6. Oktober 2014 Studie · Web & Cloud · Privatsphäre & Krypto · 10 Min. Lesezeit

Erstellung und Dekodierung von CMS-Containern in Javascript

Archiviert. Dieser Inhalt wird nicht mehr aktualisiert. Er bleibt zur Information verfügbar.

Quelle https://technology2.a-sit.at/veroeffentlichungen/cms-javascript/

Das Projekt demonstriert die Umsetzung von Cryptographic Message Syntax über Javascript im Browser.

Web-Applikationen ersetzen zunehmend traditionelle Desktop-Applikationen. Dahingehend findet oftmals auch eine Verarbeitung sensibler Daten im Browser statt. Applikationen, die sicherheitskritische Daten verarbeiten, fehlen jedoch wichtige kryptographische Komponenten. Basiskomponenten wie RSA oder AES Verschlüsselung sind zwar mittlerweile ausreichend durch Javascript-Bibliotheken abgedeckt, jedoch werden keine darauf aufbauenden Bibliotheken beispielsweise zur Unterstützung von Cryptographic Message Syntax Containern angeboten. Im Zuge dieses Projektes wurde eine Basiskomponente entwickelt, die basierend auf bestehenden Kryptographie Bibliotheken eine Verschlüsselung und Entschlüsselung von Daten in CMS-Containern ermöglicht.

Als beispielhafte Anwendung von CMS-Funktionalität im Browser kann die Ablage von Dokumenten bei Cloud-Speichern wie beispielsweise Dropbox oder Google Drive genannt werden. Abbildung 1 skizziert eine mögliche Implementierung einer Web-Applikation zur Verschlüsselung und sicheren Ablage von Dokumenten. Dazu erfolgt die Verschlüsselung der Daten im Browser mit einem zufälligen, von der Applikation generierten symmetrischen Schlüssel (Content Encryption Key). Dieser wird mit dem öffentlichen RSA-Schlüssels des Empfängers bzw. der Empfänger des Dokuments verschlüsselt. Der von der Web-Applikation erstellte CMS-Container enthält somit die verschlüsselten Daten und den verschlüsselten Content Encryption Key. Eine Entschlüsselung kann nur mit dem privaten RSA-Schlüssel des bzw. der Empfänger erfolgen. Daraus ergibt sich, dass weder bei Angriffen auf den Cloud-Speicher noch durch direkten Zugriff des Cloud-Betreibers auf gespeicherte Daten ein Zugriff auf das entschlüsselte Dokument möglich ist.

Um die Voraussetzungen für den oben genannten Use Case zu erfüllen, wurde im Zuge dieses Projektes eine Javascript-Komponente implementiert, die das Erstellen und Parsen von CMS-Containern im Browser ermöglicht. Dabei werden die folgenden Schritte unterstützt:

- Erstellen von CMS-Containern
- Erstellung zufälliger AES-Schlüssel und AES-Verschlüsselung von Daten im Browser
- Erstellung des CMS-Containers (CMS ASN.1 Struktur)
- Parsen von CMS-Containern
- Parsen der ASN.1 Struktur von CMS-Containern
- Extraktion des verschlüsselten Content Encryption Key
- Extraktion der verschlüsselten Daten, verwendeten Algorithmen etc.
- Entschlüsseln der Daten mit entschlüsselten Content Encryption Key

Für die Verschlüsselung und Entschlüsselung des Content Encryption Key kommt eine Java-Anwendung zum Einsatz. Der dazu notwendige Java-Code kann als Basis für eine Implementierung bei einem serverseitigen Dienst zur Verwaltung von kryptographischen Schlüssel verwendet werden. Der zu verschlüsselnde oder zu entschlüsselnde Content Encryption Key wird Base64 kodiert mit der Java-Anwendung ausgetauscht.

