VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ŠIFROVÁNÍ WEBOVÝCH STRÁNEK NA STRANĚ PROHLÍŽEČE

WEB PAGES IN-BROWSER ENCRYPTION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. TOMÁŠ PEKAŘ**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/16900/2016/xpekar10

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Pekař Tomáš, Bc.**
Obor: Informační systémy
Téma: **Šifrování webových stránek na straně prohlížeče**
Web Pages In-Browser Encryption
Kategorie: Web

Pokyny:

1. Seznamte se s HTML5 rozhraním Web Cryptography API. Prozkoumejte možnosti šifrování webových stránek pomocí tohoto rozhraní.
2. S vyžitím HTML5 navrhněte knihovnu pro šifrování statických webových stránek (celých i jejich částí) a jejich zpřístupnění v prohlížeči s různými možnostmi správy klíčů (přímé zadání klíče při operaci, asymetrické klíče, klíčenky soukromé i sdílené, atd.). Zaměřte se také na snadnou použitelnost pro vývojáře i uživatele (nastavení způsobu zobrazení míst s šifrovaným obsahem, chování procesu šifrování/dešifrování, transparentnost, atp.).
3. Po konzultaci s vedoucím implementujte knihovnu v jazyce JavaScript/HTML vč. několika příkladů použití. Výsledek integrujte do některého z generátorů statických webových stránek (např. Hugo či Jekyll). Řešení otestujte.
4. Zdokumentujte a zveřejněte výsledky jako open-source pod svobodnou licencí.

Literatura:

- *Web Cryptography API*. W3C, 2014. [http://www.w3.org/TR/WebCryptoAPI/]
- Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. ShadowCrypt: Encrypted Web Applications for Everyone. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, USA, 2014. [http://dx.doi.org/10.1145/2660267.2660326]
- Ondřej Žára. *JavaScript: programátorské techniky a webové technologie*. Computer Press, 2015. ISBN 978-80-251-4573-9

Při obhajobě semestrální části projektu je požadováno:

- Body 1, 2 a započaté řešení bodu 3.

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

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

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

Vedoucí: **Rychlý Marek, RNDr., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Cílem této práce je popsat současné možnosti kryptografie na straně prohlížeče a zaměřit se na využití nově vznikajícího standardu Web Cryptography API. S využitím těchto technologií pak navrhnout a implementovat knihovnu umožňující autorizovaný přístup k webovým stránkám nebo jejich částem za použití kryptografie.

Abstract

The aim of this work is to describe current opportunities of in-browser encryption and focus on usage of new emerging standard Web Cryptography API. By using these new technologies we going to design and implements software library enabling authorized access to web pages or their part by cryptography.

Klíčová slova

kryptografie, web, javascript, řízení přístupu, bezpečnost, web cryptography api

Keywords

cryptography, web, javascript, access control, security, web cryptography api

Citace

PEKAŘ, Tomáš. *Šifrování webových stránek na straně prohlížeče*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Rychlý Marek.

Šifrování webových stránek na straně prohlížeče

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doktora Marka Rychlého. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Pekař
23. května 2017

Poděkování

Chtěl bych poděkovat svému vedoucímu práce panu doktoru Rychlému za odborné vedení, rady a náměty, které mi během konzultací poskytl. Poděkovat bych mu chtěl ale také za to, že mi umožnil pracovat na tématu diplomové práce, které mě opravdu bavilo.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Webové aplikace jsou v posledních letech čím dál, tím více rozšířené. Běžně se setkáváme s tím, že obyčejnému uživateli stačí k jeho práci webový prohlížeč, který mu poskytuje prakticky vše, co potřebuje. Využití webu jako platformy je obrovské a autoři webových aplikací si uvědomují, že rizika s ním spojená rozhodně nejsou zanedbatelná. Za posledních pár let se začalo hodně využívat zabezpečeného spojení, speciálních autentizačních metod a jiných bezpečnostních mechanismů.

S bezpečností je úzce spjata kryptografie, která se stará o autentizaci a zabezpečení komunikace mezi klientem a serverem. To, co ale schází, je možnost využít kryptografii na aplikační úrovní na straně klienta. Tento požadavek je čím dál více aktuální, a začaly proto vznikat knihovny pro klientské aplikace zajišťující kryptografické funkce. Problém ale bylo zabezpečení samotných knihoven a právě z toho důvodu začal před několika lety vznikat standarad Web Cryptography API. Ten zpřístupňuje kryptografické funkce pomocí API přímo v prohlížeči a plně tak nahrazuje dříve vytvořené aplikační knihovny.

Velká část současného webu je dynamická a statických stránek je méně hlavně kvůli jejich komplikovanější údržbě (z pohledu uživatele) a omezeným možnostem. Statický web má ale i několik výhod, vývoj je rychlejší, provoz je levnější a také bezpečnější. Dnes navíc máme běžně dostupné generátory statických webových stránek, které jsou uživatelsky poměrně přívětivé. Ty nám zajišťují snadnější údržbu a aktualizaci statické webové prezentace.

Tato práce se věnuje rozšíření možností statického webu v oblasti autorizovaného přístupu, navíc s možností využití generátorů statických stránek. Druhá kapitola pojednává o kryptografii na straně prohlížeče. Rozebírá možné problémy dostupných nástrojů a také se věnuje novému webovému standardu v oblasti kryptografie Web Cryptography API. Třetí kapitola se zabývá detailním popisem návrhu celého systému pro autorizovaný přístup. Popisuje principy jeho fungování a také návrh jednotlivých částí, konkrétně šifrovací program a knihovnu pro zpřístupnění šifrovaného obsahu. Čtvrtá kapitola pojednává o implementaci celého systému. Pátá kapitola se zaměřuje na popis integrace výsledného řešení do generátoru statických webových stránek Hugo. Šestá, předposlední kapitola, se věnuje testování celého systému. V závěru práce jsou pak shrnuty výsledky a popsány možnosti dalšího vývoje.

Kapitola 2

Kryptografie v prohlížeči

Kryptografie v prohlížeči není žádnou novinkou. První kryptografické knihovny, resp. aplikace fungující na straně prohlížeče, začaly vznikat již téměř před 10 lety. Od té doby se věci okolo kryptografie vyvíjely až do dnešní podoby, kdy máme nový kryptografický standard pro webové prohlížeče, Web Cryptography API.

V této kapitole se postupně seznámíme s problematikou kryptografie v prohlížeči, kdy si rozebereme pro nás podstatné části teorie kryptografie, nastíníme principy kryptografie v prohlížeči před nástupem Web Cryptography API a poté se seznámíme s novým kryptografickým standardem W3C, který řeší několik doposud zásadních problémů.

2.1 Kryptografie obecně

Věda zabývající se šifrováním a dešifrováním informace se obecně nazývá kryptologie. Tato vědní disciplína se pak dělí na kryptografii, která za pomocí matematických technik poskytuje základní aspekty utajení informace, a krytoanalýzu, sloužící k získání skryté informace, utajené pomocí kryptografie bez znalosti algoritmu nebo klíče, kterým byla zpráva zašifrována. Základní cíle kryptografie jsou definovány následovně [24]:

- **Důvěrnost** je schopnost utajit informaci pro všechny neoprávněné strany. V oblasti kryptografie je toho většinou dosaženo za použití matematických algoritmů, které učiní výstup nesrozumitelným.
- **Integrita** je schopnost zabránit neoprávněné změně dat. Aby byla zajištěna celistvost, daný algoritmus musí být schopen detekovat manipulaci s těmito daty neoprávněnými stranami. To zahrnuje mazání, vkládaní či nahrazování znaků.
- **Autenticita** je schopnost prokázat svou identitu. Autentizační funkce se vztahuje jak na entity, tak na samotnou informaci. Dvě strany vstupující do komunikace, by se měly navzájem indetifikovat a informace poskytované v kanálu by měly být ověřeny původem, datem, obsahem dat, apod. Z těchto důvodů je tento aspekt kryptografie obvykle rozdělen do dvou tříd: autentizace entit a ověření původu, které implicitně poskytuje integritu dat (změnila-li se zpráva, změnil se zdroj).
- **Nepopiratelnost** je schopnost prokázat, že dané operace nebo akce byly provedeny danou stranou. V případě, že je třeba prokázat provedení nebo naopak neprovedení dané operace, je zapotřebí třetí důvěryhodná strana.

V této práci se zabýváme hlavně důvěrností, integritou a autentizací, ale nesmíme zapomínat ani na nepopiratelnost. Vzhledem k tomu, že je kryptologie velice obsáhlá, budeme se pro potřeby naší práce zabývat pouze vybranými oblastmi.

2.1.1 Generátor pseudonáhodných čísel

Bezpečnost mnoha kryptografických systémů závisí na generování nepředvídatelných hodnot. Tyto hodnoty získáváme pomocí tzv. generátorů náhodných čísel (RNG), které bychom mohli rozdělit na tři skupiny: zcela náhodné, pseudonáhodné a pseudonáhodné, které jsou vhodné kryptografické účely.

True Random Number Generator (TRNG)

TRNG využívají náhodnost, která se vyskytuje v některých fyzikálních jevech. Tyto hodnoty můžeme získat například měřením uplynulého času mezi emisemi částic během radioaktivního štěpení, měřením tepelného šumu z polovodičové diody nebo rezistoru nebo může pro získání hodnot použít jiné metody. TRNG jsou prakticky nejlepší generátory náhodných čísel, které můžeme pro kryptografii využít. [24]

Hlavní problémem těchto generátorů je jejich omezená dostupnost a také nízká efektivita.

Pseudorandom Number Generator (PRNG)

Pseudonáhodný bitový generátor PRBG (*pseudorandom bit generator*) můžeme definovat jako deterministický[1] algoritmus, který má na vstupu náhodnou binární sekvenci (*key stream*) o délce k a na výstupu binární sekvenci $l \gg k$. Tato výstupní sekvence by měla být taktéž náhodná (pseudonáhodná) s pravděpodobností $1/2$ na každou 0 či 1. Vstupní sekvenci bitů můžeme označit jako *seed*[2] a výstup jako pseudonáhodnou sekvenci bitů. Prakticky zde žádný významný rozdíl mezi PRBG a PRNG není. Jako příklad PRNG můžeme uvést lineární kongruentní generátor pseudonáhodných čísel:

$$x_n = ax_{n-1} + b \bmod m, \quad n \geq 1 \tag{2.1}$$

hodnoty a, b a m jsou parametry, které specifikují daný generátor a hodnota x_0 je výchozí hodnota generátoru (*seed*). Výstup takového generátoru jsme pak schopni přepovědět, aniž bychom znali jeho parametry a, b a m a není tak vhodný pro kryptografické účely. [24]

Cryptographically Secure Pseudorandom Number Generator (CSPRNG)

Kryptograficky bezpečný pseudonáhodný generátor je speciálním typem PRNG, který má tu vlastnost, že je nepředvídatelný. Přesněji řečeno, neexistuje žádný algoritmus, který by dokázal v polynomiálním čase předpovědět generovaný *key stream* bitů pro další bit s_{n+1} s lepší pravděpodobností jak $1/2$. [25]

Podmínka nepředvídatelnosti CSPRNG je jedinečná pro účely kryptografie. V ostatních případech, kdy je potřeba získat pseudonáhodná čísla, není tato podmínka nutná. To je hlavní rozdíl mezi PRNG a CSPRNG. Někteří, zejména ti, kteří nejsou obeznámeni s problematikou kryptografie, si tyto rozdíly neuvědomují. Musíme proto myslet na to, že téměř všechny PRNG, které byly navrženy pro jiné použití jak pro kryptografii, nejsou CSPRNG.

[1] Deterministický zde znamená, že generátor vrátí vždy stejnou sekvenci bitů pro danou výchozí hodnotu
[2] Semínko - výchozí počáteční hodnota

[25] Příkladem kryptograficky bezpečného generátoru je ANSI X9.31 PRBG (starší a již zastaralý), Micali-Schnorr PRNG nebo Fortuna. [23]

Kvalitu CSPRNG určuje nejen použitý algoritmus, ale také výchozí hodnota generátu, která by měla být co možná nejvíce náhodná. Zdroje takové náhodnosti jsme si již představili a zjistili jsme, že je nereálné je v praxi hromadně využívat. Jako zdroj náhodnosti v klasickém počítači a tedy v operačním systému můžeme použít:

1. systémové hodiny
2. uplynulý čas mezi stisky klávesy či pohyby myši
3. obsah vstupní/výstupní vyrovnávací paměti
4. vstup uživatele
5. hodnoty operačního systému (vytížení procesoru, statistiky sítě)

Ideální je pak použít kombinaci výše uvedených metod pro dosažení opravdu náhodných výsledků.

Statistické testy

Vzhledem k tomu, že je prakticky nemožné získat matematický důkaz o tom, že PRNG je opravdu náhodný, musíme použít statistické testy, které nám pomohou odhalit slabiny daného generátoru. Statistických testů je celá řada a každý test rozhoduje jiným způsobem o tom, zda je generátor opravdu náhodný či nikoli. Pro otestování můžeme použít například testovací balíček amerického NIST[3] [27], který obsahuje řadu statistických testů a můžeme podle něj určit kvalitu generátoru.

2.1.2 Symetrická kryptografie

Symetrická kryptografie, někdy označovaná jako *private key cryptography*, je založena na šifrování a dešifrování pomocí jednoho klíče. To vyžaduje, aby obě strany (šifrující a dešifrující strana) znaly jeden stejný klíč. Šifrovací funkce E a dešifrovací funkce D jsou vyjádřeny následující rovnicí:

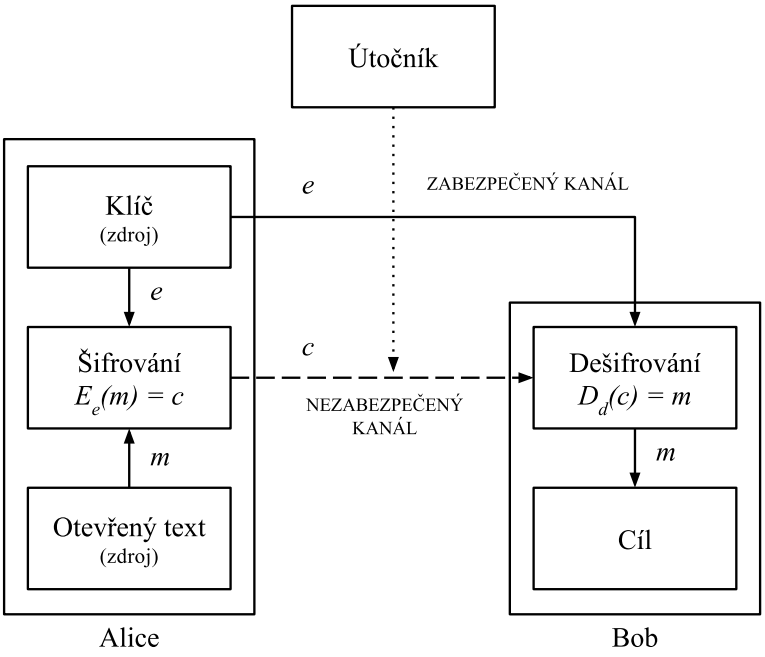
$$E_e(m) = c, \qquad D_d(c) = m \tag{2.2}$$

kde m je otevřený text (*message*) a c je šifrovaný text (*ciphertext*). Příklad komunikace pomocí symetrické kryptografie můžeme vidět na obrázku 2.1.

Symetrická kryptografie běžně rozděluje šifry na dva druhy: proudové a blokové šifry. Proudové šifry zpracovávají jednotlivé bity zprávy samostatně tak, že z původního vstupního klíče (*seed*) je vygenerován nový klíč (*key stream*) odpovídající délky a ten je aplikován pomocí operace XOR na danou zprávu. Tento typ šifer má tu výhodu, že se zde neprojevuje přenos chyby, tak jak je tomu u blokových šifer, a navíc je lze jednoduše hardwarově implementovat. Hlavním požadavkem tohoto druhu šifer je, aby byl klíč opravdu náhodný.

Blokové šifry pracují trochu odlišně. Zpráva je rozdělena na bloky o fixní délce, které jsou zašifrovány pomocí určitého algoritmu. Blokové šifry jsou specifické tím, že pracují v určitém módu, kde mezi základní patří ECB (*electronic codebook*), CBC (*cipher-block chaining*) nebo CFB (*cipher feedback*). Každý z těchto módů má své výhody, nevýhody a také skrývá různá rizika, nicméně jejich popis je nad rámec této práce. Blokové šifry bychom mohli dále dělit

[3] National Institute of Standards and Technology



Obrázek 2.1: Komunikace dvou stran využívající symetrickou kryptografii se zabezpečeným kanálem pro výměnu klíče. Zdroj [24] (upraveno)

na transpoziční, substituční a polyalfabetické. Mezi známé zástupce blokových šifer patří například DES (*Data Encryption Standard*) nebo AES (*Advanced Encryption Standard*), který je popsán dále.

Délka klíče

Délka klíče je jedním z hlavních kritérií kvality šifrovacího algoritmu. Pokud je algoritmus prakticky bezchybný, pak jedinou možností, jak danou šifru prolomit, je útok silou. Dříve bylo naprosto dostačující, že šifrovací algoritmy používaly klíče o délce řádově desítek bitů. Například algoritmus DES, který je dnes již zastarý, používá klíč o délce 56 bitů. Nyní je taková délka nedostačující, a proto se dnes volí délka klíče přesahujcící 100 bitů, standardně 128 bitů.

Advanced Encryption Standard

AES je moderní šifrovací algoritmus s délkou bloku 128 bitů a délkou klíče 128, 192 nebo 256 bitů. Algoritmus je složen z několika vrstev, přičemž každá vrtstva pracuje se všemi 128 bity daného bloku. V první vrstvě se provede sečtení (operace XOR) 128 bitového *round key* s původní zprávou (část zprávy bloku), kde *round key* je derivován z původního vstupního klíče (v dalších kolech z přechozího *round key*). V další vrstvě je provedena substituce pomocí specifikováné tabulky (S-Box), vytvořené na základě matematických modelů. Následně jsou *promíchány* sloupce a řádky. To všechno je vykonáno v několika kolech v závislosti na délce klíče. [25]

AES je blokovou šifrou, takže stejně jako ostatní blokové šifry může pracovat v určitém módu. Prakticky se používají všechny módy až na ECB, který obsahuje zranitelnost

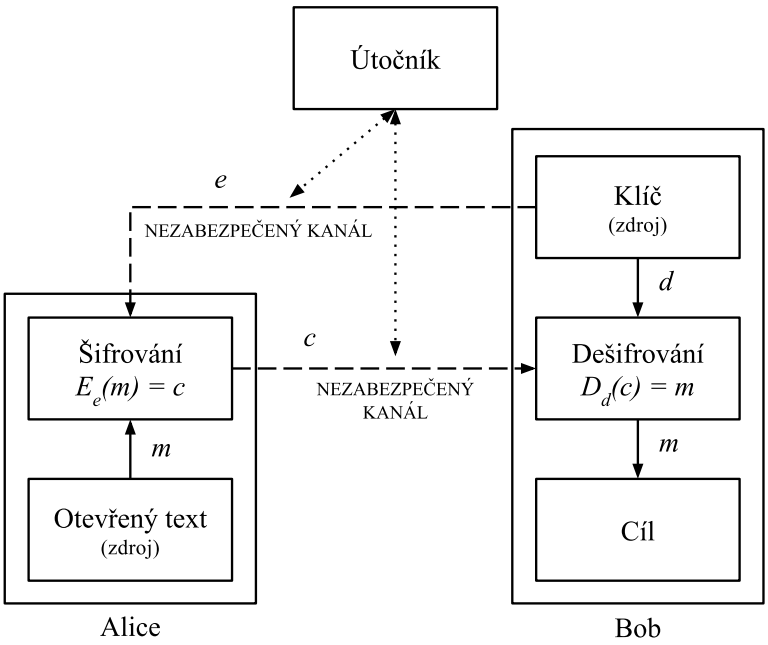
a není tak bezpečný. Některé z módů, jako jsou CBC (*Cipher Block Chaining*) nebo GCM (*Galois Counter Mode*), vyžadují pro svou činnost inicializační vektor (*IV*). Ten musí být vždy nutně zcela náhodný a nesmí se opakovat. Při opakovaném použití, by mohlo dojít k prolomení šifry. Je nutné ještě zmínit také jednu výhodu módu GCM. Mód implicitně obsahuje autentizační schéma, takže je vždy zajištěna autentizace a tudíž i integrita.

