

Bachelorarbeit

Angriffe auf Microsoft RMS

Kevin Böhl

Datum: 11. März 2016

Erstprüfer: Prof. Dr. Jörg Schwenk
Zweitprüfer: M. Sc. Christian Mainka

Betreuer: M. Sc. Christian Mainka
B. Sc. Martin Grothe

Ruhr-Universität Bochum



Lehrstuhl für Netz- und Datensicherheit
Prof. Dr. Jörg Schwenk
Homepage: www.nds.rub.de

Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Datum

Unterschrift

Abstract

In dieser Bachelorarbeit werden Schwachstellen der Rights Management Services von Microsoft in Windows Server 2012 R2 untersucht. Ein erstes Ziel ist es ein Programm vorzustellen, welches es einem Angreifer erlaubt, auf ein mit RMS geschütztes Word-Dokument, auf welches er nur lesend zugreifen darf, Vollzugriff zu erlangen. Des Weiteren werden zwei weitere Angriffe untersucht, nämlich ob ein Benutzer ein RMS geschütztes Dokument im Namen eines anderen Benutzers erstellen kann und ob es möglich ist, Vollzugriff auf ein geschütztes Dokument zu erlangen, wenn nicht einmal das Leserecht gesetzt ist.

Inhaltsverzeichnis

1 Einführung	1
1.1 Motivation	1
1.2 Ziele dieser Arbeit	1
1.3 Gliederung dieser Arbeit	2
2 Grundlagen	3
2.1 Enterprise Rights Management	3
2.2 Open Office XML	3
2.3 Active Directory Rights Management Services	4
2.4 DPAPI	6
3 Testumgebung	8
4 Vollzugriff auf ein RMS geschütztes Word-Dokument erlangen, wenn nur das Lese- recht gesetzt ist	10
4.1 Angriffsidee	11
4.2 Voraussetzungen für den Angriff	12
4.3 Durchführung des Angriffs	12
4.4 Struktur des Word-Dokuments	13
4.5 Implementierung	15
4.6 Fazit zum ersten Angriff	22
5 Weitere Angriffsszenarien	24
5.1 Übernehmen der Benutzeridentität des Opfers	24
5.2 Durchführung der Angriffe, wenn privater Schlüssel des SPC des Opfers bekannt ist	25
5.3 Extrahieren des privaten Schlüssels des SPC	26
5.4 Verwenden der DPAPI zum Entschlüsseln der Registry-Einträge	27
5.5 Weitere Untersuchungen	30
5.6 Fazit	30
6 Zusammenfassung	31
6.1 Ausblick	31
Abbildungsverzeichnis	32
Literaturverzeichnis	34

1 Einführung

In diesem Kapitel geht es um die Motivation der Arbeit. Auch werden hier die Ziele sowie die Gliederung der Arbeit vorgestellt.

1.1 Motivation

Das Thema Rechteverwaltung spielt in Unternehmen eine immer wichtigere Rolle. So ist es besonders in größeren Unternehmen erwünscht, dass der Zugriff auf digitale Dokumente für die Mitarbeiter eingeschränkt wird. Nicht jeder Mitarbeiter soll Vollzugriff auf alle firmeninternen Ressourcen besitzen, sondern Jeder sollte im Idealfall immer nur die Rechte haben, welche wirklich notwendig sind. In der Praxis oft eingesetzt werden die *Active Directory Rights Management Services (AD RMS)* von Microsoft, um diese Rechteeinschränkungen zu regeln. Bei einer Analyse der genauen Funktionsweise der AD RMS wurden potentielle Schwachstellen entdeckt[1], die in dieser Bachelorarbeit näher untersucht werden sollen. Es soll herausgefunden werden, ob diese Schwachstellen in der Praxis ausnutzbar sind, sodass Unbefugte Rechte für Dokumente erlangen können, die der Administrator so nicht vorgesehen hat. So wurde in Jan Kaiser's Bachelorarbeit dargelegt, dass es wahrscheinlich möglich ist, dass ein Benutzer, welcher zwar ein mit RMS geschütztes Dokument anzeigen kann, aber sonst keine Berechtigungen für dieses Dokument besitzt, diese Rechteeinschränkung theoretisch umgehen kann. Dies wäre für das Unternehmen kritisch, wenn dieser Mitarbeiter das Unternehmen verlässt und so sensible Firmendokumente stehlen kann, welche er dann auf dem Schwarzmarkt oder an die Konkurrenz verkaufen könnte.

1.2 Ziele dieser Arbeit

Das erste Ziel der Arbeit ist eine Anwendung mit dem RMS-SDK zu entwickeln, welche es ermöglicht, ein mit RMS geschütztes Word-Dokument zu entschlüsseln, falls derjenige Benutzer, der die Anwendung ausführt, mindestens das Leserecht für das Dokument besitzt. Es soll gezeigt werden, wie dieser Angriff genau funktioniert, welche Voraussetzungen gegeben sein müssen und wie ein solcher Angriff in der Praxis aussieht. Ein weiteres Ziel dieser Arbeit ist die Analyse von zwei weiteren Angriffen. Einerseits soll untersucht werden, ob ein Benutzer ein mit RMS geschütztes Word-Dokument im Namen eines anderen Benutzers erstellen kann. Andererseits soll herausgefunden werden, ob ein Nutzer auch Zugriff auf ein geschütztes Word-Dokument erhalten kann, wenn dieser nicht einmal das Leserecht für das entsprechende Dokument besitzt. Getestet werden sollen diese Angriffe in einer Testumgebung, in der Windows Server 2012 R2 läuft.

