testovacího SSL/TLS klienta nebo server apod [9].

OpenSSL podporuje SSLv3, TLS 1.0, TLS 1.1, TLS 1.2, DTLS 1.0 a DTLS 1.2. Dále podporuje velké množství různých kryptografických algoritmů. Mezi známější algoritmy, které OpenSSL podporuje patří [25]:

- **Šifry** — DES (Data Encryption Standard), RC2 (Ron's Code 2), RC4, RC5, Triple DES, AES (Advanced Encryption Standard)
- **Kryptografické hašovací funkce** — MD2 (Message-Digest 2), MD4, MD5, SHA-1 (Secure Hash Algorithm 1), SHA-2, SHA-224, SHA-256, SHA-384, SHA-512,
- **Kryptografie s veřejnými klíči** — RSA, DSA (Digital Signature Algorithm), DH (Diffie-Hellman) key exchange, EC (Elliptic curve).

OpenSSL je využíváno velkou částí serverů, jelikož jej využívá Apache HTTP Server[7]. Dále jej využívá např. Red Hat ve svých operačních systémech Red Hat Enterprise Linux[8].

2.3.2 GnuTLS

GnuTLS[9] je implementace SSL, TLS a DTLS protokolů s otevřeným zdrojovým kódem. Původně bylo vytvářeno jako součást projektu GNU[10], ale v roce 2012 po neshodách s Free

[3] https://www.bouncycastle.org/
[4] https://msdn.microsoft.com/en-us/library/windows/desktop/ms678421
[5] https://www.openssl.org/
[6] Certificate signing request
[7] https://httpd.apache.org/
[8] https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux
[9] http://www.gnutls.org/
[10] https://www.gnu.org/

Software Foundation[11] oznámil hlavní vývojář, že další vývoj bude probíhat mimo projekt GNU [22].

GnuTLS je napsáno v jazyce C. Nabízí aplikační uživatelské rozhraní, aby jej mohly aplikace využívat pro zabezpečené spojení, přístup k certifikátům standardu X.509, PKCS #12 (souborový formát pro archivaci několika kryptografických objektů v jednom souboru), OpenPGP (standard pro šifrování emailů) apod. Obdobně jako OpenSSL, tak i GnuTLS nabízí nástroj pro příkazovou řádku, s pomocí kterého lze pracovat s certifikáty X.509 standardu, spustit testovací SSL/TLS klienta a server nebo generovat náhodné klíče a hesla.

Podporuje SSLv3, TLS 1.0, TLS 1.1, TLS 1.2, DTLS 1.0 a DTLS 1.2. Nabízí CPU asistovanou kryptografii a podporu pro kryptografický akcelerátor */dev/crypto*, který poskytuje přístup k hardwarovým ovladačům přes rozhraní *ioctl*, což je systémové volání specifické pro operace, kterých nelze dosáhnout pomocí obvyklých systémových volání [24]. Dále poskytuje např. podporu pro zabezpečení tzv. *smart cards* (čipová karta kapesní velikosti, např. SIM karty do mobilních telefonů nebo kreditní karty) [26]. GnuTLS je používáno např. v GNOME, Wireshark, Lynx nebo Emacs [22].

2.3.3 NSS [23]

NSS[12], celým názvem Network Security Services, je implementace SSL, TLS, DTLS a S/-MIME (standard pro podepisování a šifrování pomocí veřejných klíčů) s otevřeným zdrojovým kódem. Původně patřila pod tři licence — Mozilla Public Licence 1.1, GNU General Public License a GNU Lesser General Public License. Dnes je dostupná pod licencí Mozilla Public Licence 2.0.

Network Security Services je napsána v jazyce C a jazyku symbolických adres. Je to sada knihoven navržená pro podporu multiplatformního vývoje aplikací, které mají mít možnost zabezpečené komunikace. Obdobně jako GnuTLS, tak i NSS podporuje kryptografický akcelerátor pro servery a zabezpečení pro *smart cards* pro klienty.

NSS podporuje kromě obvyklých SSLv3, TLS 1.0, TLS 1.1, TLS 1.2, DTLS 1.0 a DTLS 1.2 i TLS 1.3, který je v tuto chvíli ve fázi návrhu. Proto v každé nové verzi NSS bývají přidávány další vlastnosti TLS 1.3, které s každým novým návrhem přibývají. NSS je používání v Mozilla produktech, což je Firefox, Thunderbird, SeaMonkey a Firefox pro mobilní telefony. Dále je používán ve webovém prohlížeči Opera, komunikačním nástroji Pidgin, Sun Java Web Server apod. Dříve byl také používám ve webových prohlížečích od firmy Google — Google Chrome a Chromium, ale ty už dnes používají vlastní implementaci SSL/TLS protokolu — BoringSSL[13].

2.3.4 Tlslite-ng

Tlslite-ng[14] je knihovna s otevřeným zdrojovým kódem, která implementuje SSL a TLS kryptografické protokoly. Autorem této knihovny je Hubert Kario z firmy Red Hat. Tlslite-ng je větev z TLS Lite, jejímž hlavním autorem byl Trevor Perrin, a je distribuována pod licencí GNU LGPLv2.

Tlslite-ng je napsána v čistém Pythonu. Lze ji použít pomocí dedikovaných rozhraní nebo jako backend pro ostatní knihovny. Tlslite-ng lze využít s Python knihovnami, které slouží jako akcelerátory kryptografických operací — m2crypto, pycrypto a gmpy.

[11] http://www.fsf.org/
[12] https://nss-crypto.org/
[13] https://boringssl.googlesource.com/boringssl/
[14] https://github.com/tomato42/tlslite-ng

Tato knihovna podporuje SSLv3, TLS 1.0, TLS 1.1 a TLS 1.2, většinu šifrovacích sad a rozšíření, ale oproti ostatním zmíněným implementacím SSL/TLS protokolu zde chybí např. šifrování Camellia nebo Kerberos, což je síťový autentizační protokol umožňující komunikujícímu v nezabezpečené síti prokázat bezpečně svou identitu někomu dalšímu. Tlslite-ng knihovna je využita pro fuzz testování SSL/TLS v tlsfuzzeru [14].

Kapitola 3

Fuzz testování

Fuzz testování, nebo také fuzzing, je technika testování softwaru. Často se jedná o automatické nebo alespoň částečně automatické testování, které testuje programy pomocí velkého množství chybných, nečekaných nebo náhodných dat na vstupu, která jsou nazývána fuzz. Fuzz testování je běžně používáno pro testování bezpečnostních problémů u programů. Takto testován může být libovolný vstup programu. Mezi zajímavé vstupy fuzzování patří formáty souborů, proměnná prostředí, pořadí zmáčknutých tlačítek v programu nebo vstupy z klávesnice a myši [21].

Samotné testování softwaru můžeme rozdělit do tří kategorií podle pohledu, jaký má tester při vytváření testů [28]:

1. **White-box testování** — při vytváření testů má tester přístup ke zdrojovému kódu a na jeho základě vytváří testy. Vidí i reakce systému, díky čemuž ztrácí pohled uživatele, ale může lépe odhadnout, kde chyby hledat.

2. **Black-box testování** — zaměřuje se, jak se software chová navenek. U tohoto testování tester vidí pouze vstupy a výstupy.

3. **Grey-box testování** — kombinace white-box a black-box testování, kdy např. tester nemá k dispozici celý zdrojový kód, ale ví o matematických principech, které jsou použity v aplikaci.

Fuzz testování nelze přímo zařadit do jedné z výše uvedených kategorií, ale velmi často je využíváno jako black-box testování, kdy na vstup aplikace posíláme náhodná data a sledujeme, jak se s nimi aplikace nakládá.

Pro fuzz testování se využívají nástroje, tzv. fuzzery. Jedním ze známých open source fuzzerů je **radamsa**[1], který upravuje vstup tak, aby vypadal jako poškozený nebo zlomyslný a vynutit takto chybu nebo pád programu.

```
$ echo "Hello World!" | radamsa
Hell"o World!
```

Kód 3.1: Příklad použití a výstupu fuzzeru ramdsa

V kódu 3.1 je uveden příklad použití fuzzeru radamsa na textový řetězec. Textový řetězec je pomocí unixové roury vložen na vstup fuzzeru a na výstupu je vytisknut upravený textový řetězec. V této ukázce jsou do textového řetězce vloženy uvozovky, což by mohlo v programu způsobit konec čtení řetězce. Mezi jiné úpravy, které radamsa provádí patří např. smazání znaku, opakování některého znaku z řetězce nebo smazání celého řetězce.

[1] https://github.com/aoh/radamsa

3.1 Historie fuzz testování

Jeden z prvních zdokumentovaných pokusů o fuzz testování byl v roce 1983 *The Monkey* — příslušenství, které vytvářelo náhodné události ve spuštěné aplikaci, která byla zapnuta na Macintoshi. Toto příslušenství simulovalo náhodné klikání na myší a na klávesnici a generovalo tahy myši na náhodné pozice. Později bylo toto zařízení rozšířeno o možnost otevření menu a přesuny oken. Ze začátku jednoduše systém spadl, ale po opravách chyb, který byly tímto způsobem objeveny, byl systém schopen běžet přes 20 minut. Po této době se většinou zařízení dostalo k příkazu vypnutí počítače a vypnulo jej [11].

V praxi se však testování bezpečnosti a spolehlivosti softwaru do devadesátých let neaplikovalo, protože pády softwaru byly přijatelné a jejich aktualizace jednoduchá. Prvním skutečně testovacím nástrojem, který na vstup softwaru posílal chybná nebo neočekávaná data, byl nástroj zvaný *Fuzz*, který byl zveřejněn roku 1990 výzkumnou skupinou Bartona Millera z University of Wisconsin. Tímto nástrojem se inspirovala OUSPG (Oulu University Secure Programming Group), která v roce 1999 zahájila projekt PROTOS. Komunita kolem projektu PROTOS přišla s nápadem tvorby fuzzing testovací sady pro různé rozhraní. Tuto testovací sadu by první zveřejnili pro dodavatele softwaru a po opravě chyb by ji zpřístupnili i pro veřejnost. Během následujících let bylo s pomocí projektu PROTOS vytvořeno několik volných testovacích sad pro protokoly WAP-WSP (WSP je standard pro udržování relací a WAP je WWW relace od připojení po odpojení uživatele z konkrétní URL), WMLC (formát pro vysílání rádiových stanic), HTTP-reply (odpověď na HTTP požadavek), LDAP (protokol pro ukládání a přístup k datům na adresářovém serveru), SNMP (slouží pro potřeby správy sítí), SIP (protokol určený pro přenos signalizace v internetové telefonii), H.323 (definuje protokoly pro audio-vizuální relace komunikace), ISAKMP/IKE (IKE je protokol používaný pro nastavení sdílených bezpečnostních atributů mezi dvěma entitami používaný v IPsec a ISAKMP je součást IKE, která specifikuje mechanismus výměny klíčů) a DNS (hierarchický systém doménových jmen). Projekt PROTOS pomohl najít několik závažných bezpečnostních chyb (velká část byla v implementacích SNMP), čímž získal úspěch, kterým upozornil bezpečnostní komunitu na tuto "novou"techniku testování zvanou fuzzing [6].