2.1.3 Asymetrická kryptografie

Asymetrická kryptografie, označovaná jako *public key cryptography*, je založena na šifrování a dešifrování pomocí dvou různých klíčů. Každá entita účastnící se komunikace tak má svůj veřejný klíč e a k tomu odpovídající soukromý klíč d. Šifrování pomocí veřejného klíče E_e a děšifrování pomocí soukromého klíče D_d je vyjádřeno následovně [24]:

$$E_e(m) = c, \qquad D_d(c) = m \tag{2.3}$$

kde m je otevřený text a c je šifrovaná zpráva. Při komunikaci je potřeba nejdříve distribuovat veřejné klíče všech komunikujích stran. Následně je otevřený text zašifrován veřejným klíčem protistrany a odeslán. Příjemce pak tento šifrovaný text rozšifruje pomocí svého soukromého klíče. Ani jedna zpráva v tomto případě nemusí jít přes zabezpečený kanál. Demostraci můžeme vidět na obrázku 2.2.



Obrázek 2.2: Komunikace dvou stran využívající asymetrickou kryptografii s použitím dvou klíčů (veřejný a soukromý). Zdroj [24] (upraveno)

Asymetrické algoritmy jsou založeny na matematickém složitě spočetném problému. Takovým problémem může být například faktorizace velkých čísel, problém diskrétního logaritmu nebo eliptických křivek. Toto jsou jedny z hlavních, nejvíce používaných problémů, na kterých staví dnes nejpoužívanější algoritmy. Využití těchto algoritmů není jen pro šifrování a dešifrování zpráv, nově je můžeme použít i pro tzv. elektronický podpis nebo

výměnu klíčů. Není to ale rozhodně tak, že by asymetrická kryptografie plně nahradila symetrickou. Asymetrická kryptografie je výpočetně náročnější a nehodí se tak ke zpracování velkého množství dat. Její přednosti jsou hlavně v oblasti digitálních podpisů a algoritmů pro výměnu klíčů, které jsou pak následně použity pro symetrické algoritmy. Mezi nejznámější asymetrické algoritmy patří RSA, Elgamal, ECDH (*Elliptic Curve Diffie-Hellman key exchange*) nebo ECDSA (*Elliptic Curve Digital Signature Algorithm*).

Délka klíče

Délka klíče u asymetrických algoritmů hraje stejně velkou roli, jako tomu bylo u symetrických. Rozdíl je ale v počtu bitů. Pokud si v symetrické kryptografii vystačíme s klíčem do délky 256 bitů, tak v asymetrické kryptografii to stačit nebude. Minimální délka pro relativně bezpečný klíč je 2048 bitů.

Při porovnání více druhů algoritmů se obvykle používá pojem *úroveň zabezpečení*. Ta říká, že k prolomení algoritmu, který využívá klíč s n bity, je potřeba 2^n kroků. To je ale poměrně jednoduchá definice a v asymetrické kryprografii ten vztah není úplně jednoznačný. Porovnání jednotlivých algoritmů můžete vidět v tabulce 2.1, kde je úroveň pro asymetrické algoritmy správně přiřazena. [25]

Druh algoritmu	**Kryptosystém**	**Úroveň zabezpečení (bity)**			
		80	128	192	256
Faktorizace čísel	RSA	1024	3072	7680	15360
Diskrétní logaritmus	DH, DSA, Elgamal	1024	3072	7680	15360
Eliptické křivky	ECDH, ECDSA	160	256	384	512
Symetrická kryptografie	AES, 3DES	80	128	192	256

Tabulka 2.1: Úroveň zabezpečení algoritmů v závislosti na délce klíče. Zdroj [25]

RSA

Algoritmus RSA (*Rives-Shamir-Adleman*) je jedním z nejvíce používaných asymetrických algoritmů a umožňuje jak šifrování dat, tak i podepisování. RSA je postaveno na problému faktorizace velkých čísel. Aby mohlo RSA správně fungovat, je potřeba vygenerovat sadu čísel, podle algoritmu 2.1. Šifrování a dešifrování je pak provedeno pomocí vzorce:

$$c = m^e \ mod \ n, \qquad m = c^d \ mod \ n \tag{2.4}$$

kde c je šifrovaná zpráva a m je otevřený text. Hodnoty e, d a n jsou hodnoty získané při generování klíče pomocí algoritmu 2.1. [24]

RSA je v základu (*textbook RSA*) nedokonalé a zranitelné, např. slovníkovým útokem. Musíme tak nejprve provést předzpracování vstupní zprávy. To můžeme provést technikou OAEP (*Optimal Asymetric Encryption Padding*) definovanou standardem PKCS#1 (*Public Key Cryptography Standard #1*). [25]

Algoritmus 2.1 Generování klíče pro RSA

1. Vygeneruj dvě náhodná prvočísla p a q, které mají přihližně stejnou velikost.
2. Vypočítej $n = pq$ a $\phi = (p-1)(q-1)$.
3. Vyber náhodné číslo e, pro které platí $1 < e < \phi$ a kde $gcd(e, \varphi) = 1$.
4. Za pomocí rozšířeného eukleidovského algoritmu vygeneruj d, pro které platí $1 < d < \phi$ a kde $ed \equiv 1(mod\phi)$.
5. Veřejný klíč je pak (n, e) a soukromý d.

Formát pro ukládání veřejných a soukromých RSA klíčů

Vzhledem k tomu, že potřebujeme generované klíče jistým způsobem přenášet (např. veřejný klíč komunikující strany), je potřeba dané informace patřičně kódovat. Způsob, jakým data ukládat, nám definuje standard PKCS#1. Syntaxi ve formátu ASN.1 můžeme vidět na příkladu 2.1.

Příklad 2.1 Syntaxe formátu veřejného a soukromého RSA klíče [14]

```
1  RSAPublicKey ::= SEQUENCE {
2      modulus           INTEGER,  -- n
3      publicExponent    INTEGER   -- e
4  }
5
6  RSAPrivateKey ::= SEQUENCE {
7      version           Version,
8      modulus           INTEGER,  -- n
9      publicExponent    INTEGER,  -- e
10     privateExponent   INTEGER,  -- d
11     prime1            INTEGER,  -- p
12     prime2            INTEGER,  -- q
13     exponent1         INTEGER,  -- d mod (p-1)
14     exponent2         INTEGER,  -- d mod (q-1)
15     coefficient       INTEGER,  -- (inverse of q) mod p
16     otherPrimeInfos   OtherPrimeInfos OPTIONAL
17 }
```

V současné době je pro ukládání klíčů lepší použít textový formát, oproti binárnímu. K tomu se využívá formát souboru PEM, který nám uchovává data v textové reprezentaci. PEM se používá hlavně u certifikátů X.509, které mají svůj specifický formát a proto je důležité rozlišit, že se v souboru nachází RSA klíč. Způsob, jakým se to provede, je uveden v příkladu 2.2. Data uvozená značkami jsou zakódována v Base64. [1]

Příklad 2.2 Formát PEM

```
1  -----BEGIN RSA PUBLIC KEY-----
2  BASE64 ENCODED DATA
3  -----END RSA PUBLIC KEY-----
4
```

```
-----BEGIN RSA PRIVATE KEY-----
BASE64 ENCODED DATA
-----END RSA PRIVATE KEY-----
```

2.1.4 Hašovací funkce

Kryptografická hašovací funkce hraje základní roli v moderní kryprografii. Hašovací funkce h mapuje řetězce konečné délky na řetězce fixní délky n bitů. Funkce mapuje více vstupů na jeden výstup, což znamená, že existence kolize je nevyhnutelná. Je tedy možné, že dva náhodné řetězce, budou mít stejný výsledný haš. Základní myšlenka kryptografických hašovacích funkcí spočívá v tom, že hodnota haše slouží jako reprezentativní obraz vstupního řetězce a může být použit jako jednoznačný identifikátor. Hašovací funkce jsou používané pro zajištění integrity dat, spolu s modely pro digitální podedpisování. Kryptografická hašovací funkce musí splňovat tři základní parametry [24]

1. **preimage resistence** Funkce musí být jednosměrná, tzn. nesmí být jednoduché získat původní zprávu z výsledné haše. Takže pro jakékoli $h(x) = y$ nesmí být možné nalézt x při známém y.
2. **2nd-preimage resistence** Pro jeden vstup x nesmí být vypočítány dva různé výstupy, tzn. $h(x) = y$ a $h(x) = y'$.
3. **collision resistence** Pro dva vstupy x a x' nesmí být vypočítán jeden výstup, tzn. $h(x) = h(x')$.

Pro správnou funkčnost je nezbytené, aby byly splněny uvedené podmínky. Nejrozšířenější kryptografické hašovací funkce jsou SHA-1 (160 bitová), SHA-256 (256 bitová) a SHA-512 (512 bitová). Dnes je již dokázané, že SHA-1, která vychází z MD4/MD5, nesplňuje pomínku kolize a je proto nevhodná.

2.2 Bezpečnost kryptografie v prohlížeči

Kryptografie v prohlížeči není žádnou novinkou. Už dříve se uvažovalo o implementaci kryptografických funkcí v Javascriptu a byly proto implementovány mnohé knihovny s různou funkčností. Pokud ale mluvíme o kryptografii, mluvíme také o bezpečnosti. Je proto velice důležité, zamyslet se nad tím, zda je možné kryptografické funkce, a celkově tyto přístupy, v prohlížeči bezpečně implementovat. Hned z počátku vyvstanou tyto problémy:

1. jak bezpečně doručit knihovny na stranu klienta
2. jak získat kryptograficky bezpečný PRNG
3. jak bezpečně uchovávat kryptografický materiál a jak s ním pracovat

Vyjma tří výše zmíněných se musíme potýkat i s dalšími problémy. Javascript je sám o sobě interpretovaný jazyk a logicky tak bude problém efektivně vykonávat kryptografické operace, které vyžadují více iterací. Příkladem může být funkce pro derivaci kryptografického klíče PBKDF2, která potřebuje minimálně tisíckrát po sobě vypočítat kryptografickou haš. Tato operace pak potrvá delší dobu, což může být někdy problematické. Navíc v reálném prostředí nemusíme zůstat u generování, resp. derivování, pouze jednoho klíče. Podobných problémů by se dalo najít více. Výše zmíněné prolbémy jsou rozebrány v následujících podkapitolách.

2.2.1 Komunikace klient-server

Prvním problémem, na který narazíme při práci s kryptografickými knihovnami, je jejich doručení klientovi (klientská aplikace - prohlížeč). Pokud pomineme, že může být kompromitován samotný server nebo klientská aplikace, je v případě útoku nejpravděpodobnější, že bude obsah změněn při samotném doručování kódu klientovi. Knihovna může být správně implementována a nemusí obsahovat žádné zranitelnosti, ale např. útokem Man-in-the-middle (MITM) může být její kód pozměněn. Tomu lze samozřejmě zabránit použitím TLS/SSL. Komunikace klient-server je během spojení šifrována a v ideálním případě zabezpečena proti útoku typu MITM.

2.2.2 Pseudonáhodná čísla

V Javascriptu je běžně dostupný PRNG *Math.random*. Tento PRNG je postaven funkci *XorShift128+*, která je rychlá a pro běžné použití naprosto dostačující. Nehodí se však pro kryptografické účely. V základu tedy nemáme vhodný CSPRNG, který bychom mohli použít a musíme proto hledat jiné alternativy.

V Javascriptu nemáme přístup k CSPRNG operačního systému, ale víme, že můžeme jistými způsoby dostat relativně náhodnou hodnotu, která by posloužila jako *seed*. Lze ji získat pomocí technik uvedených v 2.1.1. Některé z nich jsou prakticky nepoužitelné, ale jiné techniky lze v Javascriptu implementovat.

Problém s nedostupností bezpečného generátoru můžeme vyřešit vlastní implementací CSPRNG. Mohli bychom např. použít generátor Fortuna, který by pro inicializaci sbíral data z vybraného uživatelského vstupu (pohyb myši na stránce, stisky kláves, aj.).

2.2.3 Práce s kryptografickým materiálem

Jedná se asi o nejzávažnější problém celého konceptu kryptografie v prohlížeči. Javascript neposkytuje bezpečné prostředí pro práci s kryptografickým materiálem. Vzhledem k tomu, že Javascript používá ke správě paměti GC (*garbage collector*), nemáme nad ní žádnou kontrolu. Nevíme a není také zaručeno, kdy GC začne uvolňovat pamět a odstraní případné kryptografické klíče uložené v paměti.

Dalším problémem je samotný „režim“, ve kterém Javascript pracuje. Pro zajištění ideální bezpečnosti potřebujeme, aby bylo k dispozici speciální prostředí, které zajistí bezpečné uchování (ne nutně perzistentní) kryptografického materiálu. Klíče tak nemohou být ukládány do standardní paměti, a proto s nimi musí být nakládáno jiným, bezpečnějším způsobem.

Takových problémů bychom mohli najít více, např. absence typu pro práci s velkými čísly (64 a více bitů). Daný datový typ je nutný pro jakékoli kryptografické operace (minimálně pro asymetrickou kryptografii). Nemá proto smysl vyhledávat další problémy související s kryptografií v Javascriptu, protože jich bylo demonstrováno tolik, že lze automaticky usuzovat, že Javascript alespoň v základní podobě, v jaké ho známe, není pro kryptograrfii příliš vhodný.

2.3 Kryptografické knihovny pro Javascript

Kryptografických knihoven implementovaných v Javascriptu je celá řada. Liší se však svou robustností a také hlavně bezpečností. Knihovny implementují standardní kryptografické algoritmy pro symetrické a asymetrické šifrování, hašovací funkce i autentizační schémata.

Stanford Javascript Crypto Library [8] je pravděpodobně nejlepší běžně dostupnou implementací kryptografických funkcí. Knihovna byla vyvinuta týmem Stanfordské univerzity v roce 2009 a nabízí standardní šifrovací funkci AES pro 128, 192 a 256 bitů v módu CCM, OCB, GCM a hašovací funkce SHA-1, SHA-256 a SHA-512. Dále poskytuje i autentizační funkci HMAC. Nechybí ani funkce pro posílení/derivaci klíče HKDF a PBKDF2. K dispozici jsou i funkce asymetrické kryptografie postavené na eliptických křivkách. Hlavní předností knihovny je silný generátor. Základem CSPRNG je derivát generátoru Fortuna, který jako *seed* používá funkci *Math.random* v kombinaci s informacemi o pohybu myši. Tím je docíleno opravdu náhodné počáteční hodnoty. SJCL je knihovna, kterou můžeme považovat za bezpečnou.

Clipperz - Javascript Crypto Libraray [2] je další poměrně dobře implementovanou knihovnou pro kryptografické účely. Jsou zde použity i jiné algoritmy než u SJCL, ale funkčnost je téměř totožná. Podobným způsobem také využívá generátor Fortuna.

CryptoJS [3] je poměrně robustní knihovna, která obsahuje velké množství kryptografických funkcí jak pro symetrickou, tak pro asymetrickou kryptografii, několik druhů hašovacích funkcí, ale i varianty autentizačního kódu HMAC. Knihovna má ovšem jednu velkou slabinu, kterou je její PRNG. Na rozdíl od dvou předchozích knihoven nepoužívá žádnou implementaci CSPRNG a spoléhá se jen na generátor *Math.random*. Knihovna může mít sebelepší implementaci kryptografických funkcí, nicméně použití slabého generátoru ji znehodnocuje.

Další dostupné knihovny implementují kryptografické funkce podobným způsobem. Za mnohými bohužel nestojí kvalifikovaný tým specialistů, jako u SJCL, a proto si nemůžeme být stoprocentně jisti jejich bezpečností. Pokud se rozhodneme pro použití jedné z těchto knihovny, stojí za to první analyzovat generátor náhodných čísel, protože to může být prvotní signál toho, že knihovna nemusí splňovat potřebné požadavky.

S nástupem Web Cryptography API velká část knihoven začala používat generátor poskytovaný tímto API, který je určený pro kryptografické účely.

2.4 Web cryptography API

W3C Web Cryptography API [12] je nově vznikající standard v oblasti kryptografie. Standard nám definuje kryptografická primitiva napříč prohlížeči, která jsou dostupná skrze prostředí Javascriptu. Standard začal vznikat v roce 2012, kdy vyšla první *Draft* verze a nyní, v roce 2017, je standard téměř schválen. V současné době je ve verzi *Prososed recomendation* a jedná se tak o pravděpodobného kandidáta výsledného standardu.

Web Cryptography API je nízkoúrovňové API, které nám definuje rozhraní pro práci s kryprografickým materiálem spravovaným nebo generovaným prohlížečem. API jako takové v současné době neposkytuje perzistentní úložiště pro kryptografické klíče, ale pouze rozhraní pro přístup k funkcím pro šifrování, dešifrování, podpis, ověření, derivaci a generování klíčů nebo hašovací funkce. API je popsáno pomocí pomocí WebIDL[4] a jeho hlavní funkce uvedené ve specifikaci jsou následující [12]:

1. kryptograficky bezpečný generátor pseudonáhodných čísel
2. rozhraní pro práci s kryptografickými klíči

[4] Web Interface Description Language

3. kryptografické funkce

V následujícím textu jsou rozebrány jednotlivé části API spolu s ukázkami, jak s ním pracovat. Mimo popis funkčnosti jsou uvedeny i příklady použití, analýzy současné podpory API napříč prohlížeči nebo bezpečnostní analýzy.