1.3 Gliederung dieser Arbeit

Zunächst werden in den Grundlagenkapiteln dieser Arbeit die Begriffe erklärt, welche im Rahmen dieser Bachelorarbeit benötigt werden. Dazu zählt die Funktionsweise der AD RMS, da das Verständnis dieser für die Analyse der beschriebenen Angriffe notwendig ist. Auch wird die Funktionsweise der DPAPI, einer in Microsoft Windows integrierten Programmierschnittstelle, erläutert; diese wird bei der Analyse des zweiten und dritten Angriffs benötigt. Im dritten Kapitel wird eine Testumgebung vorgestellt, welche ein vereinfachtes Netzwerk in einem Unternehmen simulieren soll, in welchem die AD RMS verwendet wird. Die in dieser Arbeit getesteten Angriffe wurden alle in dieser Testumgebung analysiert. Im vierten Kapitel wird eine in Visual C++ entwickelte Anwendung vorgestellt, welche es einem Benutzer ermöglicht, auf ein mit RMS geschütztes Word-Dokument Vollzugriff zu erlangen, sofern dieser Benutzer das Leserecht für das Dokument besitzt. Des Weiteren werden in diesem Kapitel alle notwendigen Voraussetzungen für diesen Angriff diskutiert. Die beiden weiteren Angriffe werden im fünften Kapitel zusammen behandelt, da die Vorgehensweise ähnlich ist. Schließlich werden im letzten Kapitel die Ergebnisse der Arbeit zusammengefasst.

2 Grundlagen

In diesem Kapitel werden einige grundlegende Begriffe erläutert. Dies sind zum einen *Enterprise Rights Management*, sowie *Open Office XML*, da die Angriffe auf mit RMS geschützte Microsoft Word-Dokumente im *.docx*-Format getestet werden, welche grundsätzlich auf diesem Format basieren. Danach wird auf die grundlegende Funktionsweise der AD RMS von Microsoft eingegangen, es werden diejenigen Details erläutert, welche für das Verständnis der Angriffe notwendig sind. Eine vollständige Analyse der Funktionsweise der AD RMS wird hier nicht behandelt, diese ist in Jan Kaiser's Bachelorarbeit zu finden. Schließlich wird die Funktionsweise der DPAPI erläutert, diese spielt bei dem zweiten und dritten Angriff eine entscheidende Rolle. Alle weiteren Begrifflichkeiten sowie spezielle Details werden an den Stellen in der Arbeit erklärt, an denen sie gebraucht werden.

2.1 Enterprise Rights Management

Besonders in Unternehmen spielt das Thema Rechteverwaltung eine zentrale Rolle. So müssen dort für jeden Mitarbeiter die Zugriffe auf digitale Dokumente und Verzeichnisse eingeschränkt werden. Beispielsweise können bestimmte firmeninterne Dokumente so geschützt werden, dass Mitarbeiter nur lesend auf diese Dokumente zugreifen dürfen. Diese Art der Rechteverwaltung wird *Enterprise Rights Management (ERM)* genannt; ein in der Praxis oft verwendetes ERM-System sind die Active Directory Rights Management Services von Microsoft, um welche es in dieser Bachelorarbeit gehen wird.

2.2 Open Office XML

Open Office XML (auch ISO/IEC 29500 Information technology – Office Open XML formats oder ECMA-376 Office Open XML File Formats) beschreiben Dateiformate auf XML-Basis, welche seit der Einführung von Microsoft Office 2007 existieren. Die Dateierendungen sind *.docx* für *Microsoft Word*, *.xlsx* für *Microsoft Excel* und *.pptx* für *Microsoft PowerPoint*. Das Format bringt einige Vorteile mit sich, wie beispielsweise eine stark reduzierte Dateigröße, da das Open XML Format die ZIP-Komprimierungstechnologie verwendet[2]. Ein weiterer Vorteil ist beispielsweise die verbesserte Wiederherstellung beschädigter Dateien, da die Dateien so aufgebaut sind, dass die einzelnen Komponenten voneinander getrennt sind.

2.3 Active Directory Rights Management Services

Die Microsoft Active Directory Rights Management Services (RMS) sind eine von fünf verschiedenen Serverrollen, welche durch den Netzwerkadministrator in Windows Server hinzugefügt werden können. RMS basiert auf einer *public key infrastructure (PKI)*, bestehend aus verschiedenen Zertifikaten und Lizenzen (siehe Abbildung).

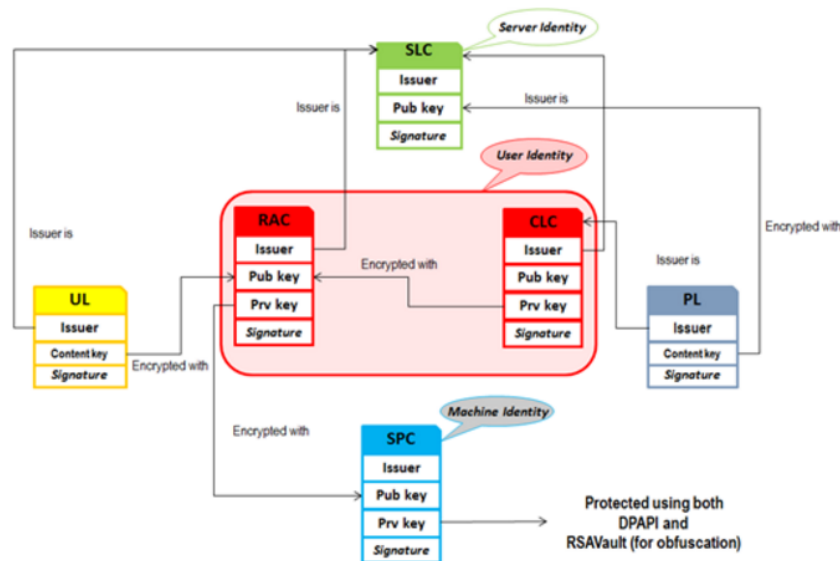


Abbildung 2.1: Abhängigkeiten der Zertifikate und Lizenzen
[3]