V roce 2012 firma Google oznámila ClusterFuzz[2], infrastrukturu pro fuzz testování na cloud základu, pro webový prohlížeč Chromium. Pomocí tohoto nástroje mohou bezpečnostní výzkumníci nahrávat své fuzzery a sbírat odměny za chyby, které najde ClusterFuzz s pomocí jejich fuzzerů. V roce 2013 byl zveřejněn afl[3] (american fuzzy lop) od Michała Zalewského. Tento fuzzer využívá tzv. evoluční algoritmy, které používají techniky napodobující evoluční procesy z biologie, aby efektivně zvýšil pokrytí kódu v testech. Tento fuzzer s otevřeným zdrojovým kódem pomohl najít několik významných chyb v projektech jako např. OpenSSL (např. CVE-2015-1789[4]), Firefox (např. CVE-2014-1564[5]), PHP (např. CVE-2015-0232[6]) nebo Qt (např. QTBUG-43513[7]) [19]. V prosinci 2016 firma Google zveřejnila OSS-Fuzz[8], který umožňuje fuzz testování několika bezpečnostně kritických open-source projektů [21]. OSS-Fuzz již našel několik chyb v SSL/TLS implementaci GnuTLS, např. CVE-2017-5334[9] (dekódování speciálně vytvořeného certifikátu standardu X.509 může

[2] https://github.com/google/clusterfuzz-tools
[3] http://lcamtuf.coredump.cx/afl/
[4] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1789
[5] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1564
[6] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0232
[7] https://bugreports.qt.io/browse/QTBUG-43513
[8] https://github.com/google/oss-fuzz
[9] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5334

vést ke dvojitému uvolnění paměti) a CVE-2017-5335[10] (pomocí speciálně vytvořeného OpenPGP certifikátu může útočník způsobit přetečení zásobníku a haldy, což vede k DoS útoku).

3.2 Typy fuzzerů

Fuzzery můžou být děleny na základě dvou kriterií:

1. Vektor útoku
2. Složitost testovací sady

Vektor útoku určuje, na co se fuzzer zaměřuje. Fuzzer může testovat stranu klienta nebo serveru. Při testování strany klienta u HTTP nebo SSL/TLS protokolu je testování cíleno na webový prohlížeč. Na straně serveru je testován webový server.

Rozdělení fuzzerů podle složitosti testovací sady se zaměřuje na rozdílné vrstvy testovaného softwaru. Jednoduchým testováním může být testování vstupů aplikace, např. když místo číselné hodnoty zadáme nečíselný znak. Nebo mohou být upraveny struktury zpráv, pomocí kterých aplikace komunikuje, což může způsobit chybu při parsování této zprávy. Složitější fuzzery pak upravují i pořadí jednotlivých zpráv [6].

3.3 Fuzz testování síťových protokolů

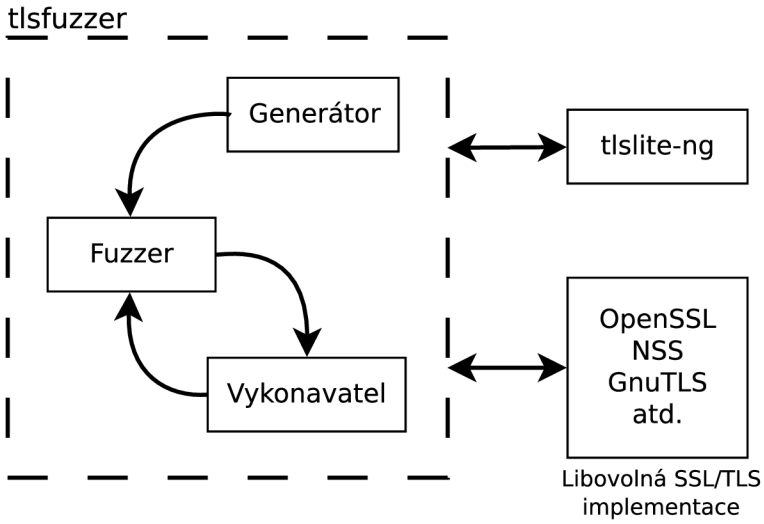
Fuzz testování síťových protokolů je složitější, protože fuzzery, které je testují, je složité udržovat, aktualizovat a po odeslání jedné zprávy vyžadují zahájení nového spojení. Proto se většina fuzzerů zaměřuje primárně na bezstavové protokoly (např. *HTTP*) nebo na protokoly, které mají málo stavů (např. *SMTP*) [14].

Použití čistého fuzz testování na SSL/TLS protokol je složité, protože u SSL/TLS může být platných vstupů několik a ty jsou závislé na předchozích datech. Po handshaku jsou navíc data zašifrovány a obsahují kontrolní součet sloužící pro ověření, zda je informace úplná. Ve fuzzeru pro SSL/TLS protokol tedy potřebujeme i implementaci samotného protokolu [12].

3.4 Tlsfuzzer

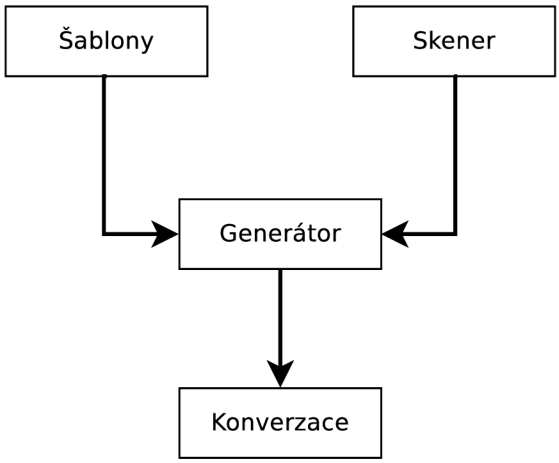
Tlsfuzzer[11] je fuzzer a testovací sada pro implementace SSL/TLS protokolu. Autorem tohoto nástroje je Hubert Kario z firmy Red Hat. Tento fuzzer je napsán v jazyce Python a využívá knihovny tlslite-ng (viz. 2.3.4). Tento fuzzer nedělá náhodné změny v odesílaných paketech, ale zaměřuje se na jednotlivé zprávy a jejich obsah [14].

[10] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5335
[11] https://github.com/tomato42/tlsfuzzer



Obrázek 3.1: Architektura tlsfuzzeru [13]

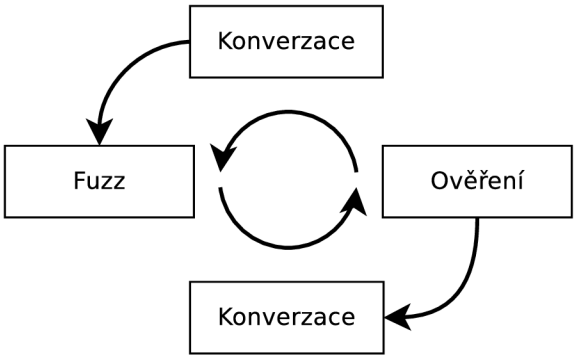
Na obrázku 3.1 je znázorněna plánovaná architektura tlsfuzzeru. Tlsfuzzer spolupracuje s knihovnou tlslite-ng, která obstarává komunikaci, a jinou libovolnou SSL/TLS implementací, kterou chceme testovat.



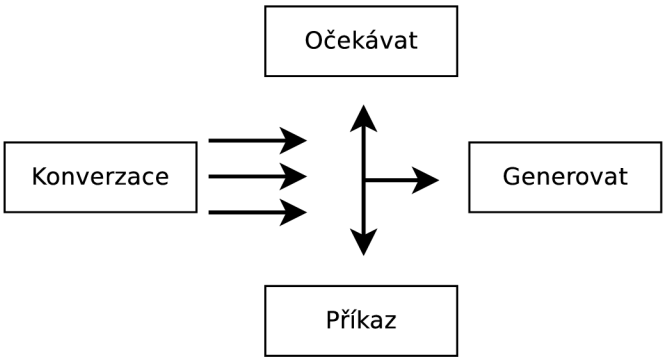
Obrázek 3.2: Architektura generátou [13]

Celý běh tlsfuzzeru začíná v *generátoru*, který je znázorněn na obrázku 3.2. Na jeho vstupu jsou šablony pro připojení, ve kterých je uvedena např. IP adresa protější strany a port, na který se má tlsfuzzer připojit, a skener, který zjistí informace o protější straně, např. jaké verze SSL/TLS nebo jaká rozšíření jsou podporována. Z těchto informací vytvoří zprávy a očekávané odpovědi na tyto zprávy — konverzaci, které je vstupem do *fuzzeru* znázorněného na obrázku 3.3. Fuzzer tyto zprávy náhodně upravuje a ověřuje, zda-li na danou upravenou zprávu má stále očekávat stejnou odpověď nebo jestli po daných úpravách je nutno změnit i očekávanou odpověď. Po těchto úpravách jsou zprávy vloženy do konverzace. Tato konverzace, rozšířena o změněné zprávy, je vstupem *vykonavatele*, který je na obrázku 3.4. Vykonavatel prochází konverzaci a rozřazuje jednotlivé části do kategorií. První kategorie je *Očekávat*, kdy očekáváme zprávu od druhé strany, např. odpověď na naši žádost

nebo očekávané upozornění o chybě. Další kategorií je *Generovat*, kdy ze strany tlsfuzzeru je vygenerována a odeslána zpráva, např. požadavek o renegociaci nebo o ukončení spojení. Poslední kategorií je *Příkaz*, který slouží pro práci s tlsfuzzerem, např. připojení k protější straně před zahájením handshaku nebo určení maximální velikosti záznamu (je-li velikost zprávy větší, pak je rozdělena na více částí).



Obrázek 3.3: Architektura fuzzeru [13]



Obrázek 3.4: Architektura vykonavatele [13]

Tento běh tlsfuzzeru je zatím pouze v plánu. Z jednotlivých částí funguje pouze *Vykonavatel*, pro který lze vytvářet a spouštět testy, které jsou závislé na možnostech tlsfuzzeru a knihovny tlslite-ng [13].

Kapitola 4

Útoky na SSL/TLS protokol

Útoků zaznamenaných na tento protokol existuje mnoho. Tato kapitola je stručným výčtem útoků. Vybrané a reprodukované útoky jsou detailněji popsány v kapitole 5. Mezi útoky patří jak přerušení komunikace útočníkem, tak i získání privátního klíče, díky kterému může útočník dešifrovat nebo zahájit komunikaci.

Útoky můžeme rozřadit do čtyř kategorií — útoky na PKI, útoky využívající chyb ve webovém prohlížeči a HTTP protokolu, útoky využívající chyb v implementaci protokolu a útoky na protokol. Útok, který nelze jednoduše zařadit do žádné z uvedených kategorií, protože je mnoha útoky využíván, je tzv. Man-in-the-Middle útok (MitM) [16].

Man-in-the-Middle útok Během komunikace dvou stran, serveru a klienta, může v cestě stát strana třetí, útočník — muž uprostřed. V případě, že útočník probíhající komunikaci pouze odposlouchává, ale nijak do ní aktivně nezasahuje, pak mluvíme o *pasivním útoku*. Tento útok je nejúčinnější při odposlouchávání nezašifrované komunikace. Pokud útočník komunikaci upravuje nebo do ní jakkoliv jinak zasahuje, pak mluvíme o *aktivním útoku*. Při tomto útoku se většinou útočník snaží oklamat klienta, že on je server, a dohodnout se na vlastní dvojici klíčů, kterými by si komunikaci dešifroval.

Útoky se označují jako *CVE-rok-řadová číslice* (Common Vulnerabilities and Exposures), kde *rok* je rokem zveřejnění útoku. CVE je seznam jmen veřejně známých bezpečnostních problémů. Každému útoku je při zařazení mezi CVE přidělen identifikátor, pomocí kterého lze útok dohledat v databázi [1].

4.1 Útoky na PKI

Tento typ útoků se zaměřuje na kompromitaci certifikační autority. Ve většině případů jde o její oklamání, kdy autoritu přesvědčíme o vlastnictví konkrétní domény. V případě úspěchu nám autorita vystaví platný certifikát. V některých případech je tento útok mířen na celé certifikační autority. Pokud se nám podaří certifikační autoritu kompromitovat, pak si můžeme s její pomocí vytvořit certifikát pro libovolnou webovou stránku. U těchto útoků se tedy ve většině případů jedná spíše o sociální inženýrství [16]. Jako příklad jsou zde uvedeny dva zaznamenané útoky — VeriSign Microsoft Code-Signing Certificate a Thawte login.live.com.