2.4.1 Rozhraní Crypto

Rozhraní nám poskytuje přístup k dvěma objektům. Jedním je objekt *SubtleCrypto*, který je rozebraný v 2.4.3, a druhým kryptograficky bezpečný generátor pseudonáhodných čísel (*RandomSource*). Tento generátor je implementací CSPRNG s výchozí hodnotou, která splňuje všechny bezpečnostní požadavky uvedené v 2.1.1.

Objekt *crypto*, resp. *mscrypto*, je dostupný globálně skrze objekt *Window* a obsahuje metodu *getRandomValues*, která generuje náhodná čísla pomocí uvedeného generátoru. Vstupem metody je *typedArray*, např. *Uint8Array* nebo *Uint32Array*. Toto pole je pak naplněno výslednými čísly.

Není doporučeno používat metodu *RandomSource.getRandomValues* pro generování kryptografických klíčů, k tomuto účelu slouží metody rozhraní *SubtleCrypto*, které přistupují ke generovaným hodnotám jiným, bezpečnějším způsobem. Příklad získání 10 pseudonáhodných čísel můžeme vidět na příkladu 2.3.

Příklad 2.3 Generování pseudonáhodných čísel [10]

```
1 <script type="text/javascript">
2   var array = new Uint32Array(10);
3   window.crypto.getRandomValues(array);
4 </script>
```

2.4.2 Rozhraní CryptoKey

CryptoKey objekt reprezentuje kryptografický klíč, který je spravován přímo prohlížečem. Jedná se o zásadní změnu, protože kryptografický materiál již není součástí běžného prostředí, ale je s ním nakládáno zcela odlišným způsobem. Tohle je důvod, proč není doporučeno generovat klíče pomocí *getRandomValues*. Pokud bychom tímto způsobem generovali daný klíč, pak by pro výslednou hodnotu neplatila pravidla pro daný bezpečný režim (hodnota by se standardně uložila do paměti a byla spravována GC).

Ke klíčům tak není možné přistupovat přímo skrze Javascript, a proto musíme použít dostupné metody. Jedinná výjimka, kdy můžeme získat klíč přímo, je při jeho exportu. Docílíme toho nastavením vlastnosti *extractable*. Musíme ovšem počítat s tím, že se jedná o bezpečnostní riziko. [17]

Klíče jsou rozděleny na tři typy podle jejich použití. Jsou jimi *public*, *private* a *secret*, kdy je dvojice *public-private* použita pro asymetrickou kryptografii a *secret* pro symetrickou. Klíče, které nejsou typu *private* nebo *secret* nebo mají hodnotu *extractable* nastavenou na *true*, mohou být po exportu běžně přístupné.

Dále objekt obsahuje vlastnost *KeyUsage*, která nám specifikuje, pro jaký typ operace je klíč určen (šifrování, dešifrování, podpis, aj.). Zjednodušený popis *CryptoKey* můžeme najít v příkladu 2.4.

Příklad 2.4 CrytoKey WebIDL (zjednodušená verze) [17]

```
KeyType { public, private, secret };
KeyUsage { encrypt, decrypt, sign, verify, deriveKey, deriveBits,
    wrapKey, unwrapKey };
CryptoKey { KeyType type; boolean extractable; object algorithm;
    object usages; };
```

2.4.3 Rozhraní SubtleCrypto

Toto rozhraní je jádrem celého API a je přístupné skrze objekt *crypto*. Zpřístupňuje nám kryptografické operace umožňující šifrovat, dešifrovat, podepisovat nebo ověřovat, generovat klíče, apod. Obvykle se tak děje na základě uvedeného algoritmu, klíče a vstupních dat (to však závisí na uvedené operaci). Na příkladu 2.5 můžeme najít zjednodušený popis poskytovaných metod.

Příklad 2.5 SubtleCrypto WebIDL (zjednodušená verze) [17]

```
encrypt(algorithm, key, data);
decrypt(algorithm, key, data);
sign(algorithm, key, data);
verify(algorithm, key, signature, data);
digest(algorithm, data);
generateKey(algorithm, extractable, keyUsages);
deriveKey(algorithm, baseKey, derivedKeyType, extractable, keyUsages
    );
deriveBits(algorithm, baseKey, length);
importKey(format, keyData, algorithm, extractable, keyUsages);
exportKey(format, key);
wrapKey(format, key,wrappingKey, wrapAlgorithm);
unwrapKey(format, wrappedKey, unwrappingKey,
unwrapAlgorithm, unwrappedKeyAlgorithm, extractable, keyUsages);
```

Na rozdíl od *RandomSource*, jsou všechny tyto operace poskytované rozhraním *SubtleCrypto* asynchronní. Je to z toho důvodu, že se jedná o výpočetně náročnější operace a je logické, aby se tak dělo asynchronně. Toho je docíleno využitím *Promise*. Jednoduchou ukázku toho, jak Web Cryptography API a *Promise* fungují, můžeme vidět na příkladu 2.6.

Příklad 2.6 Příklad zašifrovaní zprávy [10]

```
const ptUtf8 = new TextEncoder().encode('my plaintext');
const pwUtf8 = new TextEncoder().encode('my password');

const pwHash = await crypto.subtle.digest('SHA-256', pwUtf8);

const iv = crypto.getRandomValues(new Uint8Array(12));
const alg = { name: 'AES-GCM', iv: iv };
const key = await crypto.subtle.importKey('raw', pwHash, alg, false,
    ['encrypt']);

const ctBuffer = await crypto.subtle.encrypt(alg, key, ptUtf8);
```

2.4.4 Podpora Web Cryptography API

Web Cryptography API je již implementováno ve všech hlavních prohlížečích. Přehled podpory API můžete najít v tabulce 2.2, kde jsou uvedeny nejpoužívanější prohlížeče. API je podporovnáno také v prohlížečích pro mobilní zařízení, jedná se však většinou o poslední aktuální verze.

	Chrome	**Mozilla**	**Opera**	**Safari**	**IE**	**Edge**
Verze	37+	34+	24+	8+	11	12+

Tabulka 2.2: Podpora Web Cryptography API v prohlížečích. Zdroj [9]

Názvy některých objektů se můžou napříč prohlížeči lišit a je proto nutné tomu věnovat zvýšenou pozornost. Příkladem může být Internet Explorer, kde je objekt *crypto* dostupný pod názevm *msCrypto* nebo Safari, kde ho máme vedený pod *webkitCrypto* (v předchozích verzích).

Standard specifikuje také nové výjimky, které nejsou v prohlížečích ještě plně implementovány, a které může API vracet. Přehled chybových hlášení a jejich význam, se kterými API pracuje, je v tabulce 2.3.

Název	**Popis**
NotSupportedError	Požadovaný algoritmus není podporován
SyntaxError	Chybí vstupní parametr funkce nebo je mimo rozsah
InvalidAccessError	Požadovaná operace se neslučuje se specifikovaným klíčem
DataError	Vstupní data neodpovídají daným požadavkům
OperationError	Operace selhala ze specifického důvodu

Tabulka 2.3: Chybová hlášení Web Cryptography API. Zdroj [12]

Problémem některých chybových hlášení je to, že z nich nelze poznat, kde nastala chyba. Příkladem může být *OperationError*. Tato výjimka je vrácena při jakémkoli neúspěšném pokusu například dešifrovat vstupní řetězec. To znamená, že zdrojem chyby může být více faktorů a nelze přesně určit, kde nastala. Příkladem může být dešifrování pomocí AES-GCM, kdy je výjimka *OperationError* vrácena v těchto případech:

- tag nemá předepsanou délku
- inicializační vektor nemá předepsanou délku
- dodatečná data nemají předepsanou délku
- autentizace zprávy selhala (neplatný klíč, pozměněná zpráva)

V takových případech pak nelze ihned určit příčinu chyby a je nutné krok po kroku analyzovat daný kód.

Podporované kryptografické algoritmy

Podpora algoritmů během tvorby standardu prošla řadou změn. Výsledný seznam, který by měl být standardně podporován je v tabulce 2.4. Během vývoje byly ze seznamu odstraněny algoritmy jako RSAES-PKCS1-v1_5, CONCAT, DH, AES-CMAC a další.

Použití	**Algoritmus**
Šifrování a zapouzdření	RSA-OAEP, AES-CTR, AES-CBC, AES-GCM, AES-KW
Podpis	RSASSA-PKCS1-v1_5, RSA-PSS, ECDSA, HMAC
Hašování	SHA-1, SHA-256, SHA-384, SHA-512
Derivace klíče	ECDH, HKDF, PBKDF2

Tabulka 2.4: Podporované algoritmy. Zdroj [12]

2.4.5 Případy užití Web Cryptography API

Vícefaktorová autentizace

Webová aplikace by mohla roršířit nebo nahradit stávající schémata ověřování založená na uživatelských jménech a heslech metodami autentizace postavených na prokázání se osobním certifikátem nebo jiným materiálem. Místo ověřování pomocí přenosové vrstvy, může webová aplikace upřednostit uživatelsky přívětivější varianty přímo v aplikaci. [12]

Pomocí Web Cryptography API by aplikace mohla najít vhodné klientské klíče, které mohly být již vygenerovány samotnou aplikací nebo doručeny jiným způsobem klientovi. Aplikace by pak mohla provádět kryptografické operace spojené s autentizací účastníka komunikace. Komunikace by samozřejmě mohla být ještě posílena pomocí TLS.

Elektronická výměna dat

Při elektronické výměně dat, typicky při odesílání různých elektronických dokumentů obsahující citlivé nebo osobní informace, můžeme pomocí webové aplikace chtít zajistit, aby byly dokumenty přístupné pouze oprávněným stranám i poté, co byly bezpečně odeslány a přijaty, např. přes TLS. Webová aplikace tak může učinit zašifrováním pomocí symetrického klíče a jeho následným zabalením (*key wrap*) pomocí veřejného klíče protistrany. [19]

Cloudové úložiště

Při ukládání dat na vzdálené servery mohou chtít uživatelé zajistit jejich důvěrnost. Aplikace tak může data na straně klienta šifrovat a klíč uchovat pro budoucí dešifrování. Toto použití je podobné elektronické výměně dat. [19]

Podpis dokumentů

Webová aplikace může vyžadovat práci s elektronickými podpisy a také může chtít dokumenty podepisovat na základě vystaveného certifikátu. Nicméně API v tomhle ohledu

poskytuje pouze základní nástroje a pro realizaci opravdu funkční a reálné implementace budou potřeba navíc další funkční prvky. Ty lze přidat např. jako rozšíření pro prohlížeč, které načte HW klíč tak, jak je popsáno v [20].

Ochrana integrity dat

V případě, že klientská aplikace pracuje s lokálními daty, které dříve získala ze serveru, může zde být potřeba kontrolovat jejich integritu. To lze provést například tak, že data zašifrujeme soukromým klíčem přímo na serveru a klientské aplikaci poskytneme veřejný klíč pro ověření integrity dat. [19]

Zabezpečená výměna zpráv

Zatímco TLS může být využito pro zabezpečení zpráv mezi klientem a serverem, uživatelé mohou chtít zabezpečovat zprávy pomocí schémat jako například OTR (*off-the-record*). API umožňuje použití kryptografických funkcí, kterými lze docílit implementace tohoto schématu. [19]

Javascript Object Signing and Encryption (JOSE)

Aplikace může chtít pracovat se strukturou nebo zprávami ve formátu definovaném pomocí IETD JavaScript Object Signing and Encryption (JOSE). Pomocí Web Cryptography API může aplikace číst a importovat klíče kódované ve formátu JSON (JWK), validovat zprávy, u kterých je integrita chráněna pomocí digitálních podpisů nebo MAC (JWS) nebo dešifrovat zprávy, které byly pomocí JOSE zašifrovány (JWE). [12]

JOSE je celý framework, který poskytuje nástroje pro formátování a předávání klíčů JSON Web Key (JWK), nástroje pro digitální podpis a autentizační kódy JSON Web Signature (JWS) nebo nástroje pro šifrování JSON Web Encryption (JWE). JOSE obsahuje i JSON Web Token, který může být použit jako prostředek pro autentizaci a autorizace pomocí modelu OAuth 2.0. [4]

2.4.6 Bezpečnost Web Cryptography API

Web Cryptography API rozhodně zlepšuje situaci na poli kryptografie v prohlížeči. Oproti předchozímu stavu, kdy byly kryptografické funkce implementovány pomocí Javascriptovách knihoven, je současný stav z hlediska bezpečnosti mnohem lepší. Důležité je také zmínit, že s nástupem API se nám dostává kvalitní implementace kryptografických funkcí, které se nám dříve nemuselo dostávat.

Pokud se podíváme zpět do 2.2 a projdeme si jednotlivé body, zjistíme, že většina bezpečnostních rizik byla odstraněna. V prvním případě se jedná o kryptograficky bezpečný generátor pseudonáhodných čísel. Generátor využívá vhodných výchozích hodnot a jeho implementace je efektivní. V textu jsou dále rozebrány rizika popsané v 2.2 a také způsob, jakým je API odstraňuje.

Bezpečné doručení kódu

Dalším z problémů je doručení kódu s kryptografickými funkcemi na stranu klienta. Tento problém se prakticky vyřešil tím, že kryptografické funkce byly implementovány přímo v prohlížeči. Nastal ale druhý problém a tím bylo doručení samotného kódu, který s API

pracuje. Kdyby byl kompromitován ten, znamenalo by to stejný problém, jak kdyby byly pozměněny samotné kryptografické funkce.

V dřívější verzi specifikace se tento problém neřešil, ale s novou aktualizací, která přišla koncem roku 2016, bylo rozhodnuto, že kryptografické funkce budou přístupné pouze z bezpečného zdroje. *Secure Contexts* [7] je také nově vznikající standard, stejně jako Web Cryptography API, a je zaměřen právě na komunikaci klient-server. Jeho cílem je ochrana uživatele při komunikaci se serverem. *Secure contexts* prakticky vynucuje zabezpečenou komunikaci, bez které není API přístupné. Jinými slovy, bez zabezpečeného spojení není přístupné rozhraní *SubtleCrypto* a *CryptoKey*. Rozhraní *Crypto* i *RandomSource* zůstává přístupné i bez zabezpečeného spojení.

Je otázkou, zda je *Secure contexts* řešením problému. Existuje celá řada útoků na zabezpečené spojení s cílem získat nad ním kontrolu a pokud není server správně nakonfigurován, je velice pravděpodobné, že daná bezpečnostní opatření nebudou plnit svůj účel.

Zavedením *Secure contexts* se také omezuje použití Web Cryptography API, protože ne všechny weby používají zabezpečené spojení. I když tu dnes máme certifikáty Let's encrypt, webů bez zabezpečeného spojení je stále poměrně mnoho (samotný certifikát může být zdarma, ale mnohé hostingy tuto službu zpoplatňují). Použití API je tak omezeno i v případě, kdy není potřeba taková míra zabezpečení.

Práce s kryptografickým materiálem

Řešení problému práce s kryptografickým materiálem bylo nastíněno v 2.4.2. Web Cryptography API přistupuje k danému materiálu zcela odlišným způsobem a používá k tomu několik bezpečnostních mechanismů.

Prvním je samotný způsob uložení kryptografických klíčů, které nejsou vystaveny přímo, ale jsou dostupné pouze skrze API. Je tak zaručen bezpečný režim práce s kryptografickým materiálem, který nám dříve chyběl. Jedinou výjimkou, jak už bylo zmíněno, je parametr *extractable*, díky kterému můžeme klíč exportovat. Samotné úložiště není perzistentní, takže při přechodu na jinou stránku budou všechny uložené klíče ztraceny. Klíče mohou být také odstraněny stejným způsobem jako se odstraňuje obsah *LocalStorage* nebo *Cookies*. Je to dáno tím, že pro ukládání klíčů je použita implementace *Structured Clone* algoritmu, který se využívá taktéž pro perzistentní úložiště. [17]

Druhým bezpečnostním mechanismem je model *Same Origin Policy* (SOP). Objekt typu *CryptoKey* je dostupný pouze ze stejného zdroje. Jinými slovy, pokud budou mít dvě webové stránky uloženy své kryptografické klíče, tak ani jedna nebude mít přístup ke klíčům té druhé. Stejného modelu je využito také k přístupu k perzistentím úložištím jako je *LocalStorage* nebo *IndexedDB*. Příklad fungování SOP můžeme vidět na tabulce 2.5, kdy je jako výchozí adresa brána `http://store.company.com/dir/page.html`. [6, 5]

Pokud potřebujeme perzistentní úložiště pro kryptografický materiál, jedinou možností je použít *IndexedDB*, která nám umožňuje uložit objekt typu *CryptoKey*. V tom je výhoda tohoto úložiště, neboť nám dovoluje uložit klíč tak, abychom ho nemuseli přímo vystavit standardnímu prostředí. Tento způsob ukládání ale není úplně ideální a jeho možná rizika jsou rozebrána dále.

Zranitelnosti Web Cryptography API

Pravděpodobně největší zranitelností API je *Cross-site scripting* (XSS), konkrétně DOM-based XSS. Pokud se v aplikaci vyskytuje tato zranitelnost, je Web Cryptography API proti útoku prakticky bezbranné. XSS je jednou ze současných hrozeb se kterou musí vývojář

URL	Výsledek	Zdůvodnění
http://store.company.com/dir2/other.html	úspěch	
http://store.company.com/dir/inner/another.html	úspěch	
https://store.company.com/secure.html	neúspěch	protokol
http://store.company.com:81/dir/etc.htm	neúspěch	port
http://news.company.com/dir/other.html	neúspěch	hostitel

Tabulka 2.5: Příklad fungování Same Origin Policy. Zdroj [5]

počítat, zvláště v tomto případě. Veškerý kryptografický materiál je uložen v objektu typu *CryptoKey*, který je zpravidla non-perzistentní, ale může být uložen do *IndexedDB* a tím se stává perzistentním (viz předchozí text). Útok XSS může být veden s cílem získat tento materiál.

Není to ale tak jednoduché jak se zdá. Útočník se ke klíči může dostat, ale pokud byl klíč vytvořen jako *non-extractable*, nelze získat jeho hodnotu přímo a jeho použití je tak omezeno pouze na danou stránku (není ho možné exportovat, resp. číst).