Ein AD RMS Server ist gekennzeichnet durch ein *Server Licensor Certificate (SLC)*, welches im Fall von Windows Server 2012 R2 selbstsigniert ist. Jeder Client Computer hat ein sogenanntes *Security Processor Certificate (SPC)*, dieses identifiziert den jeweiligen Computer; durch zwei weitere Zertifikate *Rights Account Certificate (RAC)* und *Client Licensor Certificate (CLC)* wird der Benutzer identifiziert. Diese Zertifikate (SPC, RAC und CLC) werden erstellt, wenn ein Benutzer zum ersten Mal eine Datei mit RMS schützt oder zum ersten Mal eine mit RMS geschützte Datei öffnet. Alle Zertifikate und Lizenzen haben einen *Issuer*, dieser ist der Ersteller des jeweiligen Zertifikats bzw. der jeweiligen Lizenz. Zudem hat jedes Zertifikat einen öffentlichen und einen privaten Schlüssel. Der private Schlüssel steht nie im Zertifikat selbst, sondern er ist verschlüsselt mit dem öffentlichen Schlüssel eines anderen Zertifikats (siehe Grafik). Eine Ausnahme ist hier der private Schlüssel des SPC, dieser liegt mit der *DPAPI* und *RSVault* geschützt in der Registry des jeweiligen Benutzers. Wenn eine Datei mit RMS geschützt wird, wird eine sogenannte *Publishing License (PL)* erstellt. Diese enthält den Namen des Autos der Datei, die Rechte, die der Autor für andere Benutzer vergeben hat, den Content-Key, mit welchem die entsprechende Datei geschützt wird (der Content-Key ist mit dem öffentlichen Schlüssel des SLC geschützt), sowie eine Signatur über die Publishing

License, erstellt mit dem privaten Schlüssel des CLC. Diese PL wird an den Server gesendet, des Weiteren steht sie in dem entsprechenden geschützten Dokument. Wenn ein Benutzer eine geschützte Datei öffnet, so sendet der Server diesem Benutzer eine sogenannte *Use License (UL)*, diese enthält den Content-Key der Datei, welcher mit dem öffentlichen Schlüssel des RACs des Benutzers verschlüsselt wird, sowie die Rechte, die der Autor der Datei für diesen Benutzer vergeben hat. Der Content-Key wird daraufhin mit dem privaten Schlüssel des RACs entschlüsselt, welcher der Anwendung zusammen mit den Rechten übergeben wird. Da der Content-Key nicht an die Rechte gebunden ist und die Anwendung im Endeffekt für das Einhalten der Rechte verantwortlich ist, kann der Content-Key prinzipiell isoliert von den Rechten verwendet werden, um Dokumente zu entschlüsseln. Dies wird im vierten Kapitel dieser Arbeit betrachtet. Öffnet der Autor der Datei das geschützte Dokument selbst, so findet dies offline statt, denn die UL des Autors ist immer in der geschützten Datei vorhanden, d.h. er kann den Content-Key mit dem privaten Schlüssel seines RAC entschlüsseln. Es werden in diesem Fall also keine Informationen vom Server an den Benutzer geschickt.

2.4 DPAPI

Die DPAPI (Data Protection Application Programming Interface) ist eine einfache Programmierschnittstelle, die in Windows 2000 und allen nachfolgenden Versionen verfügbar ist. Sie stellt zwei Funktionen bereit, die von Benutzer- und Systemprozessen verwendet werden können. Diese zwei Funktionen sind eine Verschlüsselungs- und eine Entschlüsselungsfunktion mit jeweils weiteren Optionen. Die Funktionalitäten werden durch das Betriebssystem selber bereitgestellt, d.h. es sind keine zusätzlichen Bibliotheken notwendig, um die Funktionen der DPAPI verwenden zu können; es muss lediglich die *Crypt32.dll* geladen werden, denn die DPAPI-Schnittstelle ist ein Teil dieser DLL (Dynamic Link Library), die *Crypt32.dll* wiederum ist ein Teil der *CryptoAPI*. Im Folgenden Schaubild ist die Funktionsweise der DPAPI zu sehen.

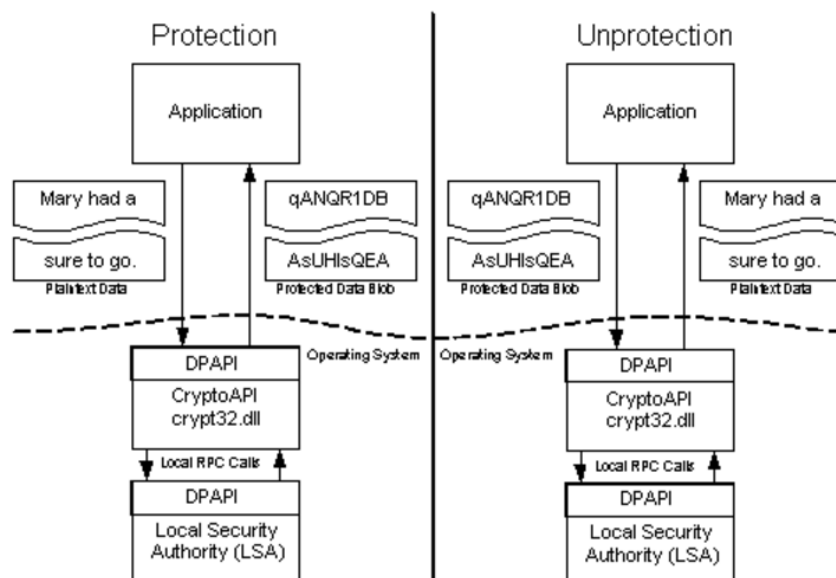


Abbildung 2.2: Funktionsweise der DPAPI

[4]