4.1.1 VeriSign Microsoft Code-Signing Certificate

V roce 2001 se podařilo oklamat certifikační autoritu VeriSign, která vydala dva certifikáty osobě, která se vydávala za představitele Microsoftu [3]. Tato osoba musela znát systém vydávání certifikátu a být dostatečně přesvědčivá, jelikož pro vystavení musela vlastnit průkaz s falešnou identitou a podařilo se jí oklamat alespoň jednoho člověka z VeriSign, který tuto žádost o certifikát potvrdil [16].

4.1.2 Thawte login.live.com

Bezpečnostnímu výzkumníkovi jménem Mike Zusman se roku 2008 podařilo získat certifikát pro *login.live.com*, což je přihlašovací stránka patřící pod Microsoft.

Využil dvou skutečností — Certifikační autorita Thawte používala pro ověření vlastnictví webové stránky email a Mikovi se podařilo registrovat si email *sslcertificates@live.com*, který použil při vyřizování certifikátu [16].

V roce 2015 byl tento útok reprodukován na doméně *live.fi* [18].

4.2 Útoky využívající chyb ve webovém prohlížeči a HTTP

Tento typ útoků je založen na využití chyb zavedených při tvorbě webové stránky nebo špatného chování uživatele na internetu. HTTP protokol totiž neumožňuje šifrování ani zabezpečení integrity dat, čehož může útočník využít.

4.2.1 Sidejacking [16]

Sidejacking je speciální případ ukradení relace webové aplikace. Při tomto útoku útočník ukradne token relace (session token), který získá z nezašifrované komunikace. Tento útok je jednoduchý, pokud webová stránka nepoužívá šifrování vůbec. Pokud je stránka šifrovaná částečně, pak můžou nastat dva typy chyb, kterých může útočník využít:

Únik relace díky špatnému návrhu
: Některé stránky používají šifrování pouze pro zabezpečení hesel, ale po přihlášení uživatele komunikují nezabezpečeně. V tomto případě útočník nemůže ukrást přihlašovací údaje, ale *tokeny relace*, kterými se může prokázat stránce, která ho bude považovat za přihlášeného uživatele.

Únik relace díky chybě
: I když se tvůrce stránky snaží využít šifrování na celém webu, stále může vzniknout chyba, kdy s některým zdrojem pracuje nezašifrovaně. Útočník toho může využít a ukrást tak relaci.

4.2.2 SSL Stripping [16]

SSL Stripping je typ Man-in-the-Middle útoku, kdy útočník přinutí oběť pro komunikaci přes HTTP protokol. Tohoto lze docílit změnou *https://* na *http://* v URL. Útoky tohoto typu jsou mířeny na uživatele, kteří nepoznají rozdíl mezi tím, jestli si prohlíží stránku se zabezpečením nebo ne. Uživatelé, kteří tento rozdíl poznají, můžou být napadeni pomocí přesměrování na podvržený "zabezpečený"web, který má však pod kontrolou útočník. Tento útočníkův web může vypadat stejně jako požadovaná stránka oběti, jen její adresa se může lišit např. v jednom znaku. Jedním řešením, jak se bránit proti tomuto útoku, je HSTS

(HTTP Strict Transport Security), což je bezpečnostní mechanismus, který umožňuje, aby webový server vynutil v prohlížeči komunikaci pomocí šifrovaného HTTPS.

4.3 Útoky na protokol

Útoky tohoto typu se zaměřují na zneužití některých funkcí SSL/TLS protokolu. Některé funkce, které bylo možné využít k útoku, byly upraveny v novějších verzích protokolu nebo se přestaly využívat, ale dosud nebyly z protokolu zcela odstraněny.

4.3.1 BEAST [16]

V roce 2011 byla zveřejněna technika nového útoku, který mohl být využit na TLS 1.0 a starší verze protokolu. Tento útok využívá předpověditelné konstrukce inicializačního vektoru u *CBC (Cipher Block Chaining)*. Tento problém byl vyřešen ve verzi TLS 1.1.

O tomto problému se vědělo už dříve, ale nikdo mu nevěnoval příliš velkou pozornost. Až díky dvěma výzkumníkům, Duongu a Rizzovi, kteří tento útok uskutečnili, bylo demonstrováno, že útoky se stále zlepšují a ignorace malých nedostatků v protokolu může vést na katastrofální problémy.

4.3.2 CRIME [16]

Roku 2012 stejní výzkumníci, kteří stáli za útokem BEAST, Duong a Rizzo, ukázali, jak lze zneužít komprese dat v SSL/TLS protokolu. Tímto útokem lze získat *cookie* uživatele a tu využít pro připojení k webové stránce jako oběť přes HTTPS protokol.

CRIME byl jedním z útoků zneužívající komprese dat u protokolu. Díky těmto útokům, které ukázaly problémy s bezpečností při využití komprese dat, se jí přestalo využívat u SSL/TLS protokolu.

4.3.3 POODLE [16]

Google Security Team zveřejnil v roce 2014 útok *POODLE — Padding Oracle On Downgraded Legacy Encrytion*, který umožňuje útočníkovi získat část šifrovaných dat. Útok využívá chybného návrhu *CBC*, který zanechává zarovnání pro blokovou šifru nezabezpečené.

4.4 Útoky na chyby v implementaci

Software je v dnešní době často vyvíjen co nejrychleji, aby byl co nejlevnější, ale jeho bezpečnost se ve většině případů zanedbává. Dokumentace mnohdy obsahují úryvky zdrojových kódu i s bezpečnostními chybami.

Mezi nejrozšířenější implementace SSL/TLS protokolu v dnešní době patří OpenSSL a NSS. Samotné OpenSSL je používáno denně miliony lidí, (např. v *Apache HTTP Server*). Bohužel je málo zdokumentováno a složité na použití.

I přesto, že jsou tyto knihovny denně používány a spoléháme se na ně, že nám poskytnou zabezpečenou komunikaci, i přesto se v nich objevují stále nové chyby, kterých může útočník využít.

4.4.1 Heartbleed [16]

Tento útok, zveřejněn roku 2014, je považován za zatím nejhorší incident TLS protokolu.

Útok zneužívá rozšíření TLS protokolu *Heartbeat* v OpenSSL. Umožňuje útočníkovi získat až 64 kB dat z paměti serveru v jednom Heartbeat požadavku. Tento požadavek může útočník posílat neustále, díky čemuž může získat hesla, TLS klíče relací nebo i privátní klíč serveru.

4.4.2 Protocol Downgrade Attack [16]

Aktivní útok typu Man-in-the-Middle může vést k dohodnutí použití nižší a méně bezpečnější verze protokolu během handshaku. Jelikož webové prohlížeče se snaží o úspěšnou komunikaci s každým serverem, takže s ním začnou komunikovat i se starší verzí SSL/TLS protokolu, které většina serverů podporuje kvůli zpětné kompatibilitě se staršími systémy. Tento útok je možný ve starších verzích OpenSSL.

Kapitola 5

Implementace vybraných útoků na SSL/TLS

Z kategorií útoků na SSL/TLS protokol uvedených v kapitole 4 byly reprodukovány útoky, které patří do kategorie, které využívají chyb v implementaci. Ostatní kategorie nejsou vhodné pro reprodukování — u útoků na PKI je hlavním principem oklamání autority, jde tedy spíše o sociální inženýrství; útoky, které využívají chyb ve webovém prohlížeči a HTTP, jsou závislé na chování uživatele a míří více na klientskou stranu, což zatím není v tlsfuzzeru podporované. Testovat útoky, které využívají problémů v protokolu, není složité, protože ve většině případů stačí ověřit, zda-li protější strana podporuje některou ze zastaralých nebo nebezpečných funkcí, proto testů pro tyto útoky již existuje dostatečné množství, a proto se na ně tlsfuzzer nezaměřuje.

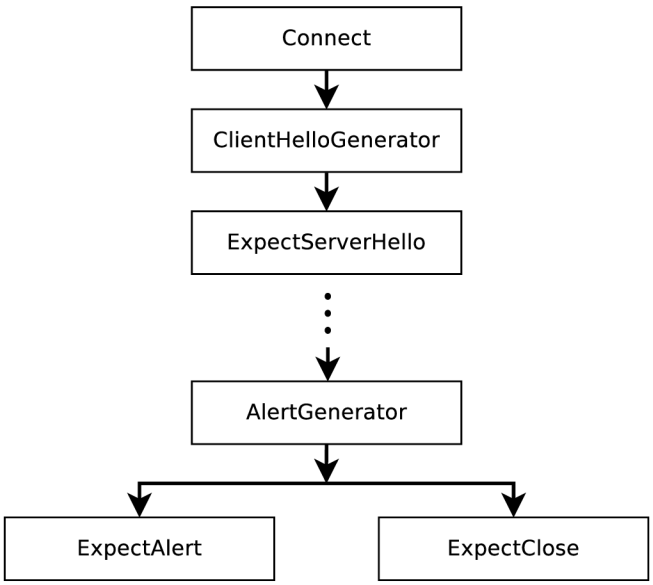
Vybrané útoky jsou implementovány jako testy pro tlsfuzzer. Tlsfuzzer je aktuálně v rané alfa verzi — z částí uvedených v podkapitole 3.4 funguje pouze část *Vykonavatel*, chybí podpora některých funkcí (ať už funkcí, které jsou nebo nejsou implementovány v knihovně tlslite-ng) a je možné testovat pouze stranu serveru.

V kódu 5.1 je uveden příklad modelování konverzace v tlsfuzzeru. Na prvním řádku je do konverzace vložen objekt, který inicializuje TCP spojení na uvedenou adresu a port. Po přidání reference na konverzaci do jiné proměnné a inicializaci pole se šifrovacími sadami, je na řádku číslo 4 přidán k úvodnímu uzlu potomek, který bude vykonán po vytvoření spojení. Tento potomek je generátor pro vytvoření zprávy ClientHello. V tomto generátoru je použit parametr pro definování šifrovacích sad, které budou nabídnuty serveru pro připojení. Kromě tohoto parametru lze definovat např. požadovanou verzi SSL/TLS, rozšíření nebo metody komprese. Na následujícím řádku je očekávání zprávy ServerHello od serveru. V této očekávané zprávě můžeme definovat např. rozšíření, které vyžadujeme od serveru. V případě, že by nám server toto rozšíření neposlal v ServerHello, čímž by sdělil, že nechce vytvořit spojení s tímto rozšířením, pak je to na straně tlsfuzzeru zaznamenáno jako chyba a spojení je ukončeno. Po ServerHello je očekáván certifikát, ve kterém lze specifikovat očekávaný typ certifikátu, např. certifikát standardu X.509, a ServerHelloDone, po kterém je spuštěn na řádku 8 generátor ClientKeyExchange. V tomto generátoru je vytvořena klientova část pro tvorbu šifrovacích klíčů a lze v něm pomocí parametrů specifikovat některé hodnoty, které ovlivňují tvorbu klíčů. Na řádku 9 je vložen ChangeCipherSpec generátor, ve kterém můžeme specifikovat rozšířený master secret (extended master secret). Tímto generátorem je změněno odesílání zpráv, takže všechny další odchozí zprávy z naší strany budou šifrovány. Následujícím generátorem Finished informujeme o ukončení hand-

shaku z naší strany. Na řádku 11 očekáváme od serveru ChangeCipherSpec, po kterém je změněno čtení zpráv, protože všechny další příchozí zprávy musí být šifrovány, a Finished, pro potvrzení a ukončení handshaku. Tímto je ukončen v uvedeném příkladu handshake. Další částí, která je zde uvedena od řádku 13, je korektní ukončení spojení iniciované z naší strany. První je použit generátor pro alert, který má v parametrech specifikovánu závažnost *warning* a popis `close_notify`, což vytvoří upozornění o žádosti o ukončení spojení. Z druhé strany očekáváme odpověď v podobě alertu nebo okamžité ukončení spojení. Ukončení spojení v podobě odeslání alertu `close_notify` a očekávání alertu z protější strany je považováno za řádné ukončení spojení, což některé servery nesplňují a ihned po obdržení `close_notify` alertu ukončují spojení, což je důvodem, proč jsou po odeslání alertu přijímány dva způsoby chování protější strany.

```
1  conversation = Connect(host, port)
2  node = conversation
3  ciphers = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA]
4  node = node.add_child(ClientHelloGenerator(ciphers))
5  node = node.add_child(ExpectServerHello())
6  node = node.add_child(ExpectCertificate())
7  node = node.add_child(ExpectServerHelloDone())
8  node = node.add_child(ClientKeyExchangeGenerator())
9  node = node.add_child(ChangeCipherSpecGenerator())
10 node = node.add_child(FinishedGenerator())
11 node = node.add_child(ExpectChangeCipherSpec())
12 node = node.add_child(ExpectFinished())
13 node = node.add_child(AlertGenerator(AlertLevel.warning,
14                                       AlertDescription.close_notify))
15 node = node.add_child(ExpectAlert())
16 node.next_sibling = ExpectClose()
```

