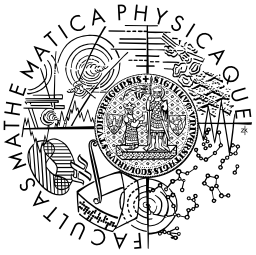


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

**BAKALÁŘSKÁ PRÁCE**

Jan Oupický

# Kryptografické útoky na TLS protokol

Katedra algebry

| | |
|---|---|
| Vedoucí bakalářské práce: | doc. RNDr. Jiří Tůma, DrSc. |
| Studijní program: | Matematika |
| Studijní obor: | Matematika pro informační technologie |

Praha 2019

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Chtěl bych poděkovat svému vedoucímu práce doc. RNDr. Jiřímu Tůmovi, DrSc. a konzultantovi Mgr. Adolfu Středovi za přátelské jednání, důležité rady a v neposlední řadě za umožnění zpracování tohoto tématu, které mě velice zajímá.

Název práce: Kryptografické útoky na TLS protokol

Autor: Jan Oupický

Katedra: Katedra algebry

Vedoucí bakalářské práce: doc. RNDr. Jiří Tůma, DrSc., Katedra algebry

Abstrakt: Cílem této práce je seznámit čtenáře s protokolem TLS (dříve SSL) a vybranými útoky na tento protokol. V první části práce si zavedeme nezbytné kryptografické definice použité v následujících kapitolách. V druhé části si stručně představíme historii protokolů TLS a SSL, a poté se blížeji podíváme na to, jak fungují. Poslední část se týká rozboru vybraných kryptograficky zajímavých útoků (Padding oracle na CBC mód, POODLE, BEAST a CRIME) na protokoly TLS a SSL.

Klíčová slova: TLS, šifrování, kryptoanalýza, postranní kanály

Title: Crytographic attacks on TLS protocol

Author: Jan Oupický

Department: Department of Algebra

Supervisor: doc. RNDr. Jiří Tůma, DrSc., Department of Algebra

Abstract: The aim of this work is to introduce the reader to the protocol TLS and a few selected attacks against the protocol. In the first part we will define the necessary cryptographic definitions used in the following chapters. In the second part we will briefly talk about the history of procotols TLS and SSL and then we will closely look into how they work. The last part is about the analysis of the chosen cryptographically interesting attacks (Padding oracle on CBC mode, POODLE, BEAST and CRIME) against protocols TLS and SSL.

Keywords: TLS, encryption, cryptanalysis, side-channel

# Obsah

# Úvod

Šifrování je v dnešním světě všude kolem nás, aniž by si to většina lidí uvědomovala. Pokud chcete obyčejnému člověku vysvětlit, co vlastně studujete, tak se nejvíce nabízí příklad s internetovým bankovnictvím. Obvykle se k internetovému bankovnictví připojujeme právě pomocí webového prohlížeče nebo aplikace na chytrém telefonu. Během připojení se potřebujeme nějakým způsobem prokázat, abychom se dostali ke svému účtu. To se obvykle dělá pomocí hesla nebo certifikátu. Tyto autentifikátory by měl znát pouze majitel účtu, v ideálním světě ani banka.

Potřebujeme, aby tato data nemohl přečíst kdokoliv. Na pomoc přichází protokol TLS („Transport Layer Security“), dříve SSL („Secure Sockets Layer“). Ještě specifičtěji ve většině případů protokol HTTPS („Hypertext Transfer Protocol Secure“), jehož součástí je právě protokol TLS/SSL.

V této práci se nejprve seznámíme se základními kryptografickými definicemi, zkratkami a značením, které budeme v celé práci používat.

Následně se budeme zabývat protokolem TLS/SSL. Stručně popíšeme jeho historii vývoje. Dále se stručně podíváme na to, jak rychle reálný svět reaguje na objevy nových bezpečnostních hrozeb souvisejících s protokolem TLS/SSL.

V další části si podrobně vysvětlíme, jak tento protokol funguje, jelikož se nám to bude hodit pro porozumění průběhu vybraných útoků. Nejprve se budeme zabývat protokolem SSL 3.0, který tvoří základ pro novější verze TLS. Popíšeme si jeho 2 hlavní komponenty „Record protocol“ a „Handshake protocol“. Následně si vysvětlíme, jak moc se liší SSL 3.0 od novějších verzí TLS.

Poslední část práce se týká bližšího popisu vybraných útoků na protokol TLS a SSL. Během více než 20 let používání protokolu TLS/SSL bylo objeveny desítky různých útoků. Z nich jsme vybrali ty kryptograficky nejzajímavější a vysvětlili jsme, jaké specifické zranitelnosti v protokolu využívají. Autoři útoků často nezpracovávají popisy daných útoků moc dopodrobna, proto si útoky rozebereme podrobněji a vysvětlíme, jak fungují. Útoky byly vybrány celkem 4.

První útok „Padding oracle na CBC mód“ je teoretického charakteru, ale stal se základním kamenem pro ostatní objevené útoky. Druhý útok „POODLE“ je v podstatě idea prvního útoku aplikovaná na reálnou situaci. Třetí útok „BEAST“ využívá „školácké“ chyby v designu protokolu týkající se inicializačních vektorů. Poslední útok „CRIME“ využívá podpory kompresních funkcí v protokolu.

# 1. Slovníček a zkratky

Některé pojmy nemají dostatečně výstižné pojmenování v češtině, proto zde uvádíme přehled cizích slov, která se vyskytují v práci, a jejich význam.

Dále pro zjednodušení a lepší srozumitelnost budeme používat zkratky, které zde uvedeme.

## 1.1 Slovníček

- session: v doslovném překladu znamená „zasedání“ nebo „sezení“. Tento termín je používaný pro časové období, kdy spolu komunikují dvě strany.
- cipher spec: označení pro dvojici algoritmů, které se používají pro autentifikaci a šifrování dat v protokolu TLS/SSL.
- cipher suite: označení pro daný cipher spec společně s algoritmem pro výměnu klíčů, který se dá použít také k podepisování.
- cache: označení pro dočasnou paměť počítače.
- autentifikace: proces, při kterém si ověříme identitu dané entity.
- autentizace: proces, který zaručuje integritu dat a zároveň autentifikuje data.
- plaintext (otevřený text): označení pro data, která nejsou šifrována.
- ciphertext (šifrový text): označení pro data, která vznikla z plaintextu po zašifrování.
- cookie: označení pro data, která obvykle slouží k identifikaci uživatele na webové stránce prostřednictvím protokolu HTTP(S).

## 1.2 Zkratky

- SSL: „Secure Sockets Layer“
- TLS: „Transport Layer Security“
- HTTP: „Hypertext Transfer Protocol“
- HTTPS: „Hypertext Transfer Protocol Secure“
- CBC: „Cipher Block Chaining“
- ECB: „Electronic Codebook“
- CFB: „Cipher Feedback“
- GCM: „Galois/Counter Mode“
- IV: „Inicializační vektor“

- MAC: „Message Authentication Code“
- HMAC: „Hash-based Message Authentication Code“
- PKCS: „Public Key Cryptographic Standards“
- DH: „Diffie-Hellman“

# 2. Definice

V této kapitole se seznámíme s definicemi, které budeme používat.

**Definice 1** (Symetrická šifra)**.** *Nechť* $\boldsymbol{K}$ *množina klíčů,* $\boldsymbol{P}$ *je množina otevřených textů a* $\boldsymbol{C}$ *množina šifrových textů. Mějme zobrazení*

$$E : \mathbf{K} \times \mathbf{P} \to \mathbf{C}$$
$$\textit{takové že,}$$
$$(\exists D : \mathbf{K} \times \mathbf{C} \to \mathbf{P}) : (\forall k \in \mathbf{K}), (\forall x \in \mathbf{P}) : D(k, E(k, x)) = x$$

*Zobrazení E se nazývá šifrovací funkce (algoritmus) a zobrazení D se nazývá dešifrovací funkce (algoritmus). Výrazem „symetrická šifra“ označujeme tuto dvojici (E, D).*

**Značení.** *Pro pevné* $k \in \mathbf{K}$ *se někdy používá značení* $\forall x \in \mathbf{P} : E_k(x) \coloneqq E(k, x)$ *a* $\forall y \in \mathbf{C} : D_k(y) \coloneqq D(k, y)$.

**Značení.** *Nechť* $n \in \mathbb{N}$*, pak* $\{0,1\}^n$ *značí množinu všech posloupností nul a jedniček (bitů) délky n.*

**Značení.** $\{0,1\}^*$ *značí množinu všech posloupností nul a jedniček (bitů) konečné délky.*

**Definice 2** (Bloková šifra)**.** *Nechť* $n$,$k \in \mathbb{N}$*. Bloková šifra je symetrická šifra, pro kterou platí*

$$E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$$
$$D : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$$

*kde n nazýváme délkou bloku a k nazýváme délkou klíče.*

Z definice blokové šifry vyplývá, že je v této podobě použitelná pouze na šifrování/dešifrování zprávy délky $n$. Samozřejmě ale chceme šifrovat data libovolné délky, a proto existují tzv. *operační módy*. Operační mód je algoritmus, který používá zobrazení $E$ a $D$ a umožňuje šifrování zpráv libovolných délek. Nejznámějšími typy jsou módy ECB, CFB, GCM a CBC. V další práci budeme potřebovat pouze znalost CBC, takže si představíme blíže pouze tento.

**Značení.** *„*$||$*“ značí operaci zřetězení. Budeme tím obvykle značit zřetězení bitů nebo bytů.*

**Značení.** *„*$\oplus$*“ značí logickou operaci XOR provedenou na bity řetězců.*

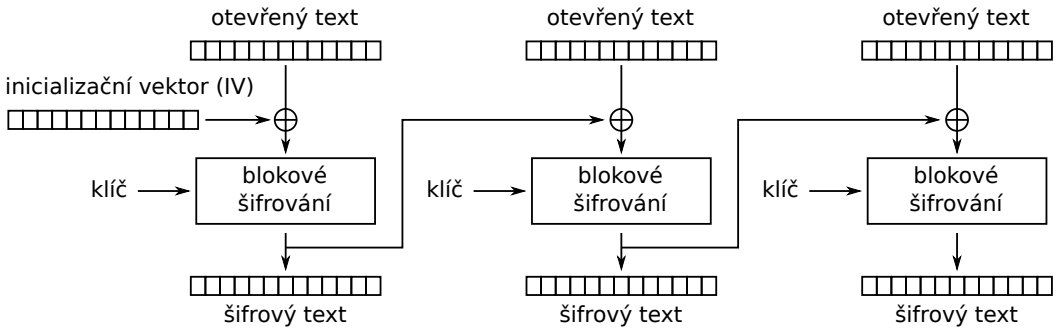
**Značení.** *Nechť* $x \in \{0,1\}^n$*. Poté* $|x| = n$*. V tomto případě by to znamenalo, že* $|\cdot|$ *značí délku x v bitech. V závislosti na kontextu tato operace bude značit obdobně délku v bytech.*

**Definice 3** (CBC mód)**.** *Nechť $E$, $D$ je bloková šifra, $N \in \mathbb{N}$ je délka bloku v bitech, $n \in \mathbb{N}$, $x = x_1||x_2||\ldots||x_n, \forall i \in \{1,\ldots,n\} : |x_i| = N$ je plaintext, $IV \in \{0,1\}^N$ je inicializační vektor, $k \in \mathbb{N}$ je délka klíče v bitech, $K \in \{0,1\}^k$ je klíč. Výsledný ciphertext po aplikování šifrování v módu CBC je tvaru $y = y_0||y_1||y_2||\ldots||y_n$ a byl spočítán následovně:*

$$y_0 = IV$$
$$\forall i \in \{1,\ldots,n\} : y_i = E_k(y_{i-1} \oplus x_i)$$

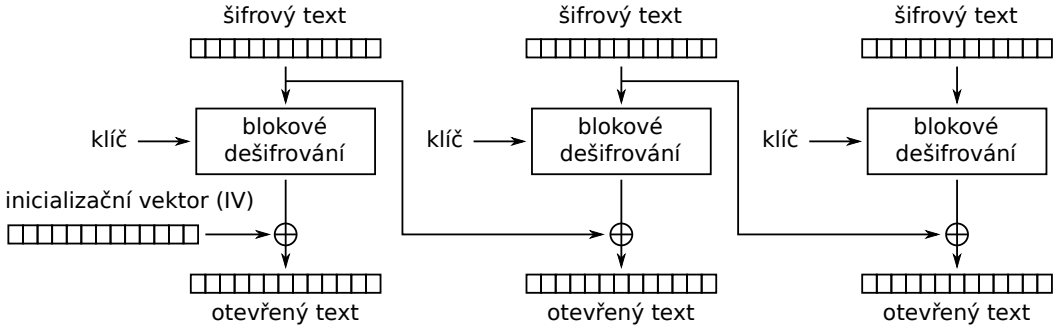
*Dešifrování probíhá obdobně:*

$$\forall i \in \{1,\ldots,n\} : x_i = D_k(y_i) \oplus y_{i-1}$$




Šifrování v režimu řetězení šifrových bloků (CBC)

Obrázek 2.1: Schéma šifrování v módu CBC.
Zdroj: `https://commons.wikimedia.org/wiki/File:CBC_encryption_cs.svg`




Dešifrování v režimu řetězení šifrových bloků (CBC)

Obrázek 2.2: Schéma dešifrování v módu CBC.
Zdroj: `https://commons.wikimedia.org/wiki/File:CBC_decryption_cs.svg`

*Poznámka.* IV neboli inicializační vektor nemusí být tajný, ale je důležité, aby byl nepředvídatelný neboli volen náhodně. Jestliže tomu tak není, tak CBC mód není bezpečný, viz například BEAST.

*Poznámka.* Plaintext obvykle nemá délku, která je násobkem délky bloku. Proto se rozšiřuje na nějaký násobek délky bloku, aby šla data zašifrovat. Tomuto rozšíření se říká *padding* nebo aplikování paddingu.

**Definice 4** (PKCS#7 padding)**.** *Nechť $N \in \mathbb{N}$ je délka bloku v bytech, $x \in \{0,1\}^* : n > 0$, $x = x_0||x_1||\ldots||x_n$, $\forall i \in \{0,\ldots,n-1\} : |x_i| = N, |x_n| \leq N$ jsou data, na které chceme aplikovat padding. Data po aplikování paddingu ($x'$) jsou tvaru:*

1. *Pokud $|x_n| = N$, tak $x' = x_0||x_1||\ldots||x_n||y$, kde $y = \underbrace{N||N||\ldots||N}_{N}$. Neboli je přidán na konec nový blok, který obsahuje N bytů, které mají hodnotu N. Z toho plyne, že padding je dobře definován pokud $N < 256$.*

2. *Pokud $|x_n| < N$, tak $x' = x_0||x_1||\ldots||x_{n-1}||y$, kde $k = N - |x_n| > 0$ a $y = x_n||\underbrace{k||k||\ldots||k}_{k}$. Neboli je poslední blok doplněn k byty, které mají hodnotu k.*

**Definice 5** („SSL 3.0“ padding)**.** *Nechť $N \in \mathbb{N}$ je délka bloku v bytech, $x \in \{0,1\}^* : n > 0$, $x = x_0||x_1||\ldots||x_n$, $\forall i \in \{0,\ldots,n-1\} : |x_i| = N, |x_n| \leq N$ jsou data, na které chceme aplikovat padding. Data po aplikování paddingu ($x'$) jsou tvaru:*

1. *Pokud $|x_n| = N$, tak $x' = x_0||x_1||\ldots||x_n||y$, kde $y = b_1||b_2||\ldots||b_{N-1}||N-1$, kde $\forall i \in \{1, N-1\} : b_i$ jsou libovolné byty.*

2. *Pokud $|x_n| < N$, tak $x' = x_0||x_1||\ldots||x_{n-1}||y$, kde $k = N - |x_n| > 0$ a $y = x_n||b_1||b_2||\ldots||b_{k-1}||k-1$, kde $\forall i \in \{1, k-1\} : b_i$ jsou libovolné hodnoty.*

*Stručně řečeno: Blok je doplněn libovolnými byty na potřebnou délku a poslední byte obsahuje délku paddingu.*

*Poznámka.* Proudová šifra se od blokové liší v tom, že její vstup je libovolné délky. Součástí proudové šifry je algoritmus, který má na vstupu klíč pevné délky, a ten vygeneruje tzv. keystream neboli „proud hesla“. Keystream je nagenerován pro potřebnou délku vstupu. Výstupem šifry je obvykle XOR vstupu a keystreamu.

**Definice 6** (Zanedbatelná funkce [1, str. 39])**.** *$\epsilon : \mathbb{N} \to \mathbb{R}$ je zanedbatelná funkce, pokud pro každý polynom $p$ existuje $n_p \in \mathbb{N}$ takové, že $\forall n \in \mathbb{N} : n \geq n_p$ platí*

$$\epsilon(n) < \frac{1}{p(n)}$$

*Poznámka.* Zanedbatelná funkce nám může posloužit jako měřítko obtížnosti problému. Příkladem je definice jednosměrné funkce, kde obtížnost invertovatelnosti se přeloží jako zanedbatelná pravděpodobnost nalezení vzoru pro $x$ z uniformního rozdělení a libovolný polynomiální algoritmus. Podobně budeme přistupovat i k dalším problémům, byť, z důvodu přehlednosti a dostatečnosti intuitivní představy, tento překlad již nebudeme explicitně uvádět.