Eine Anwendung, welche die DPAPI benutzt, kann einen beliebigen Klartext der DPAPI Verschlüsselungsfunktion übergeben, woraufhin die Funktion einen lokalen *Remote Procedure Call* an die *Local Security Authority (LSA)* durchführt. Die LSA ist ein Systemprozess, welcher immer läuft, wenn der Computer gebootet ist, sie authentifiziert den Benutzer und loggt diesen ins lokale System ein. Auch ist sie verantwortlich, dass bestimmte Sicherheitsrichtlinien auf dem System eingehalten werden. Die eigentliche Verschlüsselung wird in der *Crypt32.dll* realisiert. Nach erfolgreicher Verschlüsselung liefert die Funktion einen geschützten *DPAPI data blob* (Im Folgenden einfach Data-Blob genannt) zurück. Die Entschlüsselungsfunktion der DPAPI kann diesen Data-Blob wieder entschlüsseln, der Vorgang ist analog zur Verschlüsselung. Die DPAPI ist darauf ausgelegt Verschlüsselungsfunktionen für den Benutzer bereitzustellen, daher spielt bei der Verschlüsselung das Login-Passwort des jeweiligen Nutzers eine zentrale Rolle. Das Passwort direkt wird

allerdings nicht verwendet, sondern ein Hash des Passworts. Eine Anwendung, könnte theoretisch einen geschützten Data-Blob entschlüsseln, welcher von einer anderen Anwendung erstellt worden ist, sofern beide Anwendungen unter dem gleichen Benutzer laufen. Da es Fälle gibt, bei denen dies nicht erwünscht ist, ist es möglich sekundäre Entropie, also ein zusätzliches Geheimnis, zu verwenden, wenn Daten mit der DPAPI-Verschlüsselungsfunktion geschützt werden sollen. Die Entschlüsselung kann daraufhin nur erfolgreich durchgeführt werden, wenn der DPAPI-Entschlüsselungsfunktion diese sekundäre Entropie wieder als Parameter übergeben wird. Jede Anwendung ist selbst dafür verantwortlich, wo der entstandene Data-Blob und die eventuell genutzte sekundäre Entropie abgespeichert wird. Wenn es gewünscht ist, dass keine anderen Anwendungen den geschützten Data-Blob entschlüsseln sollen, so darf die Entropie nicht einfach ungeschützt in einer Datei abgespeichert werden.

Schlüsselgenerierung:

Die DPAPI erstellt einen Master-Key, dieser wird aus dem Passwort des jeweiligen Benutzers abgeleitet. Das Verfahren, welches dazu verwendet wird nennt sich *Password-Based Key Derivation*, welches in PKCS #5 beschrieben wird. Dieser abgeleitete Master-Key ist mit dem Passwort verschlüsselt, als Verfahren wird Tripple-DES verwendet. Werden Daten mit der DPAPI verschlüsselt, wird allerdings nicht der Master-Key selbst benutzt, sondern es wird ein Session-Key generiert, welcher vom Master-Key selber, von zufälligen Daten und von der eventuell benutzten sekundären Entropie anhängt. Die zufälligen Daten werden in dem geschützten Data-Blob abgespeichert. Wird die Entschlüsselungsfunktion auf diesen Data-Blob angewendet, werden diese zufälligen Daten aus dem Data-Blob extrahiert, um damit den Session-Key zu generieren, welcher dann für die Entschlüsselung verwendet wird. Der Session-Key selber wird normalerweise nicht abgespeichert. Die Struktur der entstehenden Data-Blobs ist von Microsoft nicht dokumentiert, allerdings konnte *passcape*[5] die Struktur vollständig rekonstruieren.

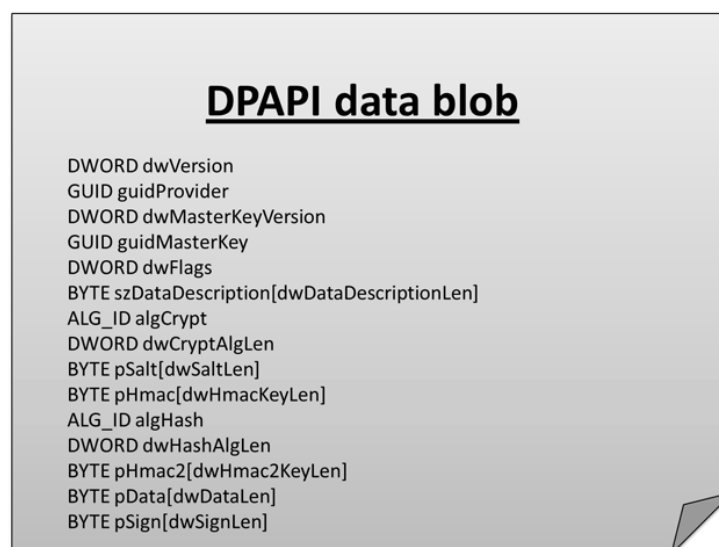


Abbildung 2.3: Struktur der Data-Blobs nach *passcape*[5]

3 Testumgebung

Um die Angriffe auf die Rights Management Services von Microsoft zu testen, wird eine Testumgebung benötigt, die hier kurz erläutert werden soll. Diese Testumgebung soll ein Netzwerk in einem Unternehmen simulieren, in welchem die AD RMS verwendet werden. Grundsätzlich ist diese Testumgebung die gleiche wie in Jan Kaiser's Bachelorarbeit *Analyse von Microsofts Rights Management Services in Windows Server 2012 R2*, dort ist auch eine Anleitung für die Einrichtung dieser Testumgebung zu finden.