Kód 5.1: Příklad tvorby handshaku a ukončení komunikace v tlsfuzzeru



Obrázek 5.1: Příklad rozhodovacího stromu pro zdrojový kód tlsfuzzeru

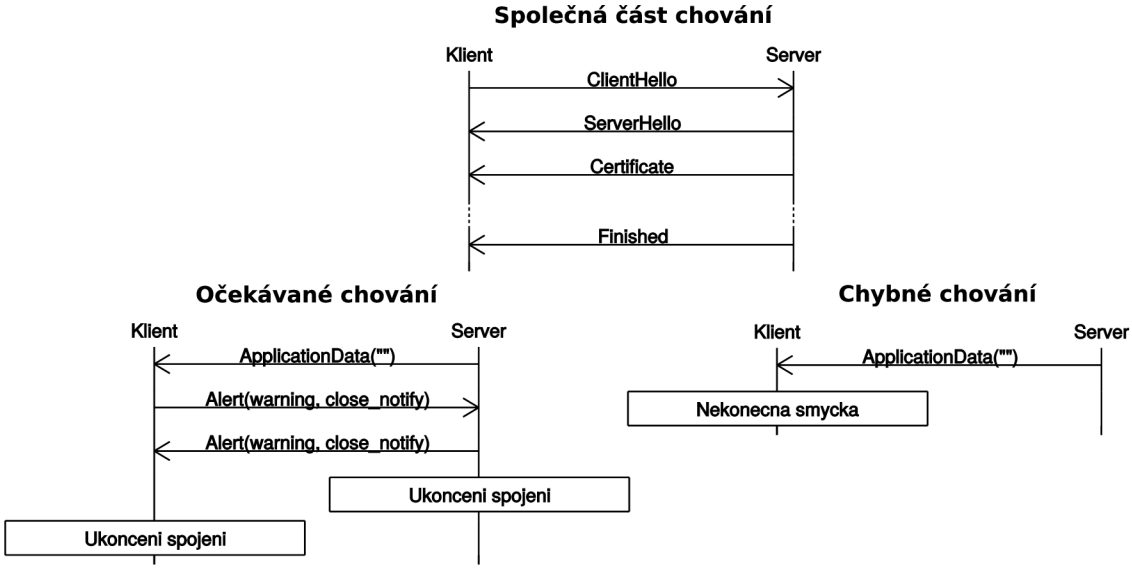
Tímto zdrojovým kódem je vytvořen rozhodovací strom, který je bez rozvětvení s výjimkou poslední ukončovací části, kde k uzlu přidáme druhou možnost pomocí vložení do `next_sibling`. Rozhodovací strom z výše uvedeného zdrojového kódu je ve zkrácené podobě uveden na obrázku 5.1.

Pro implementaci byly vybrány takové CVE (Common Vulnerabilities and Exposures), pro které v tlsfuzzeru ještě není vytvořen test a zároveň je umožňují schopnosti tlsfuzzeru a tlslite-ng. Pokud vytvoření testu vyžadovalo další funkčnost tlsfuzzeru nebo tlslite-ng a bylo možné ji přidat bez rozsáhlých změn, pak tato funkčnost byla realizována a je součástí této práce.

Nové funkce a testy byly vyvíjeny ve forku[1] tlsfuzzeru a do upstreamu[2] jsou změny přidávány pomocí pull requestů. Některé změny byly již přijaty a ostatní čekají na schválení.

5.1 CVE-2016-6305: SSL_peek() hang on empty record

Prvním reprodukovaným CVE je CVE-2016-6305[3]. Toto CVE je způsobeno chybou v kryptografické knihovně OpenSSL verze 1.1.0, kdy funkce `ssl3_read_bytes` v souboru `record/rec_layer_s3.c`[4] umožňuje útočníkovi způsobit DoS útok (typ útoku, kdy cílem je cílovou službu znepřístupnit ostatním uživatelům) v podobě nekonečné smyčky ve funkci `SSL_peak` tím, že útočník zašle v průběhu komunikace záznam s nulovou velikostí dat. Chování CVE a jaké je očekávané chování je uvedeno na obrázku 5.2.



Obrázek 5.2: Očekávané chování a chování CVE-2016-6305: SSL_peek() hang on empty record

Při testování této chyby na OpenSSL však nebylo zjištěno neobvyklého chování ani u verzí OpenSSL, které byly u CVE uvedeny jako zranitelné. Po konzultaci a analýze zdrojových kódů OpenSSL bylo zjištěno, že problémová funkce ve skutečnosti není využívána

[1] https://github.com/mildass/tlsfuzzer
[2] https://github.com/tomato42/tlsfuzzer
[3] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6305
[4] https://github.com/openssl/openssl/blob/master/ssl/record/rec_layer_s3.c#L1149

u serverů. Tuto chybu tedy zatím nelze otestovat s pomocí tlsfuzzeru, proto tento test nebyl prozatím přidán mezi ostatní testy do hlavního repozitáře tlsfuzzeru[5].

5.1.1 Implementace

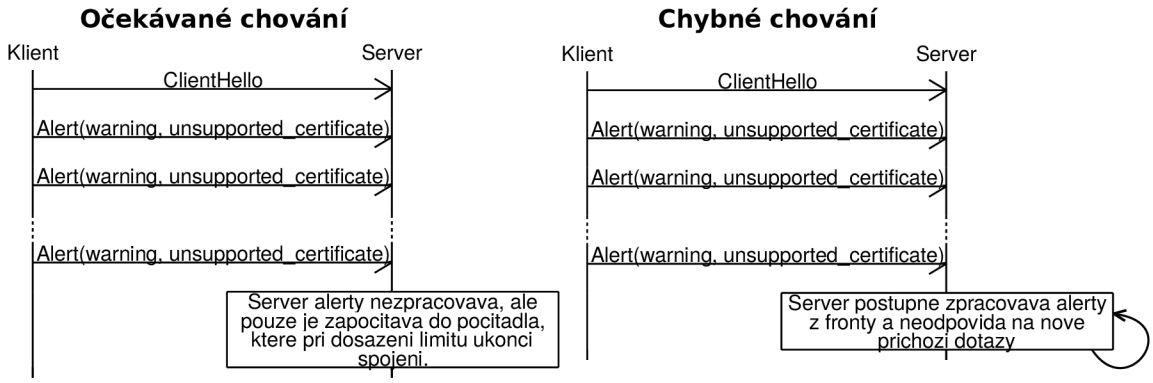
Pro toto CVE je vytvořen test v souboru `scripts/test-cve-2016-6305.py`. V úvodní části testu jsou zpracovány argumenty skriptu, připojení k serveru a handshake. Po handshaku je hlavní část testu — vytvoření aplikačních dat s nulovou délkou s pomocí generátoru aplikačních dat. Příklad tvorby aplikačních dat nulové délky z testu je uveden v kódu 5.2. Po této části je očekáváno řádné ukončení spojení, abychom byli ujištěni, že vše proběhlo v pořádku a protější strana neuvázla v nekonečné smyčce.

```
empty_record = b""
node = node.add_child(ApplicationDataGenerator(empty_record))
```

Kód 5.2: Tvorba záznamu s aplikačními daty nulové délky v tlsfuzzeru

5.2 CVE-2016-8610: SSL Death Alert

Dalším reprodukovaným CVE je CVE-2016-8610[6], které bylo nazváno jako *SSL Death Alert*. CVE se vyskytuje ve OpenSSL verzích 1.1.0, 1.0.2 až 1.0.2h, ve všech verzích 1.0.1 (1.0.1 až 1.0.1u) a ve všech 0.9.8 (0.9.8 až 0.9.8zh). Chyba je způsobena nevhodnou manipulací s upozorněními závažnosti *warning* během handshaku. Díky tomu může útočník opakovaným zasíláním alertů v průběhu handshaku dosáhnout vytížení až 100% CPU (centrální procesorová jednotka) na straně serveru, čímž dosáhne DoS útoku. CVE chování a očekávané chování je na obrázku 5.3.



Obrázek 5.3: Očekávané chování a chování CVE-2016-7054: SSL Death Alert

Tímto testem byl zjištěn počet alertů, kolik je pro OpenSSL knihovnu považován za zlomový — 4 alerty zaslané ihned po ClientHello nevyvolaly u OpenSSL žádnou reakci, ale 5 alertů způsobilo po ServerHelloDone odeslání alertu úrovně *fatal* a ukončení spojení. Tento test byl schválen a přidán[7] do oficiálního repozitáře tlsfuzzeru.

[5] https://github.com/tomato42/tlsfuzzer
[6] http://seclists.org/oss-sec/2016/q4/224
[7] https://github.com/tomato42/tlsfuzzer/pull/95

5.2.1 Implementace

Pro toto CVE jsou v souboru `scripts/test-ssl-death-alert.py` vytvořeny 2 testy. Před samotnými testy jsou opět zpracovány argumenty skriptu. První test provede připojení, ve kterém vyžaduje po protější straně verzi TLS 1.2, pro ujištění, že protější strana typ příchozích alertů rozpozná, a následně je zavolán generátor `ClientHelloGenerator`. Tímto je započat handshake a vzápětí jsou zaslány alerty, jejichž počet je možné určit při volání skriptu. Zasílání upozornění je uvedeno v kódu 5.3. Při zasílání upozornění je důležitá závažnost *warning* a popis, protože upozornění závažnosti *fatal* nebo s popisem, který i při závažnosti *warning* bývá vyhodnocen jako problémový, způsobí ukončení spojení. Z možných popisů upozornění je možné použít uvedený `unsupported_certificate` (nepodporovaný typ certifikátu) nebo např. `certificate_unknown` (nastala nespecifikovaná chyba při zpracovávání certifikátu). Po odeslání upozornění je očekáván normální průběh handshaku (nejprve ServerHello od protější strany s pomocí `ExpectServerHello` atd.) a po handshaku je očekáváno normální ukončení spojení. Tento test slouží ke zjištění, kolik upozornění je možné TLS implementaci zaslat, aniž bychom vyvolali u protější strany chybu a ukončení spojení.

```
for _ in range(number_of_alerts):
    node = node.add_child(
        AlertGenerator(AlertLevel.warning,
                       AlertDescription.unsupported_certificate))
```

Kód 5.3: Zasílání alertů v průběhu handshaku

Druhý test, obdobně jako první test, po připojení a zaslání ClientHello začne posílat alerty, jejich počet je o 1 větší než v předchozím testu. Po zaslání těchto alertů je očekáván od protější strany úvod handshaku — ServerHello, Certificate a ServerHelloDone. Po tomto úvodu je očekáván alert se závažností *fatal* a okamžité ukončení spojení. Tímto testem je zjištěn počet alertů, který je pro implementaci již závažný a vyvolá na ně reakci v podobě alertu a okamžitého ukončení spojení.