**Definice 7** (Jednosměrná funkce [1, str. 42])**.** *Funkce* $f : \{0,1\}^* \to \{0,1\}^*$ *je jednosměrná, pokud je:*

1. *spočítatelná v polynomiálním čase*
2. *těžko invertovatelná neboli pro každý pravděpodobnostní polynomiální algoritmus A platí*

$$P[A(f(x)) \in f^{-1} \circ f(x)] < \epsilon(n)$$

*Kde* $\epsilon(\cdot)$ *je zanedbatelná funkce a* $x$ *je uniformě náhodně vybraný řetězec délky* $n$*.*

**Definice 8** (Kryptografická hašovací funkce)**.** *Nechť* $n \in \mathbb{N}$*. Kryptografická hašovací funkce* $H$ *je zobrazení, pro které platí*

$$H : \{0,1\}^* \to \{0,1\}^n$$

*a má následující vlastnosti:*

1. *Pre-image resistance: H je jednosměrná funkce. Tedy pro dané h je „těžké“ nalézt m tž.* $H(m) = h$*.*
2. *Second pre-image resistance: Pro dané* $m_1$ *je „těžké“ nalézt* $m_2$ *tž.* $H(m_1) = H(m_2)$*.*
3. *Odolnost vůči nalezní kolize: Je „těžké“ nalézt pár* $(m_1,m_2)$ *tž.* $H(m_1) = H(m_2)$*. Takový pár nazýváme kolize funkce H.*

*Poznámka.* Jelikož nebudeme s hašovací funkcí pracovat v obecnější informatické definici, která mj. neklade požadavky na jednosměrnost funkce, bude vždy dále hašovací funkcí myšlena funkce ve smyslu předchozí definice.

**Definice 9** (MAC)**.** *Nechť* $l, n \in \mathbb{N}$*,* $m \in \{0,1\}^*$ *zpráva. MAC je trojice algoritmů* $G, T, V$*, kde*

- $G$ *je algoritmus generující klíč* $k \in \{0,1\}^n$
- $T$ *je algoritmus vytvářející tzv. „tag“* $t \in \{0,1\}^l$*, který autentizuje* $m$ *v závislosti na klíči* $k$ *neboli* $t = T(k,m)$
- $V$ *je algoritmus pro verifikaci tagu. Jeho výstupem je buď* 1 *(tag je validní) nebo* 0 *(tag není validní). Vstupy* $V$ *jsou* $t, k, m$*.*

*Poznámka.* MAC je obvykle konstruován tak, aby tag byl nerozeznatelný od náhodné posloupnosti bitů.

*Poznámka.* Slovem MAC se často označuje výstup (tag) algoritmu $T$.

*Poznámka.* Algoritmy $G, V, T$ v MAC mají polynomiální složitost.

*Poznámka.* Výrazem „útočník“ chápeme intuitivně nějaký abstraktní algoritmus.

**Definice 10** (Bezpečný MAC)**.** *Nechť $(G, T, V)$ je MAC. Nechť $G$ vygeneruje $k \in \{0,1\}^n$. Řekneme, že MAC je bezpečný, pokud pro každého útočníka $A$ platí*

$$P[A^{T(k,\cdot)} \to (m,t), m \notin Q(A^{T(k,\cdot)}) : V(t,k,m) = 1] < \epsilon(n)$$

*Kde $A^{T(k,\cdot)} \to (m,t)$ značí to, že útočník $A$ po komunikaci s orákulem $T(k,\cdot)$ vydal výstup $(m,t)$, kde $m$ je zpráva a $t$ její tag. $Q(A^{T(k,\cdot)})$ značí množinu zpráv, které $A$ poslal orákulu $T(k,\cdot)$. $T(k,\cdot)$ chápejme jako algoritmus $T$, který dostává na vstupu $m$ a vrací $t = T(k,m)$. $\epsilon(\cdot)$ je zanedbatelná funkce.*

*Poznámka.* Populární konstrukcí MAC je tzv. HMAC, který pro výpočet tagu využívá kryptograficky bezpečné hašovací funkce.

*Poznámka.* Bezztrátová kompresní funkce $f$ je zobrazení, pro které platí

$$f : \{0,1\}^* \to \{0,1\}^*$$

a pro většinu[1] vstupů $x$ délky $n \in \mathbb{N}$ platí, že výstup $y = f(x)$ je délky $m \in \mathbb{N}$ a $m < n$. Dále se předpokládá, že existuje polynomiální algoritmus pro výpočet $y = f(x)$ a $x = f^{-1}(y)$. Funkce $f$ musí být prostá (bezztrátovost).

**Definice 11** (Rodina pseudonáhodných funkcí (PRF family))**.** *Rodinu (množinu) funkcí $\{f_s : \{0,1\}^{|s|} \to \{0,1\}^{|s|}\}_{s \in \{0,1\}^*}$ nazveme pseudonáhodnou, pokud platí:*

1. *$\forall s \in \{0,1\}^*, \forall x \in \{0,1\}^{|s|}$ je hodnota $f_s(x)$ spočítatelná v polynomiálním čase*

2. *pro každého útočníka $A$, který se snaží rozlišit instanci $f_s$ od náhodné funkce $RF$, a $\forall n \in \mathbb{N}$ platí*

$$P[A^{f_s(\cdot)} = 1] - P[A^{RF(\cdot)} = 1] < \epsilon(n) \tag{2.1}$$

   *Kde $s$ je uniformě náhodně vybraný řetězec délky $n$, $A^{f_s(\cdot)}$ značí výstup útočníka po komunikaci s funkcí $f_s$, $RF(\cdot)$ je uniformě náhodně vybraná funkce z $\{0,1\}^n$ do $\{0,1\}^n$ , $A^{RF(\cdot)}$ značí výstup útočníka po komunikaci s funkcí $RF$. $\epsilon(\cdot)$ je zanedbatelná funkce.*

*Jinými slovy: neexistuje polynomiální algoritmus, který by dokázal rozlišit instanci $f_s$ od náhodně vybrané funkce stejného typu.*

**Definice 12** (Pseudonáhodná funkce (PRF))**.** *Nechť*
*$F := \{f_s : \{0,1\}^{|s|} \to \{0,1\}^{|s|}\}_{s \in \{0,1\}^*}$ je rodina pseudonáhodných funkcí. Každou funkci $f \in F$ nazveme pseudonáhodnou funkcí.*

[1] Výraz „většinu“ chápeme intuitivně jako v průměru. Zřejmě není možné vytvořit perfektní bezztrátovou kompresní funkci, která pro každý vstup dokázala vytvořit výstup kratší délky.

# 3. Protokol TLS/SSL

## 3.1 Historie

Protokol SSL začal být vyvíjen firmou Netscape Communications Corporation v první polovině 90. let 20. století. První verze s číslem 1.0 nikdy nebyla zveřejněna, jelikož zaměstnanci Netscape Communications Corporation objevili bezpečnostní chyby ještě předtím, než byl protokol představen veřejnosti. Vylepšenou verzí měla být verze 2.0, která také nevydržela příliš dlouho.

První „oficiální“ verze ve smyslu, že lze dohledat její RFC[1], byla verze s označením 3.0 (RFC 6101) vydaná v roce 1996. Tato verze tvoří základ pro její nástupce.

Často je v souvislosti s TLS a SSL kladena otázka: „Jaký je rozdíl mezi SSL a TLS?“. Odpověď na ni není zajímavá. Díky růstu popularity internetu se Netscape a Microsoft[2] dohodly, že vývoj protokolu SSL předají organizaci IETF[3]. V rámci této dohody se rozhodlo, že se již nebude používat název SSL, jelikož je spojen s firmou Netscape a IETF plánovala udělat pár změn v SSL 3.0. Tak vzniklo v roce 1999 TLS 1.0. Ve specifikaci TLS 1.0 je zmíněno, že změny oproti SSL 3.0 nejsou dramatické, ale jsou významné z hlediska bezpečnosti.

TLS bylo od svého vzniku v roce 1999 upraveno zatím 3krát.

Hlavní změna v TLS 1.1 oproti 1.0 je zlepšení bezpečnosti CBC módu. V protokolech TLS 1.0 a SSL 3.0 se používá tzv. implicitní IV. Použití implicitního IV znamená, že se jako inicializační vektor pro každou novou zprávu, kromě první v komunikaci, vybere poslední blok šifrového textu v poslední zprávě. Dále se upravilo, jak protokol hlásí chybu paddingu při dešifrování, aby nebylo možné provést padding oracle útok. TLS 1.0 bylo vydáno v roce 2006.

Nedlouho po vydání TLS 1.1, v roce 2008, bylo zveřejněno TLS 1.2. Hlavním rozdílem těchto verzí je postupné nahrazení hašovacích funkcí MD5 a SHA-1 za novější a bezpečnější verze jako SHA-2. Také byl vylepšen mechanismus výběru hašovací funkce mezi účastníky komunikace. Dále byla přidána podpora pro módy autentizovaného šifrování s připojenými daty[4].

Nejnovější verze protokolu byla vydána nedávno, v roce 2018. Jelikož 10 let je dlouhá doba, hlavně v kryptografickém světě, tak protokol verze 1.3 obsahuje více změn než jeho předchůdci. Zejména byl upraven seznam použitelných algoritmů, byly odebrány ty zastaralé a přidány nové. Například byly úplně zakázány algoritmy MD5, SHA-224 a DSA[5]

[1] RFC je zkratka za „Request for Comments“. Pomocí RFC se publikuje většina specifikací protokolů a jiných technologií. Každý RFC dokument má své unikátní číslo.

[2] Netscape a Microsoft byly jediné 2 firmy, které v té době vyvíjely internetové prohlížeče.

[3] Zkratka pro „Internet Engineering Task Force“.

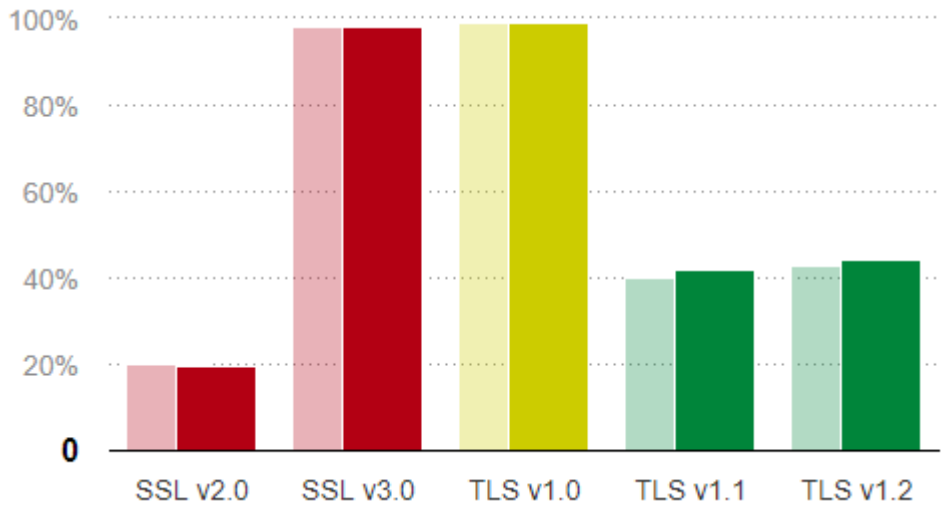
[4] Známé pod zkratkou AEAD neboli „Authenticated Encryption with Additional Data.“

[5] MD5 a SHA-224 jsou hašovací algoritmy a DSA je algoritmus pro digitální podpis. V protokolech SSL/TLS se používají právě kombinace těchto typů algoritmů. Více v další kapitole.

## 3.2 Implementace změn

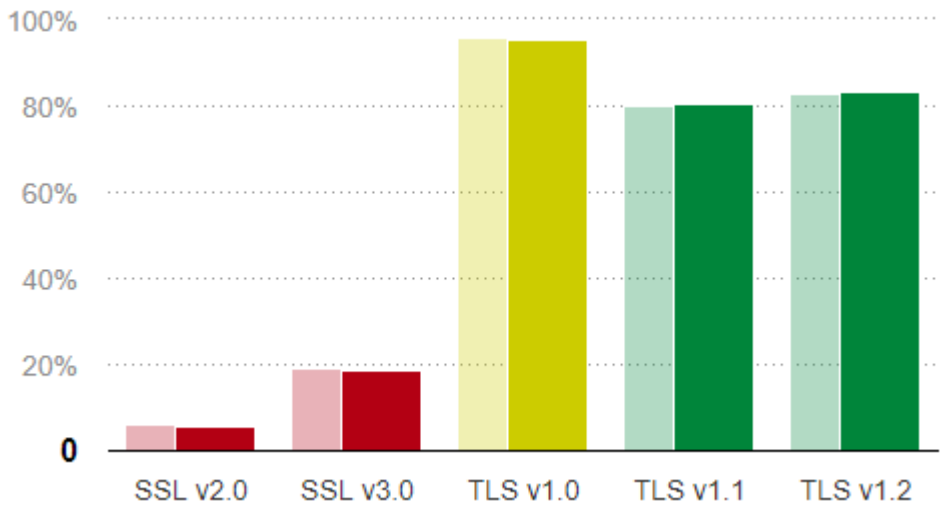
Důležité je zmínit, že protokol je sice vyvíjen relativně rychlým tempem, tak to ale neznamená, že je rychle implementován. Protokol má v sobě zabudovaný mechanismus zpětné kompatibility, který povoluje přejít na starší verze protokolů. Například právě starší verze TLS jako 1.0 povolují přejít na SSL 2.0. Kvůli objevení kritických chyb v SSL 2.0 bylo vydáno v roce 2011 RFC 6176, které přímo zakazuje přechod na SSL 2.0. Pokud se ale podíváme na statistiku z října 2014 (celé 3 roky po vydání „zákazu“), tak vidíme, že stále skoro 20 % webových stránek povoluje použití protokolu SSL 2.0[6]. Podobná situace se stala v roce 2015 s protokolem SSL 3.0, potom co byl objeven útok POODLE. Bylo vydáno také RFC s číslem 7568. Na počátku roku 2017 skoro 19 % webových stránek povolovalo použití SSL 3.0.

Situace se ale v posledních několika letech dramaticky zlepšuje. V době psaní této práce cca 2 % stránek povolují SSL 2.0 a 8 % SSL 3.0. Zastoupení protokolu TLS 1.0 už také mizí ve prospěch bezpečnějších verzí 1.2 a 1.3.

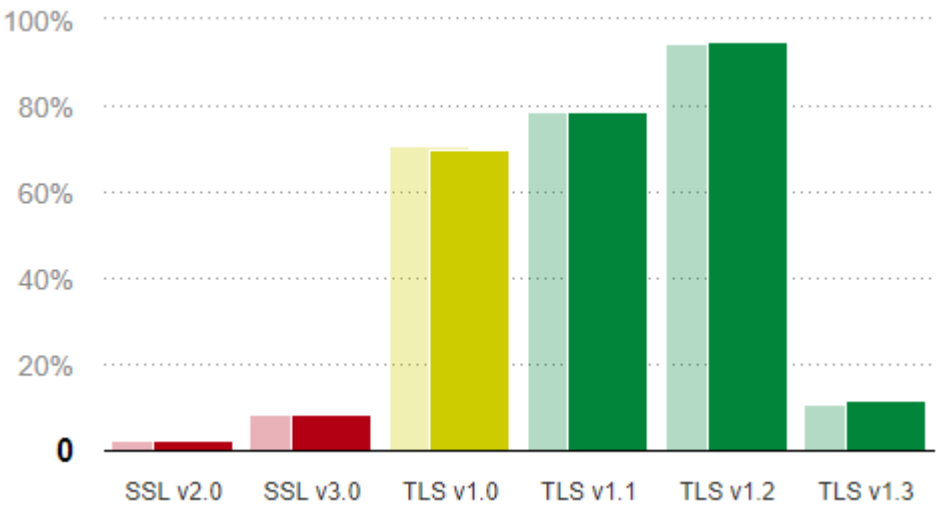



Obrázek 3.1: Statistika podpory verzí protokolu na zhruba 150 000 nejnavštěvovanějších stránek na internetu. Říjen 2014. Zdroj: `https://www.ssllabs.com/ssl-pulse/`

[6]Zdroj: `https://www.ssllabs.com/ssl-pulse/`




Obrázek 3.2: Statistika podpory verzí protokolu na zhruba 150 000 nejnavštěvovanějších stránek na internetu. Leden 2017. Zdroj: `https://www.ssllabs.com/ssl-pulse/`




Obrázek 3.3: Statistika podpory verzí protokolu na zhruba 150 000 nejnavštěvovanějších stránek na internetu. Únor 2019. Zdroj: `https://www.ssllabs.com/ssl-pulse/`

## 3.3 SSL 3.0

TLS/SSL protokol zajišťuje bezpečnou komunikaci na internetu mezi dvěma účastníky[7]. TLS/SSL lze aplikovat na každý protokol, který je z definice spolehlivý[8]. Například HTTP je z definice spolehlivý protokol, takže máme bezpečnou

[7] Obvykle se tito dva účastníci označují klient a server. Nemusí to být ani server v klasickém smyslu, ale klient je ten, kdo zahajuje komunikaci.