Die entwickelte Basiskomponente verwendet, soweit möglich, bestehende Technologien. Konkret werden folgende Javascript-Bibliotheken verwendet:

- **CryptoJS** ist eine der gängigsten Javascript Crypto-Bibliotheken. Unter anderem bietet CryptoJS AES und DES Verschlüsselung, Unterstützung für die Berechnung von HMAC-Codes, eine Implementierung der PBKDF2 Funktion zur Schlüsselableitung und mehrere Hash-Algorithmen. Dieses Projekt verwendet CryptoJS für die AES Verschlüsselung und Entschlüsselung der Daten.
- **jsrsasign** bietet einen ASN.1 Encoder und vereinfacht somit die Erstellung von CMS-Containern. Zusätzlich unterstützt jsrsasign das Parsen von X.509 Zertifikaten und vereinfacht dadurch die Erstellung der ASN.1 Struktur *IssuerAndSerialNumber*.
- **asn1js** stellt einen ASN.1 Parser zur Dekodierung von DER oder BER kodierten ASN.1-Strukturen zur Verfügung und wird zum Parsen von CMS-Containern verwendet.
- **jQuery** wird für die DOM-Manipulation des implementierten Demonstrators verwendet. Zusätzlich findet das **jQuery Base64** Plugin zur Kodierung und Dekodierung von Base64 Sequenzen Anwendung.

Einschränkungen

Die entwickelte Basiskomponente stellt keine vollständige CMS-Bibliothek dar sondern soll lediglich die Machbarkeit dieser demonstrieren. Nachfolgend werden Einschränkungen der implementierten Javascript-Komponente aufgelistet:

- Es wird lediglich der Typ EnvelopedData in Verbindung mit KeyTransRecipientInfos unterstützt.
- Als zusätzliche Einschränkung wird nur ein RecipientInfo Objekt pro CMS-Container unterstützt, eine Erweiterung auf mehrere RecipientInfo Objekte ist jedoch ohne große Probleme möglich.
- Die Verschlüsselung der Daten erfolgt in der momentanen Implementierung über AES im CBC Modus kann jedoch auf weitere Verschlüsselungsalgorithmen oder Modi erweitert werden.
- Die Verschlüsselung des Content Encryption Key erfolgt in einer zusätzlichen Java-Anwendung. Dabei ist ein manuelles Kopieren des verschlüsselten Schlüssels und anschließend des entschlüsselten Resultats notwendig. Sollten diese Operationen serverseitig z.B. unter Verwendung eines zentralen Schlüsselspeichers durchgeführt werden, kann der zur Verfügung gestellte Code weiterverwendet werden. Jedoch müsste dieser dazu um einen Mechanismus zum Matching des passenden Schlüssels anhand der Recipient Info erweitert werden.
- Momentan wird zur Schlüsselgenerierung die in CryptoJS verwendete Funktion Math.rand() verwendet. Math.random() kann jedoch nicht als kryptographisch starker Zufallszahlengenerator betrachtet werden. Bei Verwendung der entwickelten Basiskomponenten in einem Produktionsumfeld bzw. zur Verschlüsselung kritischer Daten, ist in jedem Fall auf Math.random() als Zufallszahlengenerator zu verzichten. Mögliche Alternativen sind beispielsweise die W3C Crypto API, Clipperz oder SJCL.

Links

Generell muss noch darauf hingewiesen werden, dass die Durchführung kryptographischer Operationen im Browser als problematisch angesehen werden kann. Für mehr Informationen werden folgende Beiträge empfohlen:

- http://matasano.com/articles/javascript-cryptography/ (EN)
- http://tonyarcieri.com/whats-wrong-with-webcrypto (EN)

1. Einleitung

Web-Applikationen ersetzen zunehmend traditionelle Desktop-Applikationen. Dahingehend findet oftmals auch eine Verarbeitung sensibler Daten im Browser statt. Applikationen, die sicherheitskritische Daten verarbeiten, fehlen jedoch wichtige kryptographische Komponenten. Basiskomponenten wie RSA oder AES Verschlüsselung sind zwar mittlerweile ausreichend durch Javascript-Bibliotheken abgedeckt, jedoch werden keine darauf aufbauenden Bibliotheken beispielsweise zur Unterstützung von *Cryptographic Message Syntax* [2] Containern angeboten.