5.3 CVE-2017-3733: Encrypt-Then-MAC renegotiation crash

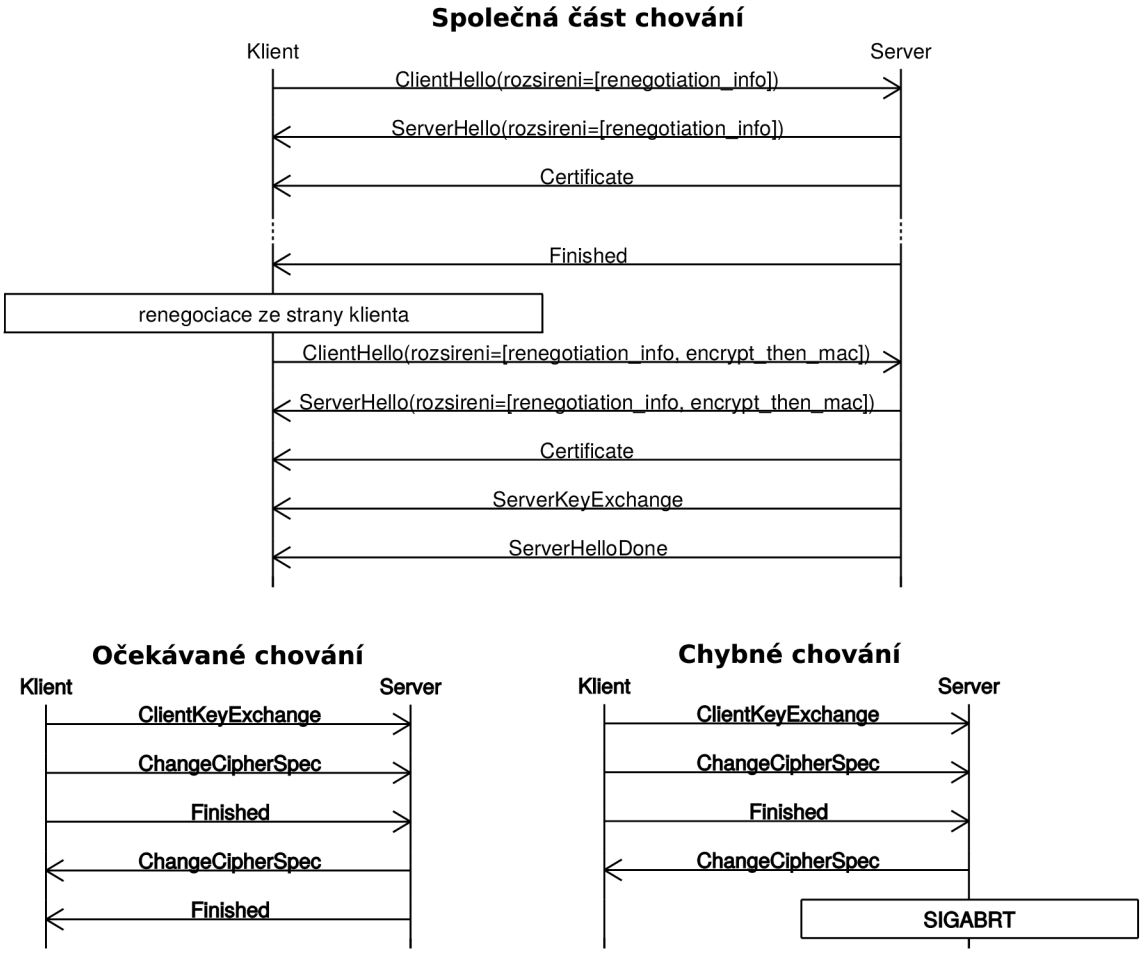
CVE-2017-3733[8] popisuje chybu v OpenSSL ve verzích 1.1.0 až 1.1.0d, která při použití vybraných šifrovacích sad způsobuje pád OpenSSL. Tuto chybu lze vyvolat při dohodnotí Encrypt-then-MAC rozšíření (viz. 2.2.5) při renegociaci, pokud toto rozšíření nebylo dohodnuto v původním handshaku. Očekávané chování a chování CVE je zobrazeno na obrázku 5.4.

Pro tento test byl vytvořen pull request[9] do oficiálního repozitáře tlsfuzzeru. Byly provedeny požadované úpravy a test čeká na schválení a přidání do tlsfuzzeru.

5.3.1 Implementace

Encrypt-Then-MAC rozšíření je implementováno v knihovně tlslite-ng, ale nebylo podporováno v tlsfuzzeru, který po dohodnutí rozšíření Encrypt-then-MAC rozšíření nezměnil způsob vytváření ověřeného šifrování a stále využíval výchozí MAC-then-Encrypt, proto bylo první potřeba tuto možnost doplnit. Pro toto rozšíření byl vytvořen atribut

[8] https://www.openssl.org/news/secadv/20170216.txt
[9] https://github.com/tomato42/tlsfuzzer/pull/122



Obrázek 5.4: Očekávané chování a chování CVE-2017-3733: Encrypt-Then-MAC renegotiation crash

`encrypt_then_mac` ve třídě `ConnectionState` v souboru `tlsfuzzer/runner.py`, který je datového typu boolean a výchozí hodnoty *False*. Tento atribut je měněn s každým příchozím ServerHello — ve třídě `ExpectServerHello` a jeho metodě `process(state, msg)`, která zpracovává ServerHello zprávu, je tento atribut získán z parametru `state` a v úvodu vždy nastaven na výchozí *False*, protože Encrypt-then-MAC rozšíření se vztahuje pouze na dané sezení, ve kterém bylo dohodnuto. Po tomto nastavení na výchozí hodnotu jsou zkontrolovány všechny rozšíření ze ServerHello a ověřeno, zda-li byly nabídnuty ze strany klienta v ClientHello a pokud je jedno z rozšíření Encrypt-then-MAC, pak je i atribut `encrypt_then_mac` změněn na hodnotu *True*. Tento atribut je dále využit ve třídě `ChangeCipherSpecGenerator` v metodě `post_send(status)`, která je zavolána po odeslání ChangeCipherSpec zprávy. V této metodě je hodnota z vytvořeného atributu `status.encrypt_then_mac` vložena do `status.msg_sock.encryptThenMAC`, což je atribut ze třídy, která zajišťuje činnosti při odesílání a přijímání zpráv jako je např. jestli je pro zprávy použito ověřené šifrování MAC-then-Encrypt nebo Encrypt-then-MAC.

```
node = node.add_child(ApplicationDataGenerator(
    bytearray(b"GET␣/␣HTTP/1.0\r\n\r\n")))
```

```
node = node.add_child(ExpectApplicationData())
```

Kód 5.4: Žádost o aplikační data a jejich očekávání

Pro toto CVE jsou v souboru `scripts/test-encrypt-then-mac-renegotiation.py` vytvořeny 2 testy. První test je tzv. *sanity test*, kterým je zjištěno, zda-li protější strana podporuje renegociaci, rozšíření Encrypt-then-MAC a správně komunikuje, což je zjištěno správným handshakem, odpovědí na naší žádost o aplikační data, která je uvedena v kódu 5.4 a ukončením spojení. Žádost o možnou budoucí renegociaci z klientské strany je možné sdělit pomocí přidání šifrovací sady *TLS_EMPTY_RENEGOTIATION_INFO_SCSV* k dalším šifrovacím sadám nebo s pomocí rozšíření `renegotiation_info`. V obou případech nám server potvrzení odešle v ServerHello jako rozšíření `renegotiation_info`.

Ve druhém testu, který již reprodukuje uvedené CVE, je proveden handshake, ve kterém je ze strany klienta odesláno ClientHello s rozšířením `renegotiation_info`. Toto rozšíření očekáváme též v příchozí zprávě ServerHello jako potvrzení o použití renegociace. Po ukončení handshaku je započata renegociace, před kterou je spuštěn příkaz `ResetHandshakeHashes`, který obnoví hodnoty handshake hašů pro hašovací algoritmy. Při renegociaci jsou v ClientHello odeslány rozšíření `renegotiation_info`, které musíme odeslat, aby protější strana věděla, že navazujeme na předchozí handshake, a `encrypt_then_mac`, pomocí které se pokusíme dohodnout rozšíření Encrypt-then-MAC. Obě tyto rozšíření očekáváme v příchozím ServerHello. Dále je očekáván běžný průběh handshaku, aplikační data na naší odeslanou žádost (viz. 5.4) a ukončení spojení.

Tyto 2 testy jsou spuštěny v pořadí sanity test, CVE test a opět sanity test, čímž zjistíme, zda-li protější strana před a po CVE testu je ve stejném stavu — prvním sanity testem jsme si ověřili, že server podporuje renegociaci a Encrypt-then-MAC, a druhým sanity testem je ověřeno, že server stále komunikuje jako před otestováním CVE.

5.4 CVE-2014-0160: Heartbleed

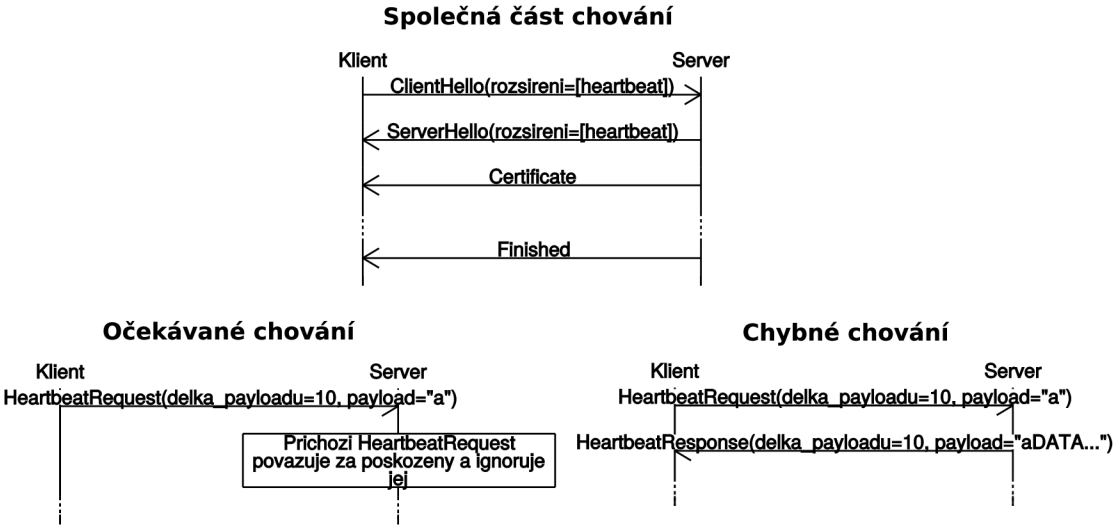
CVE-2014-0160[10], které bylo nazváno Heartbleed, zneužívá chybného zacházení se zprávami rozšíření Heartbeat 2.2.5. Tato chyba je v SSL/TLS implementaci OpenSSL ve verzích 1.0.1 až 1.0.1f a 1.0.2-beta. Chyba využívá chybějící kontroly mezí, kdy je čteno více dat něž by mělo být povoleno. Zranitelnosti lze zneužit při zadání chybné hodnoty velikost payloadu, která neodpovídá reálné velikosti. Např. pošleme-li Heartbeat požadavek v němž je uvedena velikost payloadu 25 bajtů, ale payload obsahuje data o velikosti pouze 10 bajtů, pak nám protější strana, která je zranitelná vůči tomuto útoku, v odpovědi zašle náš payload (velikost 10 znaků) a data z paměti (velikost 15 znaků). Správné chování a chování u serveru zranitelného na Heartbleed je na obrázku 5.5. Tímto způsobem lze získat ze serveru např. privátní klíče pro dešifrování komunikace nebo hesla.