Další zranitelnost nesouvisí přímo s API jako takovým, ale s dostupnými kryptografickými algoritmy. Některé algoritmy nejsou za jistých okolností bezpečné a očekává se, že autor, který vytváří jakýkoli kód, pracující s API, bude obeznámen s touto problematikou. Příkladem můžou být algoritmy AES v módech CBC a CTR. V těchto módech je zaručena pouze důvěrnost zprávy, ale už není zajištěna její integrita. Tyto módy, na rozdíl od GCM, neobsahují autentizační kódy a pro zajištění plné bezpečnosti musí být doimplementovány. Obdobně je na tom *PBKDF2*, kde aby byla zaručena požadující bezpečnost, je nutné provést minimálně tisíc iterací zadané hašovací funkce. Přehled doporučených algoritmů je v tabulce 2.6. Dále je doporučeno používat pro symetrické šifry a eliptické křivky klíče o délce minimálně 256 bitů a pro RSA klíče použít minimálně 2048 bitů. [17]

Použití	Algoritmus
Symetrické šifrování	AES-GCM
Asymetrické šifrování	RSA-OAEP
Digitální podpis	RSA-PSS
Výměna klíčů	ECDH
Autentizační kódy	HMAC

Tabulka 2.6: Doporučené symetrické a asymetrické algoritmy. Zdroj [17]

2.4.7 WebCrypto Key Discovery

Standardně kryptografické klíče vznikají tak, že jsou importovány nebo přímo generovány v prohlížeči. WebCrypto Key Discovery ale počítá s kryptografickým materiálem, který bude

přístupný dávno předtím. Zjednodušeně bychom to mohli přirovnat k Trusted Platform Modulu (TPM). Prohlížeč bude poskytovat klíče, které budou *named*, *specific-origin* a *pre-provisioned*, jinými slovy:

- **named** - pojmenované nebo-li rozlišitelné
- **specific-origin** - vázané k danému zdroji (webové stránce)
- **pre-provisioned** - předem připravené k použití (již vygenerované)

Tímto způsobem budou přístupné klíče, které lze použít různými způsoby (např. zapouzdření klíče). WebCrypto Key Discovery je standard vznikající po boku Web Cryptography API od roku 2013. V prohlížečích ale zatím nebyl implementován a na jeho vývoji se stále pracuje. [13]

Rizika WebCrypto Key Discovery

Tento mechanismus předem připravených klíčů ale vytváří rizika spojená s ohrožením soukromí uživatele. Tyto klíče jasně daného uživatele identifikují a může být pomocí nich sledován nebo penalizován.

Kapitola 3

Návrh

Kapitola se věnuje návrhu aplikace pro šifrování webových stránek a jejich částí a návrhu knihovny, která zpřístupní tento šifrovaný obsah autorizovanému uživateli. Nejprve jsou v 3.1 analyzovány požadavky na výsledné řešení a následně je v dalších podkapitolách řešen samotný návrh aplikace a knihovny.

3.1 Analýza požadované funkčnosti

Cílem práce je vytvořit Javascriptovou knihovnu, která bude rozšiřovat možnosti statických webových stránek tak, že za pomocí kryptografie přidá možnost autorizovaného přístupu k jejich obsahu. Měla by tak být přidána možnost omezit přístup k celým webovým stránkám nebo pouze jejich částem, popřípadě i ke stahovatelnému obsahu. K tomu by měla využívat nový standard Web Cryptography API popsaný v 2.4.

3.1.1 Požadavky na funkčnost

Práce s knihovnou by měla být relativně snadná jak pro vývojáře, tak pro uživatele daných webových stránek. Knihovna by vývojáře neměla nijak omezovat a měla by umožňovat jistou míru *customizace*. To znamená, že by měla například dávat možnost bez problému zobrazit místa webové stránky, která obsahují šiforvaný obsah tak, jak je vývojář nadefinuje.

Měl by být vytvořen také systém řízení přístupu, aby bylo jasně definované, který uživatel nebo skupina uživatelů, má k jakému zdroji, resp. webové stránce nebo její části, přístup. Tohle by mělo být navrženo tak, aby byl výsledný model přístupu přehledný a nepůsobil zbytečné komplikace.

Knihovna by také měla zohlednit to, že vývojáři si nemusí být vědomi bezpečnostních rizik při volbě algoritmu nebo volbě jejich parametrů. V základu by tedy měly být definovány bezpečné algoritmy s vhodně zvolenými parametry. To stejné se týká i šifrovacích klíčů, kdy by nemělo docházet k volbě slabého šifrovacího klíče.

3.1.2 Požadavky na implementaci

Vzhledem k tomu, že je knihovna určena pro statické webové stránky, je jasnou volbou implementačního jazyka Javascript, resp. HTML5. Dalším implemetačním požadavkem by mělo být také to, aby byla k dispozici minifikovaná forma knihovny a nevyžadovalo se příliš mnoho závislotí, které by komplikovaly její reálné nasazení.

Požadavkem je také to, aby bylo výsledné řešení integrovatelné do nějakého z generátorů statických webových stránek, např. Hugo nebo Jekyll. Vzhledem k tomu, že je Jekyll dostupný pouze na OS Linux, je vhodnější volbou rozšířenější Hugo, který je multiplatformní. Problémem ale je, že nepodporuje svou rozšiřitelnost pomocí modulu. Musí se tak najít způsob, jak vhodně integrovat výsledné řešení.

3.2 Obecný návrh řešení

Nejprve je třeba navrhnout koncept celého řešení. V 3.1 jsme si přibližně představili, co všechno by mělo výsledné řešení obsahovat. Hned při prvním prozkoumání je jasné, že to všechno nepůjde implementovat v rámci jedné knihovny, ale bude potřeba vytvořit dva samostatné programy, resp. aplikaci pro šifrování a knihovnu pro dešifrování. Při návrhu budeme muset řešit několik problémů:

- jak identifikovat obsah stránky určený k zašifrovaní
- jakým způsobem uložit šifrovaný obsah bloku nebo stránky
- jakým způsobem na stránce definovat místo s šifrovaným obsahem
- jak pracovat s uživateli a zajistit jejich přístup k různým zdrojům
- jak zajistit bezpečnost celé knihovny vzhledem ke známým skutečnostem

Všechny uvedené problémy jsou poměrně zásadní a je třeba s nimi při návrhu počítat a ideálním způsobem se s nimi vypořádat. Musíme zdůraznit, že důležitým faktorem při návrhu je samotný uživatel a také vývojář, který bude s knihovnou pracovat. Cílem není vytvořit něco úplně nového, ale navrhnout výsledné řešení tak, aby bylo blízké aktuálním trendům.

Celý proces zpracování webové prezentace můžeme logicky rozdělit na dvě části tak, jak budeme implementovat aplikaci pro šifrování a knihovnu pro dešifrování. V první části procesu, kterou bychom mohli nazvat *postprocessingem*, se provede zašifrování webové stránky a její příprava pro publikaci online. Druhá část procesu již bude probíhat ve webovém prohlížeči, kdy se jednotlivé stránky dešifrují a budou se zobrazovat uživateli podle toho, zda má oprávnění k přístupu. Toto je obecný koncept řešení, nyní přejděme na popis hlavních bodů.

3.2.1 Konfigurace a konfigurační soubory

Celý systém využívá několik druhů konfiguračních souborů. V prvé řadě se jedná o centrální (hlavní) konfigurační soubor a dále o konfigurační soubory pro autorizovaný přístup (viz 3.2.2). Veškeré konfigurační soubory jsou ve formátu JSON a to z důvodu jeho podpory ze strany webových technologií. Ukázku hlavního konfiguračního souboru můžeme vidět na příkladu 3.1.

Příklad 3.1 Hlavní konfigurační soubor

```
1 {
2     "site-name": "J3A Demo",
3     "uri-base": "https://praserx.github.io/j3a/demo",
4     "uri-boot": "security/boot.script.html",
5     "uri-acl": "security/acl.json",
6     "uri-roles": "security/roles.json",
```

```
7       "uri-version": "security/version.json",
8       "uri-users-dir": "security/users",
9       "uri-resources-dir": "security/resources",
10      "denied-info-element": "security/deniedWarning.html",
11      "denied-info-page": "security/denied.html",
12      "allow-cache": "true",
13      "auto-logout": "true",
14      "algorithms": {
15          "public-key-encryption": "RSA-OAEP",
16          "private-key-encryption": "AES-GCM",
17          "digest": "SHA-512",
18          "sign": "",
19          "key-derivation": "PBKDF2"
20      },
21      "perm-groups": [],
22      "file-perm-groups": []
23  }
```

Parametry s prefixem `uri` obsahují důležité cesty k dalším konfiguračním souborům nebo zdrojům. Parametry s prefixem `denied` zase cesty k šablonám pro zobrazení nepřístupného obsahu. Důležitým parametrem je také `algorithms`, který definuje, jaké algoritmy mají být při šifrování použity.

Konfigurační soubory pro autententizaci a autorizaci jsou ve výsledku vždy ve dvou variantách. První varianta je k dispozici před procesem šifrování a obsahuje všechny podstatné informace pro daný proces (např. výchozí heslo uživatele). Druhá varianta pak nahrazuje první a obsahuje důležité šifrovací klíče. Pro jednoduchost budeme dále nazývat první variantu jako *plaintext variantu*, a druhou jako *ciphertext variantu*.

3.2.2 Autorizovaný přístup

Prakticky na každém webu, který není statický, mají uživatelé možnost se přihlásit a provádět po přihlášení další akce nebo zobrazovat obsah, který byl do té doby nepřístupný. V našem případě jde spíše o druhou možnost, i když první není úplně vyloučena. Systém by měl umožnit uživatele registrovat (evidovat), autentizovat a následně autorizovat pro přístup k daným částem webu.

Aby bylo možné tyto operace provádět, je potřeba zavést určitý systém řízení přístupu (*Access Control*). Variant se nám v tomto směru dostává hned několik:

- Mandatory Access Control (MAC)
- Discretionary Access Control (DAC)
- Role-Based Access Control (RBAC)
- Cryptographic Access Control (CAC)
- a další

Modely MAC a DAC pro nás nejsou příliš ideální, protože nesplňují náš požadavek na současné trendy a také nejsou ideální proto, jakým způsobem fungují. Vhodným řešením se zdá být RBAC, který je prakticky kombinací MAC a DAC. RBAC přiděluje uživatelům role, na základě kterých jsou jim povoleny/odepřeny různé operace (např. čtení souboru). RBAC se nejvíce podobá současnému stavu, kdy se v informačních systémech nejčastěji setkáváme s přidělováním práv právě pomocí definovaných rolí.

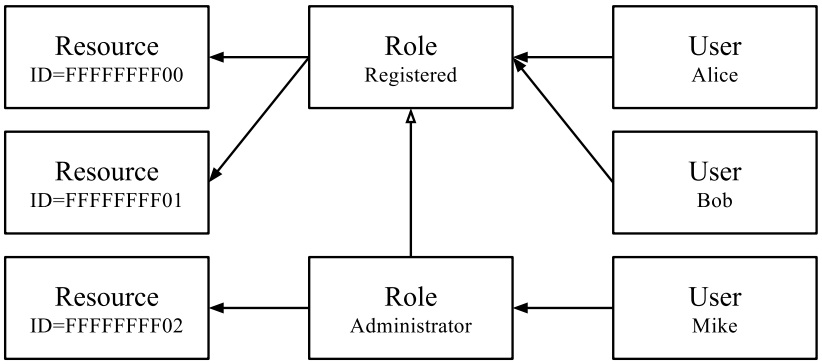
Navíc tu máme ještě poměrně nový model, kterým je CAC. Jedná se o obecnější model řízení přístupu schopný poskytovat zabezpečení v různých kontextech, aniž by byla vyžadována rozsáhlá změna základní architektury systému. Bezpečnostní schémata CAC jsou obvykle modelována jako částečně uspořádané množiny tříd zabezpečení, které představují skupiny uživatelů. Kryptografické klíče pro dané zdroje jsou pak přiřazeny jednotlivým skupinám. Mějme částečně uspořádanou množinu $(S, \preceq)$, kde $S = \{U_0, U_1, ..., U_n\}$ reprezentuje skupinu uživatelů. Pak platí $U_i \preceq U_j$, což znamená, že skupina uživatelů U_j může získat přístup k informacím určeným pro skupinu U_i, ale nikoli obráceně. [21]

I přesto, že je CAC založen na kryptografii a sdílení klíčů mezi uživateli v rámci skupin, jeví se stále jako nejvhodnější RBAC. Budeme proto volit kombinaci řízení přístupu založeného na rolích a řízení přístupu postaveném na kryptografii.

Model řízení přístupu

Navržené řízení přístupu vychází z modelů RBAC a CAC. Oproti ostatním systémům je ten náš mnohem jednodušší vzhledem k povaze statického webu. Prakticky máme k dispozici jen jednu dostupnou operaci, kterou je čtení. I když model vychází z RBAC, nepřebírá všechny jeho nevýhody a rizika, a to proto, že využíváme jen jednu operaci pro čtení obsahu, resp. přístupu.

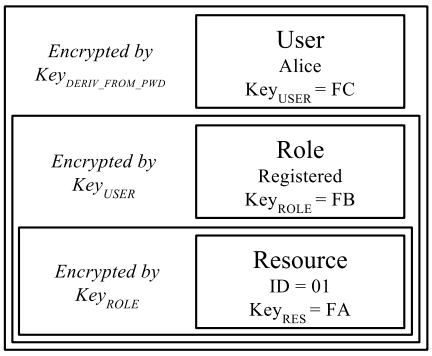
Model obsahuje tři prvky: zdroj (*resource*), roli (*role*) a uživatele (*user*). Zdrojem je myšlena webová stránka nebo její část, ke které chceme přistupovat. Každá role má pak přiřazeno, ke kterým zdrojům má přístup, tzn. oprávnění číst. Uživatel je pak entita, které může mít až n rolí. Příklad lze vidět na obrázku 3.1.



Obrázek 3.1: Řízení přístupu RBAC

Z obrázku 3.1 plyne ještě jedna důležitá skutečnost, a tou je dědičnost rolí. Každá role může dědit oprávnění od jedné či více jiných rolí a získá tak přístup k jejich zdrojům.

Do celého modelu musí být také zakomponována samotná kryptografie. Musíme si uvědomit, že každý zdroj musí být šifrován vlastním klíčem, každá role musí být šifrována vlastním klíčem a pak také uživatel musí mít vlastní klíč. Celkem tu máme 3 druhy kryptografických klíču (prakticky je jich více, viz dále). Nejprve celý proces zašifruje zdroj a jeho klíč předá oprávněným rolím, poté se zašifrují role a jejich klíče jsou předány uživatelům, kterým jsou role přiřazeny. Způsob *obalování* klíčů je ilustrován na obrázku 3.2.



Obrázek 3.2: Příklad obalování kryptografických klíčů

Uživatelský účet

Pokud budeme vycházet ze standardního modelu informačního systému, použijeme pro autentizaci uživatele jméno a heslo a při autorizaci budeme kontrolovat příslušnost k požadované roli. Nemusíme navíc zůstávat jen u autentizace pomocí jména a hesla. Pro přístup můžeme využít i asymetrickou kryptografii a autentizovat tak uživatele pomocí certifikátu.

Veškeré informace o registrovaných uživatelích jsou vedeny v jedné složce, kterou definuje konfigurační soubor (`uri-users-dir`). Ta obsahuje soubory s jednotlivými uživateli v definovaném formátu. Plaintext variantu můžeme vidět na příkladu 3.2

Příklad 3.2 Ukázka definice uživatele admin (plaintext varianta)

```
1 {
2     "username": "admin",
3     "password": "foxtrot",
4     "roles": [ "Administrator" ]
5 }
```

V případě, že by se chtěl uživatel přihlašovat pomocí certifikátu, namísto parametru `password` by pak soubor obsahoval parametr `certificate` s cestou k příslušnému certifikátu (veřejnému klíči) pro šifrování.

Ciphertext varianta je od plaintext varianty odlišná, to můžeme vidět na příkladu 3.3. Soubor obsahuje `username`, `roles`, `key-type` a navíc `secret-algorithm`, popisující algoritmus použitý při zašifrování důvěrných klíčů a `secret`, který obsahuje ony důvěrné šifrovací klíče k rolím uživatele. Parametr `secret` je vlastně *ciphertext*, který šifrován klíčem derivovaným z uživatelského hesla a v případě asymetrické kryptografie je šifrován přiloženým veřejným klíčem.

Příklad 3.3 Ukázka definice uživatele admin (ciphertext varianta)

```
1 {
2     "username": "admin",
3     "roles": [ "Administrator" ],
4     "key-type": "password",
5     "salt": "",
6     "secret-algorithm": {
```

```
7           "name": "AES-GCM",
8           "iv": "2F732D5BA17EC2DCE6AF7341B69CEA2F",
9           "tag": 128
10      },
11      "secret": "0AB4E66C9A6F5A6D98F24CD...",
12  }
```

Zaměřme se ještě na parametr `secret`. Ten neobsahuje pouze šifrovací klíče k daným rolím, ale také navíc popis algoritmu, kterým byly role zašifrovány. Parametr `secret-algorithm` naopak popisuje algoritmus, kterým byl zašifrován samotný parametr `secret`.

Pokud šifrujeme `secret` pomocí asymetrické kryptografie, je výstupní ciphertext varianta doplněna o parametry `key-secret`, obsahující zašifrovaný (zabalený) symetrický klíč a `key-algorithm`, obsahující popis použitého asymetrického algoritmu. Odpovědí na otázku, proč nešifrujeme data přímo, ale šifrujeme klíč je, že délka klíče je úzce spjata s délkou vstupních dat. V případě klíče šifrujeme vždy relativně krátký řetězec konstantní délky.

Definice rolí

Soubor obsahující všechny role je oproti souboru s informacemi o uživateli poměrně jednoduchý. Prakticky se jedná o pole, které obsahuje jednotlivé jména rolí a případně jejich dědičnosti. Každá role může dědit přístupová práva jiné role. Toho docílíme tak, že všechny role, od kterých máme dědit, uvedeme do pole `inherits`. Při zpracování souboru pak budou získány všechny závislosti.

Příklad 3.4 Ukázka konfiguračního souboru pro definice rolí (plaintext varianta)

```
1  [
2      { "role": "Zamestnanec" },
3      { "role": "Externista" },
4      {
5          "role": "Administrator",
6          "inherits": [ "Zamestnanec", "Externista" ]
7      }
8  ]
```

V případě ciphertext varianty je situace stejná jako u souborů s uživateli. Navíc zde také přibyl parametr secret, který obsahuje šifrovací klíče a popisy šifrovacích algoritmů k přiřazeným zdrojům.

Příklad 3.5 Ukázka konfiguračního souboru pro definice rolí (ciphertext varianta)

```
1  [
2      {
3          "role": "Zamestnanec",
4          "secret": "72840346BB512E465AE58C7EB2F1A...",
5          "inherits": []
6      }, {
7          "role": "Externista",
8          "secret": "555E1B0710F72AE4C44B2F1A3B25A...",
9          "inherits": []
```

```
10      }, {
11          "role": "Administrator",
12          "secret": "0AE601B09341592A15D9AB7F8DF8D...",
13          "inherits": [ "Zamestnanec", "Externista" ]
14      }
15  ]
```

3.2.3 Neveřejné části webových stránek

Na počátku celého procesu postprocessingu je vždy k dispozici webová prezentace, která je nachystána pro zveřejnění. V prvé řadě musíme v dané prezentaci rozlišit, který obsah šifrovat a skrýt ho tak před neoprávněnými uživateli, a který naopak zobrazit. Nejen, že danou oblast musíme nějakým způsobem rozlišit, ale musíme také uvést, která skupina uživatelů bude mít přístup. Pokud uživatel nebo skupina uživatelů nebude mít ke stránce přístup, je nutné definovat, jak se v takovém případě zachovat.