Als beispielhafte Anwendung von CMS-Funktionalität im Browser kann die Ablage von Dokumenten bei Cloud-Speichern wie beispielsweise Dropbox oder Google Drive genannt werden. Abbildung 1 skizziert eine mögliche Implementierung einer Web-Applikation zur Verschlüsselung und sicheren Ablage von Dokumenten. Dazu erfolgt die Verschlüsselung der Daten im Browser mit einem zufälligen, von der Applikation generierten symmetrischen Schlüssel (*Content Encryption Key*). Dieser wird mit dem öffentlichen RSA-Schlüssels des Empfängers bzw. der Empfänger des Dokuments verschlüsselt. Der von der Web-Applikation erstellte CMS-Container enthält somit die verschlüsselten Daten und den verschlüsselten *Content Encryption Key* . Eine Entschlüsselung kann nur mit dem privaten RSA-Schlüssel des bzw. der Empfänger erfolgen. Daraus ergibt sich, 1 IETF, RFC 5652 http://tools.ietf.org/html/rfc5652 2 IETF, RFC 5652 http://tools.ietf.org/html/rfc5652 Bankverbindung: BAWAG P.S.K. Konto Nr. 01810-904-220, BLZ 14000 dass weder bei Angriffen auf den Cloud-Speicher noch durch direkten Zugriff des Cloud-Betreibers auf gespeicherte Daten ein Zugriff auf das entschlüsselte Dokument möglich ist.