Pro tento test byl vytvořen pull request pro přidání Heartbeatu do upstreamu tlslite-ng[11]. Další pull request pro přidání Heartbeatu a testu na Heartbleed do tlsfuzzeru budou přidány po schválení a přidání rozšíření Heartbeatu do tlslite-ng.

5.4.1 Implementace

Nejprve bylo potřeba přidat podporu pro Heartbeat rozšíření do knihovny tlslite-ng. Pro Heartbeat rozšíření je vytvořena třída `HeartbeatExtension` v souboru

[10] https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160
[11] https://github.com/tomato42/tlslite-ng/pull/155



Obrázek 5.5: Očekávané chování a chování CVE-2014-0160: Heartbleed

`tlslite/extensions.py`. Tato třída obsahuje metody pro vytvoření Heartbeat rozšíření, zakódování rozšíření pro přidání do zprávy ClientHello nebo ServerHello, a zpracování příchozího Heartbeat rozšíření. Třída má jeden atribut, ve kterém je určeno, zda-li lze přijímat Heartbeat požadavky nebo ne.

Další částí bylo přidání Heartbeat protokolu do tlslite-ng. Pro tento protokol je vytvořena třída `Heartbeat` v souboru `tlslite/messages.py`. První metodou této třídy je vytvoření Heartbeat zprávy, ve které si lze zvolit typ (požadavek nebo odpověď), payload a velikost vycpávky[12]. Další metoda slouží pro vytvoření odpovědi na Heartbeat žádost. V této metodě je u Heartbeat žádosti změněn typ a vytvořena vycpávka. Další dvě metody slouží podobně jako u Heartbeat rozšíření pro zakódování a pro zpracování. Tyto metody zpracovávají Heartbeat zprávu, ať aby ji bylo možné odeslat nebo abychom z příchozí zprávy zjistili, co přesně obsahuje. Poslední metoda kontroluje příchozí Heartbeat odpovědi. V této metodě je použit mechanismus zpětného volání, pomocí kterého lze parametrem při volání metody specifikovat funkci, která má být zavolána v těle metody.

Třídy vytvořené pro zpracování Heartbeat rozšíření a Heartbeat protokolu byly dále propojeny s implementací tlslite-ng. Rozšíření je přidáváno v případě tlslite-ng v pozici klienta do ClientHello a v pozici serveru do ServerHello, pokud o rozšíření projevila zájem protistrana v ClientHello. Dále byla přidána část mezi zpracovávání zpráv, kdy v případě obdržení Heartbeat požadavku je vytvořena a odeslána odpověď a v případě Heartbeat odpovědi je tato odpověď zaznamenána.

Dále byla přidána podpora Heartbeat rozšíření do tlsfuzzeru, v němž byly vytvořeny dvě třídy — `HeartbeatGenerator`, který vytvoří Heartbeat zprávu, která je připravena na odeslání vykonavatelem, a `ExpectHeartbeatResponse`, která očekává Heartbeat odpověď a lze jí specifikovat payload, který očekáváme v této odpovědi. V případě, že payload nesouhlasí, pak je vyvolána výjimka.

Pro Heartbleed je v tlsfuzzeru vytvořen test, který po handshaku, ve kterém je nabídnuto v ClientHello a očekáváno v ServerHello rozšíření Heartbeat, vytvoří Heartbeat požadavek s payloadem velikosti 1 s pomocí třídy `HeartbeatGenerator`. Tento požadavek je poté upraven s pomocí funkce `fuzz_message`, se kterou lze upravit zprávu po zakódování

[12]Vycpávka jsou v Heartbeat zprávě náhodné bajty o velikosti alespoň 16 B

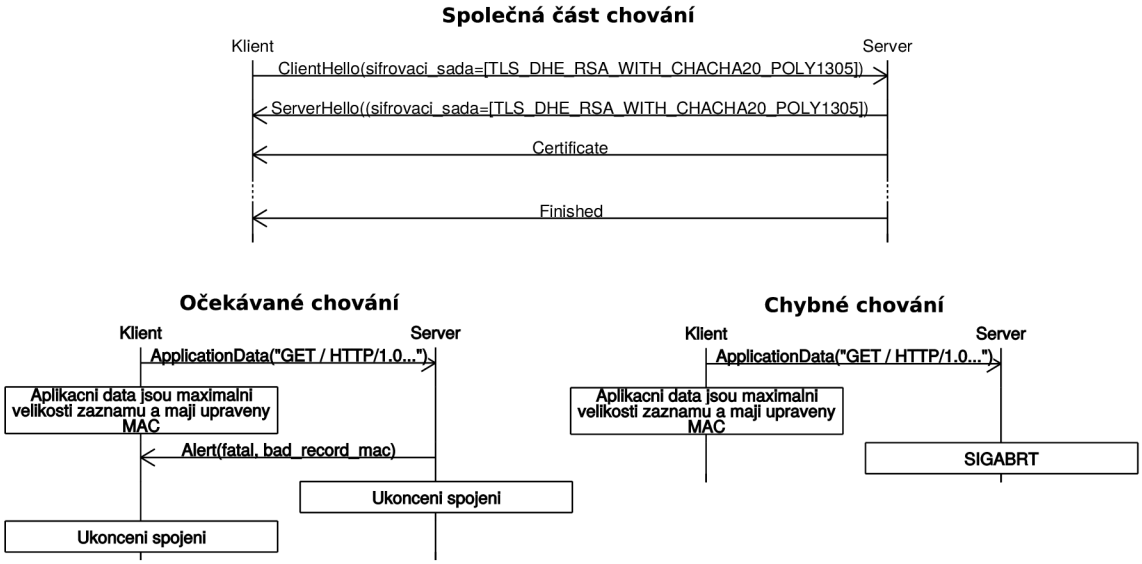
```
node = node.add_child(fuzz_message(HeartbeatGenerator(data),
                                   substitutions={2:2}))
```

Kód 5.5: Vytvoření Heartbeat požadavku pro Heartbleed

do bajtů. V této funkci je využit parametr `substitutions`, do kterého je vložen slovník, který specifikuje úpravu třetího bajtu (první bajt určuje typ a následující 2 velikost) na hodnotu 2, což změní hodnotu velikosti payload v Heartbeat požadavku z 1 na 2. Po tomto požadavku je očekáváno ukončení spojení. Není očekávána Heartbeat odpověď, protože Heartbeat požadavek byl poškozen. Příklad tvorby poškozeného Heartbeat požadavku je v kódu 5.5.

5.5 CVE-2016-7054: ChaCha20/Poly1305 buffer-overflow

Toto CVE[13] způsobuje chyba, která může způsobit pád OpenSSL při poškození větších zpráv, které jsou šifrovány šifrovací sadou **-CHACHA20-POLY1305*, kde CHACHA20 slouží pro tvorbu proudových šifer s pomocí POLY1305 je vytvořen MAC pro ověření zprávy. Chyba se vyskytuje ve verzích OpenSSL 1.1.0 až 1.1.0b. Očekávané chování a chybné chování je na obrázku 5.6.



Obrázek 5.6: Očekávané chování a chování CVE-2016-7054: ChaCha20/Poly1305 buffer-overflow

Pro tento test byl vytvořen pull request[14]. Byly provedeny požadované úpravy a test čeká na schválení do tlsfuzzeru.

[13] https://www.openssl.org/news/secadv/20161110.txt
[14] https://github.com/tomato42/tlsfuzzer/pull/126

5.5.1 Implementace

Pro CVE byl již vytvořen test a pull request[15] od autora Babak Amin Azad, ale obsahoval řadu nedostatků, které udržovatel tlsfuzzeru požadoval před schválením tohoto pull request. Autor pull requestu ale nereagoval, což bylo důvodem, proč bylo navázáno na tento pull request a změněn test podle požadavků.

```
data_length = 2**14 - 28
node = node.add_child(ApplicationDataGenerator(
    b"GET / HTTP/1.0\r\n" +
    b'X-test: ' + data_length * b'A' +
    b'\r\n\r\n'))
```

Kód 5.6: Vytvoření požadavku na aplikační data, který je maximální velikosti

Skript je v souboru `scripts/test-cve-2016-7054.py` a obsahuje dva testy. První je *sanity test*, kde je zjištěno, zda-li protější strana správně komunikuje. Je tedy proveden handshake, kdy v ClientHello je nabídnuta šifrovací sada z kategorie **-CHACHA20-POLY1305* (konkrétně *TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256*). Po handshaku je vytvořen a odeslán požadavek na aplikační data a ty jsou též očekávány od protější strany. Požadavek na aplikační data je maximální velikosti záznamu. Žádost o aplikační data je uvedena v kódu 5.6. Maximální velikost záznamu je 2^{14} B, ze které musíme ale odečíst 28 B, což je velikost dat pro vytvoření požadavku o aplikační data ("GET / HTTP/1.0\r\nX-test: "a ukončení požadavku).

```
fuzzes = [(x, 2**y) for x in range(-16, 0) for y in range(8)]
```

Kód 5.7: Vytvoření listu pro úpravu všech bitů v 16 bajtech

Druhý test je pro CVE. Tento test probíhá ve `for` cyklu, kdy procházíme hodnoty ze dvouprvkového listu, který je vytvořen s pomocí generátoru, který je v kódu 5.7. Tento list slouží pro postupnou úpravu jednotlivých bitů, které se vyskytují v 16 B tagu. Tento tag se vyskytuje za šifrovaným textem v šifrovacích sadách **-CHACHA20-POLY1305* (vždy posledních 16 B) a slouží jako MAC. V cyklu vždy proběhne handshake se stejnou šifrovací sadou jako je v sanity testu a jsou odeslány aplikační data, ve kterých je upraven 1 bit z tagu. Úprava aplikačních dat je provedena s pomocí `fuzz_encrypted_message`, který je podobný jako `fuzz_message` použitý v 5.4, ale upravuje zprávu, která je již zašifrována a obsahuje MAC. Použití je znázornění v kódu 5.8. Ve funkci `fuzz_encrypted_message` je použit parametr `xors`, který provádí operaci XOR na vybrané pozici s konkrétní hodnotou. Po odeslání těchto poškozených dat je očekáván od protější strany alert `bad_record_mac` úrovně *fatal* a okamžité ukončení spojení.

```
node = node.add_child(fuzz_encrypted_message(
    ApplicationDataGenerator(
        b"GET / HTTP/1.0\r\n" +
        b'X-test: ' + data_length * b'A' +
        b'\r\n\r\n'),
    xors={possition: value}))
```

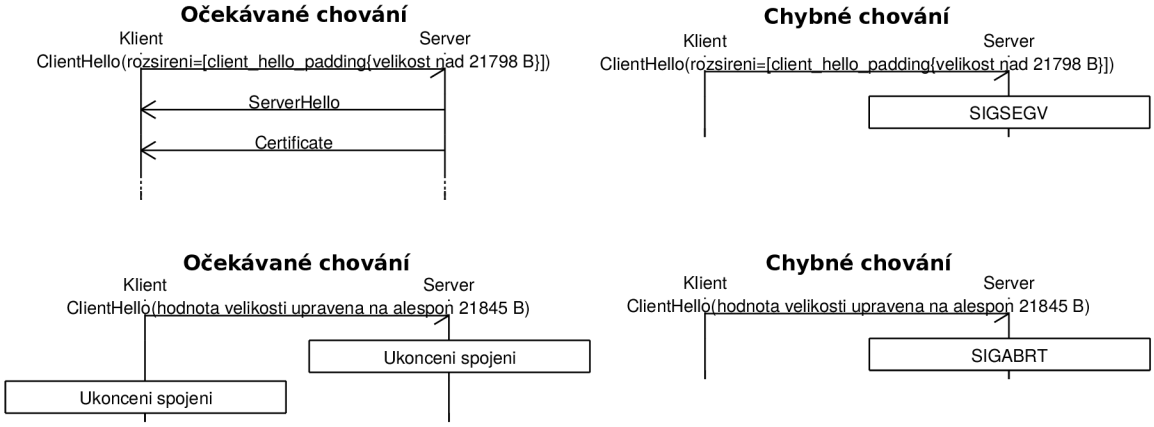
Kód 5.8: Odeslání velkých aplikačních dat s poškozeným MAC

[15] https://github.com/tomato42/tlsfuzzer/pull/94

5.6 CVE-2016-6309: Use After Free for Large Message Sizes

CVE-2016-6309[16] je způsobeno chybou v OpenSSL verze 1.1.0a. Tato chyba může způsobit DoS útok nebo spuštění libovolného kódu od útočníka. Chyby lze docílit při odeslání větší zprávy (nad 16 kB) od klienta. Server neprovede přesunutí paměťového bloku po realokaci, což způsobí zápis do chybného místa paměti.