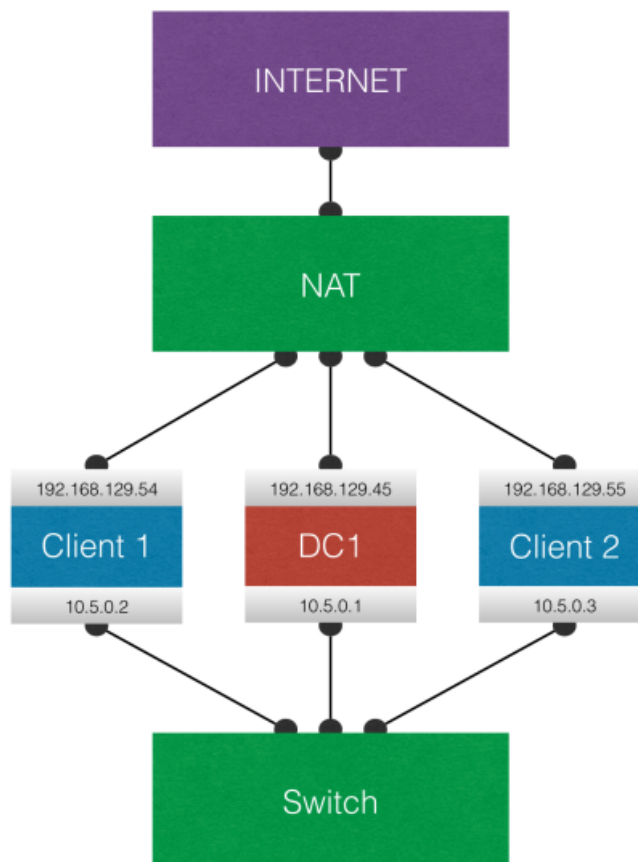



Abbildung 3.1: Aufbau der Testumgebung
[1]

Die Testumgebung besteht aus drei virtuellen Maschinen, wobei zwei davon als Clients konfiguriert sind, auf welchen jeweils Windows 7 Enterprise in der 64bit Version läuft. Die andere virtuelle Maschine ist als Server konfiguriert, als Betriebssystem wird hier Windows Server 2012 R2 verwendet. Damit die Maschinen miteinander kommunizieren können, sind diese über einen Switch miteinander verbunden, es ist also eine Netzwerkkarte dementsprechend konfiguriert, sodass diese Kommunikation möglich ist. Damit die beiden Clients und der Server auf das Internet zugreifen können, haben die Maschinen jeweils eine weitere Netzwerkkarte, die entsprechend konfiguriert ist. Der Zugriff auf das Internet selbst findet über einen NAT-Router statt. Es sind drei Benutzerkonten vorhanden, welche auf beiden Clients verwendet werden können, diese Benutzerkonten heißen *Max Mustermann*, *Erika Musterfrau* und *Hans Meier*. Um die Angriffe zu testen ist noch weitere Software notwendig, die auf den beiden Clients installiert werden muss. Zum einen wird Microsoft Office benötigt, da es in dieser Bachelorarbeit um Microsoft Word-Dokumente im *.docx*-Format geht, auf welche die jeweiligen Angriffe getestet werden; hier verwendet wird die Version 2013. Um eine Anwendung entwickeln zu können, welche ein mit RMS geschütztes Word-Dokument entschlüsselt, wird eine geeignete Entwicklungsumgebung benötigt, verwendet wurde hier Microsoft Visual Studio 2015 Professional. Als letztes wird noch das RMS SDK 2.1 benötigt, da diese die Funktionen bereitstellt, um eine Anwendung mit RMS-Funktionalitäten zu erstellen, hier verwendet wurde die Version vom 11.12.2015.

4 Vollzugriff auf ein RMS geschütztes Word-Dokument erlangen, wenn nur das Leserecht gesetzt ist

In diesem Kapitel wird das Szenario betrachtet, dass ein Benutzer in einer Domäne auf ein mit RMS geschütztes Word-Dokument lesend zugreifen kann, aber sonst keine Berechtigungen für dieses Dokument hat. Der Benutzer kann sich den Inhalt des Dokuments wie gewohnt anzeigen lassen, indem er es mit *Word* öffnet. Im folgenden Beispiel wird ein Word-Dokument namens *vonErikaAnMax.docx* betrachtet, welches auf Client2 unter Erika Musterfrau erstellt wurde und auf Client1 mit dem Benutzer Max Mustermann geöffnet werden soll. Es wird daraufhin eine Anwendung vorgestellt, welche den Schutz des Dokuments entfernen kann, sodass Max Vollzugriff auf das Dokument erlangt. Außerdem werden alle notwendigen Voraussetzungen besprochen, welche notwendig sind, um diesen Angriff durchzuführen. Versucht Max Mustermann die

 EINGESCHRÄNKTER ZUGRIFF Berechtigung ist zurzeit eingeschränkt. Nur bestimmte Benutzer haben Zugriff zu diesem Inhalt.