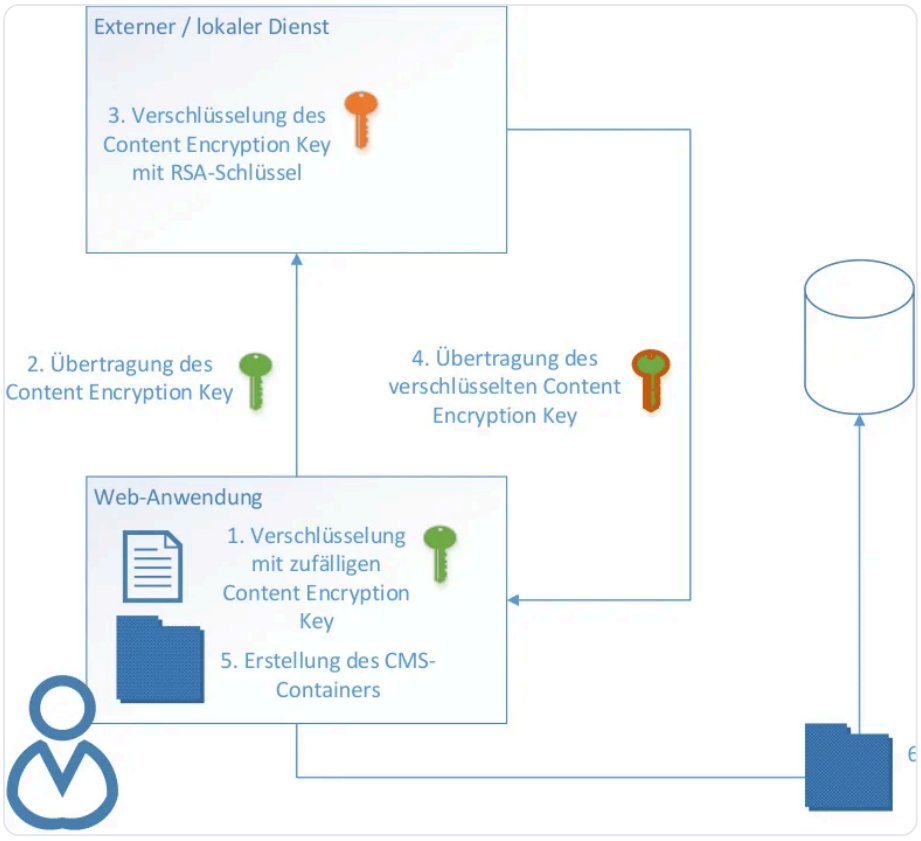


Abbildung 1: Use Case Cloud-Speicher Um die Voraussetzungen für den oben genannten Use Case zu erfüllen, wurde im Zuge dieses Projektes eine Javascript-Komponente implementiert, die das Erstellen und Parsen von CMS-Containern im Browser ermöglicht. Dabei werden die folgenden Schritte unterstützt:

Cloud-Speicher Containers



- Erstellen von CMS-Containern

o Erstellung zufälliger AES-Schlüssel und AES-Verschlüsselung von Daten im Browser o Erstellung des CMS-Containers (CMS ASN.1 Struktur)

- Parsen von CMS-Containern

o Parsen der ASN.1 Struktur von CMS-Containern o Extraktion des verschlüsselten *Content Encryption Key* o Extraktion der verschlüsselten Daten, verwendeten Algorithmen etc.

o Entschlüsseln der Daten mit entschlüsselten *Content Encryption Key* Für die Verschlüsselung und Entschlüsselung des *Content Encryption Key* kommt eine Java-Anwendung zum Einsatz. Der dazu notwendige Java-Code kann als Basis für eine Implementierung bei einem serverseitigen Dienst zur Verwaltung von kryptographischen Schlüssel verwendet werden. Der zu verschlüsselnde oder zu entschlüsselnde *Content Encryption Key* wird Base64 kodiert mit der Java-Anwendung ausgetauscht.

2. Umsetzung

2.1. Verwendete Technologien

Die entwickelte Basiskomponente verwendet, soweit möglich, bestehende Technologien. Konkret werden folgende Javascript-Bibliotheken verwendet:

- **CryptoJS** [3] ist eine der gängigsten Javascript Crypto-Bibliotheken. Unter anderem bietet CryptoJS AES und DES Verschlüsselung, Unterstützung für die Berechnung von HMAC-Codes, eine Implementierung der PBKDF2 Funktion zur Schlüsselableitung und mehrere Hash-Algorithmen. Dieses Projekt verwendet CryptoJS für die AES Verschlüsselung und Entschlüsselung der Daten.
- **jsrsasign** [4] bietet einen ASN.1 Encoder und vereinfacht somit die Erstellung von CMS-Containern. Zusätzlich unterstützt jsrsasign das Parsen von X.509 Zertifikaten und vereinfacht dadurch die Erstellung der ASN.1 Struktur *IssuerAndSerialNumber* .
- **asn1js** [5] stellt einen ASN.1 Parser zur Dekodierung von DER oder BER kodierten ASN.1-Strukturen zur Verfügung und wird zum Parsen von CMS-Containern verwendet.
- **jQuery** [6] wird für die DOM-Manipulation des implementierten Demonstrators verwendet. Zusätzlich findet das **jQuery Base64** [7] Plugin zur Kodierung und Dekodierung von Base64 Sequenzen Anwendung.

2.2. Code-Dokumentation

Neben Kommentaren im Quellcode wird im nachfolgenden Abschnitt eine kurze Übersicht der implementierten Methoden und Klassen gegeben. Die Basiskomponente verfügt momentan über zwei Klassen:

- ***KeyTransRecipientInfos:*** Eine Instanz der Klasse *KeyTransRecipientInfos* wird durch Aufruf *von parseKeyTransRecipientInfos(asn1)* erstellt und an den Aufrufer retourniert. Als Übergabe Parameter ist die ASN.1 Struktur von *KeyTransRecipientInfos* in Form eines *ASN1* Objektes (asn1js Datei asn1.js) anzugeben.

o *toKJURASN1()* retourniert die ASN.1 Repräsentation des *KeyTransRecipientInfos* Typs.