Na obrázku 5.7 je zobrazeno očekávané chování a chybné chování. Chybu lze vyvolat dvěma způsoby — odesláním ClientHello s rozšířením `client_hello_padding` a použitím velkého paddingu nebo pomocí ClientHello s upravenou hodnotu velikosti. Hodnoty pro velikost paddingu a ClientHello byly zjištěny experimentálně. S menšími hodnotami nebylo chybné chování reprodukováno.



Obrázek 5.7: Očekávané chování a chování CVE-2016-6309: Use After Free for Large Message Sizes

Pro tento test je vytvořen pull request do repositáře tlsfuzzeru[17].

5.6.1 Implementace

Pro toto CVE byly vytvořeny v souboru `scripts/test-cve-2016-6309.py` 3 testy. První je sanity test pro kontrolu podpory rozšíření `client_hello_padding` a kontroly správné komunikace serveru. Ve druhém testu je využito rozšíření `client_hello_padding`, kde jeho je velikost tohoto paddingu 21 798 B, což je nejmenší experimentáně zjištěná hodnota, která způsobuje SIGSEGV na straně serveru. Rozšíření je použito v ClientHello a není očekáváno v ServerHello, neboť toto rozšíření pouze přidává vycpávku za zprávu ClientHello. Vytvoření takto rozšířeného ClientHello je uvedeno v kódu na 5.9. Po ClientHello je očekáván normální průběh handshaku a ukončení spojení běžným způsobem.

```
padding_len = 21798
ext = {ExtensionType.client_hello_padding: PaddingExtension()
                        .create(padding_len)}
node = node.add_child(ClientHelloGenerator(ciphers,
                        extensions=ext))
```

Kód 5.9: Použití velkého paddingu v ClientHello s pomocí rozšíření client_hello_padding

[16] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6309
[17] https://github.com/tomato42/tlsfuzzer/pull/129

Ve třetím testu je upravena velikost ClientHello. Pro úpravu velikosti je využita funkce `fuzz_message` (viz. 5.10). Velikost dat v ClientHello je uvedena ve 3 B a uložena jako big-endian. Při úpravě velikosti není ještě zpráva obalena informacemi o protokolu, verzi SSL/TLS ani celkové délce zprávy, proto hodnota pro velikost začíná na druhém bajtu. Velikost je upravena na hodnotu 0x5555 (decimálně 21 845 B) — nejmenší velikost, která způsobuje SIGABRT. Tato úprava je provedena s pomocí parametru `substitutions`, která na určených indexech změní hodnoty na požadované. Po takto upraveném ClientHello je očekáváno okamžité ukončení spojení, protože server by správně po přijmutí takového ClientHello měl očekávat další části zprávy, aby jejich celková velikost byla rovna uvedené velikosti. Z naší strany ale další zprávy nejsou odeslány, takže server neodpoví v podobě ServerHello a spojení ukončí.

```
node = node.add_child(fuzz_message(
                      ClientHelloGenerator(ciphers),
                      substitutions={2:0x55, 3:0x55}))
```

Kód 5.10: Změna hodnoty velikosti v ClientHello

Kapitola 6

Testování SSL/TLS implementací

Funkčnost vytvořených útoků byla otestována na implementacích OpenSSL, GnuTLS a NSS. Z každé implementace bylo vybráno několik verzí, které byly otestovány na implementované útoky, které jsou uvedeny v kapitole 5.

6.1 Příprava pro testování

Využití tlsfuzzeru pro testovací účely vyžaduje instalaci knihovny tlslite-ng a následujících závislostí:

1. Python 2.6 nebo vyšší,
2. Python knihovna ecdsa[1].

Tlslite-ng a tlsfuzzer lze nainstalovat pomocí příkazu `$ python setup.py install` ve složce, ve které se tlslite-ng (tlsfuzzer) nachází.

Pro spuštění SSL/TLS serveru potřebujeme RSA certifikát a s ním související RSA klíč. Příkazem uvedeným v kódu 6.1 tuto dvojici vytvoříme s pomocí OpenSSL konzolové služby. Uvedeným příkazem vytvoříme žádost o tvorbu certifikátu standardu X.509 (*req -x509*), se kterou chceme vytvořit nový RSA klíč (*-newkey rsa*). Žádost o certifikát zašleme na naší adresu (*-subj /CN=localhost*), protože nechceme vytvořit platný certifikát s platností u certifikační autority. Přepínačem *-nodes* zakážeme šifrování výstupního klíče a přepínačem *-batch* vypneme všechny dotazy na informace pro tvorbu žádosti o vystavení certifikátu (u tvorby platné žádosti o vystavení certifikátu musíme zadat informace o státě, městě, organizaci atd.).

```
$ openssl req -x509 -newkey rsa -keyout rsa.key -out rsa.crt \
-subj /CN=localhost -nodes -batch
```

Kód 6.1: Tvorba RSA certifikátu a klíče

OpenSSL server lze spustit příkazem uvedeným v kódu 6.2, s pomocí kterého využijeme vytvořený RSA certifikát a klíč. S pomocí *-www* zapneme odpovídání na požadavky o aplikační data (např. "GET / HTTP/1.0\r\n\r\n"). Ve výchozím nastavení je tento server spuštěn na localhostu a naslouchá na portu 4433.

```
$ openssl s_server -key rsa.key -cert rsa.crt -www
```

Kód 6.2: Spuštění OpenSSL serveru

[1] https://pypi.python.org/pypi/ecdsa

Příkaz pro spuštění GnuTLS serveru je uveden v kódu 6.3. Soubor *gnutls-serv* se vyskytuje po přeložení ve složce *src*. GnuTLS serveru pomocí přepínače *–http* vynutíme, aby se choval jako HTTP server. Dále musíme nastavit port, na kterém bude naslouchat pomocí parametru *-p* a pomocí *–disable-client-cert* je vypnuto požadování certifikátu od klienta, protože vytvořené testy neposkytují certifikát ze strany klienta.

```
$ gnutls-serv --http -p 4433 --x509keyfile rsa.key \
--x509certfile rsa.crt --disable-client-cert
```

Kód 6.3: Spuštění GnuTLS serveru

Pro spuštění NSS serveru musíme vytvořit databázi certifikátů a klíčů a vložit do ní certifikát klienta. V kódu 6.4 je popsán postup pro tvorbu databáze, klientského certifikátu a následného vložení klientského certifikátu do vytvořené databáze.

Nejprve je vytvořena složka pro databázi. Dále je vytvořena nová databáze (přepínač *-N*) do vytvořené složky (*-d sql:db*), přístupná bez hesla (*–empty-password*). Pak je s pomocí OpenSSL nástroje vytvořen klientský certifikát pro localhost. Nakonec pomocí nástroje *pk12util* je klientský certifikát *rsa.p12* vložen do databáze s pomocí prázdného hesla (*-W ”*).

```
$ mkdir db
$ certutil -N -d sql:db --empty-password
$ openssl pkcs12 -export -passout pass: -out rsa.p12 \
-inkey rsa.key -in rsa.crt -name localhost
$ pk12util -i rsa.p12 -d sql:db -W ''
```

Kód 6.4: Tvorba databáze pro certifikáty a klíče, vytvoření klientského certifikátu a jeho propojení s databází

S vytvořenou databází lze spustit NSS server s pomocí příkazu 6.5. Na linuxové distribuci Fedora je nástroj pro spuštění serveru dostupný pomocí */usr/lib/nss/unsupported-tools/selfsev* nebo */usr/lib64/nss/unsupported-tools/selfsev*. Při spouštění je použita databáze a nastaven port pro naslouchání.

```
$ selfserv -d sql:./nssdb -p 4433 -n localhost
```

Kód 6.5: Spuštění NSS serveru

Jednotlivé testy lze spustit s pomocí interpretu jazyka Python. U všech testů lze určit IP adresu a port cíle, ale ve výchozím nastavení je test spuštěn pro server, který běží na localhostu a portu 4433. Informace o přepínačích testu lze získat při spuštění s *–help*.

6.2 Výsledky testování

Výsledky testování jsou uvedeny v Tabulce 6.1. V případě, že test proběhl v pořádku, pak byl označen zeleným polem s nápisem *Ok*. Pokud implementace ve výchozím stavu nepodporuje rozšíření, které je potřeba pro reprodukci útoku, pak je označen šedě. Implementace nepodporuje rozšíření, které jsou novějšího vydání než samotná implementace, nebo rozšíření, které nejsou ve většině případů potřebné pro běh serveru (např. rozšíření Heartbeat u TLS je u novějších implementací vypnuto, ale lze ho přidat nebo vynutit jeho přidání při instalaci a spuštění serveru). U testů, které proběhly neúspěšně je výsledek zobrazen jako *Chyba* v červeném políčku.

	CVE-2016-6305	SSL Death Alert	EtM renegotiation	Heartbleed	CVE-2016-7054	CVE-2016-6309
OpenSSL-1.0.0	Ok	Ok	-	-	-	-/Ok
OpenSSL-1.0.0t	Ok	Ok	-	-	-	-/Ok
OpenSSL-1.0.1	Ok	Chyba	-	Chyba	-	-/Ok
OpenSSL-1.0.1u	Ok	Chyba	-	Ok	-	-/Ok
OpenSSL-1.0.2	Ok	Chyba	-	Ok	-	-/Ok
OpenSSL-1.0.2k	Ok	Ok	-	Ok	Ok	-/Ok
OpenSSL-1.1.0	Ok	Chyba	Chyba	-	Chyba	Ok
OpenSSL-1.1.0a	Ok	Ok	Chyba	-	Chyba	Chyba
OpenSSL-1.1.0e	Ok	Ok	Ok	-	Ok	Ok
GnuTLS-3.3.27	Ok	Ok	-	-	-	Ok
GnuTLS-3.5.0	Ok	Ok*	Ok	-	Ok	Ok
GnuTLS-3.5.11	Ok	Ok	Ok	-	Ok	Ok
NSS-3.15.1 (Fedora 19)	Ok	Ok*	-	-	-	Ok
NSS-3.27.0 (Fedora 25)	Ok	Ok*	-	-	Ok	Ok
NSS-3.30.2	Ok	Ok*	-	-	Ok	Ok

* Spojení není ukončeno ani po obdržení velkého množství upozornění, ale je s upozorněními zacházeno správně (nedochází k velkému vytížení CPU)

Tabulka 6.1: Výsledky testování vybraných SSL/TLS implementací

V Tabulce 6.1 je vidět, že CVE-2016-6305 se nevyskytuje v žádném testovaném serveru. Tato chyba by se měla vyskytovat na straně klienta, proto bylo očekáváno, že u žádné z implementací nebude nalezena.