[8] V jazyku síťových protokolů to v znamená, že se odesílatel zprávy dozví zda byla zpráva úspěšně doručena příjemci.

nadstavbu HTTPS. Pokud protokol TLS/SSL chceme zařadit do OSI modelu[9], tak leží někde mezi transportní a aplikační vrstvou.

Tento bezpečný kanál komunikace by měl zaručovat tyto vlastnosti:

- Autentifikaci: Klient si je schopen ověřit identitu serveru. Obvykle autentifikace funguje jednosměrně, ale existuje i možnost pro server si ověřit identitu klienta.
- Důvěrnost: Všechna data poslaná kanálem po dohodě spojení jsou utajena třetím stranám.
- Integritu: Třetí strana nemůže manipulovat s daty bez toho, aniž by to účastníci komunikace zjistili.

TLS/SSL lze rozdělit na 2 části. Jednu část tvoří protokol nižší vrstvy[10], tzv. „Record protocol“. Pomocí record protokolu se posílají všechny zprávy v TLS/SSL. Druhou částí jsou 4 protokoly vyšší vrstvy, jelikož jejich zprávy jsou právě také transportovány pomocí record protokolu. Tyto 4 subprotokoly se nazývají:

1. **Handshake protocol**: slouží k domluvení parametrů pro šifrování a autentizaci dalších zpráv.
2. **Change cipher spec protocol**: slouží k oznámení druhé straně, že následující zprávy už budou chráněny domluvenými algoritmy v handshake protokolu.
3. **Alert protocol**: slouží k výměně informací o chybách, které se nastaly během komunikace.
4. **Application protocol**: slouží k zapouzdření dat, které chce klient/server posílat pomocí TLS/SSL protokolu.

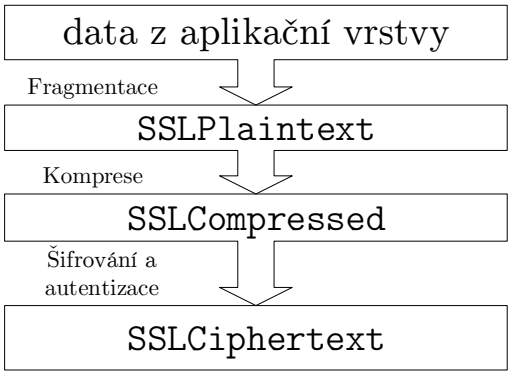
Protokoly **Change cipher spec protocol**, **Alert protocol** a **Application protocol** jsou protokoly spíše z formálního hlediska. Jsou tak jednoduché, že nemá cenu se jimi blíže zabývat. Obvykle tyto části protokolu nejsou ani rozlišovány.

Důležité je zmínit, že základ protokolů TLS 1.x je víceméně stejný s protokolem SSL 3.0, proto si popíšeme fungování protokolu SSL 3.0. V další kapitole se seznámíme s rozdíly TLS 1.x oproti SSL 3.0.

### 3.3.1 Record protocol

Jak již bylo zmíněno, record protokol má na starost zapouzdření dat protokolů vyšších vrstev. Toto zapouzdření se dá rozdělit na tři základní kroky:

1. **Fragmentace**
2. **Komprese**
3. **Šifrování a autentizace**




Obrázek 3.4: Schéma kroků v record protokolu.

Tyto kroky se aplikují za sebou, jak jsou očíslovány viz 3.4.

Důležitým pozorováním je, že SSL aplikuje kompresi před šifrováním a autentizací. Opačně by to ani nedávalo smysl. Kompresní algoritmy využívají pravidelnosti v datech, ale výstupem šifry jsou náhodná data, ve kterých z definice není možné nalézt pravidelnost. Jejich komprese byla tedy kontraproduktivní. Toto pozorování využijeme pro útok na SSL 3.0 až do TLS 1.2[11] viz CRIME.

### Fragmentace

Record protokol dostane od některé z vyšších vrstev data. Tato data nejsou vůbec strukturovaná pro přenos pomocí tohoto protokolu, tudíž si je musí sám předpřipravit. O toto předpřipravení se stará fragmentační vrstva. Rozdělí data na bloky o velikosti $2^{14}$ bytů[12]. Každý blok se opatří příslušnou hlavičkou. Daný objekt se nazývá `SSLPlaintext`, ten obsahuje následující informace:

1. `type`: označuje protokol vyšší vrstvy (subprotokol), ze kterého pocházejí fragmentovaná data. Může nabývat čtyř hodnot viz seznam protokolů vyšší vrstvy.
2. `version`: označuje verzi protokolu TLS/SSL.
3. `length`: označuje délku fragmentu v bytech.
4. `fragment`: blok fragmentovaných dat.

Zde je vidět, že více zpráv stejného typu mohou být součástí jednoho `SSLPlaintext` objektu. Například v handshake protokolu (o kterém si povíme více v další kapitole) mohou být některé zprávy, brané jako samostatné z pohledu handshake protokolu, spojeny do jednoho `SSLPlaintext` objektu a poslány

[9]OSI (Open Systems Interconnection) model je označení modelu, který se snaží charakterizovat komunikaci po síti. Model pracuje se sedmi vrstvami, kde každá vrstva má vlastní účel.

[10]Nižší vrstva v rámci protokolu TLS/SSL

[11]TLS 1.3 je jediná verze, která ve specifikaci nemá kompresní krok. V implementacích je ale kompresní krok vynecháván i ve starších verzích protokolu.

[12]To odpovídá zhruba 16 kB.

najednou. To je mimo jiné důvod, proč **Change cipher spec protocol** je označován jako samostatný protokol. Má vlastní `type` hodnotu, takže nemůže být součástí `SSLPlaintext` spolu se zprávami handshake protokolu[13].

### Komprese

V tomto kroku se na blok `fragment` z `SSLPlaintext` aplikuje aktuálně vybraný kompresní algoritmus. Výchozím algoritmem pro kompresi je *null*[14], to znamená, že tento krok hodnoty v `SSLPlaintext` nezmění neboli žádná kompresní funkce nebyla použita. Pokud se ale server s klientem dohodnou na algoritmu, tak se potom daná kompresní funkce aplikuje na již zmíněný blok `fragment` dat. Výstupem tohoto kroku je objekt s názvem `SSLCompressed`. Jeho struktura je skoro identická se strukturou `SSLPlaintext` objektu.

Kompresní algoritmus/funkce musí být samozřejmě bezztrátová, jelikož o datech nic nevíme a cílem record protokolu je jejich bezpečné doručení mezi komunikujícími stranami. Kompresní funkce navíc nesmí zvětšit délku fragmentu o více než 1024 bytů. To má za následek, že hodnota `length` v `SSLCompressed` může být větší než v `SSLPlaintext` o právě 1024 bytů.

### Šifrování a autentizace

Posledním krokem zapouzdření dat v record protokolu je šifrování a autentizace dat pomocí zvoleného cipher spec. Cipher spec je ale vybrán až po skočení handshake protokolu. Výchozím cipher spec je, stejně jako u kompresní funkce, cipher spec[15] s označením `SSL_NULL_WITH_NULL_NULL`. Při začátku komunikace tedy není aplikováno žádné šifrování ani autentizace na posílaná data. Výstup po aplikaci šifrování a autentizace je pojmenován `SSLCiphertext`. Tento objekt má strukturu skoro stejnou jako `SSLCompressed`, jediný rozdíl je v atributu `fragment`. Ten je nyní vlastní objekt s potřebnými atributy v závislosti na typu použité šifry. Dělí se na 2 typy, podle toho, jaký typ šifry je používán:

- **Proudová šifra** (například RC4)
- **Bloková šifra v módu CBC** (například 3DES)

V případě proudové šifry je `fragment` typu `GenericStreamCipher` a má tyto atributy:

1. `content`: označuje nezašifrovaná data `fragment` z objektu `SSLCompressed`.
2. `MAC`: označuje MAC dat, která se nacházejí v `content`.

Tento fragment je poté zašifrován vybranou šifrou. Zde je vidět, že protokol používá metodu „MAC then encrypt“.

Druhou možností je použití blokové šifry v CBC módu. Poté je `fragment` typu `GenericBlockCipher` a má tyto atributy:

[13]Toto rozdělení bylo uděláno hlavně kvůli zabránění špatné implementaci protokolu. Zpráva **Change cipher spec** protokolu nemusí čekat na zbylé zprávy například handshake protokolu, a tudíž má vlastně vyšší prioritu.

[14]V oficiální specifikaci SSL 3.0 je *null* jediný definovaný kompresní algoritmus.

[15]Přesněji je to název cipher suite, ale šifrování a autentizace se týká jen část cipher spec z cipher suite, tudíž v této části jsou tyto termíny často zaměňovány.

1. `content`: stejné jako v `GenericStreamCipher`.
2. `MAC`: stejné jako v `GenericStreamCipher`.
3. `padding`: padding, aby délka `fragment` byla násobkem velikosti bloku pro zašifrování danou blokovou šifrou.
4. `padding_length`: označuje délku paddingu v bytech.

Jak má padding vypadat, není popsáno ve specifikaci SSL 3.0. Také zde nikde není hodnota pro IV. V protokolu je totiž implicitně definováno, že IV pro první zprávu se spočítá z *master secret*, které získáme po provedení handshake protokolu. Pro další zprávy se používá jako inicializační vektor poslední blok ciphertextu (tedy hodnoty `fragment`) v předchozím recordu. Toto je bezpečnostní chyba, která se dá využít viz BEAST.

### Závěr record protokolu

To, co se předává transportní vrstvě[16], jako data z record protokolu neboli z celého TLS/SSL protokolu, je objekt `SSLCiphertext`.

Zbývá doplnit, jak se generují klíče pro symetrickou šifru, MAC a případně inicializační vektory. Dále také, jakým způsobem se počítá MAC.

Provedením handshake protokolu získáme sdílený klíč, který se nazývá *master secret*. Ten nám dává zdroj entropie pro generování potřebných parametrů pro šifrování. Vygeneruje se blok bytů s názvem *key_block*, který se rozdělí na části dle potřebných parametrů. Generuje se následovně:

$$\begin{gathered} key\_block \leftarrow \\ md5(master\_secret \,||\, sha1(\text{"}A\text{"} \,||\, master\_secret \,|| \\ client\_random \,||\, server\_random)) \,|| \\ md5(master\_secret \,||\, sha1(\text{"}BB\text{"} \,||\, master\_secret \,|| \\ client\_random \,||\, server\_random)) \,|| \\ md5(master\_secret \,||\, sha1(\text{"}CCC\text{"} \,||\, master\_secret \,|| \\ client\_random \,||\, server\_random)) \,|| \\ md5(master\_secret \,||\, sha1(\text{"}DDDD\text{"} \,||\, master\_secret \,|| \\ client\_random \,||\, server\_random)) \,|| \\ \ldots \end{gathered} \tag{3.1}$$

Kde

- *md5* značí hašovací funkci MD5.
- *master_secret* značí *master secret* z handshake protokolu.
- *sha* značí hašovací funkci SHA-1.
- *client_random* značí náhodnou hodnotu klienta z handshake protokolu.
- *server_random* značí náhodnou hodnotu serveru z handshake protokolu.

[16]Transportní vrstva je označení pro protokol nižší vrstvy, jako je např. TCP.

Tato kontrukce teoreticky může maximálně dát $16 \cdot 26 = 416$ bytů[17]. Ve specifikaci není popsáno, co dále, jelikož se předpokládá, že 416 bytů by mělo být dostatek pro všechny parametry. Těmi parametry jsou:

- klíč klienta pro MAC
- klíč serveru pro MAC
- klíč klienta pro symetrickou šifru
- klíč serveru pro symetrickou šifru
- inicializační vektor klienta[18]
- inicializační vektor serveru

Výpočet MACu vychází z klasické HMAC konstrukce:

$$mac \leftarrow H(key \mid\mid opad \mid\mid H(key \mid\mid seq\_num \mid\mid type \mid\mid length \mid\mid fragment)) \tag{3.2}$$

Kde

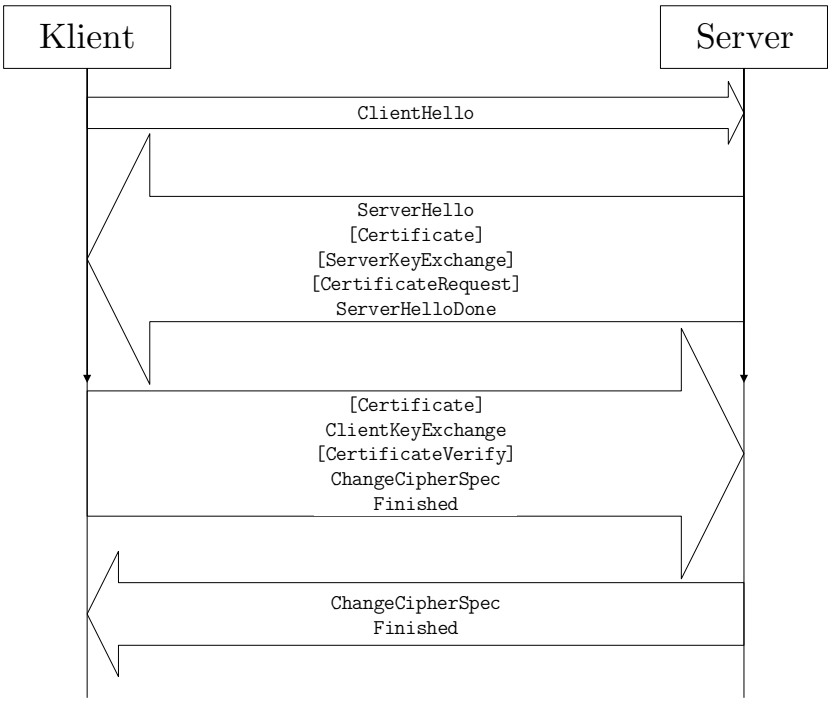
- $H$ značí vybranou hašovací funkci v cipher suite.
- *key* značí klíč dané strany pro MAC.
- *ipad*, *opad* jsou konstanty stejné jako v definici schématu HMAC[19].
- *seq_num* značí pořadové číslo zprávy v aktuální session.
- *type* je hodnota `type` z objektu `SSLCompressed`.
- *length* je délka `fragment` z objektu `SSLCompressed`.
- *fragment* je `fragment` z objektu `SSLCompressed`.

### 3.3.2 Handshake protocol

V této části si popíšeme fungování Handshake protokolu. Tento protokol se podílí na navázání spojení mezi klientem a serverem. Pomocí protokolu se dohadují a autentifikují potřebné parametry pro šifrování v record protokolu.

Zde je diagram klasického průběhu tohoto protokolu:

[17] Výstup funkce MD5 má délku 16 bytů a písmen v anglické abecedě je 26.
[18] Inicializační vektory jsou potřeba pouze v případě, že symetrická šifra je bloková.
[19] Definice HMAC schématu: `https://tools.ietf.org/html/rfc2104`.




Obrázek 3.5: SSL handshake diagram. Inspirován [2, str. 47]

V diagramu výše jsou zprávy v hranatých závorkách nepovinné.

### ClientHello, ServerHello

Klient naváže spojení se serverem zprávou `ClientHello`. Součástí zprávy `ClientHello` jsou tyto informace:

1. `client_version`: specifikuje danou verzi TLS/SSL protokolu, kterou klient chce používat.

2. `random`: skládá se ze dvou hodnot. Jedna je aktuální čas klienta a druhá je 28 bytů, které jsou vygenerované kryptograficky bezpečným (pseudo) náhodným generátorem čísel[20].

3. `session_id`: identifikátor session. Nabízí možnost se odkazovat na předchozí spojení pomocí tohoto identifikátoru. Výhodou toho je, že při obnovení předchozího spojení již není potřeba znovu generovat potřebné klíče. Pokud chce klient založit novou session, tak je tento údaj prázdný.

4. `cipher_suites`: seznam cipher suites podporovaných klientem seřazených sestupně dle preference.

5. `compression_method`: seznam algoritmů pro kompresi dat podporovaných klientem seřazených sestupně dle preference. Pokud klient nechce používat kompresi, tak je tento údaj prázdný.

[20] Anglicky CSRNG respektive CSPRNG neboli „cryptographically secure random number generator“ respektive „cryptographically secure pseudo-random number generator“.