- ***EncryptedContentInfo*** : Eine Instanz der Klasse *EncryptedContentInfo* wird durch Aufruf von *parseEncryptedContentInfo(asn1)* erstellt und an den Aufrufer retourniert. Als Übergabe Parameter ist die ASN.1 Struktur von *EncryptedContentInfo* in Form eines *ASN1* Objektes (asn1js Datei asn1.js) anzugeben.

o *toKJURASN1()* retourniert die ASN.1 Repräsentation des *EncryptedContentInfo* Typs.

3 CryptoJS https://code.google.com/p/crypto-js/ 4 Jsrsasign http://kjur.github.io/jsrsasign/ 5 asn1js https://github.com/lapo-luchini/asn1js 6 jQuery http://jquery.com/ 7 jQuery Base64 http://plugins.jquery.com/base64/ Um einen CMS-Container erstellen bzw. dekodieren zu können, werden folgende Methoden benötigt (diese beiden Methoden dienen als Einsprungspunkt für Anwendungen, die die entwickelte Basiskomponente verwenden):

- createCMS(x509cert_pem_str, encrypted_aes_key_base64, aes_iv_b64, encrypted_content_hex_str) erstellt einen CMS-Container in PEM Kodierung.
- *parseCMSString(cms_value)* : Retourniert eine Instanz der Objekte *EncryptedContentInfo* und *KeyTransRecipientInfo* . Die beiden Objekte werden aus der PEM Repräsentation des CMS-Containers geparst.

Methoden zur AES Verschlüsselung:

- *generateAESRandomKey(bitlen)* erstellt einen zufälligen AES Schlüssel. Retourniert den Schlüssel und den generierten Initialisierungsvektor in Base64 Kodierung.
- *decryptAES(symmetric_key_b64, iv_hex_str, cipher_hex_str)* entschlüsselt die Hex-Repräsentation des Ciphertexts (wie in CMS-Container angegeben).
- *aesEncryptToOctetString(key_base64, iv_base64, plaintext_str)* verschlüsselt einen String und retourniert den Ciphertext als Hex-String (wie für die Erstellung des Containers benötigt).

Weitere Methoden:

- *getIssuerName(issuerString)* parst die in X.509 Zertifikaten vorhandene Repräsentation des Issuers und retourniert den Issuer String als ASN.1 SEQUENCE, mit den einzelnen Felders als SET.
- *hex2a(hexx)* stellt den entschlüsselten Hex-String als Ascii dar.
- *a2hex(str)* erstellt einen Hex-String aus einem Ascii String. Wird für die Verschlüsselung benötigt.

2.3. Einschränkungen

Die entwickelte Basiskomponente stellt keine vollständige CMS-Bibliothek dar sondern soll lediglich die Machbarkeit dieser demonstrieren. Nachfolgend werden Einschränkungen der implementierten Javascript-Komponente aufgelistet:

- Es wird lediglich der Typ *EnvelopedData* in Verbindung mit *KeyTransRecipientInfos* unterstützt.
- Als zusätzliche Einschränkung wird nur ein *RecipientInfo* Objekt pro CMS-Container unterstützt, eine Erweiterung auf mehrere *RecipientInfo* Objekte ist jedoch ohne große Probleme möglich.
- Die Verschlüsselung der Daten erfolgt in der momentanen Implementierung über AES im CBC Modus kann jedoch auf weitere Verschlüsselungsalgorithmen oder Modi erweitert werden.
- Die Verschlüsselung des *Content Encryption Key* erfolgt in einer zusätzlichen Java-Anwendung. Dabei ist ein manuelles Kopieren des verschlüsselten Schlüssels und anschließend des entschlüsselten Resultats notwendig. Sollten diese Operationen serverseitig z.B. unter Verwendung

eines zentralen Schlüsselspeichers durchgeführt werden, kann der zur Verfügung gestellte Code weiterverwendet werden. Jedoch müsste

dieser dazu um einen Mechanismus zum Matching des passenden Schlüssels anhand der *Recipient Info* erweitert werden.