Vymezení důvěrného obsahu

Při vymezování obsahu stránky musíme počítat s tím, že daný systém musí jít aplikovat na vybraný generátor statických webových stránek. Některá řešení jsou tak nereálná, protože tyto generátory z pohledu obyčejného uživatele nepracují s HTML kódem, ale vlastními šablonami. Vymezení bloku nebo strány, by tak mělo jít aplikovat i v těchto šablonách. Pokud ale tento problém pomineme, máme několik možností, jak blok uvodit:

- použijeme stávající značky HTML rozšířené o námi definované atributy
- uvodíme požadovaný kód definovanými značkami (kódem)
- uvodíme požadovaný kód vhodně formátovanými HTML komentáři

První způsob by pravděpodobně nefungoval. Není nijak zaručeno, jak by se systém generátoru zachoval, kdybychom v šabloně uvedli HTML kód. Pravděpodobně by značky převedl na escape sekvence a nám by tak znemožnil efektivně uvozený blok rozeznat. Uvodit požadovaný blok námi definovaným kódem také není ideální řešení. Pokud by autor nebo vývojář na stránce ještě pracoval a potřeboval ji by zobrazit, naše uvozovací značky by pravděpodobně narušily strukturu stránky.

Ideálním způsobem se tak zdá uvození pomocí HTML komentářů, protože je velká pravděpodobnost, že komentáře nebudou převáděny na escape sekvence a také nebudou narušovat výslednou stránku. Příklad použití komentářů pro uvození bloku můžeme vidět na příkladu 3.6.

Příklad 3.6 Vymezení obsahu stránky, který je určen jako důvěrný

```
1  <!--EE:BEGIN-->
2  <!--PERM:Externista,Administrator-->
3  <!--ODA:W-->
4  <p>Duverny obsah na strance.</p>
5  <!--EE:END-->
```

Celý blok je uvozen mezi značky `EE:BEGIN` a `EE:END`. Značka `PERM` (permission) určuje, kdo má k danému obsahu přístup. `ODA` (on-denied-action) nám říká, jak se máme zachovat v případě, že uživatel nebude mít přístup.

Poslední co zbývá vyřešit je způsob, jak znemožnit přístup na celou stránku. Prakticky si nemůžeme dovolit zašifrovat celou stránku včetně hlavičky, protože bychom museli řešit dost problematických věcí. Lepším způsobem, je uvodit stránku hned za značkou `body` a nastavit patřičně parametr `ODA`. Na stránku se tak bez problému dostaneme, ale přístup k obsahu nezískáme, pokud k tomu nemáme oprávnění.

Encrypted Element

Jakmile máme blok stránky zašifrovaný, potřebujeme na stránce zaznamenat místo kde se nacházel. Z určitých důvodů není možné na stránce nechat bajty zašifrovaného textu, a proto tento text nahradíme novým elementem s definovanými atributy. Ukázku *encrypted elementu* můžeme vidět na příkladu 3.7.

Příklad 3.7 Encrypted Element

```
1 <encrypted-element
2     resource-id="FFAAFFAAFFAA00CC"
3     oda="W">
4 </encrypted-element>
```

Element obsahuje dva atributy. Jedním je již známý `ODA` a druhým je `resource-id`. Namísto toho, aby byla do stránky přímo vložena zašifrovaná zpráva, byl vložen pouze element obsahující identifikátor určující skrytý obsah.

Zobrazování nepřístupného obsahu

K nastavení chování zobrazení nepřístupného obsahu slouží již zmiňovaný *On Denied Action*. Vývojář by měl mít možnost definovat vlastnosti chování v případě, že uživateli nebude umožněn přístup k danému obsahu. Máme tři možnosti jak se zachovat:

R (redirect) - uživatel bude přesměrován na stránku *Denied* definovanou pomocí konfiguračního souboru

W (warning) - blok s encrypted elementem bude nahrazen varovným hlášením definovaným pomocí konfiguračního souboru

H (hide) - obsah se jednoduše skryje

Konfigurační soubor zde hraje zásadní roli, obsahuje totiž adresy šablon, resp. jedné stránky a jedné šablony, které zobrazí namísto obsahu. Vzhledem k tomu, že uživatel v konfiguračním souboru tyto odkazy sám definuje, může si tak vytvořit vlastní hlášení nebo stránku dle svých představ a není ničím omezován.

Ukládání šifrovaného obsahu

Poslední věcí, kterou zbývá vyřešit je, co s šifrovaným obsahem. Jedna z možností by byla, nechat šifrovanou zprávu v *encrypted elementu* a jen ji skrýt. My jsme ale zvolili jinou

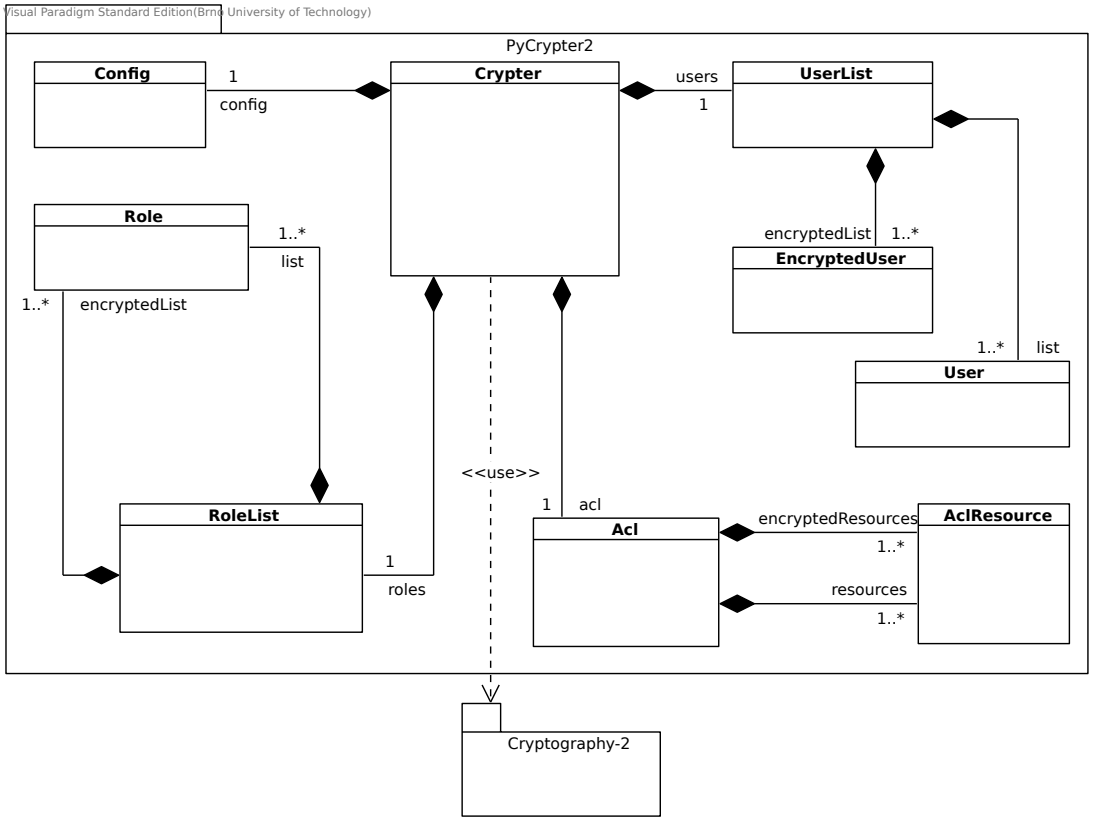
možnost. Obsah části stránky je v šifrované podobě uložen do samostatného souboru s unikátním identifikátorem. Důvod, proč bylo zvoleno právě toto řešení bude vysvětlen dále. Ukázku souboru, který je také ve formátu JSON, můžeme vidět na příkladu 3.8.

Příklad 3.8 Šifrovaný obsah stránky uložený v samostatném JSON souboru

```
1  { "ciphertext" : "0C0A598299C9B0D0877B8674893..." }
```

3.3 Aplikace pro šifrování webových stránek

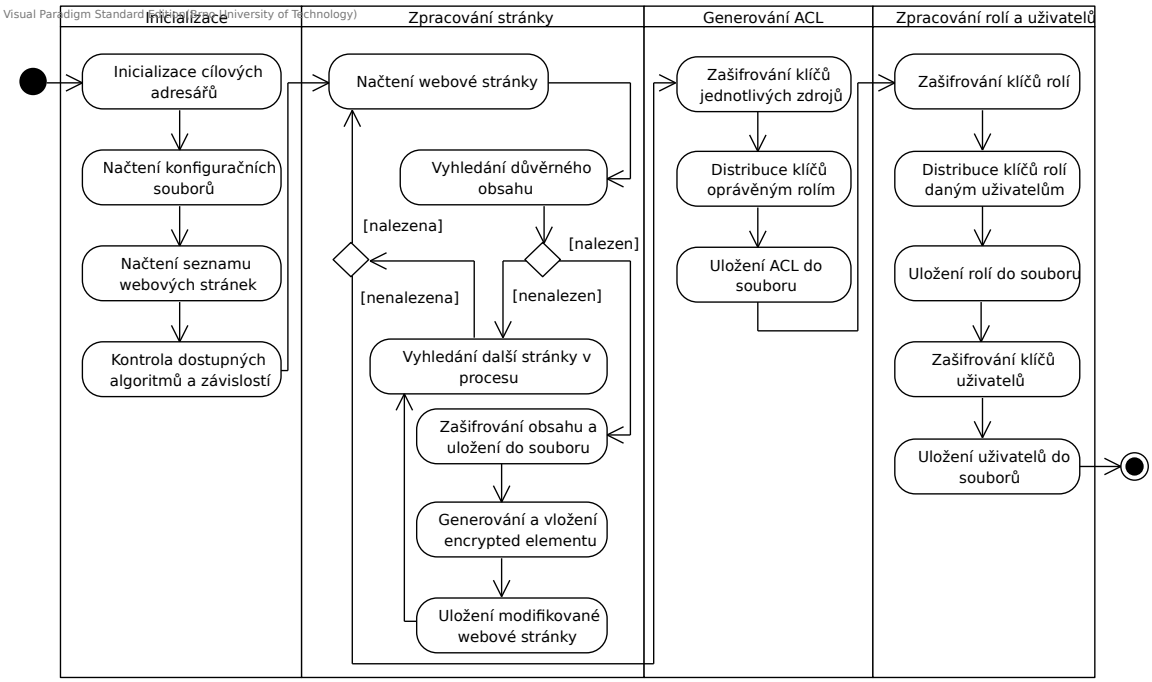
Aplikace pro šifrování se stará o postprocessing webové stránky. Zjednodušený model celé aplikace je na obrázku 3.3, kompletní model je pak v příloze na obrázku B.2. Model obsahuje několik tříd, ale tou nejdůležitější je třída `Crypter`, která zajišťuje celý proces zpracování a obstarává šifrování. Třída `Config` obsahuje načtená konfigurační data, `UserList` seznam registrovaných uživatelů, resp. objektů uživatelů, `RoleList` seznam rolí a jejich dědičností a `Acl` je třída, která je postupně plněna jednotlivými zdroji, ve strukturované podobě, načtených z HTML souborů.



Obrázek 3.3: Zjednodušený model aplikace pro šifrování webových stránek

3.3.1 Postprocessing webové stránky

Proces zpracování webové stránky není nijak extrémě složitý. Teoreticky je celý proces jednoduchý. Musíme načíst všechny HTML nebo HTM soubory obsahující zdrojové texty, analyzovat je a zašifrovat učená místa dle předchozího návrhu. Prakticky je ten proces komplikovanější. Celý postup ilustruje obrázek 3.4.



Obrázek 3.4: Proces zpracování webových stránek

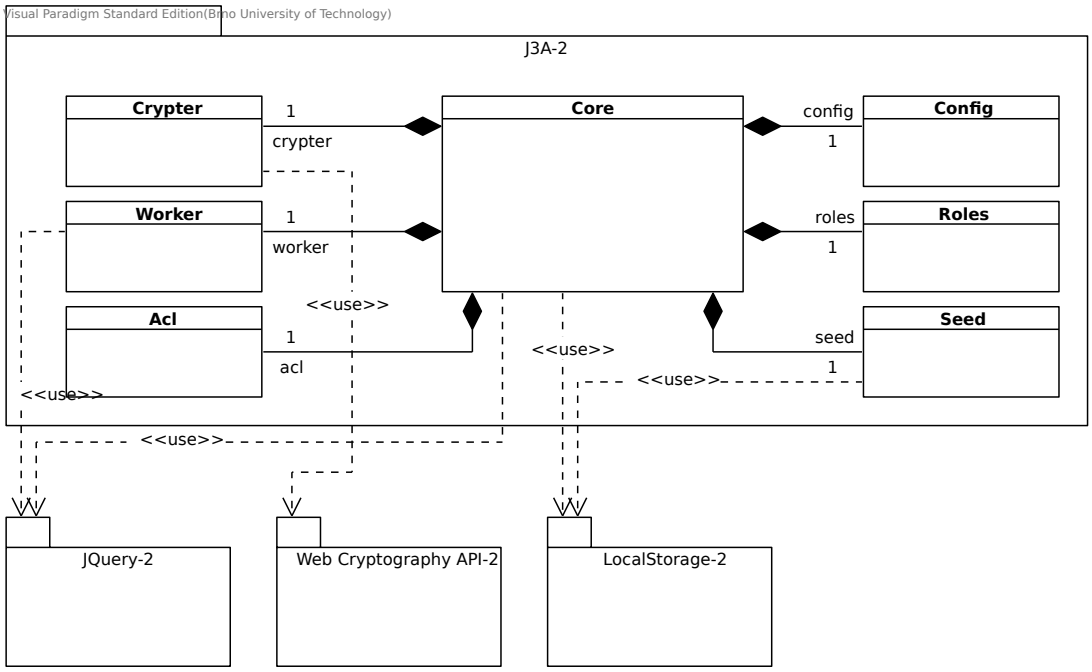
Proces zpracování můžeme rozdělet na 4 fáze. V první fázi se provede incializace, při které se načtou všechny konfigurační soubory a provede se kontrola závislostí a kompatibility algoritmů. V druhé fázi se provede zpracování webové stránky, uložení šifrovaného zdroje a uložení modifikované verze obsahující namísto *plaintextu* `encrypted-element`. Ve třetí fázi se vygeneruje ACL soubor, který obsahuje klíče k jednotlivým zdrojům a také seznam oprávněných rolí, které mohou ke zdroji přistupovat. V poslední, čtvrté fázi, se uloží soubor s rolemi obsahující klíče k odpovídajícím zdrojům. Stejný postup se uplatní s jednotlivými soubory, které definují uživatele.

3.4 Knihovna pro dešifrování webových stránek

Šifrování (postprocessing) stránky probíhá na straně uživatele, mimo webový prohlížeč, takže není potřeba extrémní opatrnost při zpracování dat. Předpokládáme totiž, že stroj, na kterém postprocessing probíhá, nebyl kompromitován. Při dešifrování stránek v prohlížeči musíme dbát na bezpečnost. V 2.2 jsme se seznámili s hrozbami a riziky, které nás mohou postihnout. Návrh tak musí probíhat s ohledem na bezpečnost a ochranu soukromí uživatele.

Zjednodušený návrh knihovny je na obrázku 3.5, výsledný model pak v příloze na obrázku B.1. Základem je třída `Core`, která se stará o inicializaci a načítání potřebných dat. Třídy `Config`, `Acl` a `Roles` v sobě pouze uchovají načtená data z konfiguračních souborů. Důležitější jsou zbylé tři třídy. `Crypter` zajišťuje veškrou kryptografickou funkčnost

knihovny a `Seed` (*Site Encrypted Element Database*) slouží pro získávání dat (důvěrných částí stránek) z lokálního úložiště (*LocalStorage*, *IndexedDB*). Třída `Worker` se stará o načítání a zpracování dat ze stránky a také o jejich nahrazování.



Obrázek 3.5: Zjednodušený model knihovny pro dešifrování webových stránek

3.4.1 Autentizační proces

V 2.2 a 2.4.6 jsme si představili bezpečností rizika spojená s kryptogafií v prohlížeči. Z hlediska bezpečnosti je autentizační proces nejrizikovější částí a našim úkolem je navrhnout takový autentizační proces, který minimalizuje tato nám známá rizika.

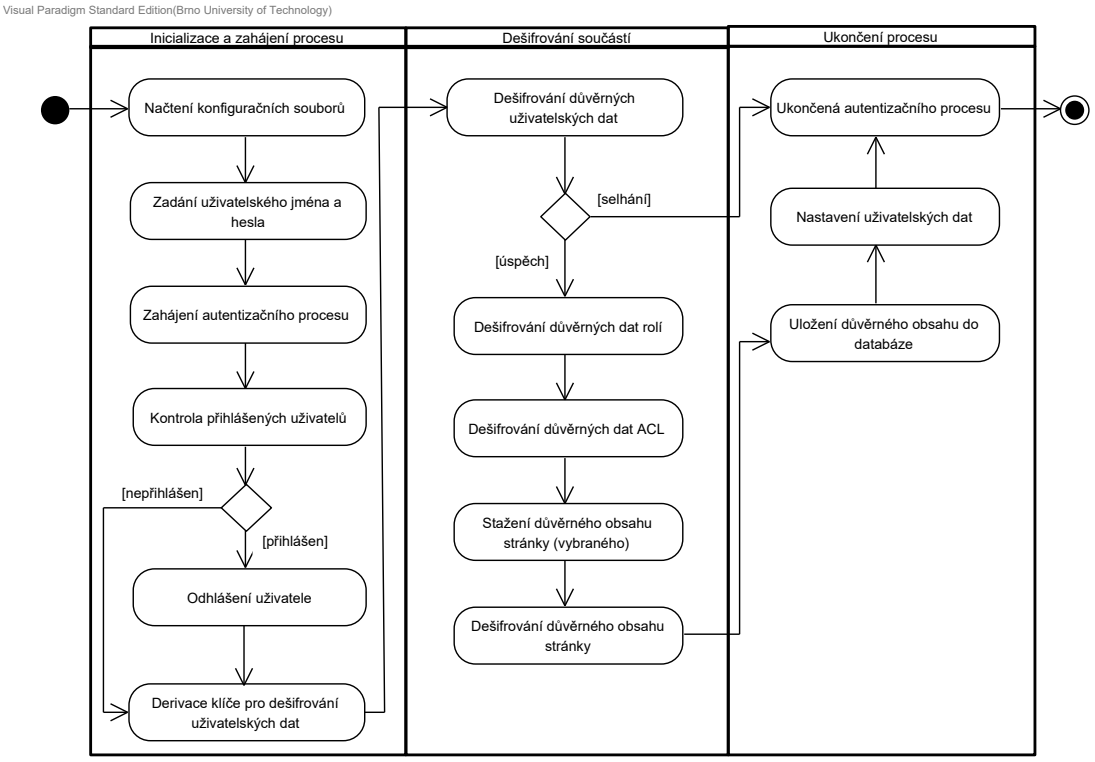
Nejprve se musíme zamyslet nad způsobem, jakým bude dešifrovat důvěrný obsah. Potřebujeme, aby uživatel nemusel zadávat klíč při každé operaci, protože by to pro něj bylo prakticky nereálné (proveditelné ano). Chceme aby se uživatel cítil jako na běžném webu, kde se přihlásí a o zbytek se již postará systém.