Po odeslání `ClientHello` zprávy klient čeká na odpověď serveru ve formě `ServerHello`. Tato zpráva má stejnou strukturu jako `ClientHello`, ale význam hodnot se liší:

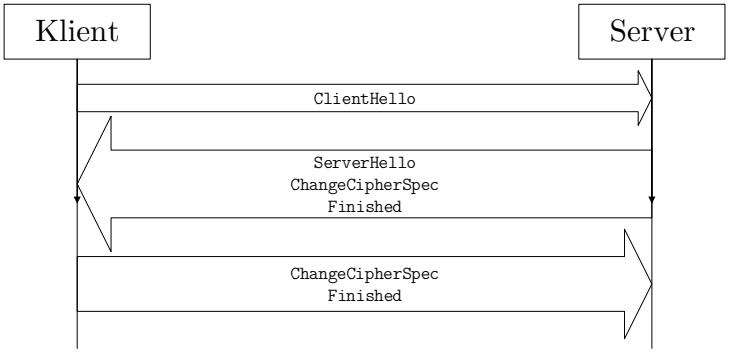
1. `server_version`: specifikuje danou verzi TLS/SSL protokolu, která se bude používat dále v komunikaci. Z definice tato hodnota musí být menší nebo rovna hodnotě `client_version` a zároveň ta největší, kterou server podporuje.

2. `random`: stejně jako `random` v `ClientHello` má tato hodnota 2 části, které jsou získány stejným způsoben, ale na straně serveru. Tyto hodnoty musí být nezávislé na hodnotách v `ClientHello`.

3. `session_id`: pokud klient vyplnil tuto hodnotu v `ClientHello`, tak se server podívá do své cache, zdali má tuto session ještě v paměti. Pokud ano, tak server vyplní stejné `session_id` jako klient. Toto značí, že se mohou přeskočit některé kroky v 3.5 a provede se zjednodušená verze (tzv. „obnovená session“) viz 3.6. Pokud server dané `session_id` nezná, tak buď vygeneruje nový identifikátor, nebo tento údaj nevyplní. Pokud je tento údaj prázdný, tak to značí klientovi, že tuto session nebude možné v budoucnu obnovit.

4. `cipher_suite`: identifikátor vybraného cipher suite serverem. Server vybere nejpreferovanější cipher suite ze seznamu v `ClientHello`, který podporuje. V případě obnovené session je zřejmě tento identifikátor shodný s údajem, který byl uložen v paměti po předchozí dohodě.

5. `compression_method`: ekvivalentní význam jako `cipher_suite`, akorát pro algoritmus používaný pro kompresi.




Obrázek 3.6: Zjednodušený SSL handshake diagram. Inspirován [2, str. 50]

### Certificate

Zpráva `Certificate` není povinná, ale obvykle je nutná, pokud klient neověří bezpečnosti spojení mezi ním a serverem[21]. Jak název napovídá, tato zpráva obsahuje informaci o identitě serveru neboli certifikát:

[21]Specifičtěji je klient přesvědčen, že po komunikačním kanálu nemá nikdo jiný možnost posílat zprávy (kromě klienta serveru). Tento případ je samozřejmě v reálném životě velmi vzácný.

1. `certificate_list`: obsahuje tzv. „certificate chain“ neboli řetězec certifikátů, které svědčí o identitě serveru a také může obsahovat potřebné parametry pro výměnu klíče. Server může mít více certifikátů v závislosti na podporovaných algoritmech pro výměnu klíče. Pokud uživatel nechce používat algoritmus RSA, ale místo toho Diffie-Hellman, tak server pošle adekvátní certifikát (pokud tento cipher suite podporuje).

### ServerKeyExchange

Po zprávě s certifikátem server může poslat další zprávu s názvem `ServerKeyExchange`. Zda je tato zpráva poslána a co obsahuje, je závislé na vybraném cipher suite. Tato zpráva obsahuje nutné parametry, aby mohl být vybraný algoritmus pro výměnu klíče proveden. Například pokud byl vybrán algoritmus RSA, tak server v této zprávě pošle:

1. `rsa_modulus`: vygenerovaný parametr serverem pro provedení algoritmu. K němu si spočítal adekvátní privátní klíč.

2. `rsa_exponent`: veřejný klíč vybraný serverem pro provedení algoritmu. Tento parametr může být statický a nemusí si ho volit server pokaždé, když chce provést algoritmus RSA.[22]

Obvyklou součástí této zprávy ještě je digitální podpis těchto hodnot. Algoritmus použitý pro tento podpis má své potřebné parametry specifikované v certifikátu.

Ukážeme si příklady vybraných cipher suites pro lepší pochopení toho, kdy jsou zprávy `Certificate` a `ServerKeyExchange` poslány.

Identifikátor cipher suite například vypadá takto:

`SSL_RSA_WITH_3DES_EDE_CBC_SHA` (3.3)

1. `SSL`: značí, k jakému protokolu tato cipher suite patří.

2. `RSA`: značí algoritmus použitý pro výměnu klíče i algoritmus pro podepisování.

3. `3DES_EDE_CBC`: znamená, že pro šifrování dat (po výměně klíče) bude použit algoritmus 3DES (EDE[23] označuje, že iterace DES[24] jsou v pořadí šifrování - dešifrování - šifrování) v módu CBC.

4. `SHA`: značí MAC[25] algoritmus.

[22] Na této hodnotě bezpečnost RSA nezávisí, pokud to samozřejmě není jedna z triviálních hodnot. Typicky se univerzálně volí relativně malá hodnota `65537`, jelikož se s ní rychle počítá.

[23] Anglicky „encryption-decryption-encryption“

[24] 3DES je iterovaná verze algoritmu DES.

[25] Kryptografická hašovací funkce SHA sama o sobě není MAC algoritmus. Na to, jak přesně vypadá daný MAC algoritmus, se podíváme v další kapitole.

Dalším příkladem je:

`SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA` (3.4)

Tento cipher suite se od 3.3 liší v tom, že pro výměnu klíče je používán statický Diffie-Hellman algoritmus a pro podepisování se používá RSA. Statický Diffie-Hellman znamená, že parametry používané v tomto algoritmu nejsou dočasné, tedy mohou být použity znovu při novém spojení. Obvykle jsou součástí certifikátu ve zprávě `Certificate`, takže není potřeba zpráva `ServerKeyExchange`.

Naopak v 3.3 je používán algoritmus RSA pro podepisování i pro výměnu klíče. Zda se pošle zpráva `ServerKeyExchange`, rozhoduje to, jestli je veřejný klíč v certifikátu serveru možno používat pouze na podepisování.

`SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA` (3.5)

Zde je použitý tzv. efemérní[26] Diffie-Hellman algoritmus pro výměnu klíčů. Rozdíl mezi statickým a efemérním DH je „trvanlivost“ parametrů.

Ve statickém Diffie-Hellman algoritmu je pevně daný pár DH klíčů, který se používá po delší dobu. Obvykle je veřejný klíč součástí certifikátu, a tudíž je implicitně autentizován.

V efemérním Diffie-Hellman algoritmu se pro každou session generuje nový pár klíčů (parametrů). Tyto parametry je potřeba autentizovat a to se dělá právě pomocí příslušného algoritmu pro podepisování. Efemérní Diffie-Hellman zaručuje tzv. „perfect forward secrecy“ (PFS). To znemožňuje následující možnost útoku:

Útočník odposlouchává a ukládá si komunikaci mezi klientem a serverem. Předpokládejme, že používají algoritmus RSA k podepisování i výměně klíčů. Jelikož používají SSL/TLS, tak útočník komunikaci stejně nedokáže rozšifrovat bez náležitých klíčů. Pokud se v budoucnu útočník dostane k privátnímu klíči serveru a protože klient a server nepoužívali algoritmus zaručující PFS, tak si útočník může rozšifrovat dřívější komunikaci, kterou si po nějakou dobu ukládal.

Algoritmus zaručující PFS tomuto útoku nečelí. Pro každou session jsou vygenerovány nové klíče, které se po uzavření komunikace vymažou z paměti počítače a nejsou již nikdy nepotřeba. Potenciální znalost dlouhodobých klíčů útočníkovi nijak nepomůže s dešifrováním předchozí komunikace.

Pro podepisování je použit algoritmus DSA. Pro šifrování a MAC jsou použity stejné algoritmy jako v přechozích cipher suites.

Z definice efemérního Diffie-Hellman algoritmu je zřejmé, že je v tomto případě (3.5) potřeba také poslat zprávu `ServerKeyExchange`.

### CertificateRequest, ServerHelloDone

Server má možnost autentifikovat klienta. Chce-li to udělat, musí poslat zprávu `CertificateRequest`. Server může ve zprávě specifikovat, jaké typy certifikátů podporuje a od jakých certifikačních autorit musí být podepsány. Tato situace je ale neobvyklá, tedy se jí zabývat blíže nebudeme.

Zpráva `ServerHelloDone` informuje klienta, že server už poslal všechny zprávy, co chtěl a je připraven přijímat zprávy od klienta.

[26] Neboli prchavý.

V případě, že server vyžaduje autentifikaci klienta (poslal zprávu `CertificateRequest`), tak klient pošle zprávu `Certificate`. Tato zpráva má stejný formát jako zpráva `Certificate` od serveru.

### ClientKeyExchange

`ClientKeyExchange` je analogie zprávy `ServerKeyExchange`. Co se přímo posílá v této zprávě, záléží na dohodnutém cipher suite stejně jako u `ServerKeyExchange`.

V případě, že se pro dohodu klíče vybral algoritmus RSA, tak klient vygeneruje 46 náhodných bytů a připojí k nim 2 byty specifikující verzi protokolu[27], kterou klient poslal ve zprávě `ClientHello`. Těchto 48 bytů je zašifrováno veřejným klíčem serveru. Těchto 48 bytů tvoří tzv. *premaster secret*, to je označení pro tajnou hodnotu sdílenou pouze mezi klientem a serverem, ze které se počítá výsledný *master secret*.

Pokud byl použit Diffie-Hellman[28], tak klient naopak pošle svůj veřejný klíč v této zprávě. Výsledkem DH algoritmu je společná hodnota sdílená pouze mezi serverem a klientem, tedy v DH cipher suite toto společná hodnota má význam *premaster secret*. Počet bytů závisí na výběru grupy pro provedení DH algoritmu.

Po přijetí zprávy `ClientKeyExchange`[29] mají klient i server k dispozici všechny potřebné hodnoty pro výpočet *master secret*. Ta se spočítá následovně:

$$\begin{gathered} master\_secret \leftarrow \\ md5(pre\_master\_secret \,||\, sha1(\text{"}A\text{"} \,||\, pre\_master\_secret \,|| \\ client\_random \,||\, server\_random)) \,|| \\ md5(pre\_master\_secret \,||\, sha1(\text{"}BB\text{"} \,||\, pre\_master\_secret \,|| \\ client\_random \,||\, server\_random)) \,|| \\ md5(pre\_master\_secret \,||\, sha1(\text{"}CCC\text{"} \,||\, pre\_master\_secret \,|| \\ client\_random \,||\, server\_random)) \end{gathered} \tag{3.6}$$

Kde

- *md5* značí hašovací funkci MD5.
- *sha1* značí hašovací funkci SHA-1.
- *pre_master_secret* značí výše zmíněný *premaster secret*.
- *client_random* značí hodnotu `random` poslanou ve zprávě `ClientHello`.
- *server_random* značí hodnotu `random` poslanou ve zprávě `ServerHello`.

*master secret* je 48 bytový hash, který je společný zdroj entropie pro další generování klíčů pro symetrické šifrování. Je vidět, že i v případě vybraných cipher suites, které používají statické veřejné klíče (takže *pre_master_secret* bude třeba stejné), tak tato hodnota je vždy náhodná díky hodnotám `random` od klienta a serveru.

[27] Verze protokolu se posílá kvůli detekci tzv. „roll-back“ útoku.

[28] V případě statického DH se tato zpráva může posílat, pokud nebyl jeho veřejný klíč součástí certifikátu (zdali vůbec server požadoval certifikát po klientovi).

[29] Jestliže byl použit cipher suite, který nevyžaduje zprávy `ClientKeyExchange` a `ServerKeyExchange`, tak už po přijetí zprávy `Certificate` od klienta.

### CertificateVerify

Pokud byl po klientovi požadován certifikát, tak klient potřebuje dokázat, že vlastní privátní klíč příslušící veřejnému klíči specifikovanému v certifikátu[30]. Proto klient musí poslat následně zprávu `CertificateVerify`. V této zprávě je důkaz ve formě validního podpisu dat, který si server zkontroluje. Podepsaná data jsou hash všech dosud poslaných zpráv spolu s *master secret*. Posílají se 2 různé hashe, jeden je spočítán pomocí hašovací funkce MD5 a druhý SHA-1. Je to známé HMAC schéma:

$$hash \leftarrow H(master\_secret \,||\, opad \,||\, H(msgs \,||\, master\_secret \,||\, ipad)) \quad (3.7)$$

Kde

- $H$ značí danou hašovací funkci (MD5 nebo SHA-1 v tomto případě).
- *ipad*, *opad* jsou konstanty stejné jako v definici schématu HMAC.
- *msgs* značí jeden řetězec všech zpráv v komunikaci za sebou.
- *master_secret* výše spočítaný *master secret*.

Klient tedy pošle 2 hashe a spolu s nimi odpovídající podpisy.

Poslední dvě zprávy, které musí klient i server poslat, jsou zprávy `ChangeCipherSpec` a `Finished`.

### ChangeCipherSpec

Jak již bylo zmíněno u diagramu handshake protokolu, zpráva `ChangeCipherSpec` se dá považovat za samostatný protokol. Toto odlišení je uděláno kvůli tomu, že tato zpráva nemusí být poslána jen během handshake protokolu, ale i poté. V podstatě ale jediný rozdíl mezi zprávou `ChangeCipherSpec` a ostatními je, že má jinou hodnotu v hlavičce zprávy[31]. Tato zpráva obsahuje pouze jeden byte s hodnotou 1.

Vždy, když klient nebo server tuto zprávu dostane, tak to je znamení, že druhý účastník komunikace bude už posílat jen zašifrované zprávy pomocí dohodnutých parametrů v předchozích zprávách[32].

### Finished

`Finished` je první zpráva, jejíž tělo je zašifrované a autentizované pomocí dohodnutého cipher suite. Významem této zprávy je, navzájem se předsvědčit o tom, že se obě strany správně dohodly na cipher suite a jeho parametrech.

Hodnoty poslané ve zprávě jsou, podobně jako u `CertificateVerify`, dva hashe spočítané následovně:

$$h \leftarrow H(master\_secret \,||\, opad \,||\, H(msgs \,||\, sender \,||\, master\_secret \,||\, ipad)) \quad (3.8)$$

[30] Pokud se nejedná o certifikát obsahující parametry pro statický Diffie-Hellman algoritmus. V tomto případě nemá co dokazovat a server si může ověřit autenticitu díky podpisu certifikační autority.

[31] Přesněji tato hodnota v hlavičce označuje příslušný „subprotokol“ SSL/TLS, ke kterému zpráva patří.

[32] `ChangeCipherSpec` nenese jen tuto informaci, ale detaily se zde nebudeme zabývat.

Kde

- $H$ značí danou hašovací funkci (MD5 nebo SHA-1 v tomto případě).
- *ipad*, *opad* jsou konstanty stejné jako v definici schématu HMAC.
- *msgs* značí jeden řetězec všech zpráv v komunikaci, které byly součástí handshake protokolu[33], za sebou.
- *sender* je konstanta, která může nabývat dvou hodnot, podle toho, zdali tuto zprávu poslal klient nebo server.
- *master_secret* výše spočítaný *master secret*.

Jak již bylo zmíněno, tato zpráva protokolu handshake je poslána jako součást record protokolu, ale oproti předchozím handshake zprávám je její tělo zašifrované a autentizované MAC algoritmem.

Po úspěšné verifikaci zpráv `Finished` je handshake protokol u konce a od nyní si klient a server mohou posílat jiné zprávy zašifrované a autentizované dohodnutým cipher suite.

## 3.4 TLS vs. SSL

Jak již bylo zmíněno, protokoly TLS (verze 1.0, 1.1, 1.2 a 1.3) mají stejný základ jako SSL 3.0 a v podstatě to jsou vždy vylepšené verze toho předchozího. V této kapitole si představíme nejdůležitější změny ve verzích TLS 1.0 a TLS 1.1, kterých se týkají níže popsané útoky.

### 3.4.1 TLS 1.0

Zásadní rozdílem oproti SSL 3.0 je v TLS 1.0 zadefinování pseudonáhodné funkce (PRF) používané k generování parametrů pro record protocol 3.1 a pro první autentizaci dat při handshake protokolu ve zprávě `Finished`. Pro autentizaci v record protokolu se ale používá stejné schéma HMAC viz 3.2. Ve specifikaci TLS 1.0 je oproti SSL 3.0 také zmíněno, jak má vypadat padding při použití blokových šifer.

#### TLS PRF

Tato pseudonáhodná funkce je definovaná podobným způsobem jako předchozí funkce pro generování parametrů 3.1. Je iterovaná a díky tomu délka jejího výstupu není shora omezená[34]. Definice této pseudonáhodné funkce je následující:

$$\begin{gathered} PRF(key,\ label,\ seed) = \\ expand(hash\_A,\ key\_L,\ label \mid\mid seed) \\ \oplus \\ expand(hash\_B,\ key\_R,\ label \mid\mid seed) \end{gathered} \tag{3.9}$$

[33] Tedy bez zpráv `ChangeCipherSpec`

[34] Předchozí funkce byla také „skoro“ neomezená, ale byl tam teoretický limit. U této funkce se tohle neděje.