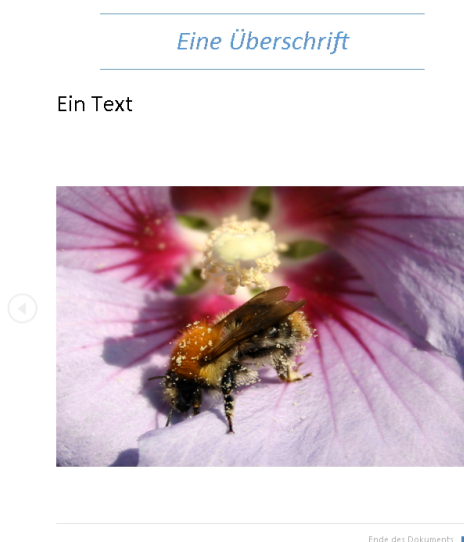


Abbildung 4.1: Max öffnet die Datei

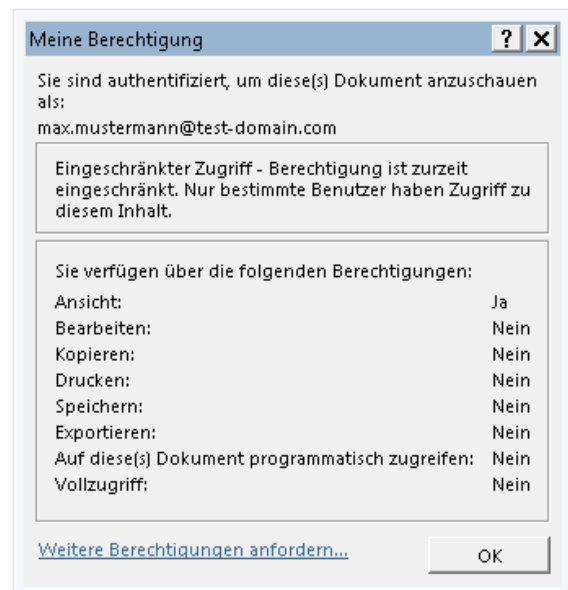


Abbildung 4.2: Berechtigungen von Max

Datei zu öffnen, so kann er sich zwar den Inhalt anzeigen lassen, jedoch hat er keine Möglichkeit die Datei zu kopieren oder zu drucken, da Erika Musterfrau ihm diese Rechte nicht eingeräumt hat. Dies wird auch ersichtlich, wenn Max Mustermann sich alle Rechte anzeigen lässt, welche er für dieses Dokument besitzt.

4.1 Angriffsidee

Da der Benutzer Max Mustermann den Inhalt des Dokuments anzeigen kann, besitzt er eine gültige UL. In dieser UL steht einerseits der Content-Key, mit welchem das Dokument geschützt ist; sie enthält auch die Rechte, die der Autor der Datei (in diesem Fall Erika Musterfrau) für Max eingeräumt hat. Die Angriffsidee besteht darin, die Rechteeinschränkung zu ignorieren und den Content-Key zu verwenden, um das geschützte Dokument vollständig zu entschlüsseln, um so den Schutz zu entfernen. Microsoft stellt ein RMS-SDK bereit, mit welchem eigene Anwendungen mit RMS-Funktionen entwickelt werden können. Für den Angriff muss eine geeignete Funktion gefunden werden, welche mit RMS geschützten Inhalt entschlüsseln kann. Es stehen im RMS-SDK 2.1 vier Funktionen zur Verfügung, welche hier kurz näher untersucht werden, um zu beurteilen, ob diese für den Angriff in Frage kommen oder nicht.

IpcfReadFile()[6]: Die *IpcfReadFile()*-Funktion ist eine File-API Funktion, d.h. der Funktion kann ein mit RMS-geschütztes Dokument übergeben werden, woraufhin das Dokument entschlüsselt wird. Leider kann diese Funktion hier nicht verwendet werden, da die File-API keine nativ geschützten Dokumente unterstützt, wozu auch Office-Dokumente gehören.

IpcfDecryptFile()[7]: Die *IpcfDecryptFile()*-Funktion ist ebenfalls eine File-API Funktion, des Weiteren wird das EXTRACT-Recht für das entsprechende Dokument benötigt, um diese Funktion erfolgreich auszuführen. Diese Funktion kommt also auch nicht in Frage.

IpcfDecryptFileStream()[8]: Hier liegt der gleiche Fall wie bei der *IpcfDecryptFile()*-Funktion vor, daher kann auch diese Funktion nicht für den Angriff verwendet werden.

IpcfDecrypt()[9]: Die letzte Funktion ist die *IpcfDecrypt()*-Funktion und die einzige die für den Angriff verwendet werden kann, da sie keine File-API Funktion ist. Allerdings ist es hier nicht möglich, der Funktion einfach das geschützte *.docx*-Dokument zu übergeben, da der Funktion die verschlüsselten Bytes übergeben werden müssen und nicht das geschützte Dokument an sich mit seiner gesamten Dokumentenstruktur.

Um den Angriff durchzuführen muss also eine Anwendung mit dem RMS SDK 2.1 entwickelt werden, welche den geschützten Inhalt in *vonErikaAnMax.docx* extrahiert und der *IpcfDecrypt()*-Funktion übergibt. Zuvor muss der Content-Key aus der UL eingelesen werden, welche der Server dem Benutzer ausstellt; dies wird erreicht, indem die *IpcfGetKey()*-Methode[10], welche das RMS-SDK 2.1 ebenfalls zur Verfügung stellt auf die in dem Dokument enthaltene Lizenz angewendet wird. Streng genommen ist in dem Dokument eine Lizenzkette über weitere, in diesem Kontext nicht relevante, Zertifikate enthalten. Der Einfachheit halber wird in den folgenden Abschnitten aber immer nur von *der Lizenz* gesprochen, die eingelesen werden muss.