Víme, že klíče nejsou perzistentní a je také komplikované je uchovávat v databázi *IndexedDB*, protože je zde riziko útoku XSS. Navíc nemáme nad úložištěm plnou kontrolu a nemůžeme zajistit, že se po vypnutí prohlížeče obsah odstraní. Pokud by uživatel odešel ze stránek bez odhlášení, cenný kryptografický materiál by zůstal nezabezpečeně uložený v prohlížeči.

Dalším problémem je samotný systém. Veškeré klíče jsou uloženy v samostatném souboru, který musíme načíst a to nelze udělat jinak, než pomocí Javascriptu v běžném nezabezpečeném režimu. Klíče bychom tedy museli nahrát do paměti a až poté z nich vytvořit instance objektu *CryptoKey*. Se stejným problémem bychom se potýkali i v případě, že by byly šifrovací klíče zadávány přímo při operaci.

Jako nejlepší řešení se zdá být rychlý autentizační proces. Provést rizikovou operaci jednou a rychle. Nebudeme proto uchovávat kryptografické klíče, ale budeme uchovávat přímo důvěrná data v otevřené podobě. Pokud bychom v databázi měli kryptografické klíče

a útočník by se k nim dostal, nebyl by pro něj problém důvěrná data dešifrovat (a to ani později). Tato varianta má navíc jednu hlavní výhodu a tou je efektivita. Proces dešifrování je náročnější operace a my chceme, aby to uživatel co nejméně pociťoval. Pokud provedeme stažení dat během autentizace, dešifrujeme je a uložíme v otevřené podobě, pak nás při jejich zobrazování nemusí nic brzdit a uživatel tak obsah uvidí téměř okamžitě. Ilustrace procesu autentizace je na obrázku 4.1.



Obrázek 3.6: Proces autentizace uživatele

Popis autentizačního procesu

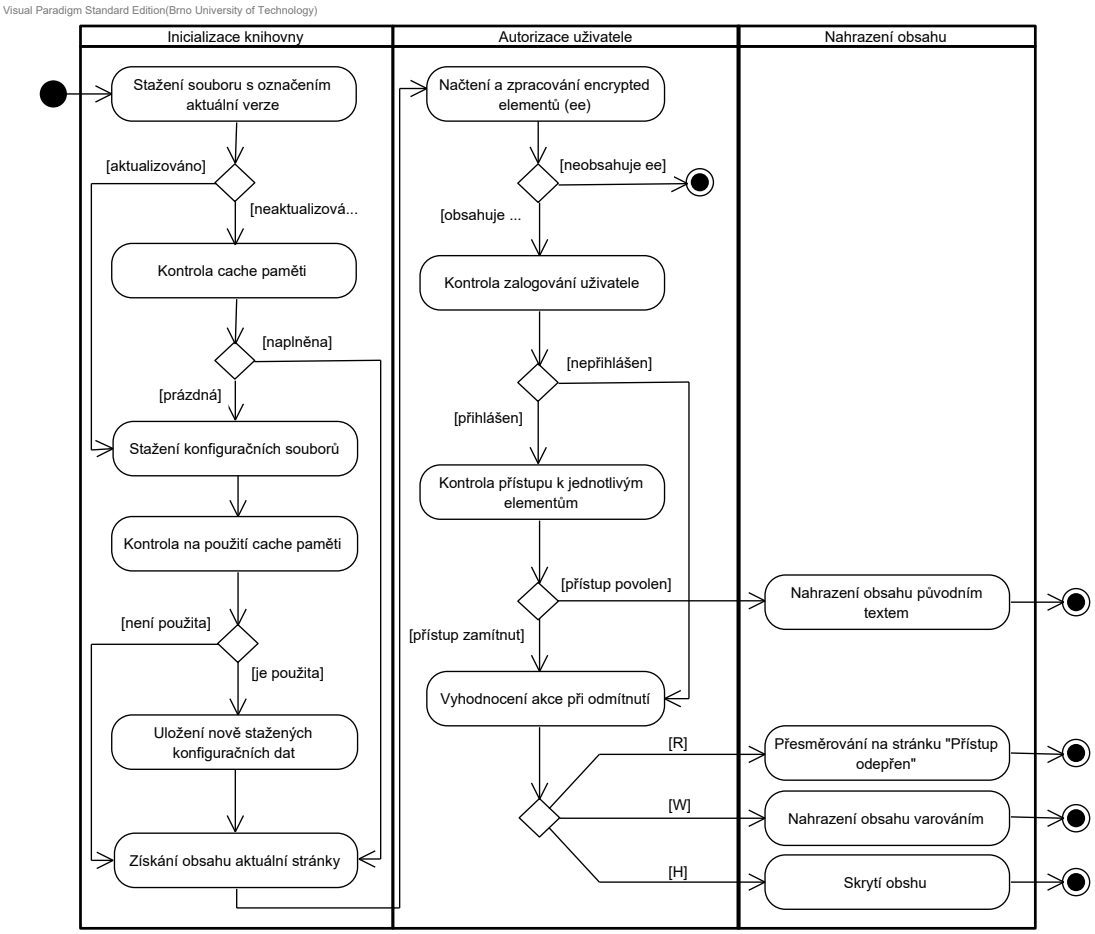
Předtím než popíšeme samotný autentizační proces je nutné popsat, jak k autentizaci vlastně dojde. Knihovna negeneruje žádné přihlašovací formuláře, ale pouze poskytuje metody pro přihlášení a odhlášení uživatele. Je tak docíleno toho, že vývojář bude mít plnou kontrolu nad tvorbou stránky a nebude omezený žádnou šablonou. V případě, že bude chtít uživatele autentizovat, tak použije metodu `Login` nebo `LoginCert`.

Při načtení stránky se nejdříve inicializují potřebná data a vyčká se na zavolání metody pro přihlášení. První je provedena kontrola, zda již není nějaký uživatel v systému přihlášen, pokud ano systém provede jeho automatické odhlášení. Poté následuje derivace klíče od zadaného uživatelského jména. V dalším kroku je proveden pokus o dešifrování uživatelských dat. Pokud selže, uživateli se nepodařilo úspěšně autentizovat a proces je ukončen. K selhání může dojít proto, že uživatel zadal špatné heslo nebo byl šifrovaný text pozměněn. Po úspěšném získání důvěrných uživatelských dat následuje proces získání zbylých šifrovacích klíčů. Poté stáhneme zdroje, ke kterým má uživatel přístup. Ty rozšifrujeme mocí klíčů z ACL a v otevřené podobě je uložíme do databáze. Na závěr nastavíme do perzistentního úložiště uživatelské jméno a role, které mu byly přiděleny. Tím proces autentizace končí,

funkce vrací úspěch a vývojář na to může reagovat například přesměrováním na úvodní stránku.

3.4.2 Autorizační proces

Autorizační proces je spuštěn vždy, když uživatel vstoupí na stránku, která obsahuje část s důvěrným obsahem. Autorizačnímu procesu přechází ale i inicializace knihovny, která je vzhledem k povaze statického webu provedena vždy při načtení stránky. Na obrázku 3.7 můžeme vidět jak inicializaci knihovny, tak autorizační proces.

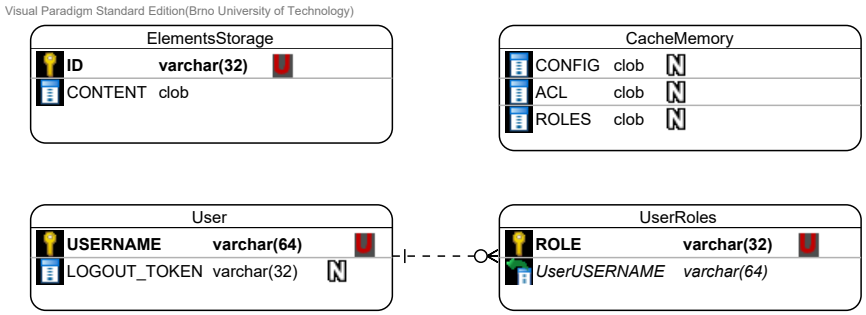


Obrázek 3.7: Proces autorizace uživatele

Proces autorizace je trochu složitější než ten autentizační. Vynechejme fázi incializace a přejděme rovnou k autorizaci. Nejprve je provedeno načtení všech *encrypted elementů*, které se na stránce vyskytují. Pokud není žádný nazelen proces skončí úspěšně. V opačném případě se kontroluje, zda je uživatel přihlášen, pokud není přecházíme k vyhodnocení akce. Pokud je přihlášen, zkontrolujeme jeho oprávnění pro dané elementy. Každý element je vyhodnocen zvlášť, uživatel totiž může mít přístup k jednomu elementu a k druhému nikoli. V případě, že alespoň jeden z elementů, ke kterým nemá přístup má příznak ODA jako R (redirect), je vždy přesměrován. Po vyhodnocení všech elementů se tedy provede jejich nahrazení za původní obsah, výpis varování nebo se element skryje.

3.4.3 Návrh databáze

Pro náš systém není potřeba žádná složitá databáze. Data, která potřebujeme ukládat jsou pouze tato: informace o přihlášeném uživateli, konfigurační soubory (*cache paměť*) a úložiště pro data důvěrných částí webových stránek. Každý blok na stránce je označen unikátním identifikátorem, který lze použít i jako identifikátor v databázi. Stačí nám ukládat jen dvojice klíč-hodnota. Návrh databáze je na obrázku 3.8.



Obrázek 3.8: Návrh databázového schématu

Kapitola 4

Implementace

Cílem této kapitoly je popsat klíčové části implementace obou navrhovaných řešení. V první části kapitoly je popsána aplikace pro šifrování webových stránek a v druhé knihovna pro jejich dešifrování. Implementační jazyk knihovny, kterým je Javascript, byl předem znám, neboť knihovnu vytváříme pro webové aplikace. Pro implementaci aplikace byl zvolen jazyk Python a to ze dvou důvodů. Prvním důvodem je to, že máme různé druhy generátorů statických webových stránek jak pro Linux, tak pro Windows. Aby bylo výsledné řešení univerzální, musíme volit multiplatformní jazyk, kterým Python je. Druhým důvodem bylo, že Python obsahuje řadu kvalitních kryptografických knihoven, které podporují současné kryptografické algoritmy.

Vzhledem k tomu, že využíváme dva různé implementační jazyky a ve výsledku máme dva různé programy, hrají konfigurační soubory velkou roli. Každý proces pracuje odlišným způsobem a konfigurační soubory jsou tak řešením, jak oba procesy *synchronizovat*. Jasným příkladem je definice algoritmů. Musíme totiž najít průnik mezi podporovanými algoritmy v Pythonu a Javascriptu, tzn. Web Cryptography API. Co se definice kryptografických algoritmů týče, tak nám API moc na výběr nedává. Vzhledem k tomu, že vyžadujeme vyšší míru bezpečnosti, tak ze symetrických algoritmů můžeme použít jen AES-GCM, protože ostatní neobsahují autentizační kódy a my bychom je tak museli sami implementovat, což nelze, protože by to znamenalo degradaci míry bezpečnosti. Knihovna *Cryptography* naštěstí šifru AES v módu GCM podporuje. Ze strany Web Cryptography API, v asymetrických variantách na výběr nemáme, k dispozici je pouze RSA-OAEP (ostatní algoritmy jsou učeny k podpisu).

V rámci implementace aplikace pro šifrování webových stránek vznikl také program pro snadné generování klíčů v požadovaném formátu. Výstupem programu jsou dva klíče, veřejný a soukromý, oba ve formátu PEM.

4.1 Aplikace pro šifrování webových stránek

Jak již bylo řečeno, implementačním jazykem aplikace pro postprocessing webových stránek je Python s využitím knihovny *Cryptography*. Ostatní použité knihovny jsou standardní součástí Pythonu a není je potřeba explicitně instalovat.

Třída `Crypter`, která má prakticky na starosti celý proces, má tři veřejné metody. Nejprve musí být zavolána metoda `Initialize`, která provede inicializaci. Následně metoda `Analyze`, která projde zdrojový adresář a vyhledá všechny důležité konfigurační soubory a webové stránky. Na závěr je možné zavolat metodu `Process`, která začne se zpracováním.

Obrázek 4.1: Aplikace PyCrypter - třída Crypter

Obecně byl způsob jakým aplikace pracuje popsán v 3.3.1 a dále si proto detailně rozbereme jen podstatné části tohoto procesu.

4.1.1 Příprava pro zpracování adresáře s webovou prezentací

Aplikace vyžaduje dva argumenty, pro určení obsahu ke zpracování a určení místa pro uložení výsledku, zdrojový a cílový adresář. Je to proto, abychom nepřepisovali původní soubory, protože pokud bychom si zašifrovali zdrojové kódy, tak by pak jejich obnova byla komplikovaná. Nejdříve je připraven cílový adresář, kterému kompletně smažeme celý obsah. Navíc, abychom zamezili nehodám v podobě publikace nešifrovaných dat, je obsah zdrojového adresáře nejprve nahrán do dočasného adresáře. Ten je pak v průběhu procesu postupně přepisován a po dokončení je jeho obsah nahrán do cílového adresáře.

4.1.2 Vyhledávání důvěrných částí a jejich nahrazování

Pokud se po úspěšném ověření kompatibility algoritmů a závislotí rolí dostaneme k vyhledávání obsahu, načteme nejdříve danou webovou stránku ze souboru. Pro vyhledávání pak používáme regulární výrazy, pomocí kterých hledáme text uvozený danými značkami, jako na příkladu 4.3. Na příkladu také můžeme vidět to, jak získáme seznam oprávěných osob a hodnotu *on denied action*.

Příklad 4.1 Regulární výraz pro získání důvěrného obsahu stránky

```
# Vyhledani duvernych casti
re.findall(r'<!--EE:BEGIN-->(.*?)<!--EE:END-->',web_page,re.DOTALL)

# Ziskani seznamu opravnenych osob
re.findall(r'<!--PERM:(.*?)-->', ee)

# Ziskani hodnoty ODA
re.findall(r'<!--ODA:(.*?)-->', ee)
```

Získaný obsah, bez uvedených komentářů, je pak zašifrován a uložen do souboru. Každý soubor nese jméno, které je identické s identifikátorem daného bloku. Ten se získá spojením URI souboru a indexu elementu na stránce. Výsledný řetězec se pak pomocí SHA-256 hašuje. Klíč, identifikátor a seznam oprávněných rolí se pak předají ACL v podobě nově vytvořeného zdroje (instance třídy `AclResouce`). Poté je část stránky nahrazena nově vygenerovaným *encrypted elementem*. Na závěr se na stránku přidá tzv. *Boot script* (viz 4.1.4), stránka se uloží a pokračuje se zpracováním další, dokud nejsou zpracovány všechny vyhledané stránky.

4.1.3 Zpracování rolí a uživatelů

Po zpracování zdrojů se nejdříve budeme zabývat rolemi, protože jejich klíče pak musíme distribuovat jednotlivým uživatelům. Pokud úspěšně proběhne zašifrování zdrojů v ACL, jsou potřebná data předána rolím. Konkrétně se jedná o popis použitého algoritmu a šifrovací klíč. Každá role pak obsahuje vlastnost `secret`, což je pole zdrojů, kde každá položka obsahuje ID, přílušný popis algoritmu a klíč. Jak je `secret` strukturován můžeme vidět na obrázku 4.2.

Příklad 4.2 Příklad uložení kryptografických klíčů ACL zdrojů u každé role

```
secret = [
  {
    "resource_id": "FFAAFF...",
    "secret":
      {
        algorithm: {
          name: "AES-GCM",
          iv: "FFAAFF...",
          tag: 128
        },
        key: "FFAAFFAA..."
      }
  },
  ...
]
```

Předtím, než se role zašifruje, je vypočítána dědičnost. O výpočet se stará metoda `compute_heredity` třídy `UserList`. Ta provede analýzu závislostí a dopočte rozdíly zdrojů daných rolí. Následně pak distribuuje klíče těm rolím, které by měly mít přístup k danému zdroji v rámci dědičnosti, ale klíč k němu ještě nevlastní.

Jakmile je role naplněna potřebnými zdroji a je vyřešena dědičnost, `secret` je zašifrován a klíč je distribuován daným uživatelům, kterým byla tato role přiřazena. U uživatelů se postupuje podobným způsobem. Také mají vlastnost `secret` a ta obsahuje popisy algoritmů a klíče k daným rolím. Namísto `resource_id`, jak tomu bylo u příkladu 4.2, je uvedeno `role`, což značí jméno role.

V případě, že je dokončen proces šifrování rolí, začne se s šifrováním uživatelů. Zde máme dvě možnosti. Uživatel je buď zašifrován klíčem derivovaným z jeho hesla nebo dostupným veřejným klíčem.

4.1.4 Boot script

Boot script je jednou z nejdůležitějších částí implementace. Slouží k vložení kódu knihovny na každou zpracovávanou stránku (i na tu, kde se nenachází důvěrný obsah). Boot script je pouhý HTML soubor, šablona, která je vložena do stránky před ukončovací tag `</body>`. Pomocí něho jsme schopni vložit do každé stránky kód pro načtení knihovny a následně i inicializační sekvenci.

Boot script nemusí sloužit jen k inicializaci, může obsahovat i kód pro obsluhu tlačítek pro přihlášení, odhlášení nebo třeba zajišťovat manipulaci s uživatelským rozhraním na základě získaných dat o uživateli (změna tlačítka `login` na `logout` po přihlášení).

Příklad 4.3 Ukázka inicializace a spuštění knihovny

```
<script src="../js/j3a.js"></script>
<script type="text/javascript">
    var configUrl = "../config.json";
    j3a = new J3A(true);
    j3a.Init(configUrl).then(function () {
        jQuery(document).ready(function () {
            j3a.RunPostProcessing().then(function (result) {
                // Everything is done! Do something else here!
            });
        });
    }).catch(function (error) {
        console.log(error);
    });
</script>
```

4.1.5 Verzování generovaných webových prezentací

S každou zpracovanou stránkou je vytvořen nový soubor `version.json`, který obsahuje identifikátor aktuální verze. Při každé inicializaci knihovny se tento soubor stáhne a jeho obsah se porovná s tím, co je uloženo v lokální databázi. Je to z toho důvodu, že systém využívá dočasnou pamět a při vygenerování nové verze stránky je potřeba uživatele odhlásit a donutit ho tak načíst nová data.

Generování nové verze stránky má také za následek změnu všech šifrovacích klíčů, které jsou generovány vždy při každém procesu šifrování. Výjimkou je klíč derivovaný z uživatelského hesla, popř. veřejný klíč.