Kde

- *key* je tajný parametr. Například *master secret*.
- *label* je konstanta závislá na tom, pro co je výsledný hash použitý.
- *seed* je náhodná hodnota, nemusí být ale tajná.[35]
- *expand* je iterovaná funkce definovaná níže.
- *hash_A*, *hash_B* jsou kryptografické hašovací funkce.
- *key_L*, *key_R* jsou označení pro levou a pravou část parametru *key*. Obě části mají stejnou délku (polovina délky *key*). Pokud je délka *key* lichá, tak se duplikuje byte uprostřed.

Tato definice je poněkud nekorektní, jelikož v ní není explicitně zmíněno, že je iterovaná a kolik iterací se vlastně provede. Je to spíše označení pro XOR dvou výstupů funkce *expand*, která jsou definována takto:

$$\begin{aligned} &expand(hash,\ key,\ seed) = \\ &hmac\_hash(key,\ A(1)\ ||\ seed)\ || \\ &hmac\_hash(key,\ A(2)\ ||\ seed)\ || \\ &hmac\_hash(key,\ A(3)\ ||\ seed)\ || \\ &\ldots \end{aligned} \tag{3.10}$$

Kde

- *hmac_hash* je HMAC funkce používající kryptografickou hašovací funkci *hash*.
- *key* má stejný význam jako v 3.9.
- *seed* má stejný význam jako v 3.9.
- $A(i)$ je iterovaná funkce definovaná v závislosti na $i \in \mathbb{N}$:

$$\begin{aligned} A(0) &= seed \\ A(i) &= hmac\_hash(A(i-1)) \end{aligned} \tag{3.11}$$

PRF jsme definovali obecně v závislosti na kryptografických hašovacích funkcích *hash_A* a *hash_B*, jelikož se jejich hodnoty mění v závislosti na verzi TLS. V TLS 1.0 je *hash_A* funkce MD5 a *hash_B* funkce SHA-1.

### Padding

Padding je používán v record protokolu, pokud byl vybrán cipher suite s blokovou šifrou. Definice paddingu je skoro identická s paddingem specifikovaným v PKCS#7, jen jsou povolené hodnoty zmenšené o 1. Takže byte, který označuje 1 bytový padding, je `0x00` místo `0x01`. Dále je součástí paddingu byte, který má hodnotu odpovídající délce paddingu (kromě tohoto posledního bytu).

[35] Seed slouží k podobnému účelu jako IV pro blokové šifry.

### 3.4.2 TLS 1.1

Zásadní změny v této verzi jsou reakcí na útoky na TLS 1.0. První typ útoku (BEAST) využívá implicitní definici inicializačního vektoru pro šifrování v CBC módu. Druhý typ (Padding oracle útok na CBC) využívá „až moc přesné“ chybové hlášky, které jsou součástí protokolu, přesněji subprotokolu **Alert protocol**.

#### Explicitní IV v CBC módu

V předchozích verzích protokolu se náhodně generoval pouze první inicializační vektor použitý pro šifrování prvního bloku. Následující inicializační vektory byly implicitně brány jako poslední bloky ciphertextu v předchozí zprávě viz 3.3.1. Toto zřejmě porušuje vlastnost, kterou by každý IV měl mít, což je nepředvídatelnost. V TLS 1.1 je proto každý IV generován pomocí kryptograficky bezpečného náhodného generátoru.

#### Alert procotol

Z důvodů popsaných v 4.1 je upřednostňováno posílání zprávy `bad_record_mac` za všech okolností místo zprávy `decryption_failed`.

# 4. Útoky

V této kapitole si představíme vybrané útoky, které jsou zajímavé z hlediska kryptografie. Nejprve si ale projdeme základní pojmy ohledně kryptografických útoků na šifry.

Typický model je, že útočník má ciphertext $C$ a chce zjistit jeho plaintext $M$, kde $C = E_k(M)$, $E$ je šifrovací zobrazení a $k$ nějaký klíč. Předpokládá se, že útočník ví, jak funguje algoritmus $E$, ale $k$ je tajné. Kryptografické útoky se dají kategorizovat podle různých kritérií. My si představíme nejklasičtější dělení podle množství informace dostupné pro útočníka.

1. **Ciphertext-only attack (COA)**: Útočník má k dispozici pouze množinu ciphertextů.

2. **Known-plaintext attack (KPA)**: Útočník má k dispozici množinu plaintextů a k nim příslušných ciphertextů.

3. **Chosen-plaintext attack (CPA)**: Útočník si může vybrat množinu plaintextů a k ní má k dispozici jejich zašifrovanou podobu.

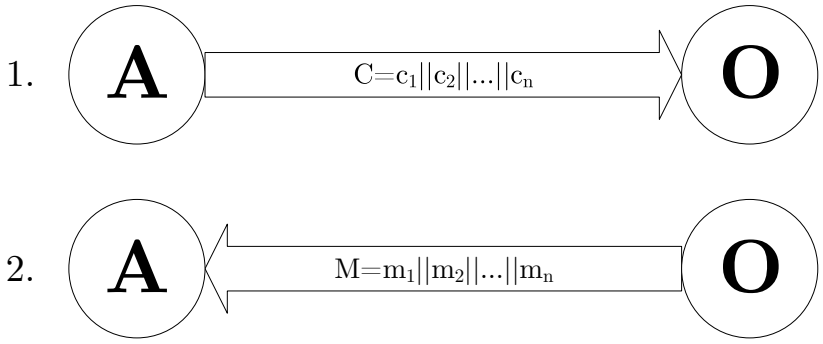
4. **(Lunchtime) Chosen-ciphertext attack (CCA1)**: Útočník si může předem vybrat množinu šifrových textů a má k dispozici jejich dešifrovanou podobu. Jinak řečeno, útočník si nemůže na základě dešifrovaných ciphertextů (nové informace) volit nové ciphertexty k dešifrování.

5. **Adaptive chosen-ciphertext attack (CCA2)**: Útočník si kdykoliv může nechat dešifrovat libovolný ciphertext.

*Poznámka.* V případech 4. a 5. se předpokládá, že si útočník nemůže nechat dešifrovat ciphertext $C$, což je jeho úkolem.

Útoky jsou uvedeny sestupně dle „jednoduchosti“ útoku pro útočníka. Předpokládá se, že pokud útočník například disponuje schopnostmi útoku CCA1, tak automaticky má schopnosti CPA, KPA a COA.

Dalším pojmem, který se typicky využívá hlavně při popisech útoků CCA1, CCA2 a CPA, je tzv. orákulum. Orákulum nazýváme algoritmus, kterému můžeme posílat vstupy a dostávat od něho výstupy. U orákula se obvykle zanedbává jeho složitost výpočtu. Je modelováno jako černá skřínka, která nám ihned vrátí výstup. Například při útoku typu CCA2 je orákulum to, pomocí čeho se dozvídáme dešifrované ciphertexty.




Obrázek 4.1: Příklad komunikace útočníka **A** s dešifrovacím orákulem **O**.

„Man-in-the-middle“ (MITM) je název pro útok, kdy útočník má možnost odposlouchávat a manipulovat s komunikací mezi serverem a klientem. Manipulací se myslí to, že dokáže vkládat zprávy, upravovat je a potlačovat. Tento typ útoku je nejčastější a velice dobře odpovídá reálné situaci. Man-in-the-middle může být například internetový zprostředkovatel, vlastník veřejné Wi-Fi nebo telefonní operátor.

Z technických důvodů budeme používat jako základní jednotku byte. Ten má $2^8 = 256$ možných hodnot, kterých může nabývat. Základní myšlenka útoků nezávisí na dané jednotce a dají se samozřejmě zobecnit.

**Definice 13** (Náhodný útočník)**.** *Náhodným útočníkem myslíme algoritmus $A$, který jako výstup vybere jeden z množiny všech možných výstupů zcela náhodně. Jinými slovy, nechť $x$ je vstup algoritmu, $B$ je množina všech možných výstupů, tak $\forall b \in B : P[A(x) = b] = \frac{1}{|B|}$.*

## 4.1 Padding oracle na CBC mód

Padding oracle na CBC mód se nedá přímo charakterizovat jako **CCA2** nebo **CPA**. Útočník zná pouze ciphertexty, ale má k dispozici i orákulum, které mu něco říká o těchto ciphertextech, i když je přímo nedešifruje. Dalo by se to tedy charakterizovat jako *ciphertext-only attack*. Model útoku je následující.

- Útočník (značíme písmenem **A**) má zprávu $C$, pro kterou platí $C = E_{K,CBC}(M)$, tedy neznámá zpráva $M$ byla zašifrována pomocí symetrické blokové šifry $(E, D)$ v CBC módu za použití paddingu dle definice PKCS#7 tajným klíčem $k$. Cílem útočníka je zjistit $M$.
- Útočník má přístup (posílá orákulu vstupy a získává výstupy) k „dešifrovacímu“ orákulu $\mathbf{O}(D_{k,CBC})$. Orákulum nazýváme „dešifrovací“, jelikož nám přímo nevrací dešifrovanou zprávu, ale informuje nás, zda dešifrování proběhlo v pořádku. Kde $\mathbf{O}(D_{k,CBC})$ je následující program:

**Algoritmus 1** „Dešifrovací“ orákulum **O** pro $(E, D)$ v módu CBC

**Vstup:** $C$
**Výstup:** 0 nebo 1
1: $M \leftarrow D_{k,CBC}(C)$
2: **if** padding zprávy M má správný formát dle definice PKCS#7 **then**
3: **return** 1
4: **else**
5: **return** 0
6: **end if**

Nechť $N$ je délka bloku $E$ v bytech. Víme, že zpráva $M$ má následující tvar $M = m_1||\dots||m_{n-1}||m_n$, kde $\forall i \in \{1,\dots,n\} : |m_i| = N$. Uvažujeme, že $m_n$ je blok, který již obsahuje padding. Buď je celý blok padding, nebo část. Provedení útoku to nijak neovlivňuje.

$C$ je tvaru $C = c_0||c_1||\dots||c_{n-1}||c_n$, kde $c_0 = IV, \forall i \in \{0,\dots,n\} : |c_i| = N$.

Nechť $i \in \{1,\dots,n\}$ značí index bloku $c_i$, který chce útočník dešifrovat. Blok $c_i$ si můžeme rozepsat po bytech $c_i = c_i[1]||\dots||c_i[N]$, kde $\forall k \in \{1,\dots,N\} : c_i[k] \in \{0,1\}^8$. Stejně tak příslušný blok $m_i$, pro který platí $c_i = E_k(m_i \oplus c_{i-1})$.

Nejprve chceme zjistit hodnotu $c_i[N]$. Nechť $b_N \in \{0,1\}^8$, zkonstruujme novou zprávu $C' := c'||c_i$, kde $c' = \underbrace{\texttt{0x00}||\dots||\texttt{0x00}}_{N-1}||b_N$. Je celkem $2^8$ způsobů, jak vyrobit tuto zprávu. Pokud $m_i[N-1] \neq \texttt{0x02}, m_i[N-1] \neq m_i[N-2]$[1], tak po $2^8$ požadavcích na **O** známe hodnotu bytu $m_i[N]$ s pravděpodobností 1. Právě pro jedno $b_N$ nám **O** odpoví, že $M'$ (dešifrovaná $C'$) má správný padding. Zpráva $M'$ má správný padding právě tehdy když $m'_2[N] = \texttt{0x01}$. Hodnotu $m_i[N]$ umíme tedy spočítat následovně:

$$M' = m'_1||m'_2 = D_{k,CBC}(C') \implies m'_2 = D_k(c_i) \oplus c'$$
$$D_k(c_i) = m_i \oplus c_{i-1} \implies$$

[1]Touto podmínkou zamezíme všechny ostatní potenciálně validní paddingy. Např. $m_i[N] = m_i[N-1] = m_i[N-2] = \texttt{0x03}$ je také validní.

$$m_i[N] = m'_2[N] \oplus c_{i-1}[N] \oplus c'[N] = \texttt{0x01} \oplus c_{i-1}[N] \oplus b_N \tag{4.1}$$

Další byty zprávy $m_i$ zjistíme podobně, stačí upravit $C' := c'||c_i$, aby $c' = \underbrace{\texttt{0x00}||\ldots||\texttt{0x00}}_{N-2}||b_{N-1}||(b_N+1),\ b_{N-1} \in \{0,1\}^8$. Pro tyto ostatní byty již vždy stačí $2^8$ požadavků na **O**. Právě pro jednu hodnotu $b_{N-1}$ bude mít $M'$ správný padding, protože poslední byte $c'$ je díky předchozímu výpočtu nastaven tak, aby $m'_2[N] = \texttt{0x02}$. $M'$ bude mít korektní padding právě tehdy, když $m'_2[N-1] = \texttt{0x02}$. Hodnotu $m_i[N-1]$ tedy získáme podobnou rovností jako výše:

$$m_i[N-1] = m'_2[N-1] \oplus c_{i-1}[N-1] \oplus c'[N-1] = \texttt{0x02} \oplus c_{i-1}[N-1] \oplus b_{N-1}$$

Je zřejmé, že tímto postupem jsme schopni dešifrovat celou zprávu $M$.

Podívejme se blíže na komplexitu útoku. Zaměříme se na dešifrování bloku $m_i$. Předpokládáme, že hodnoty bytů v bloku $m_i$ mají rovnoměrné rozdělení neboli $\forall b \in \{0,1\}^8,\ \forall k \in \{1,\ldots,N\} : \mathbf{P}[m_i[k] = b] = \frac{1}{2^8}$.

Předpoklady zmíněné výše jsou nutné, abychom s pravděpodobností 1 dešifrovali byte $m_i[N]$ po maximálně $2^8$ požadavcích na **O**. Pokud tyto předpoklady nejsou splněny, tak se nám může stát, že dostaneme tzv. „false positive“ odpověď od **O**.

Z definice zřejmě platí, že $\forall k \in \{1,\ldots,N-1\} : m'_2[k] = m_i[k]$. Například pokud $m_i[N-1] = \texttt{0x02}$, tak právě pro 2 různé hodnoty $b_{N_1}$, $b_{N_2}$ má zpráva $M'$ správný padding. Ukážeme si, že ale i v tomto „nešťastném“ případě se nám podaří správné $b_N$ nalézt. Stačí nám k tomu jeden požadavek na **O** navíc.

Máme tedy $C'_1$ příslušné k $b_{N_1}$ a $C'_2$ příslušné k $b_{N_2}$. BÚNO $b_{N_1}$ je naše hledané $b_N$.

$$D_{k,CBC}(C'_1) = M'_1 = m_1||m_2, m_2 = \ldots||\texttt{0x02}||\texttt{0x01}$$
$$D_{k,CBC}(C'_2) = M'_2 = m_1||m_2, m_2 = \ldots||\texttt{0x02}||\texttt{0x02}$$

Obě zprávy mají validní padding. Pošleme nyní modifikovanou zprávu $C' = \underbrace{\texttt{0x00}||\ldots||\texttt{0x00}}_{N-2}||\texttt{0x01}||b_{N_1}$. V tomto případě její dešifrování dopadne:

$$D_{k,CBC}(C') = M' = m_1||m_2 \text{ kde } m_2 = \ldots||\texttt{0x03}||\texttt{0x01}$$

Tedy **O** zahlásí validní padding. Další zprávu nemusíme posílat, jelikož už nyní s pravděpodobností 1 víme, že $b_{N_1}$ je správná hodnota. Poslali bychom zprávu $C' = \underbrace{\texttt{0x00}||\ldots||\texttt{0x00}}_{N-2}||\texttt{0x01}||b_{N_2}$, tak její dešifrování dopadne:

$$D_{k,CBC}(C') = M' = m_1||m_2 \text{ kde } m_2 = \ldots||\texttt{0x03}||\texttt{0x02}$$

Což není validní padding, takže **O** zahlásí špatný padding.