Výsledky testování CVE-2016-8610, neboli SSL Death Alert, nebyly závislé na výsledku testu (zda-li byl test úspěšný nebo ne), ale na chování jednotlivých serverů, protože žádné RFC nedefinuje správné chování v této situaci. Implementace, u kterých je uvedeno očekávané chování, spojení při větších počtech upozornění (např. 10 000) buď ukončili nebo zaslali upozornění úrovně *fatal* s následným ukončením spojení. Za správné chování, které je v tabulce označeno *Ok* s hvězdičkou, bylo považováno přijímání libovolného množství upozornění bez způsobení velkého vytížení CPU. Chybné chování bylo zpracování libovolného množství upozornění a nárůst vytížení CPU blížící se k hodnotě 100 % po dobu zpracování upozornění.

CVE-2017-3733: Encrypt-Then-MAC renegotiation crash nebylo možné u mnoha implementací otestovat, protože nepodporovaly rozšíření Encrypt-Then-MAC. Testy potvrzují výskyt této chyby ve verzích OpenSSL-1.1.0 a OpenSSL-1.1.0a. V novějších verzích byla tato chyba odstraněna.

Chyba Heartbleed byla objevena pouze ve verzi OpenSSL-1.0.1. Starší verze OpenSSL rozšíření Heartbeat nepodporují a v nejnovější verzi z řady OpenSSL-1.0.1 (OpenSSL-1.0.1u), nejstarší a nejnovější verzi OpenSSL-1.0.2 (OpenSSL-1.0.2 a OpenSSL-1.0.2k) je tato chyba již opravena. Poslední řada OpenSSL-1.1.0 nemá Heartbeat rozšíření povoleno ve výchozím nastavení.

CVE-2016-7054 u starších implementací nelze otestovat, protože tyto verze nepodporují šifrovací sady **-CHACHA20-POLY1305*. U novějších verzí, až s výjimkou implementace OpenSSL-1.1.0 a OpenSSL-1.1.0a, nebylo objeveno chybné chování.

V testu pro CVE-2016-6309 jsou dva testy — jeden zašle velkou zprávu ClientHello s pomocí rozšíření `client_hello_padding` a druhý pouze upraví hodnotu velikosti ClientHello. U starších implementací OpenSSL není podporování rozšíření `client_hello_padding`,

proto první test nebylo možno otestovat pro tyto verze, ale při změně velikosti ClientHello nebyl nalezen žádný problém. Pouze ve verzi OpenSSL-1.1.0a se vyskytuje chyba u obou testu. V ostatních implementacích nebyla chyba objevena.

Z výsledků testování lze vyvodit, že všechny CVE se vyskytují pouze v implementacích, s nimiž je toto CVE spojeno, a byly korektně opraveny. Ostatní implementace chybu buď vůbec neobsahují nebo ji není možné otestovat z důvodů nedostatečné podpory určitých rozšíření SSL/TLS. V implementacích GnuTLS a NSS nebyly objeveny žádné chyby, pouze zpracovávají libovolný počet upozornění, bez zátěže CPU, čímž nedochází k DoS útoku. Heartbleed chyba nebyla nalezena v novějších implementacích. Většina nových implementací rozšíření Heartbeat ve výchozím nastavení (u TLS) nepovolují, protože využití tohoto rozšíření je mířeno hlavně na DTLS.

Kapitola 7

Závěr

V této práci byly představeny kryptografické protokoly SSL/TLS, které zajišťují bezpečnou komunikaci na internetu, jejich podprotokoly (handshake, Change Cipher Spec, alert a application data protokol) a rozšíření, pomocí kterých lze přidat do TLS protokolu nové funkce, které nejsou v základní specifikaci. Dále práce seznámila s implementací SSL/TLS protokolu, hlavními zástupci těchto implementací — OpenSSL, GnuTLS a NSS — a implementací knihovny tlslite-ng.

Byly identifikovány základní kategorie útoků na SSL/TLS protokol (útoky na PKI, útoky využívající chyb ve webovém prohlížeči a HTTP, útoky na protokol a útoky na chyby v implementaci) a popsáni někteří zástupci z jednotlivých kategorií. Z těchto útoků byly vybrány útoky vhodné pro reprodukci pomocí tlsfuzzeru — fuzz testovací knihovny pro SSL/TLS protokol. Tyto útoky byly úspěšně implementovány ve formě tlsfuzzer testů a otestovány nad třemi zástupci implementací SSL/TLS protokolu — OpenSSL, GnuTLS a NSS. Z těchto implementací byly vybrány starší i nové verze, aby bylo ověřeno, že vybrané útoky jsou spojeny pouze s verzemi, ve kterých byly CVE nalezeny, a v novějších verzích již tyto chyby byly opraveny, popř. se nevyskytují v jiných implementacích. Experimenty byl ověřen výskyt daných CVE v odpovídajících verzích a současně potvrdili jejich opravu v následujících verzích. Existence těchto testů i pro již opravené chyby je důležitá, abychom zamezili případnému znovu zavedení chyb do implementace a aby se předchozí katastrofy (jako byl Heartbleed) již neopakovaly.

Výsledky práce jsou již z části integrovány v upstreamu tlsfuzzeru a knihovně tlslite-ng (jeden schválený test, tři otevřené pull requesty s testy a jeden otevřený pull request s rozšířením tlslite-ng). Pokračování práce by se pak mohlo zaměřit na sofistikovanější testování protokolu SSL/TLS nebo případnou reprodukci dalších útoků.

Literatura

[1] CVE - About CVE. [Online; navštíveno 12.5.2017].
URL http://cve.mitre.org/about/

[2] Networking 101: Transport Layer Security (TLS) - High Performance Browser Networking (O'Reilly). [Online; navštíveno 4.2.2017].
URL https://hpbn.co/transport-layer-security-tls/

[3] Microsoft Security Bulletin MS01-017 - Critical. 2001, [Online; navštíveno 4.2.2017].
URL https://technet.microsoft.com/library/security/ms01-017

[4] cryptography - SSL is half symmetric and half asymmetric? - Stack Overflow. 2010, [Online; navštíveno 11.5.2017].
URL http://stackoverflow.com/questions/3649689/ssl-is-half-symmetric-and-half-asymmetric

[5] CVE - CVE-2014-0160. 2014, [Online; navštíveno 23.3.2017].
URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[6] Ari Takanen, C. M., Jared DeMott: *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008, ISBN 978-1596932142.

[7] Dierks, T.: Tim Dierks: Security Standards and Name Changes in the Browser Wars. 2014, [Online; navštíveno 4.2.2017].
URL http://tim.dierks.org/2014/05/security-standards-and-name-changes-in.html

[8] Dierks, T.; Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008, [Online; navštíveno 4.2.2017].
URL http://www.rfc-editor.org/rfc/rfc5246.txt

[9] Guan, Z.: Crypto With OpenSSL. 2009, [Online; navštíveno 25.3.2017].
URL https://www.slideshare.net/guanzhi/crypto-with-openssl

[10] Gutmann, P.: Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366, RFC Editor, September 2014, [Online; navštíveno 12.4.2017].
URL http://www.rfc-editor.org/rfc/rfc7366.txt

[11] Hertzfeld, A.: Folklore.org: Monkey Lives. 1983, [Online; navštíveno 25.2.2017].
URL http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt

[12] Kario, H.: Huber Kario: Testing TLS (přednáška). Brno, DevConf.cz 2015. 2015, [Online; navštíveno 13.2.2017].
URL https://github.com/tomato42/tlsfuzzer/blob/master/docs/devconf2015-kario-slides.pdf

[13] Kario, H.: Huber Kario: TLS Test Framework (přednáška). Melbourne, Ruxcon 2015. 2015, [Online; navštíveno 8.4.2017].
URL https://github.com/tomato42/tlsfuzzer/blob/master/docs/ruxcon2015-kario-slides.pdf

[14] Kario, H.: Huber Kario: TLS Test Framework (přednáška). Brusel, FOSDEM'17. 2017, [Online; navštíveno 13.2.2017].
URL https://github.com/tomato42/tlsfuzzer/blob/master/docs/fosdem2017-kario-slides.pdf

[15] Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. 2017, [Online; navštíveno 12.5.2017].
URL https://tlswg.github.io/tls13-spec/

[16] Ristić, I.: *Bulletproof SSL and TLS*. Feisty Duck, 2014, ISBN 978-1907117046.

[17] Sheffer, Y.; Holz, R.; Saint-Andre, P.: Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). RFC 7457, RFC Editor, February 2015, [Online; navštíveno 23.3.2017].
URL http://www.rfc-editor.org/rfc/rfc7457.txt

[18] Vänskä, O.: A Finnish man created this simple email account - and received Microsoft's security certificate - Tivi. 2015, [Online; navštíveno 4.2.2017].
URL http://www.tivi.fi/Kaikki_uutiset/2015-03-18/A-Finnish-man-created-this-simple-email-account---and-received-Microsofts-security-certificate-3217662.html

[19] Wikipedia: American fuzzy lop (fuzzer) — Wikipedia, The Free Encyclopedia. 2017, [Online; navštíveno 7.4.2017].
URL https://en.wikipedia.org/w/index.php?title=American_fuzzy_lop_(fuzzer)&oldid=779747527

[20] Wikipedia: Comparison of TLS implementations — Wikipedia, The Free Encyclopedia. 2017, [Online; navštíveno 25.3.2017].
URL https://en.wikipedia.org/w/index.php?title=Comparison_of_TLS_implementations&oldid=778225413

[21] Wikipedia: Fuzzing — Wikipedia, The Free Encyclopedia. 2017, [Online; navštíveno 4.2.2017].
URL https://en.wikipedia.org/w/index.php?title=Fuzzing&oldid=780193478

[22] Wikipedia: GnuTLS — Wikipedia, The Free Encyclopedia. 2017, [Online; navštíveno 25.3.2017].
URL https://en.wikipedia.org/w/index.php?title=GnuTLS&oldid=772920480

[23] Wikipedia: Network Security Services — Wikipedia, The Free Encyclopedia. 2017, [Online; navštíveno 25.3.2017].

URL https://en.wikipedia.org/w/index.php?title=Network_Security_Services&oldid=774657022

[24] Wikipedia: OpenBSD Cryptographic Framework — Wikipedia, The Free Encyclopedia. 2017, [Online; navštíveno 25.3.2017].
URL https://en.wikipedia.org/w/index.php?title=OpenBSD_Cryptographic_Framework&oldid=763054334

[25] Wikipedia: OpenSSL — Wikipedia, The Free Encyclopedia. 2017, [Online; navštíveno 4.2.2017].
URL https://en.wikipedia.org/w/index.php?title=OpenSSL&oldid=779665728

[26] Wikipedia: Smart card — Wikipedia, The Free Encyclopedia. 2017, [Online; navštíveno 25.3.2017].
URL
https://en.wikipedia.org/w/index.php?title=Smart_card&oldid=778638094

[27] Wikipedia: Transport Layer Security — Wikipedia, The Free Encyclopedia. 2017, [Online; navštíveno 4.2.2017].
URL https://en.wikipedia.org/w/index.php?title=Transport_Layer_Security&oldid=779696876

[28] Wikipedie: Testování softwaru — Wikipedie: Otevřená encyklopedie. 2016, [Online; navštíveno 4.2.2017].
URL https://cs.wikipedia.org/w/index.php?title=Testov%C3%A1n%C3%AD_softwaru&oldid=14423939

Přílohy

Příloha A

Obsah přiloženého paměťového média

- **src/tlsfuzzer/** — git repozitář nástroje tlsfuzzer,
- **src/tlslite-ng/** — git repozitář knihovny tlslite-ng,
- **latex/** — zdrojové soubory textu,
- **text.pdf** — PDF soubor textu práce,
- **README** — dokument obsahující instrukce pro instalaci a spuštění, informace o git repozitářích a obsah CD,
- **LICENCE** — licence k užití práce.