4.2 Knihovna pro dešifrování webových stránek

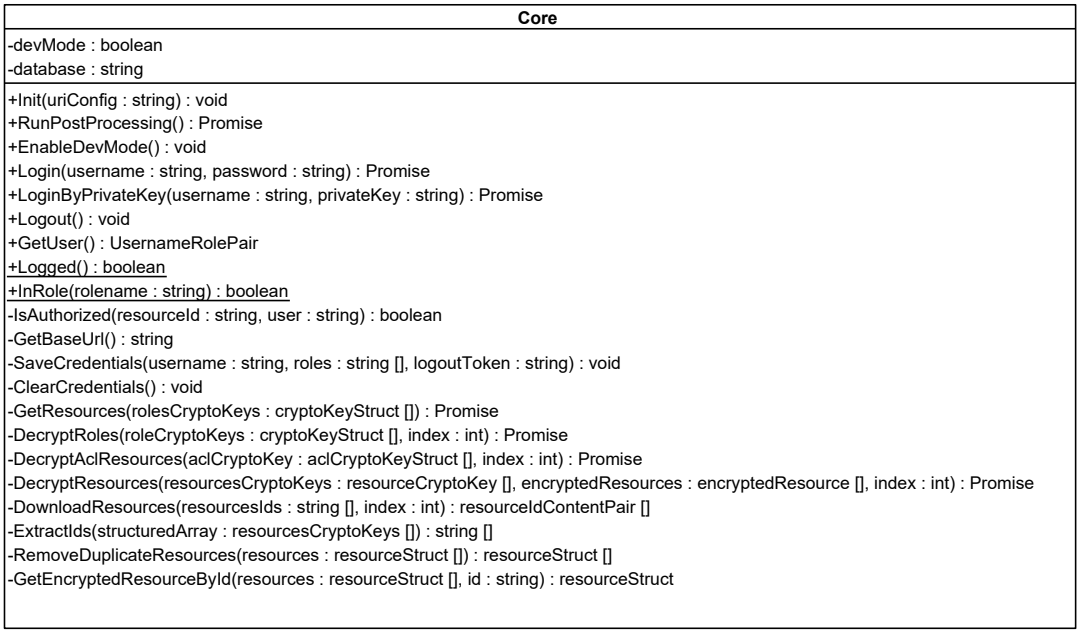
Vzhledem k požadavkům na výsledné řešení, bylo k implementaci knihovny využito několik různých technologií. Základem je Browserify, technologie která nám umožňuje skládat výsledný Javascriptový soubor z menších částí, takže nám dovoluje vytvářet samostatné soubory pro navrhované třídy. Browserify prakticky přidává možnost importu modulů či jiných souborů. O kompatibilitu se stará Babelify, který výsledný kód generovaný pomocí Browserify převede na starší verzi kompatibilní i se staršími prohlížeči. UglifyJS se na závěr postará o vygenerování minifikované verze knihovny.

Použití uvedených technologí přináší také jistá omezení. Vzhledem k tomu, že Babelify podporuje nanejvýš ECMAScript 2015 (verze 6), nelze použít některé nové konstrukce pro práci s asynchronními funkcemi. Některé části implementace jsou tak komplikovanější a méně přehledné.

4.2.1 Autentizace a autorizace uživatelů

Knihovna celý proces autentizace a autorizace nedělá plně automaticky, naopak, poskytuje vývojáři metody, které může použít tak, jak se mu to bude hodit. Není tedy ničím omezen. Autentizaci zajišťují metody `Login` a `LoginCert` a autorizace probíhá v rámci volání metody `RunPostProcessing`. K dispozici jsou také dvě statické metody `Logged` a `InRole`, které slouží k získání informace o přihlášeném uživateli, a které mohou být použity bez vytvoření instance třídy, což umožní rychlejší odezvu pro určité operace (např. práce UI).

Vzhledem k tomu, že většina metod používá asynchronní funkce, je jejich návratovou hodnotou `Promise`.



Obrázek 4.2: Knihovna J3A - třída Core

4.2.2 Perzistentní ukládání dat

Na straně klienta (klient = webový prohlížeč) potřebujeme ukládat informace o uživateli, obsah důvěrných částí stránek a také dočasná data pro zrychlení procesu. K tomu bylo využito lokálního úložiště *LocalStorage* a v dřívejší verzi implementace i *IndexedDB*.

LocalStorage slouží k uložení dat o uživateli (uživatelské jméno a role). Je to proto, že přístup k tomuto úložišti je rychlý a synchronní a nemusíme tak čekat až komplikovanější operace doběhne, jak je tomu např. u *IndexedDB*. *LocalStorage* slouží také pro uložení dočasných dat, takže jako *Cache paměť*. Ukládáme do něj data konfiguračního souboru, ACL a rolí proto, abychom je nemuseli při každém načtení stránky znova stahovat. *LocalStorage* nepodporuje ukládání složitějších typů, což nám ale nevadí, protože stahované soubory jsou ve formátu JSON, který můžeme snadno serializovat pro textové podoby.

V dřívějších verzích implementace byla pro uložení obsahu stránek použita *IndexedDB*, která na lokálním stroji (*localhost*) fungovala dobře, ale na produkčním serveru docházelo k nahodilým chybám. V některých případech se stávalo, že se databázi nepodařilo otevřít a musel se tak pokus o otevření opakovat. Někdy se ji dokonce nepodařilo otevřít vůbec. Celý proces zpracování stránky tak byl v lepším případě značně zpomalen. Proto se začalo pro ukládání dat používat *LocalStorage* a vzhledem k tomu, že potřebujeme ukládat pouze data typu klíč-hodnota, nebylo při změně nutné provést větší zásahy. Načtení z *LocalStorage* je navíc rychlejší, takže se celý proces nahrazení obsahu nepatrně zrychlil.

S lokálním uložištěm také úzce souvisí i systém *prefixů*. Vzhledem k SOP[1] není možné provozovat dva weby na jedné doméně, proto byl implementován systém prefixů, který právě tohle umožňuje. Prakticky se jedná o to, že před každý klíč je vložen prefix specifický pro daný web. Musíme mít ale na paměti, že se v některých případech jedná o bezpečnostní riziko.

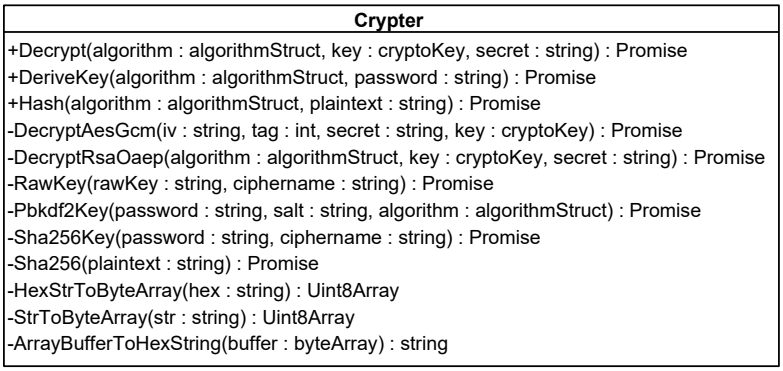
4.2.3 Implementace metod pro práci s API

S Web Cryptography API pracuje třída `Crypter`, která je k vidění na obrázku 4.3. K dispozici jsou obecné metody `Decrypt`, `DeriveKey` a `Hash`. Další metody třídy už implementují volání konkrétních metod API a musí tak mít na vstupu validní formáty požadovaných struktur. O to se starají uvedené metody, které požadované struktury vytvoří tak, aby odpovídaly požadavkům. V případě, že vstupní data neodpovídají požadavkům, je vrácena chyba.

Na obrázku 4.3 můžeme také vidět, že není implementována metoda pro derivaci klíče pro asymetrickou kryptografii. Je to z toho důvodu, že nám API dovoluje importovat klíč přímo, pokud je v zadaném formátu, konkrétně PKCS#8, což je formát soukromého klíče.

Vstupem pro data metod API jsou pole typu `Uint8Array` a klíče, které máme k dispozici jsou ve formátu `String`. Konkrétně bychom *klíčový* textový řetězec mohli označit jako `Hex String`. Metody, které převádějí tyto řetězce na pole a zpět jsou `HexStrToByteArray` a `ArrayBufferToHexString`.

[1] Same Origin Policy



Obrázek 4.3: Knihovna J3A - třída Crypter

Kapitola 5

Integrace

Kapitola se věnuje popisu vybraného generátoru statických webových stránek, ve kterém popisuje principy jeho fungování a práci s šablonami, konkrétně jejich tvorbu. Dále rozebírá způsoby integrace námi navrženého a implementovaného systému do generátoru Hugo.

5.1 Generátory statických webových stránek

V poslední době se generátory statických webových stránek těší čím dál větší popularitě. Není vždy potřeba instalovat robustní CMS a v mnohých případech si lze vystačit pouze se statickou verzí webové prezentace, což nám přináší i několik výhod [26]:

Výkon Vzhledem ke statické povaze webu, není na serveru potřeba dalšího zpracování, jako například generování obsahu nebo připojování k databázi a získání stránky tak není ničím zpomalováno.

Hosting Vzhledem k tomu, že zde nejsou prakticky žádné požadavky na webový server (např. interpret, databáze, aj.), nevyžaduje hosting složité nastavení a údržbu. Je proto možné provozovat webové stránky jednoduše a levně. K dispozici máme spoustu hostingů, které umožňují hostovat stránky zdarma, jako GitHub nebo Surge.

Bezpečnost Jestliže používáme pouze statický web, tak není na straně serveru nic, co by se dalo zneužít. Pokud jsou soubory na hostitelském serveru zabezpečeny, tak nemůže dojít k ohrožení bezpečnosti stránky.

Verzování obsahu Celý web je postaven pouze ze souborů uložených v příslušných adresářích. Není tak zapotřebí žádná databáze a to přináší jisté výhody, celý obsah můžeme verzovat pomocí systému Git.

Statické webové stránky mají jistě spoustu výhod, ale oproti dynamickému webu mají i nevýhody. Nemůžeme např. editovat nebo přidávat obsah přímo na stránce (např. psaní komentářů pod článkem) v reálném čase nebo upravovat obsah stránky v závisloti na získaných datech o přihlášeném uživateli.

V současné době máme k dispozici mnoho generátorů statických webových stránek, např. Hugo nebo Jekyll a mnohé jsou i přímo integrovány do některých systémů (GitHub - generátor Jekyll).

5.2 Generátor Hugo

Hugo je jako jeden z mála generátorů psaný v jazyce Go[1]. Generátor je vytvořen tak, aby umožňoval rychlou a jednoduchou tvorbu webové prezentace a nevyžaduje instalaci žádných dalších knihoven. [26]

Nespornou výhodou je také to, že Hugo umožňuje spustit vlastní webový server (na portu 1313). To nám velice zjednoduší práci, protože Hugo si hlídá, který soubor byl upraven a při změně automaticky přegeneruje výslednou webovou prezentaci.

Máme téměr nespočet možností nastavení generátoru a jeho výstupů, a proto se v následujícím textu zaměříme pouze na popis základní funkčnosti a struktur.

5.2.1 Generování stránek

Stránky můžeme generovat dvojím způsobem. V prvním případě můžeme využít přímo konzoli. Pomocí specifického příkazu nejprve celou prezentaci vygenerujeme a následně můžeme pomocí jiného příkazu přidat novou stránku. Celý web tak lze napsat jen pomocí konzolové aplikace.

V druhém případě si pomocí příkazu web jen vygenerujeme a poté si jednotlivé stránky manuálně vytvoříme v dané adresářové struktuře. Veškerý obsah (textový) je umístěn ve složce `content` v souborech typu `md` (např. `site.md`). Struktura adresáře `content` nám pak definuje i URL. Pokud bychom měli adresář `pages` a v něm soubor `page.md`, výsledná URL by byla `example.com/pages/page/`. Ukázku definice stránky můžeme vidět na příkladu 5.1.

Příklad 5.1 Ukázka stránky about.md

```
1  +++
2  date = "2017-04-19T21:47:52+02:00"
3  title = "About me"
4  layout = "single"
5  +++
6
7  Some heading
8  ------------
9  Lorem ipsum dolor sit amet.
```

Text oddělený pomocí `+++` označuje hlavičku obsahující metadata. Metadat může být celá řada, ale pro nás je podstatný parametr `layout`, kterým můžeme specifikovat, jaká šablona bude pro stránku použita. Více informací o nastavení lze dohledat přímo na webových stránkách generátoru.

5.2.2 Šablony

Generátor využívá systém šablon, které jsou při tvorbě webové prezentace snadno aplikovatelné. Šablonu lze definovat při vytváření webu pomocí konzole nebo posléze přímo v konfiguračním souboru. Příklad nastavení šablony v konfigurační souboru `config.toml` můžeme vidět na příkladu 5.2. Veškeré dostupné šablony musí být uloženy ve složce `themes`.

[1] Go je jazyk původně vyvinutý společností Google.

Příklad 5.2 Nastavení šablony v konfiguračním souboru config.toml

```
1  ...
2  languageCode = "en-us"
3  title = "Sampe web site"
4  theme = "lunafreya"
5  ...
```

Každá šablona má přesně definovanou strukturu, kde jsou nejdůležitější adresáře `static` a `layouts`. Adresář `static` obsahuje veškeré soubory a adresáře, které budou v kořenovém adresáři výsledné webové prezentace. Obsahuje proto zdrojové soubory definic stylů, knihovny nebo obrázky. Adresář `layouts` slouží k uložení šablon stránek, které je pak možné používat. Máme zde dva podadresáře `partials` a `_defaults`. V `partials` jsou části stránek jako *header* nebo *footer*, které jsou pak vloženy do konkrétní šablony. Adresář `_defaults` obsahuje jednotlivé šablony, které mohou být použity při definici stylu dané stránky. Ukázka šablony je v příkladu 5.3.

Příklad 5.3 Šablona stránky

```
1  {{ partial "header.html" . }}
2  <main>
3    <article>
4          <h2>{{ .Title }}</h2>
5          <section>
6              {{ .Content }}
7          </section>
8      </article>
9  </main>
10 {{ partial "footer.html" . }}
```

5.3 Integrace knihovny do generátoru

Systém lze využít bez jakéhokoli většího zásahu, protože je možné vkládat potřebné komentáře, resp. uvozovací značky, přímo do souboru `md`. Pak stačí jen do výsledné prezentace nakopírovat složku s konfiguračními soubory a spustit postprocessing. Tento způsob je ale pro uživatele komplikovanější.

Systém proto můžeme integrovat do šablony přímo. Ukázku takové integrace můžeme vidět na příkladu 5.4. Vývojář, který vytváří šablonu pro generátor, tak může do adresáře `static` umístit všechny potřebné konfigurační soubory a knihovny a definovat role. Pak stačí jen vytvořit šablonu, která umožní přístup daným rolím. Uživatel při tvorbě své prezentace jen použije šablonu definující určitý přístup a o víc se starat nemusí. Na závěr je pouze třeba definovat uživatele a přidělit jim role. Pokud se jedná o zkušenějšího uživatele, neměl by pro něj být problém šablonu trochu upravit a dodat vlastní role nebo definovat přístupová oprávnění.

Příklad 5.4 Šablona privátní stránky

```
1  {{ partial "header.html" . }}
2
```

```
3  {{ "<!--EE:BEGIN-->" | safeHTML }}
4  {{ "<!--PERM:Registered-->" | safeHTML }}
5  {{ "<!--ODA:R-->" | safeHTML }}
6
7  <main>
8  ...
9  </main>
10
11 {{ "<!--EE:END-->" | safeHTML }}
12
13 {{ partial "footer.html" . }}
```

Kapitola 6

Testování

V této kapitole se zaměříme na testování výsledného řešení. V první části popíšeme základní testování výsledného systému, kde se zaměříme na testování aplikace v nejpoužívanějších prohlížečích, testování přístupu k důvěrnému obsahu a testování metod přihlašování. Provedeme také jednoduchý penetrační test systému. Konkrétně vytvoříme stránku se zranitelností na XSS a provedeme simulaci útoku. V druhé části se zaměříme na testování výkonu aplikace a měření velikosti výstupu.

6.1 Komplexní testování systému

Testování systému probíhalo manuálně se zaměřením na testování výsledných generovaných prezentací. Nejprve byla otestována kompatibilita knihovny s aktuálními verzemi prohlížečů Chrome[1], Firefox[2], Opera[3], Edge[4] a IE 11[5]. Prohlížeče Chrome, Firefox a Opera prošly testem bez jakéhokoli prolbému. Naopak IE 11 nepodporuje *Promises* a knihovna je tím pádem nefunkční. Prohlížeč Edge v sobě skrývá nestandardní chování u operace *decrypt* Web Cryptography API. Z neznámého důvodu není možné dešifrovat AES-GCM s inicializačním vektorem o 128 bitech (při 96 bitech není problém pozorován). Z uvedeného důvodu se tak není možné přihlašovat. V rámci testování byly provedeny také testy na mobilních zařízeních. Byly úspěšně otestovány aktuální verze prohlížečů Chrome a Firefox pro operační systém Android[6].

V další fázi jsme se zaměřili na testování řízení přístupu. Bylo vytvořeno několik rolí s různými přístupovými právy, přičemž některé dědily přístupová práva od jiných. Test byl úspěšný pro všechny testované uživatele. Systém přesměrování a nahrazování obsahu také obstál. V rámci této fáze bylo také otestováno jak přihlašování pomocí hesla, tak pomocí certifikátu.

V předposlední fázi jsme otestovali implementaci prefixů. V rámci jedné domény byly publikovány dvě prezentace s vlastními prefixy. Webové stránky se při testování nijak neovlivňovaly a to ani při dvou zaráz přihlášených uživatelích.

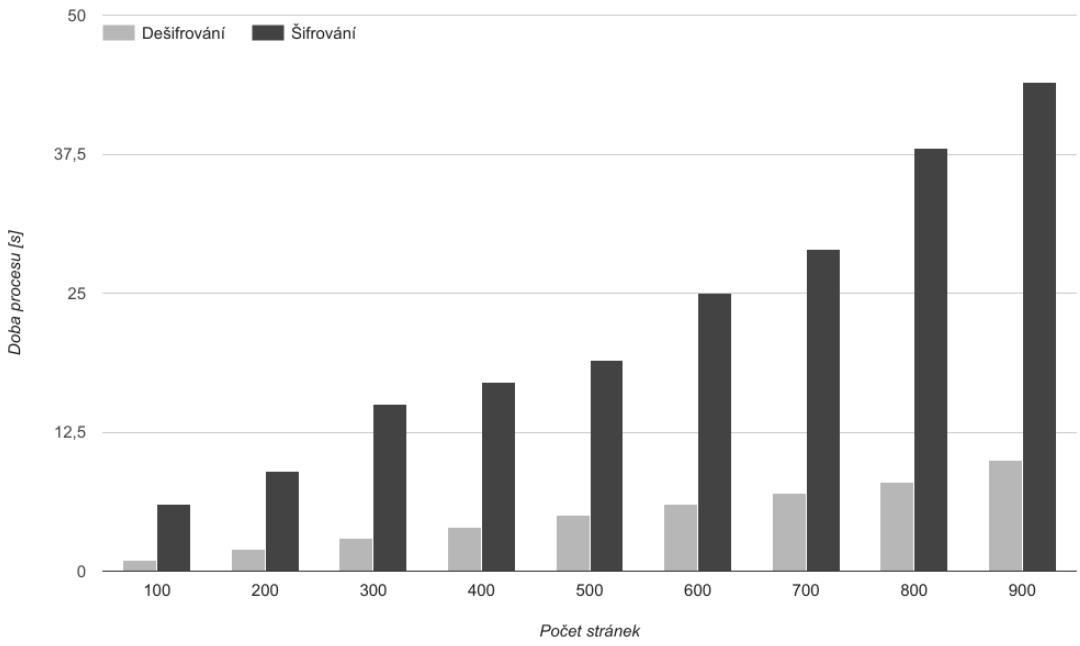
[1] Google Chrome, verze 58.0.3029.110 (64 bitů)
[2] Mozilla Firefox, verze 53.0.3 (32 bitů)
[3] Opera, verze 45.0.2552.812
[4] Microsoft Edge, verze 38.14393.1066.0
[5] Internet Explorer, verze 11.1198.14393.0
[6] Android 4.2.2; Android 5.1.1

V poslední fázi byl proveden test na zranitelnost útokem XSS. Pro účely testu byla vytvořena jednoduchá stránka, která tuto zranitelnost obsahovala. Následným simulovaným XSS útokem se podařilo získat důvěrná data přihlášeného uživatele.