Můžeme si vyjádřit pravděpodobnost toho, že $m_i[N]$ zjistíme po maximálně $2^8$ požadavcích:

**Tvrzení 1.** *Za předpokladů výše pravděpodobnost toho, že hodnotu $m_i[N]$ zjistíme po maximálně $2^8$ požadavcích na **O** s pravděpodobností 1, je rovna $2 - \frac{1-(\frac{1}{2^8})^N}{1-\frac{1}{2^8}}$. Tuto pravděpodobnost označme $x$.*

*Důkaz.* $x = 1$ právě tehdy, když $m_i$ nemá strukturu popsanou výše, při které se nám může objevit „false positive“. Spočteme pravděpodobnost toho, že to nenastane, tedy:

$$x = 1 - (\mathbf{P}[m_i[N-1] = \texttt{0x02}] + \mathbf{P}[m_i[N-1] = m_i[N-2] = \texttt{0x03}] + \ldots$$
$$+\mathbf{P}[m_i[N-1] = \cdots = m_i[1] = N]) =$$
$$1 - \sum_{k=0}^{N-2} (\mathbf{P}[m_i[N-1] = \cdots = m_i[N-1-k] = k+2]) =$$
$$1 - \sum_{k=0}^{N-2} \frac{1}{2^{8(k+1)}} = 1 - \sum_{k=1}^{N-1} \left(\frac{1}{2^8}\right)^k = 1 - \left(\frac{1 - (\frac{1}{2^8})^N}{1 - \frac{1}{2^8}} - 1\right) = 2 - \frac{1 - (\frac{1}{2^8})^N}{1 - \frac{1}{2^8}}$$

□

$N$ značí velikost dešifrovaného bloku v bytech. V dnešní době je standardní $N = 16$. Pro takové $N$ platí $x \approx 0.99607$ ($x = \frac{254}{255}$ pro $N \to \infty$)[2]. Výpočet této pravděpodobnosti je celkem zbytečný, když jsme si ukázali, že po maximálně $2^8 + 1$ požadavcích zjistíme $m_i[N]$ s pravděpodobností 1. Můžeme zformulovat následující tvrzení:

**Tvrzení 2.** *Nechť $N \in \mathbb{N}$ je délka bloku. Uvažujme 3 útočníky:*

- $\mathbf{A}_1(2^8 + 1)$ *je útočník využívající algoritmus popsaný výše. Útočník má povoleno* $2^8 + 1$ *dotazů na* $\boldsymbol{O}$.
- $\mathbf{A}_1(2^8)$ *je útočník využívající algoritmus popsaný výše. Útočník má povoleno* $2^8$ *dotazů na* $\boldsymbol{O}$.
- $\mathbf{A}_2(2^8 + 1)$ *je náhodný útočník. Útočník má povoleno* $2^8 + 1$ *dotazů na* $\boldsymbol{O}$.
- $\mathbf{A}_2(2^8)$ *je náhodný útočník. Útočník má povoleno* $2^8$ *dotazů na* $\boldsymbol{O}$.

*Cílem útočníků je dešifrovat poslední byte bloku délky $N$. Platí:*

1. $\mathbf{P}[A_1(2^8 + 1)$ *úspěšně dešifruje byte* $] = 1$
2. $\mathbf{P}[A_1(2^8)$ *úspěšně dešifruje byte* $] = 2 - \frac{1-(\frac{1}{2^8})^N}{1-\frac{1}{2^8}}$
3. $\mathbf{P}[A_2(2^8 + 1)$ *úspěšně dešifruje byte* $] = \frac{1}{2^8}$
4. $\mathbf{P}[A_2(2^8)$ *úspěšně dešifruje byte* $] = \frac{1}{2^8}$

*Důkaz.* Plyne z diskuze výše a z definice náhodného útočníka.

□

Z toho plyne, že celý blok $m_i$ dokážeme s pravděpodobností 1 dešifrovat po $(N \cdot 2^8)+1$ požadavcích na $\mathbf{O}$. V průměrném případě[3] stačí pro každý byte $\frac{2^8}{2} = 2^7$ požadavků.

[2]Tato funkce velice rychle konverguje ke své limitě. Hodnota pro $N = 16$ se od limity liší zhruba o $3 \cdot 10^{-39}$.

[3]Za předpokladu, že nenastanou výše zmíněné, a jak jsme ukázali, nepravděpodobné situace.

Celkově v průměrném případě nám stačí pro kompletní dešifrování bloku $m_i$ $N \cdot 2^7$ požadavků.

Pro dešifrování zprávy $M = m_1||m_2||\ldots||m_n$ stačí $n \cdot N \cdot 2^7$ požadavků. Útok je tedy zřejmě velice efektivní, použijeme-li O-notaci v závislosti na délce zprávy $n$, tak jeho složitost je lineární ($O(n)$). Při reálném předpokladu $N = 16 = 2^4$ nám stačí pro dešifrování zprávy délky $n$ $2^{11}n = 2048n$ požadavků na **O**.

Celý útok se dá naprogramovat jako jednoduchý algoritmus. Nejprve ale budeme potřebovat lehce komplikovanější algoritmus na zjištění posledního bytu:

**Algoritmus 2** decipherLastByte

**Vstup:** $C = c_1||c_2$ ▷ kde $C = E_{k,CBC}(M), M = m_1||m_2$
**Výstup:** $m_2[N]$

1: $candidates \leftarrow \{\}$
2: **for** $i = 0, \ldots, 255$ **do**
3: $\quad c' \leftarrow \underbrace{\texttt{0x00}||\ldots||\texttt{0x00}}_{N-1}||i$
4: $\quad$ **if** $\mathbf{O}(c'||c_2) = 1$ **then**
5: $\quad\quad$ add $i$ to $candidates$
6: $\quad$ **end if**
7: $\quad i \leftarrow i + 1$
8: **end for**
9: **if** length($candidates$) $= 2$ **then**
10: $\quad c' \leftarrow \underbrace{\texttt{0x00}||\ldots||\texttt{0x00}}_{N-2}||\texttt{0x01}||candidates[1]$
11: $\quad$ **if** $\mathbf{O}(c'||c_2) = 1$ **then**
12: $\quad\quad b \leftarrow candidates[1]$
13: $\quad$ **else**
14: $\quad\quad b \leftarrow candidates[2]$
15: $\quad$ **end if**
16: **else**
17: $\quad b \leftarrow candidates[1]$
18: **end if**
19: **return** $\texttt{0x01} \oplus c_1[N] \oplus b$

Nyní můžeme zjednodušeně napsat algoritmus pro dešifrování celého bloku.

**Algoritmus 3** decipherBlock

**Vstup:** $C = c_1||c_2$ ▷ kde $C = E_{k,CBC}(M), M = m_1||m_2$
**Výstup:** $m_2$

1: $m_2[N] \leftarrow$ decipherLastByte(C)
2: $b_N \leftarrow m_2[N] \oplus \texttt{0x01} \oplus c_1[N]$
3: **for** $i = N-1, \ldots, 1$ **do**
4: **for** $j = 0, \ldots, 255$ **do**
5: $c' \leftarrow \underbrace{\texttt{0x00}||\ldots||\texttt{0x00}}_{i-1}||j||(b_{i+1} \oplus (i+1-i))||\ldots||(b_N \oplus (N-i))$
6: **if** $\mathbf{O}(c'||c_2) = 1$ **then**
7: $b_i \leftarrow j$
8: $m_2[i] \leftarrow (N-i+1) \oplus c_1[N] \oplus b_i$
9: $i \leftarrow i+1$
10: **break**
11: **end if**
12: $j \leftarrow j+1$
13: **end for**
14: $i \leftarrow i+1$
15: **end for**
16: **return** $m_2$

A nakonec pro libovolnou zprávu $C = c_0||c_1||\ldots||c_n$, kde $C = E_{k,CBC}(M)$, $c_0 = IV,\ M = m_1||\ldots||m_n$.

**Algoritmus 4** decipherMessage

**Vstup:** $C = c_0||c_1||\ldots||c_n$
**Výstup:** $M = m_1||\ldots||m_n$

1: **for** $i = 1, \ldots, n$ **do**
2: $C' \leftarrow c_{i-1}||c_i$
3: $m_i \leftarrow$ decipherBlock($C'$)
4: **end for**
5: **return** $m_1||\ldots||m_n$

Důležitá otázka je, jestli takové orákulum, které se chová jako **O**, v reálném světě TLS/SSL existuje. Tato specifická forma tohoto padding oracle útoku je spíše teoretická. V protokolu totiž potřebujeme rozlišit chybové zprávy poslané od serveru, ty jsou ale zašifrované.

SSL 3.0 specifikuje pouze jednu chybu s názvem `bad_record_mac`. V TLS 1.0 byla přidána nová chyba `decryption_failed`, která právě značí to, že padding dešifrované zprávy byl nevalidní. V tomto případě by byl TLS 1.0 server perfektní orákulum **O**. Pokud bychom byli schopni nějak odlišit tyto 2 zprávy, mohli bychom vykonat takový útok. Bohužel tyto zprávy jsou také posílány zašifrované. Dalším problémem je, že obě tyto zprávy (i v SSL 3.0) jsou fatální, což znamená, že komunikace by měla být ihned ukončena. Tudíž máme jen jeden pokus na uhodnutí jednoho bytu.

## 4.2 POODLE

POODLE je zkratka pro „Padding Oracle On Downgraded Legacy Encryption“. Tento útok je typu *chosen-plaintext attack* a „man-in-the-middle“ (zkratka MITM). Útok lze rozdělit na 2 části.

V první části útočník potřebuje donutit klienta a server, aby pro komunikaci použili SSL 3.0. TLS verze 1.0, 1.1 a 1.2 oficiálně podporují přechod na verzi SSL 3.0. Tudíž tento předpoklad lze jednoduše zajistit za typických předpokládaných schopností MITM útočníka. Klient totiž při TLS handshake specifikuje svůj preferovaný (typicky nejnovější) typ TLS/SSL protokolu. V protokolu je definována zpráva `handshake_failure`, která oznamuje, že druhá strana nebyla schopná pomocí specifikovaných parametrů navázat spojení. Typický klient, který tuto zprávu obdrží, postupně sníží preferovanou verzi protokolu. Tato chybová zpráva je zřejmě posílána ještě před jakoukoliv autentizací, takže je zcela v schopnosti útočníka se vydávat za server a poslat klientovi tuto zprávu, čímž postupně donutí klienta k použití SSL 3.0. Od nyní budeme předpokládat, že server a klient komunikují pomocí SSL 3.0.

Jak jsme zmínili v přechozí kapitole o record protokolu, v SSL 3.0 specifikaci není definováno, jaký se má používat padding. Většina implementací používá padding definovaný výše jako „SSL 3.0“ padding. POODLE právě útočí na vlastnost toho, že pokud dokážeme zajistit, že padding zprávy bude celý blok délky $N$, tak známe hodnotu posledního bytu, což je hodnota $N-1$. Poté obdobným výpočtem jako v padding oracle dokážeme dopočítat byte zprávy.

Útok byl prezentován pomocí HTTPS[4], kdy chceme zjistit hodnotu cookie klienta příslušnou serveru **S**. Řekněme, že jde o server *https://sis.cuni.cz*. Předpokládejme, že útočník dokáže libovolně vytvářet HTTPS požadavky z klientova prohlížeče na **S**. Toto můžeme v praxi provést, pokud nějakým způsobem donutíme klientův prohlížeč spustit náš JavaScriptový[5] kód.

Nejprve potřebujeme zjistit velikost cookies, které klient posílá na **S**, protože dalším krokem bude posílání HTTPS požadavků typu „POST“, které mají následující strukturu:

$$\begin{gathered}\texttt{POST␣[PATH]}\ldots\\ \texttt{Cookie}:[\texttt{name1}]=[\texttt{value1}]\\ \texttt{Cookie}:[\texttt{name2}]=[\texttt{value2}]\\ \ldots\\ [\texttt{BODY}]\end{gathered}$$

Kde parametry `[PATH]`[6] a `[BODY]` dokážeme modifikovat. Posílaná zpráva $M$, která se bude šifrovat vypadá:

$$\texttt{POST␣}|||[\texttt{PATH}]|||[\texttt{A}]|||[\texttt{cookies}]|||[\texttt{B}]|||[\texttt{BODY}]|||[\texttt{mac}]|||[\texttt{padding}]$$

Části označené `[A]` a `[B]` nedokážeme měnit, ale známe jejich obsah, a tím pádem i velikost, díky znalosti protokolu HTTPS. Víme, že `[mac]` je v SSL 3.0

[4]HTTPS je verze HTTP protokolu, která používá TLS/SSL pro šifrování.

[5]JavaScript je skriptovací jazyk používaný společně s HTML a CSS k tvorbě dynamických webových stránek.

[6]Hranatými závorkami značíme proměnné, ne přímo textové řetezce.

výstup funkce MD5 nebo SHA-1, tedy $|[\texttt{mac}]| = 16$ nebo $|[\texttt{mac}]| = 20$ bytů. Velikost cookies například můžeme zjistit posláním několika GET HTTPS requestů, které mají podobnou strukturu:

$$req = \texttt{GET}\sqcup||[\texttt{PATH}]||[\texttt{A}]||[\texttt{cookies}]||[\texttt{B}]||[\texttt{mac}]||[\texttt{padding}]$$

Zachytíme-li tedy zašifrovaný $req$, tak známe jeho celkovou délku a také známe délku všech jeho součástí kromě [cookies] a [padding].

Z definice paddingu víme, že pokud budeme postupně dosazovat za [PATH] různé hodnoty jako „/“, „/AA“, „/AA“, tak to bude měnit velikost zprávy, která se šifruje, po jednom bytu.

Například je-li $|[\texttt{padding}]| = 1 < N$ v případě [PATH] = „/A“ a pro příslušný $req_A$ platí $|req_A| = n$ , tak v případě [PATH] = „/AA“ bude padding nucen „přetéct“ a bude pro něj platit $|[\texttt{padding}]| = N$ a tudíž zpozorujeme, že $|req_{AA}| = n + 1$. Zjistili jsme, že [padding] má v $req_{AA}$ délku $N$. Z toho můžeme dopočítat velikost [cookies], protože víme, že všechny ciphertext bloky $c_1, \ldots, c_{n-1}$ odpovídají zašifrovanému GET requestu bez paddingu:

$$req_{AA} = \texttt{GET}\sqcup||\texttt{/AA}||[\texttt{A}]||[\texttt{cookies}]||[\texttt{B}]||[\texttt{mac}]$$

U kterého známe jeho celkovou velikost a velikost všech částí, ze kterých se skládá, takže dokážeme spočítat $|[\texttt{cookies}]|$. Nyní známe přesné pozice bytů, které chceme dešifrovat.

Hlavní část útoku provedeme následovně. Dokážeme manipulovat [PATH], tudíž umíme zajistit, aby se první byte cookie nacházel na místě posledního bytu zašifrovaného bloku. Tento blok si označme $c_i$. Známe $|[\texttt{cookies}]|$ a dokážeme manipulovat s hodnotou [BODY], tudíž umíme zajistit, aby poslední blok $c_n$ šifrového textu byl celý blok zašifrovaného paddingu. To nám dává informaci, že $c_n[N] = N - 1$.

Tento POST požadavek pošleme od klienta zašifrovaně serveru. Jako MITM útočník dokážeme zachytit zprávu $C = c_0||c_1||\ldots||c_i||\ldots||c_{n-1}||c_n$ a serveru pošleme upravenou zprávu $C' = c_0||c_1||\ldots||c_i||\ldots||c_{n-1}||c_i$.

Server nenahlásí chybu `bad_record_mac` právě tehdy, když $m'_n[N] = N - 1$, jelikož v tom případě před kontrolou MACu odebere správný počet bytů z konce dešifrované zprávy $M'$. Jestliže $m'_n[N] \neq N - 1$, tak zpráva $M'$ bude mít modifikovaný MAC, jelikož server předpokládá, že i nějaké byty ze zprávy $m'_n$ tvoří MAC. Teoreticky může nastat, že tato kombinace bytů dá také validní MAC. V tom případě jsme ale dostali 2 různé validní tagy ($t$,$t'$) stejné zprávy, což je z definice MAC schématu prakticky nemožné a nemá cenu se tímto případem zabývat.

V případě, že server nenahlásí chybu, tak hodnotu prvního bytu cookie ($m_i[N]$) zjistíme opět z jednoduché rovnice:

$$D_k(c_i) = m_i \oplus c_{i-1}$$
$$m'_n = D_k(c_i) \oplus c_{n-1}, m'_n[N] = N - 1 \implies$$
$$m_i[N] = c_{i-1}[N] \oplus (N - 1) \oplus c_{n-1}[N]$$

Problém je, že pokud server nahlásí chybu `bad_record_mac`, tak přeruší SSL spojení s klientem. To má za následek, že pro další pokus se opět provede handshake a vygeneruje se nový symetrický klíč. Tím pádem nemůžeme použít žádné

informace z předchozího pokusu. Kvůli tomu nemůžeme vlastně použít klasický padding oracle útok (i když je definice paddingu jiná), ale naopak nám to otevírá další možnost.

Změní-li se klíč symetrické šifry, tak dostaneme vlastně nové pseudonáhodné zobrazení $E_k$. V každém takovém zobrazení je z definice pravděpodobnost toho, že $m_i[N] = N - 1$ rovna $\frac{1}{256} = \frac{1}{2^8}$.