Kryptographische Operationen im Web-Browser Die Durchführung kryptographischer Operationen im Browser kann als problematisch angesehen werden [8] . Falls es einer Angreiferin bzw. einem Angreifer gelingt, die Kommunikation zwischen Client und Server abzufangen, kann der übertragene Javascript-Code modifiziert werden und so die Sicherheit der kryptographischen Funktionen beeinträchtigen. Als weiterer Angriffsvektor gilt das Modifizieren von Javascript-Code über Cross-site Scripting (XSS) Attacken. Die W3C Crypto API bietet zwar Verbesserungen im Bereich von Javascript Kryptographie, da der Krypto-Code bereits im Browser implementiert ist und so während der Übertragung nicht modifiziert werden kann, jedoch wird auch die Sicherheit der W3C Crypto API aufgrund genereller Sicherheitsprobleme von Javascript Code im Browser beeinträchtigt.

Javascript Zufallszahlengenerator Die Sicherheit der Generierung des zufälligen AES Schlüssels bei der Erstellung des CMS-Containers basiert auf der Sicherheit der Implementierung der Math.random() Funktion. Math.random() kann jedoch nicht als kryptographisch starker Zufallszahlengenerator betrachtet werden [9] . Es existieren mehrere Javascript Bibliotheken, die diese Funktionalität bieten [10] . In dieser Demonstration wird jedoch der in CryptoJS verwendete Zufallsgenerator Math.random() verwendet. Denkbar wäre auch eine Sammlung von Entropie über Bewegungsmuster des Mauszeigers etc. Der Großteil der aktuellen Browser bietet mittlerweile die Funktion. *window.crypto.getRandomValues(outputarray)* der W3C Crypto API, die auf den vom Betriebssystem angebotenen Zufallszahlengenerator zugreift. Bei Verwendung der entwickelten Basiskomponenten in einem Produktionsumfeld bzw. zur Verschlüsselung kritischer Daten, ist in jedem Fall auf Math.random() als Zufallszahlengenerator zu verzichten.

3. Ausblick

Die entwickelte Basiskomponente zur Erstellung und Dekodierung von CMS-Containern bietet für simple Use Cases bereits jetzt ausreichend Funktionalität. Um jedoch Daten für mehrere Empfänger zu verschlüsseln oder Daten zu entschlüsseln, die für mehrere Empfänger bestimmt sind, sind Erweiterungen notwendig. Unter anderem ist ein Matching Mechanismus zum Finden des richtigen Empfängers zu implementieren. Des Weiteren wird der *Content Encryption Key* momentan in einer externen Java-Anwendung ver- und entschlüsselt. Die Übertragung des verbzw. entschlüsselten Schlüssels erfolgt momentan durch manuelles Copy&Paste. Die Basiskomponente könnte um einen automatischen Kommunikationskanal erweitert werden, z.B. HTTP Post an anzugebende URL oder Kommunikation über ein iFrame. Dabei sind jedoch Implikationen bezüglich Sicherheit des übertragenen Schlüssels zu beachten.

8 Kritische Betrachtungen zu Javascript Kryptographie im Browser: http://matasano.com/articles/javascript- cryptography/, http://tonyarcieri.com/whats-wrong-with-webcrypto ~~9~~ http://stackoverflow.com/questions/578700/how-trustworthy-is-javascripts-random-implementation-in- various-browsers ~~10~~ Clipperz bietet einen kryptographisch starken Fortuna Zufallszahlengenerator https://clipperz.is/security_privacy/crypto_algorithms/#prng. Als Entropie-Quellen können Mauszeigerbewegungen, Tastaturschläge oder die Angabe eines Zeitintervalls dienen.

Dokumentation zur Verwendung des PRNG https://groups.google.com/forum/#!topic/clipperz/jiAhRo0xjo8
Die Stanford Javascript Crypto Library (SJCL) verfügt auch über eine Implementierung von Fortuna: http://crypto.stanford.edu/sjcl/