4.2 Voraussetzungen für den Angriff

Um den Angriff durchzuführen muss derjenige Benutzer in der Domäne, welcher das Dokument entschlüsseln soll, mindestens das Leserecht für das Dokument besitzen, andere Rechte sind nicht notwendig. Auch das Recht *auf diese(s) Dokument programmatisch zugreifen* ist nicht erforderlich. Des Weiteren muss die aktuelle Version des RMS Clients 2.1 auf der Maschine des Angreifers installiert sein, welche zum Zeitpunkt des Schreibens dieser Bachelorarbeit die Version vom 12.11.2015 ist. Da die in diesem Kapitel vorgestellte Anwendung Funktionen der *Common Language Runtime* verwendet, muss zusätzlich das *Visual C++ Redistributable 2015* installiert sein, da dieses das Verwenden dieser Funktionen ermöglicht. Sind alle diese Voraussetzungen gegeben, kann der entsprechende Benutzer das Word-Dokument in seine XML-Struktur zerlegen (beispielsweise mit einem Programm wie *7zip*[2]) und die Anwendung ausführen und somit den Schutz der Datei entfernen. Beim Ausführen des Programms muss der Client eine Verbindung zum RMS-Server aufbauen können, der Angriff funktioniert also nicht offline.

4.3 Durchführung des Angriffs

Die Durchführung des Angriffs ist für den Benutzer sehr einfach. Es muss lediglich die Anwendung in den gleichen Ordner legen wie die XML-Dateien der verschlüsselten Word-Datei und dann die Anwendung ausführen. Daraufhin erhält der Benutzer das entschlüsselte Word-Dokument mit dem Namen *decrypted.docx*.

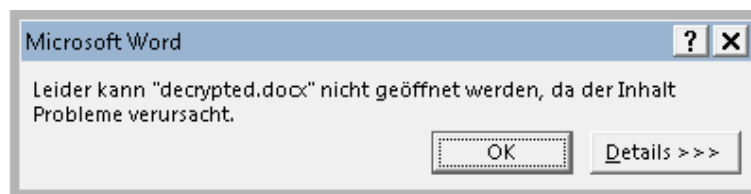


Abbildung 4.3: Fehlermeldung 1 beim Versuch die Datei zu öffnen

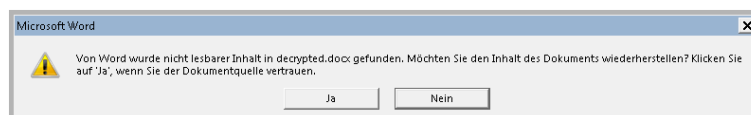


Abbildung 4.4: Fehlermeldung 2 beim Versuch die Datei zu öffnen

Ein Versuch diese Datei zu öffnen scheint zu einem Fehler zu führen, da *Word* eine Fehlermeldung ausgibt, in der es heißt, dass der Inhalt Probleme verursacht. Wird diese Fehlermeldung mit *OK* bestätigt erscheint eine weitere Meldung, dass der Inhalt wiederhergestellt werden kann. Wird diese Meldung mit *Ja* bestätigt zeigt *Word* den ungeschützten Inhalt der Datei an. Der Benutzer kann nun den Inhalt des Dokuments kopieren, da der Schutz vollständig entfernt worden ist.

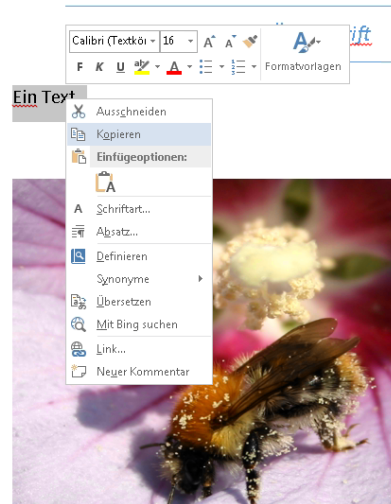


Abbildung 4.5: entschlüsselter Inhalt mit aktivierter Kopieren-Funktion in Word

4.4 Struktur des Word-Dokuments

Da die Anwendung verschiedene Bereiche der verschlüsselten Datei einlesen muss, ist es sinnvoll, das verschlüsselte Dokument zunächst einmal in seine XML-Struktur zu zerlegen. Die XML-Struktur kann beispielsweise mit einem Programm wie *7zip* angezeigt werden. In den folgenden Unterkapiteln wird dann der Aufbau der für den Angriff relevanten XML-Files näher erläutert.

Name	Größe	Gepackt...	Ordner	Dateien
[6]DataSpaces	39 056	39 232	3	4
EncryptedPackage	660 280	660 480		
[5]DocumentSummaryInformation	292	320		
[5]SummaryInformation	416	448		

Abbildung 4.6: XML-Dateien des verschlüsselten Word-Dokuments

4.4.1 [6]Primary

[6]Primary befindet sich im Unterverzeichnis [6]DataSpaces\TransformInfo\DRMEncryptedTransform. In dieser Datei ist die Lizenz enthalten, somit ist [6]Primary immer vorzufinden, wenn ein Word-Dokument mit RMS geschützt worden ist. Aus der Abbildung ist erkennbar, dass sich am Anfang der Datei Metadaten befinden, genauer gesagt befinden sich diese Metadaten immer in den ersten 172 Bytes der Datei. Die nächsten vier Bytes (0xF2, 0x96, 0x00 und 0x00) geben zusammen die Anzahl der Bytes der Lizenz an. Nun folgt die eigentliche Lizenz, welche im Fall von *vonErikaAnMax.docx* bis zum drittletzten Byte der Datei geht.