6.2 Testování výkonu

Byly provedeny dva druhy testů. Prvním testem jsme měřili rychlost šifrování a dešifrování webové prezentace a druhým velikost prezentace před a po postrocessingu. Pro testování byly generovány prezentace s různým počtem stránek (100 - 1000), kdy se na každé stránce nacházel důvěrný obsah o velikosti přibližně 4096 bajtů.

Výsledek prvního testu můžeme vidět na obrázku 6.1. Postupně byla měřena rychlost procesu šifrování na lokálním stroji[7] od 100 až po 900 stránek. Nárůst doby šifrování je dle výsledného grafu lineární. Můžeme si také povšimnout nízkých hodnot při dešifrování, což je dáno právě zpracováním dat na lokálním stroji (zdroje nejsou stahovány ze serveru). Nicméně lze podotknout, že děšifrování devítiset stránek zabere pouhých 10 sekund, přičemž se musí operace dešifrování nejméně devětsetkrát zopakovat.

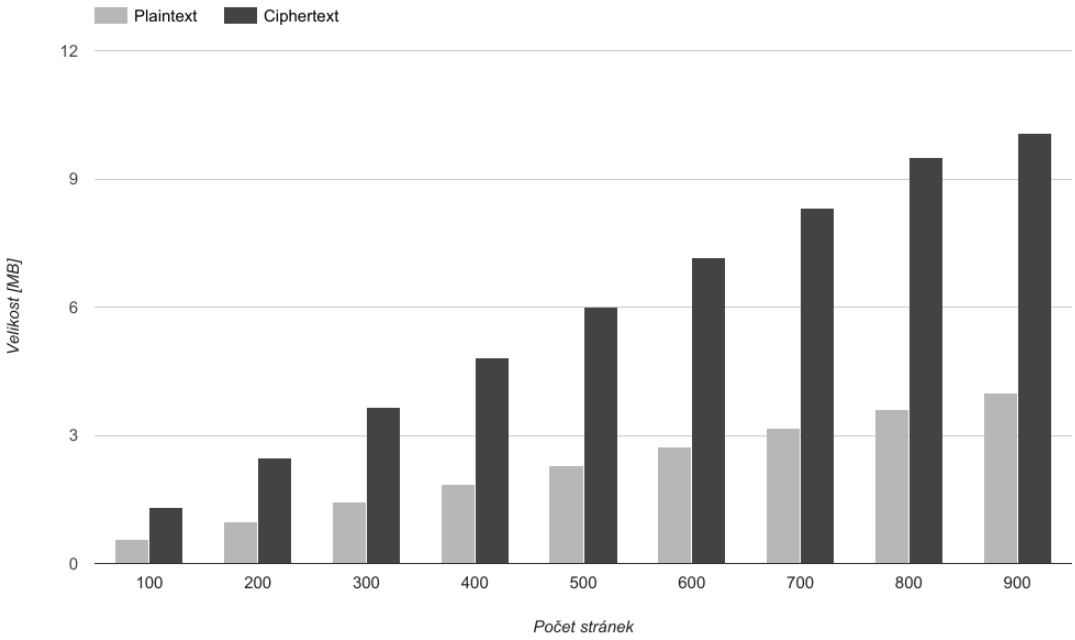


Obrázek 6.1: Doba procesu šifrování/dešifrování v závislosti na počtu stránek

Výsledek druhého testu je na obrázku 6.2. Rozsah testování byl stejný jako u předešlého testu. Zde jsme se zaměřili na měření vstupní a výstupní velikosti dat. Z grafu můžeme vidět, že nárůst velikosti webové prezentace po dokončení postprocessingu je více jak dvojnásobný. Je to dáno vlastnostmi blokové šifry AES-GCM, která byla při šifrování použita.

Pro testování bylo zašifrováno až 1 000 stránek. Rozšifrovat se je ale nepodařilo. Proces dešifrování skončil chybou, což bylo způsobeno nedostatkem místa určeného pro lokální

[7] Acer Aspire V3-772G; Procesor: Intel Core i7-4702MQ 2.2 GHz; RAM: 16 GB DDR3



Obrázek 6.2: Velikost webové prezentace v závislosti na počtu stránek

úložiště. Pro testování byl zvolen prohlížeč Firefox[8], který má nastavenu mezní hodnotu velikosti lokálního úložiště na 5.12 MB. Testování tak skončilo u hodnoty 900 stránek.

Tento problém by se částečně vyřešil použitím *IndexedDB*, která dovoluje uložit větší množství dat. Znamenalo by to ale zpomalení systému, a také bychom museli vyřešit problém nestability, se kterým jsme se potýkali během implementace.

[8]Mozilla Firefox, verze 53.0.3 (32 bitů)

Kapitola 7

Závěr

Cílem práce bylo navrhnout a implementovat systém pro šifrování a dešifrování statických webových stránek za použití Web Cryptography API a integrovat ho do jednoho z generátorů a výsledky pak zveřejnit pod volnou licencí.

V první části práce byla rozebrána problematika týkající se kryptografie na straně prohlížeče. Byly diskutovány její možnosti a související problémy. Posléze jsme se zaměřili na popis nově vznikajícího standardu Web Cryptogprahy API.

Zbytek práce popisuje návrh celého systému a jeho součástí, implementaci, integraci a také testování. V rámci návrhu jsou popsány metody a konstrukce, které byly použity při implementaci. Kapitola implementace popisuje její klíčové části a také řešení složitějších problémů. V integraci se věnujeme zapracování výsledného systému do šablon generátoru Hugo. Z testování plyne, že ne na všech prohlížečích lze systém zprovoznit. To je ve většině případů způsobeno verzí prohlížeče, která nepodporuje použité moderní (experimentální) technologie.

Ve výsledku se nám podařilo navrhnout a implementovat systém, který umožňuje autorizovaný přístup k vybraným částem webové prezentace. Řešení také obsahuje řízení přístupu založené na rolích, které je možné přiřazovat registrovaným uživatelům. Celý systém byl navržen tak, že není problém jej v rámci šablony integrovat téměř do jakéhokoli generátoru statických webových stránek. Tím se značně rozšiřují jeho možnosti. Jedinou nevýhodou je nutnost zabezpečeného spojení, které je nezbytné pro funkčnost Web Cryptography API.

Celý projekt je dostupný pod Open Source licencí na serveru Github[1] spolu s demo[2] aplikací. V rámci práce vznikla také šablona *Lunafreya*, itegrující knihovnu J3A, která je určena pro generátor Hugo. Šablona je také vydána pod lincencí Open Source a je dostupná na GitHubu[3] spolu s ukázkou[4]. Šablona si klade za cíl zvýšit povědomí o možnostech knihovny J3A a umožnit tak její rozšíření.

7.1 Možná rozšíření

Je v plánu na vývoji nadále pracovat a rozvíjet tak možnosti prezentovaného řešení. Projekt je publikován na serveru GitHub, což umožňuje snadnou rozšiřitelnost případnými přispěvateli.

[1] https://github.com/PraserX/j3a
[2] https://praserx.github.io/j3a/demo/
[3] https://github.com/PraserX/hugo-theme-lunafreya
[4] https://praserx.github.io/hugo-theme-lunafreya/demo/

Možných rozšíření systému se nabízí celá řada a uvedeme si jich tedy jen pár. V rámci prvního rozšíření by se dala zavést podpodora šifrování a následného dešifrování stažitelných souborů, které jsou nyní na webu přístupné v otevřené podobě a může si je tak kdokoli stáhnout.

Druhým rozšířením by mohla být podpora registrace uživatelů. Vzhledem k povaze statického webu není možné nové uživatele snadno registrovat. Registrace nových uživatelů by mohla být provedena dvojím způsobem. V prvním případě by se mohlo jednat o službu běžící na vzdáleném serveru, na který by se pomocí AJAXu odeslal požadavek na registraci. Následně by šifrovací program získal data o uživateli ze serveru zpět, samozřejmě zabezpečenou formou. V druhém případě by registrace mohla fungovat na principu odesílání emailů na určitou adresu, což je méně náročná varianta z hlediska potřebných technologií.

Literatura

[1] *ASN.1 key structures in DER and PEM*. [Online; navšítveno 01.05.2017].
URL
https://tls.mbed.org/kb/cryptography/asn1-key-structures-in-der-and-pem

[2] *Clipperz Javascript Crypto Library*. [Online]. [cit. 28.12.2016].
URL https://clipperz.is/open_source/javascript_crypto_library/

[3] *CryptoJS*. [Online]. [cit. 28.12.2016].
URL https://code.google.com/archive/p/crypto-js/

[4] *Javascript Object Signing and Encryption (JOSE)*. [Online]. [cit. 1.5.2017].
URL http://jose.readthedocs.io/en/latest/

[5] *Same-origin policy*. MDN, [Online]. [cit. 2.5.2017].
URL
https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[6] *Same-origin policy*. W3C, [Online]. [cit. 2.5.2017].
URL https://www.w3.org/Security/wiki/Same_Origin_Policy

[7] *Secure Contexts*. W3C, [Online]. [cit. 1.5.2017].
URL https://www.w3.org/TR/secure-contexts/

[8] *Stanford Javascript Crypto Library*. [Online]. [cit. 28.12.2016].
URL http://bitwiseshiftleft.github.io/sjcl/

[9] Web Cryptography *Can I Use*. [Online]. [cit. 25.12.2016].
URL http://caniuse.com/#feat=cryptography

[10] *Web Crypto API*. MDN, [Online]. [cit. 25.12.2016].
URL https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API

[11] *Web Cryptography*. [Online]. [cit. 25.12.2016].
URL https://msdn.microsoft.com/en-us/library/dn302338(v=vs.85).aspx

[12] *Web Cryptography API*. W3C, [Online]. [cit. 25.12.2016].
URL https://www.w3.org/TR/WebCryptoAPI/

[13] *WebCrypto Key Discovery*. W3C, [Online]. [cit. 6.5.2017].
URL https://www.w3.org/TR/webcrypto-key-discovery/

[14] *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.2*. 2016, [Online]. [cit. 09.04.2017].
URL https://tools.ietf.org/html/rfc8017

[15] ALCORN, W.; FRICHOT, C.; ORRUU, M.: *The browser hacker's handbook*. John Wiley & Sons, Inc., Indianapolis, Indiana, 2014, ISBN 978-1-118-66209-0.

[16] BHARGAVAN, K.; DELIGNAT-LAVAUD, A.; MAFFEIS, S.: *Defensive JavaScript*. [Online; navštíveno 26.12.2016].
URL https://www.doc.ic.ac.uk/~maffeis/papers/fosad14.pdf

[17] CAIRNS, K.; HALPIN, H.; STEEL, G.: Security Analysis of the W3C Web Cryptography API. In *Security Standardisation Research*, New York: Springer, 2016, s. 112-140, ISBN 978-3-319-49099-1.

[18] HAAHR, M.: *Introduction to Randomness and Random Numbers*. [Online]. [cit. 28.12.2016].
URL https://www.random.org/randomness/

[19] HALPIN, H.: The W3C Web Cryptography API: Design and Issues. WS-REST, 2014.
URL
http://ws-rest.org/2014/sites/default/files/wsrest2014_submission_11.pdf

[20] HOFSTEDE, N.; VAN DEN BLEEKEN, N.: Using the W3C WebCrypto API for Document Signing. In *Workshop on Web Applications and Secure Hardware*, UK: London, 2013.
URL http://ceur-ws.org/Vol-1011/2.pdf

[21] KAYEM, A. V. D. M.; AKL, S. G.; MARTIN, P.: *Adaptive cryptographic access control*. New York: Springer, 2010, ISBN 978-1-4419-6655-1.

[22] KÜMMEL, R.: *XSS: Cross-Site Scripting v praxi: o reálných zranitelnostech ve virtuálním světě*. Zlín: Tigris, 2011, ISBN 978-80-86062-34-1.

[23] KRHOVJÁK, J.: *Cryptographic random and pseudorandom data generators*. Dizertační práce, Masarykova univerzita, Fakulta informatiky, Brno, 2009.

[24] MENEZES, A. J.; VAN OORSCHOT, P. C.; VANSTONE, S. A.: *Handbook of Applied Cryptography*. CRC Press, 1997, ISBN 978-0-8493-8523-0.

[25] PAAR, C.; PELZL, J.: *Understanding cryptography: a textbook for students and practitioners*. Berlin: Springer, 2010, ISBN 978-3-642-04100-6.

[26] RINALDI, B.: *Static site generators*. O'Reilly Media, Inc., 2015.
URL https://hugodocs.info/documents/oreilly-static-site-generators.pdf

[27] RUKHIN, A.; SOTO, J.; OTHERS: *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. [Online]. [cit. 2.1.2017].
URL
http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf

[28] STARK, E.; HAMBURG, M.; BONEH, M.: *Symmetric Cryptography in Javascript*. [Online; navštíveno 26.12.2016].
URL https://www.emilymstark.com/papers/acsac.pdf

[29] WHITMAN, M. E.; MATTORD, H. J.: *Principles of information security. 4th ed.* Boston: Course Technology, 2012, ISBN 978-1-111-13821-9.

Přílohy

Seznam příloh

Příloha A

Obsah CD

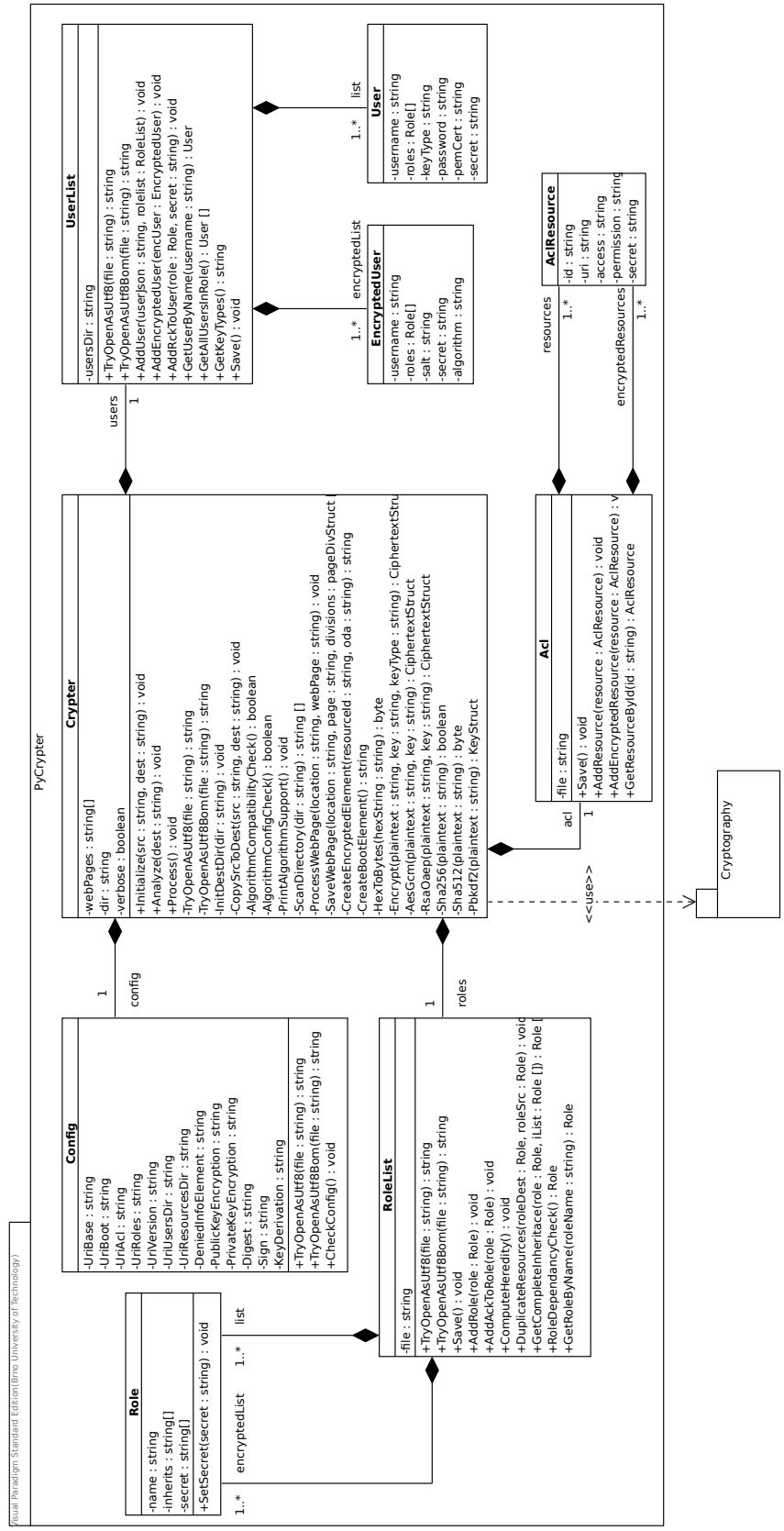
Součástí práce je přiložené CD, které obsahuje všechny dokumenty týkající se této práce. Celý text práce je uložen v hlavním adresáři pod názvem `thesis.pdf`, přiloženy jsou i zdrojové soubory textové části práce. Adresářová struktura přiloženého CD je následující:

- **thesis** zdrojové soubory textové části práce
- **source**
  - **j3a** zdrojové soubory dešifrovací knihovny (JS)
  - **j3a-crypter** zdrojové soubory šifrovacího programu (Python)
  - **j3a-key-generator** zdrojové soubory generátoru klíčů (Python)
- **examples**
  - **demo** ukázka webové prezentace
  - **demo-hugo** ukázka webové prezentace generovaná pomocí SSG Hugo
- **thesis.pdf**
- **LICENSE**

Příloha B

Výsledné modely

Obrázek B.1: Výsledný model knihovny pro dešifrování webových stránek



Obrázek B.2: Výsledný model aplikace pro šifrování webových stránek