**Tvrzení 3.** *Nechť $k \in \mathbb{N}$ značí počet pokusů. Označme $A$ pravděpodobnost toho, že po $k$ spojeních (pokusech) nastane $m_i[N] = N - 1$. Poté $A = 1 - (1 - \frac{1}{2^8})^k$*

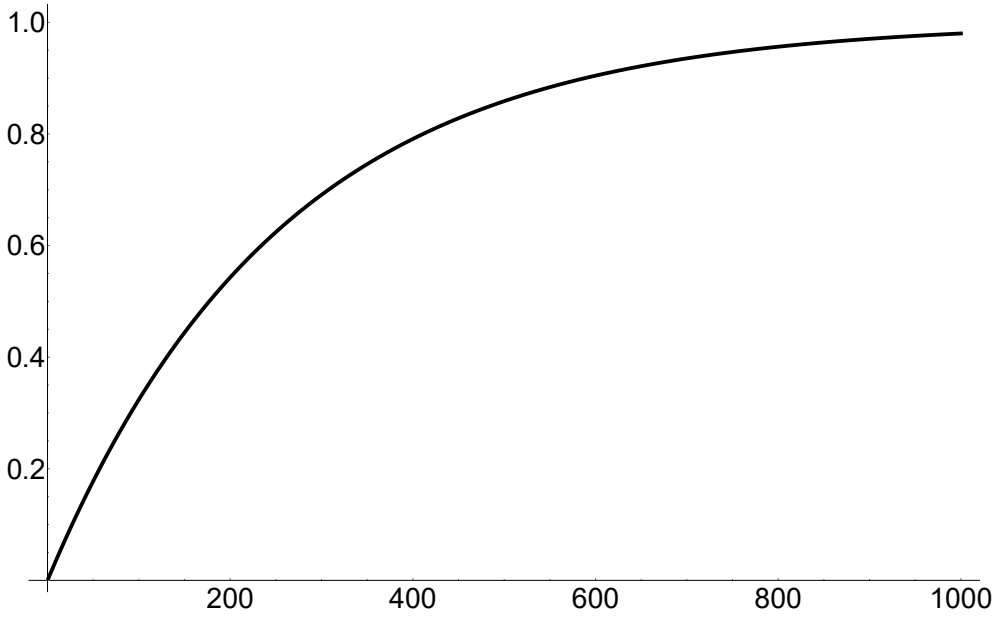
*Důkaz.* Nechť $X$ je náhodná veličina odpovídající tomu, že $P[X = k]$ značí pravděpodobnost toho, že po právě $k$ pokusech poprvé nastane $m_i[N] = N - 1$.

Nás ale zajímá jako pravděpodobonost toho, že aspoň po $k$ pokusech nastane $m_i[N] = N - 1$ neboli $P[X \leq k]$. Náhodná veličina $X$ má tedy geometrické rozdělení s parametrem $p = \frac{1}{2^8}$, takže

$$A = P[X \leq k] = 1 - (1 - \frac{1}{2^8})^k$$

□

Zde je graf této pravděpodobnosti pro $1 \leq k \leq 1000$.




Obrázek 4.2: Pravděpodobnost úspěchu uhodnutí jednoho bytu po $1 \leq k \leq 1000$ pokusech.

Pro $k \geq 178$ je pravděpodobnost toho, že $m_i[N] = N - 1$, větší než 0.5. Po 600 pokusech je $\approx 0.904$. Střední hodnota geometrického rozdělení s parametrem $p$ je $\frac{1}{p}$. Takže $E[X] = \frac{1}{\frac{1}{2^8}} = 2^8 = 256$. Tedy průměrně by nám mělo stačit 256 spojení pro uhodnutí jednoho bytu.

Jakmile získáme první byte cookie, tak opět můžeme upravit zprávu, aby se druhý byte vyskytoval na konci nějakého bloku, a opět použít stejný postup pro získání druhého. Takto můžeme získat celý obsah cookie.

V tomto případě dešifrování cookies stačí pro získání $n$ bytové cookie v průměru $256 \cdot n$ nových spojení se serverem. Průměrný počet spojení nutných k dešifrování tedy roste lineárně s počtem bytů.

Popišme si zhruba, jak útok probíhá algoritmicky. Nejprve potřebujeme zjistit délku cookies, které chceme dešifrovat. Předpokládáme, že známe zbylé parametry HTTPS requestů, takže jediná neznámá je délka paddingu. Předpokládejme, že MAC algoritmus je SHA-1, tedy délka tagu je 20 bytů. Nechť délka bloku $N = 16$ bytů.

**Algoritmus 5** getCookiesLengthInBytes

**Vstup:** $l_A, l_B$ ▷ kde $l_A$ je délka části `[A]` v GET requestu (část hlaviček) a $l_B$ druhé části `[B]`
**Výstup:** $l$ ▷ délka [`cookies`] klienta

1: $path \leftarrow$ "/"
2: sendGetRequestWithPath($path$) ▷ funkce (např. JavaScript program), který posílá námi vytvořené requesty s příslušnými délkami
3: $l_0 \leftarrow$ getLengthOfInterceptedRequestInBytes() ▷ funkce, která zachytí zašifrovaný request a změří jeho délku v bytech
4: $l_{pad} \leftarrow 0$
5: **repeat**
6: $\quad path \leftarrow path||A$
7: $\quad l_{pad} \leftarrow l_{pad} + 1$
8: **until** getLengthOfInterceptedRequestInBytes() $\neq l_0$
9: **return** $l_0 - l_{pad} - l_A - l_B - 20 - 4 - 1$ ▷ 20 značí [`mac`], 4 značí GET␣, 1 značí původní [`PATH`] „/“

Algoritmus getCookiesLengthInBytes pošle v nejhorším případě $N$ požadavků na **S**. Máme-li délku cookies, umíme nyní zmanipulovat POST request, aby vyhovoval námi výše popsaným předpokladům.

**Algoritmus 6** createPostRequest

**Vstup:** $q, l$ ▷ index bytu [`cookies`], který chceme zjistit a délka [`cookies`]
**Výstup:** $req$ ▷ POST request, který budeme posílat serveru

1: $k \leftarrow$ findPathLength($q$) ▷ funkce, která spočítá délku [`PATH`], aby $m_i[N] =$ [`cookies`][$q$] pro nějaké $1 < i < l$
2: [`PATH`] $\leftarrow$ "/$\underbrace{AA\ldots}_{k-1}$"
3: $k \leftarrow$ findBodyLength($q, l$) ▷ funkce, která spočítá délku [`BODY`], aby $|m_n| = 12$, jelikož se za tento blok ještě přidá 20-bytový tag
4: [`BODY`] $\leftarrow$ "$\underbrace{BB\ldots}_{k}$"
5: **return** `POST`␣||[`PATH`]|||[`A`]|||[`cookies`]|||[`B`]|||[`BODY`]

A poslední algoritmus pro dešifrování $q$-tého bytu cookies.

**Algoritmus 7** getCookiesByte

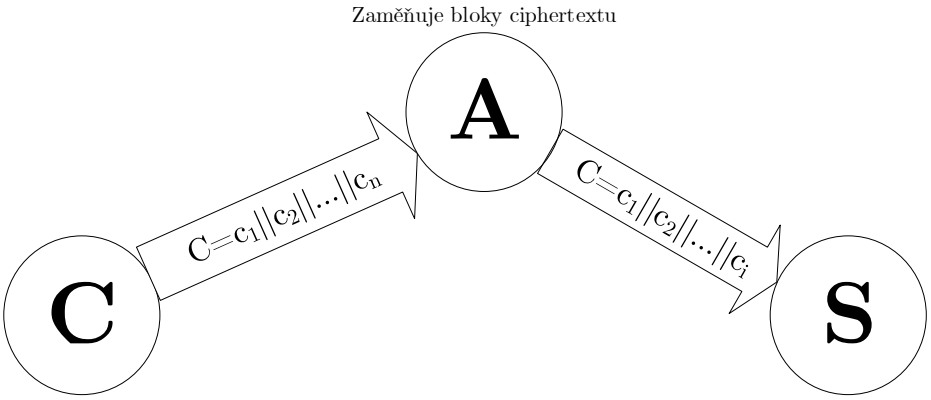
**Vstup:** $q, l_A, l_B$
**Výstup:** [`cookies`][$q$]

1: $l \leftarrow$ getCookiesLengthInBytes($l_A, l_B$)
2: **repeat**
3: $req \leftarrow$ createPostRequest($q, l$)
4: sendPostRequestAndModify($req$) ▷ funkce, která pošle šifrovaný request od klienta a poté ho zachytí a modifikuje, jak je popsáno výše, a přepošle **S**
5: **until** didServerCloseConnection() = $false$
6: $(c_{i-1}, c_{n-1}) \leftarrow$getInterceptedBlocks() ▷ $i$ je index bloku, který byl dosazen za $c_n$
7: **return** $c_{i-1}[N] \oplus (N-1) \oplus c_{n-1}[N]$

Algoritmus getCookiesByte samozřejmě můžeme postupně zavolat pro všechny byty cookies. V tomto případě by nebylo nutné volat getCookiesLengthInBytes pokaždé, ale pouze jednou na začátku.

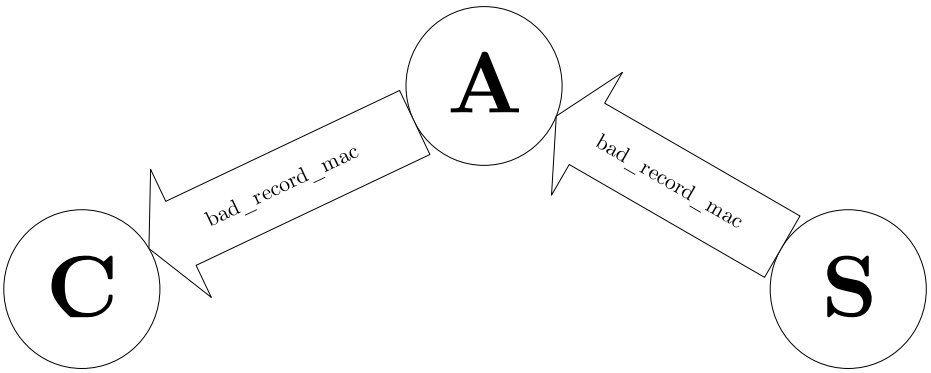
Zde schematicky zobrazena komunikace během útoku. **C** značí klienta, **A** útočníka a **S** server.


Obrázek 4.3: První fáze útoku: útočník zjišťuje velikost cookies.



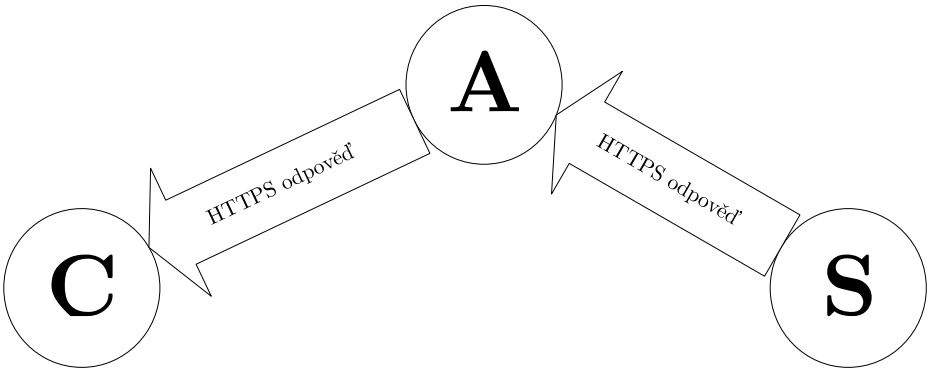

Obrázek 4.4: Druhá fáze útoku: útočník posílá požadavky od klienta a modifikuje je.

Server buď odpoví chybou a uzavře spojení,




Obrázek 4.5: Druhá fáze útoku: server nahlásí chybu a uzavírá spojení.

nebo v případě, že poslední byte byl roven $N - 1$, server odpoví klientovi standardně pomocí HTTPS:




Obrázek 4.6: Druhá fáze útoku: MAC zprávy byl validní.

POODLE znamenal konec SSL 3.0, jelikož využívá nedostatky ve specifikaci paddingu protokolu. Proti POODLE se dá bránit dvěma způsoby.

1. Nepodporovat SSL 3.0.

2. Implementovat rozšíření TLS protokolu **TLS_FALLBACK_SCSV**.

Rozšíření **TLS_FALLBACK_SCSV** zabraňuje útočníkovi donutit klienta a server používat protokol SSL 3.0, i když podporují vyšší verze. V tomto rozšíření protokolu je specifikováno, že pokud klientovi nevyšel první handshake se serverem (obvykle vlivem útočníka), tak klient v dalším spojení přidá do seznamu cipher suites novou cipher suite s názvem **TLS_FALLBACK_SCSV**. To oznamuje serveru, že preferovaná verze podporováná klientem není jeho maximální možná. Podporuje-li server vyšší verzi protokolu, tak ví, že někde musela nastat v předchozím spojení chyba (například kvůli útočníkovi). Server poté přeruší spojení.

Toto rozšíření zabraňuje možnosti útočníka přinutit server a klienta komunikovat pomocí SSL 3.0, tudíž nelze provést dešifrování pomocí orákula. Ale v případě, že klient nebo server z nějakého důvodu podporují pouze SSL 3.0, tak toto rozšíření tomu druhému nepomůže.

## 4.3 BEAST

BEAST je zkratka pro „Browser Exploit Against SSL/TLS“. Útok je stejného typu jako POODLE. Předpokládá MITM schopnosti útočníka a je to opět *chosen-plaintext attack* využívající, jak z názvu napovídá, kompromitovaný webový prohlížeč klienta. Útok je ve svém provedení velice podobný POODLE[7] útoku, a proto nebudeme některé technické aspekty popisovat tak podrobně, jelikož byly popsány v předchozí části.

BEAST využívá toho, že SSL 3.0 a TLS 1.0 používají implicitní IV pro šifrování v módu CBC, viz record protokol, tudíž útočník zná IV předtím, než byl použitý. Útok je modelován stejně jako POODLE. Předpokládáme, že chceme zjistit cookies klienta **C** k serveru **S**. Útočník **A** má možnost posílat upravené HTTPS požadavky z prohlížeče klienta a odposlouchávat jejich zašifrovanou podobu[8].

Základní princip útoku je jednoduchý. Nechť $M = m_1||\dots||m_n$ je zpráva rozdělená na bloky po $N$ bytech. Útok je typu **CPA**, takže útočník má přístup k šifrovacímu orákulu **O**, které přijímá plaintextové zprávy $M$ a vrací $C = E_{k,CBC}(M)$. Dále předpokládáme, že **O** používá implicitní IV neboli pro další šifrování zprávy použije poslední zašifrovaný blok poslední zprávy. Předpokládejme, že útočník navíc zná všechny byty v bloku $m_i$ kromě posledního ($m_i[N]$), který chce zjistit. Útočník nejprve nechá **O** zašifrovat zprávu $M$. Útočník toto může udělat, aniž by znal $M$ viz POODLE. Od **O** dostane $C = c_0||c_1||\dots||c_n$.

**Tvrzení 4.** *Nechť $b \in \{0,1\}^8$ je tip útočníka na poslední byte. Za předpokladů výše je útočník schopen provedením jednoho dotazu na **O** ověřit svůj tip.*

*Důkaz.* Díky předpokladům víme, že $c_n$ bude IV pro další zprávu. Položme $x = m_i[1]||\dots||m_i[N-1]||b$. Útočník vytvoří novou zprávu $M' = m'_1||m'_2$, kde $m'_1 = x \oplus c_n \oplus c_{i-1}$ a $m'_2$ je blok paddingu. Útočník pošle **O** zprávu $M'$ a získá $c'_1$. Pokud $c'_1 = c_i$, tak $x = m_i \iff b = m_i[N]$, protože:

$$c'_1 = E_k(m'_1 \oplus c_n) = E_k(x \oplus c_n \oplus c_{i-1} \oplus c_n) = E_k(x \oplus c_{i-1})$$
$$\text{pokud } x = m_i \implies c'_1 = E_k(m_i \oplus c_{i-1}) = c_i$$

□

*Důsledek.* Za předpokladu, že IV bude opět znát (z předchozího tipu), útočník po maximálně $2^8$ požadavcích na **O** zjistí hodnotu $m_i[N]$.

V průměrném případě zřejmě stačí $2^7$ požadavků pro zjištění jednoho bytu bloku, a tudíž v průměru $2^7 \cdot N$ pro zjištění celého bloku.

Stručně řečeno, útok má tyto předpoklady:

1. Útočník musí znát $N-1$ bytů z hledaného bloku[9].
2. Útočník musí být schopen posunout hledaný byte na určitou pozici v bloku[10].

[7] Autoři POODLE se z velké části inspirovali tímto útokem.
[8] Oproti POODLE útočník nemusí upravovat požadavky, ale stačí je jen umět číst.
[9] Teoreticky i méně, ale komplexita útoku se zřejmě zvyšuje.
[10] Pozice nemusí být poslední byte jako tomu bylo v POODLE.

3. Útočník musí mít přístup k orákulu **O** s vlastnostmi popsanými výše.

Bod 1 a 2 jdou v případě cookies provést v podstatě stejně jako u POODLE. Útočník pomocí volby [`PATH`] v HTTPS requestu může dané byty cookies posouvat postupně, dokud nebudou na konci bloku (1), a předchozí byty zná ze struktury HTTPS požadavku nebo předchozí byty již zjistil (2).

Nejsložitější je zřejmě získat takto specifické **O**. Šifrování zpráv není problém, jelikož o to se stará předpokládaný JavaScript kód v prohlížeči klienta.

Prvním problémem je ale to, že implicitní IV se používají v rámci jedné SSL/TLS session. Prohlížeč obvykle pro každý HTTPS request vytváří nové SSL/TLS spojení. Tento problém se dá vyřešit použitím protokolu WebSocket. Tento protokol zřejmě není HTTPS, ale webové prohlížeče ho také používají a mohou v rámci něho posílat cookies nebo jiné důležité hlavičky. Tento problém je čistě technický a není zajímavý z hlediska kryptografie, tudíž se jím hlouběji zabývat nebudeme.

Druhým, kryptograficky zajímavějším problémem je to, že potřebujeme mít kompletní kontrolu nad první zprávou $m'_1$, která se posílá **O**. Například HTTPS požadavek má vždy na začátku nějaké pevně dané, ale veřejně známé hodnoty. Třeba GET požadavek začíná byty „`GET␣/`“. Pro postup popsaný výše ale útočník potřebuje mít kontrolu nad celým prvním blokem. Tento problém řeší následující tvrzení.

**Tvrzení 5.** *Předpokládejme, že nedokážeme ovlivnit první byte v bloku $m'_1$, ale známe jeho konstatní hodnotu $b \in \{0,1\}^8$. Nechť $c_n$ je poslední blok předchozí zprávy, který bude použitý jako IV. Poté dokážeme ověřit náš tip $x$ na hodnotu zprávy $m_i$.*

*Důkaz.* Chceme zjistit hodnotu bloku $m_i = D_k(c_i) \oplus c_{i-1}$ z předchozí zprávy. Útok je typu **CPA**, tudíž jsme si schopni vytvořit množinu dvojic bloků plaintextů a ciphertextů $\{(p_j, s_j)\}$, kde

$$\forall j \in \{0, \ldots, 255\} : p_j[1] = j \oplus b, p_j[2], \ldots, p_j[N] \text{ libovolné}$$
$$s_j = E_k(p_j)$$

$j$ zde hraje roli hodnoty $c_n[1]$, kterou nemůžeme ovlivnit, tudíž musíme zvažovat všechny možnosti.

Nechť $k \coloneqq c_n[1]$ a $m'_1 = c_n \oplus p_k$, což z definice zaručuje, že $m'_1[1] = b$. Položme $m'_2 = s_k \oplus c_{i-1} \oplus x$ a pošleme $M' = m'_1 || m'_2$ orákulu. Platí:

$$c'_1 = E_k(m'_1 \oplus c_n)$$
$$c'_2 = E_k(m'_2 \oplus c'_1) = E_k(x \oplus c_{i-1} \oplus s_k \oplus E_k(p_k \oplus c_n \oplus c_n)) =$$
$$E_k(x \oplus c_{i-1} \oplus s_k \oplus E_k(p_k)) = E_k(x \oplus c_{i-1} \oplus s_k \oplus s_k) = E_k(x \oplus c_{i-1}) \implies$$
$$c'_2 = E_k(x \oplus c_{i-1}) = c_i \iff x = m_i$$

□

Tento postup se samozřejmě dá více specifikovat na hledání pouze jednoho bytu zprávy $m_i$, jako jsme dělali předtím, aby útok byl méně komplexní. Tento trik

je ale bohužel z praktického hlediska pro HTTPS nepoužitelný. V GET požadavku nedokážeme ovlivnit první 4 byty, což by znamenalo nutnost vytvořit tabulku o $(2^8)^4 = 2^{32} \approx 4 \cdot 10^9$ záznamech. Problém je v tom, že pro vytvoření této tabulky potřebujeme udělat lehce přes 4 miliardy požadavků na **O**, což je nereálné.

Každý požadavek na **O** totiž znamená vytvoření spojení klienta na server. Předpokládejme, že jeden požadavek trvá 50 ms. Pro provedení $2^{32}$ požadavků potřebujeme tedy $2^{32} \cdot 50$ ms $\approx 7$ let.

V případě WebSocket protokolu nedokážeme ovlivnit jen 1. byte, tudíž tento postup je aplikovatelný.

Algoritmus nebudeme explicitně popisovat, jelikož základní myšlenka byla již popsána. Hlavní příčinou toho, že je tento útok možno provést, jsou implicitní IV používané v SSL 3.0 a TLS 1.0.

Proti BEAST útoku existuje jednoduchá úprava, která ho znemožňuje. Nazývá se *1/n-1 record splitting*[11]. Idea je poslat první byte plaintextu, například „`G`“ v GET požadavku, samostatným SSL/TLS recordem viz record protokol a zbylou část plaintextu druhým. To útočníkovi znemožní znát byty v první zprávě ($m'_1$), kromě jednoho. Tento první record se zároveň stane novým IV, který nemůžeme předem znát. Součástí recordu je také MAC, který se chová jako pseudohodný generátor, tudíž jsou byty nepředvídatelné.

[11] Ideálním případem by bylo *0/n record splitting*, ale ten nelze použít, jelikož některé implementace webových klientů a serverů to nepodporují. Zdroj: `https://bugzilla.redhat.com/show_bug.cgi?id=737506`.

## 4.4 CRIME

CRIME je zkratka pro „Compression Ratio Info-leak Made Easy“. CRIME využívá toho, že všechny verze SSL/TLS protokolu, kromě verze 1.3, povolují použití kompresní funkce v record protokolu. Útok se dá kategorizovat jako *chosen-plaintext attack* a modelová situace je stejná jako v předchozích útocích. Útočník chce zjistit hodnoty cookies uživatele příslušné k nějaké stránce. Předpokládá se, že klient a server komunikují pomocí SSL/TLS a používají kompresní funkci.

Základní myšlenkou útoku je využít znalosti toho, jak se daná kompresní funkce chová, a získat díky tomu informace o šifrovém textu. Pro jednoduchost popisu použijeme vlastní kompresní algoritmus. Použití standardního algoritmu by pouze ztížilo analýzu, ale princip by byl stejný.[12].

Definujeme si místo toho jednodušší bezztrátovou kompresní funkci, kterou budeme nazývat *nc* („*naive compression*“). Předpokládejme, že komprimovaný textový řetězec neobsahuje znaky „*(*“ a „*)*“. Tyto znaky budou v našem *nc* algoritmu/funkci speciálními znaky. Algoritmus můžeme popsat následovně:

**Algoritmus 8** nc

**Vstup:** $s$ ▷ textový řetězec, který chceme zkomprimovat
**Výstup:** $cs$ ▷ zkomprimovaný řetězec

1: $cs \leftarrow s$
2: $min\_length \leftarrow 5$ ▷ konstanta, která označuje jak minimálně dlouhá slova hledáme
3: $current\_length \leftarrow \lfloor \frac{length(s)}{2} \rfloor$
4: **while** $min\_length \leq current\_length$ **do**
5: $dict \leftarrow \{\}$
6: **for** $i = 0, \ldots, (length(cs) - current\_length)$ **do**
7: $word \leftarrow substring(cs, i, current\_length)$ ▷ funkce $substring(cs, i, current\_length)$ vrátí podřetězec $cs$ začínající znakem na indexu $i$ a končící znakem na indexu $i + current\_length - 1$
8: **if** existuje záznam tvaru $(l, word)$ v $dict$ **then**
9: nahraď $word$ na indexu $i$ v $cs$ řetězcem „($l$, $current\_length$)“
10: **else**
11: přidej ($i$,$word$) do $dict$
12: **end if**
13: $i \leftarrow i + 1$
14: **end for**
15: $current\_length \leftarrow current\_length - 1$
16: **end while**
17: **return** $cs$

Například tedy máme-li řetězec $s =$„*cookie=123cookie=123*“, výsledek komprese $nc(s) =$„*cookie=123(0,10)*“. $s$ je délky 20 a $nc(s)$ je délky 16, tudíž jsme něco zkomprimovali. Této informace o změně délky při kompresi využívá právě CRIME.

[12]V TLS protokolu byly standardizovány pouze 2 možnosti komprese: žádná nebo pomocí algoritmu DEFLATE. [13]

Jak již bylo zmíněno, komprese je v SSL/TLS aplikována na plaintext, protože má větší redundanci, a tudíž komprese zmenší velikost požadavků. Řekněme, že chceme zjistit hodnotu cookie s názvem `session`[13] klienta **C** na serveru **S**. Opět jako v předchozích útocích předpokládáme, že jsme schopni z klientova webového prohlížece posílat HTTPS požadavky na **S** a umíme číst jejich zašifrovanou podobu, tudíž roli šifrovacího orákula **O** zde opět hraje klientův napadený prohlížeč. Pro zjednodušení předpokládejme, že GET požadavek vypadá následovně:

$$req = \texttt{GET␣}||\texttt{[PATH]}||\texttt{[A]}||\texttt{Cookie :␣session = secretcookie}||\texttt{[B]}$$

kde části `[A]` a `[B]` neobsahují řetězec „*session=*“.

Stejně jako v předchozích útocích dokážeme volit hodnotu `[PATH]`. Pošleme-li serveru 2 požadavky $req_1, req_2$ tvaru

$$req_1 = \texttt{GET␣}||\texttt{/session = a}||\texttt{[A]}||\texttt{Cookie :␣session = secretcookie}||\texttt{[B]}$$
$$req_2 = \texttt{GET␣}||\texttt{/session = s}||\texttt{[A]}||\texttt{Cookie :␣session = secretcookie}||\texttt{[B]}$$

a změříme jejich délky, tak požadavek $req_2$ bude mít menší délku, protože kompresní algoritmus našel dvojici odpovídajících řetězců „*session=s*“. Dále předpokládejme, že je spojení šifrováno proudovou šifrou. Díky tomu jsme schopni jednoduše rozpoznat změnu délky, jelikož proudová šifra nepoužívá žádný padding.

V případě blokových šifer jsme také schopni toto zaregistrovat, protože dokážeme manipulovat s požadavkem tak, aby poslední blok byl celý padding jako v předchozích útocích. Pokud se délka požadavku zkrátí díky kompresi, tak se zkrátí i délka paddingu, který bude součástí předchozího bloku.

**Tvrzení 6.** *Za předpokladů zmíněných výše nám stačí* $2^8$ *požadavků na* ***O*** *pro zjištění prvního bytu cookie.*

*Důkaz.* Byte má $2^8$ různých hodnot, tudíž stačí poslat $2^8$ požadavků s [`PATH`] hodnotou `session` $=||b$, kde $b \in \{0,1\}^8$.

□

*Důsledek.* Složitost v tomto případě roste lineárně s délkou hledané cookie. Pro zjistění $n$ bytů potřebujeme v průměru $n \cdot 2^7$ požadavků.

Útok se také dá zoptimalizovat jednoduchým trikem.

**Tvrzení 7.** *Za předpokladů zmíněných výše nám stačí* $2 \cdot log_2(2^8) = 16$ *požadavků na* ***O*** *pro zjištění jednoho bytu cookie.*

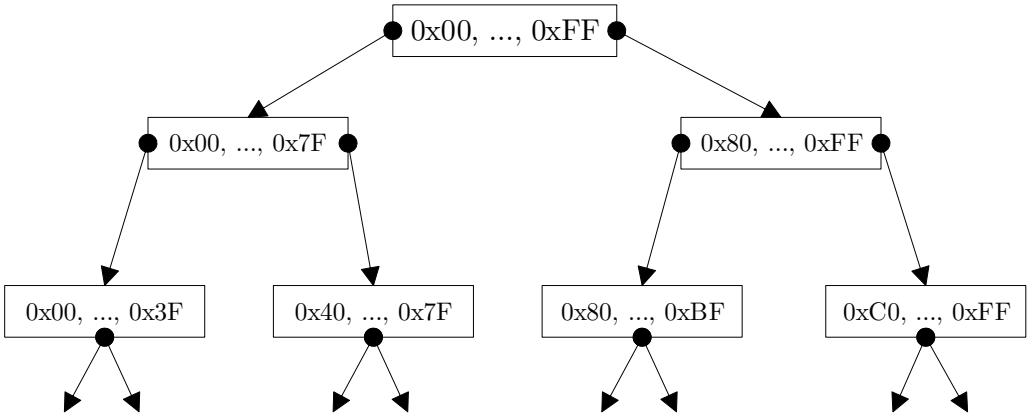
*Důkaz.* Vytvoříme 2 požadavky tvaru

$$req_1 = \texttt{GET␣}||\texttt{/session = 0x00}||\texttt{session = 0x01}||\ldots||\texttt{session = 0x7F}||$$
$$\texttt{[A]}||\texttt{Cookie :␣session = secretcookie}||\texttt{[B]}$$
$$req_2 = \texttt{GET␣}||\texttt{/session = 0x80}||\texttt{session = 0x81}||\ldots||\texttt{session = 0xFF}||$$
$$\texttt{[A]}||\texttt{Cookie :␣session = secretcookie}||\texttt{[B]}$$

[13] Názvy cookies jsou obvykle veřejně známé, tajná je jejich hodnota.

Příslušné hexadecimální hodnoty chápeme jako byty příslušných znaků. Jeden z těchto požadavků bude mít po kompresi menší délku, jelikož `s` přísluší právě jednomu bytu. Tudíž se nám zmenšila množina možných kandidátů na polovinu, a to pouze po 2 požadavcích na orákulum. Takto můžeme pokračovat, až bude množina kandidátů jednoprvková. V tomto případě nám stačí pouze $2 \cdot log_2(2^8) = 16$ požadavků na zjištění jednoho bytu, jelikož si možnosti můžeme reprezentovat ve formě binárního stromu viz 4.7. Hloubka binárního stromu je $log_2(k)$, kde $k \in \mathbb{N}$ značí počet listů (byty), a každá úroveň stromu odpovídá 2 požadavkům.

□




Obrázek 4.7: Znázornění 3 úrovní binárního stromu, kterým procházíme v optimalizovaném útoku CRIME.

# Závěr

Protokol TLS je v neustálém vývoji. Je potřeba reagovat na různé události v kryptografickém světě, např. pokud se objeví efektivní útok na hašovací funkci, kterou protokol používá. Cílem práce bylo čtenáře seznámit s fungováním protokolu. Porozumění protokolu je zásadní pro pochopení vybraných útoků, i když každý útok využívá chyby na specifickém místě v protokolu. Každý ze čtyř útoků byl detailně popsán a bylo poukázáno, které chyby v návrhu protokolu využívá.

Vybrané útoky jsou demonstrací toho, proč některé kryptografické primitivy mají nutné předpoklady jako např. unikátnost IV (BEAST), jak čistě teoretický útok (Padding oracle na CBC mód) lze modifikovat, aby šel použít v reálné situaci (POODLE) nebo jak z prvního pohledu využít vlastnosti kompresních algoritmů k dešifrování zpráv (CRIME).

Kryptografické útoky na sebe často navazují a inspirují se v některých částech od ostatních. Z toho důvodu je důležité být seznámen s těmito „základními“ typy útoků pro lepší pochopení ostatních.

# Seznam použité literatury


[1] Š. Holub. Skripta k předmětu „Složitost pro kryptografii“, 2018. https://www.karlin.mff.cuni.cz/~holub/skripta/slozitost.pdf.

[2] R. Oppliger. *SSL and TLS: Theory and practice*. Second Edition. Artech House, Norwood, MA, 2016.

[3] A. Freier, P. Karlton a P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, August 2011. https://www.rfc-editor.org/rfc/rfc6101.txt.

[4] T. Dierks a C. Allen. The TLS Protocol Version 1.0. RFC 2246, January 1999. https://www.rfc-editor.org/rfc/rfc2246.txt.

[5] T. Dierks a E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, April 2006. https://www.rfc-editor.org/rfc/rfc4346.txt.

[6] T. Dierks a E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008. https://www.rfc-editor.org/rfc/rfc5246.txt.

[7] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018. https://www.rfc-editor.org/rfc/rfc8446.txt.

[8] I. Ristić. *Bulletproof SSL and TLS*. Second Edition. Feisty Duck Limited, London, United Kingdom, 2015.

[9] S. Vaudenay. Security Flaws Induced by CBC Padding Applications to SSL, IPSEC, WTLS..., 2002. https://www.iacr.org/cryptodb/archive/2002/EUROCRYPT/2850/2850.pdf.

[10] T. Duong a K. Kotowicz B. Möller. This POODLE Bites: Exploiting The SSL 3.0 Fallback, September 2014. https://www.openssl.org/~bodo/ssl-poodle.pdf.

[11] T. Duong a J. Rizzo. Here Come The ⊕ Ninjas, May 2011. https://web.archive.org/web/20140603102506/https://bug665814.bugzilla.mozilla.org/attachment.cgi?id=540839.

[12] D. Wong. 1/n-1 split to circumvent BEAST, November 2016. https://www.cryptologie.net/article/378/1n-1-split-to-circumvent-beast/.

[13] S. Hollenbeck. Transport Layer Security Protocol Compression Methods. RFC 3749, May 2004. https://www.rfc-editor.org/rfc/rfc3749.txt.

[14] T. Duong a J. Rizzo. The CRIME attack, 2012. https://www.ekoparty.org/archive/2012/CRIME_ekoparty2012.pdf.

[15] R. Pass a A. Shelat. A course in cryptography, January 2010. https://www.cs.cornell.edu/courses/cs4830/2010fa/lecnotes.pdf.

[16] P. G. Sarkar a S. Fitzgerald. Attacks on SSL: A Comprehensive Study of BEAST, CRIME, TIME, BREACH, LUCKY 13 & RC4 biases, August 2013. https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/ssl_attacks_survey.pdf.