Nach der Lizenz folgen nur noch zwei Nullbytes. Warum sich diese Nullbytes dort befinden, konnte nicht herausgefunden werden. In allen während dieser Arbeit analysierten Word-Dokumente, waren am Ende von [6]Primary null bis drei Nullbytes vorhanden.

```

0x0000: 58 00 00 00 01 00 00 00 4C 00 00 00 7B 00 43 00 X.....L...{.C.
0x0010: 37 00 33 00 44 00 46 00 41 00 43 00 44 00 2D 00 7.3.D.F.A.C.D.-.
0x0020: 30 00 36 00 31 00 46 00 2D 00 34 00 33 00 42 00 0.6.1.F.-.4.3.B.
0x0030: 30 00 2D 00 38 00 42 00 36 00 34 00 2D 00 30 00 0.-.8.B.6.4.-.0.
0x0040: 43 00 36 00 32 00 30 00 44 00 32 00 41 00 38 00 C.6.2.0.D.2.A.8.
0x0050: 42 00 35 00 30 00 7D 00 3E 00 00 00 4D 00 69 00 B.5.0.}.>...M.i.
0x0060: 63 00 72 00 6F 00 73 00 6F 00 66 00 74 00 2E 00 c.r.o.s.o.f.t...
0x0070: 4D 00 65 00 74 00 61 00 64 00 61 00 74 00 61 00 M.e.t.a.d.a.t.a.
0x0080: 2E 00 44 00 52 00 4D 00 54 00 72 00 61 00 6E 00 ..D.R.M.T.r.a.a.n.
0x0090: 73 00 66 00 6F 00 72 00 6D 00 00 00 01 00 00 00 s.f.o.r.m.....
0x00A0: 01 00 00 00 01 00 00 00 04 00 00 00 F2 96 00 00 .....ð!..
0x00B0: 3C 3F 78 6D 6C 20 76 65 72 73 69 6F 6E 3D 22 31 <?xml version="1
0x00C0: 2E 30 22 3F 3E 3C 58 72 4D 4C 20 76 65 72 73 69 .0"?<XrML versi
0x00D0: 6F 6E 3D 22 31 2E 32 22 20 78 6D 6C 6E 73 3D 22 on="1.2" xmlns="
0x00E0: 22 3E 3C 42 4F 44 59 20 74 79 70 65 3D 22 4D 69 "><BODY type="Mi
0x00F0: 63 72 6F 73 6F 66 74 20 52 69 67 68 74 73 20 4C crosoft Rights L
0x0100: 61 62 65 6C 22 20 76 65 72 73 69 6F 6E 3D 22 33 abel" version="3
0x0110: 2E 30 22 3E 3C 49 53 53 55 45 44 54 49 4D 45 3E .0"><ISSUETIME>
0x0120: 32 30 31 36 2D 30 32 2D 32 38 54 32 30 3A 33 37 2016-02-28T20:37

```

Abbildung 4.7: Ausschnitt von [6]Primary im Hex-Editor

4.4.2 EncryptedPackage

In der *EncryptedPackage*-Datei befindet sich der verschlüsselte Inhalt. Die ersten acht Bytes geben jedoch Auskunft darüber, wie viele Bytes der Klartext enthält. Der verschlüsselte Inhalt beginnt bei Byte neun und geht bis zum Ende von EncryptedPackage. Der Inhalt ist immer mit AES-128 verschlüsselt, wobei der Schlüssel der Content-Key aus der Use-Licence ist. Als Betriebsmodus wird *Electronic Code Book* mit 16 Byte großen Blöcken verwendet, wobei der Initialisierungsvektor null ist.[11]

```

0x00000: 2F 13 0A 00 00 00 00 00 D7 AB B5 2E E8 1E D7 29 /.....x«µ.è.×)
0x00010: D2 5F 0F 85 E8 4E 90 AC 3F 8B AF 2E CC 3D 2C 23 ò_.!èN ~?|_.!=#
0x00020: 39 C8 E5 2E 1C 17 E4 1A 3F 41 F2 BB EF 66 BF 10 9Eá...ä.?Að»if¿.
0x00030: 70 78 FD 87 FA 9B 69 4B 1D DC E3 35 84 18 3F 34 pxýú!iK.Üð5!.,?4
0x00040: 63 F2 94 48 AC 1E A7 F4 DE 7D 14 F2 61 4E CA 84 cò!H~.SòP}.òaNÉ!
0x00050: 22 41 12 27 18 40 9D 50 DE 7D 14 F2 61 4E CA 84 "A.'.@ Pp}.òaNÉ!
0x00060: 22 41 12 27 18 40 9D 50 DE 7D 14 F2 61 4E CA 84 "A.'.@ Pp}.òaNÉ!
0x00070: 22 41 12 27 18 40 9D 50 DE 7D 14 F2 61 4E CA 84 "A.'.@ Pp}.òaNÉ!
0x00080: 22 41 12 27 18 40 9D 50 DE 7D 14 F2 61 4E CA 84 "A.'.@ Pp}.òaNÉ!
0x00090: 22 41 12 27 18 40 9D 50 DE 7D 14 F2 61 4E CA 84 "A.'.@ Pp}.òaNÉ!
0x000A0: 22 41 12 27 18 40 9D 50 DE 7D 14 F2 61 4E CA 84 "A.'.@ Pp}.òaNÉ!
0x000B0: 22 41 12 27 18 40 9D 50 DE 7D 14 F2 61 4E CA 84 "A.'.@ Pp}.òaNÉ!
0x000C0: 22 41 12 27 18 40 9D 50 DE 7D 14 F2 61 4E CA 84 "A.'.@ Pp}.òaNÉ!
0x000D0: 22 41 12 27 18 40 9D 50 DE 7D 14 F2 61 4E CA 84 "A.'.@ Pp}.òaNÉ!
0x000E0: 22 41 12 27 18 40 9D 50 DE 7D 14 F2 61 4E CA 84 "A.'.@ Pp}.òaNÉ!

```

Abbildung 4.8: Ausschnitt von EncryptedPackage im Hex-Editor

4.5 Implementierung

In diesem Unterkapitel werden alle relevanten Codeabschnitte der in Visual C++ erstellten Anwendung vorgestellt, um zu verstehen, wie der Angriff funktioniert. Der vollständige Code inklusive des Visual Studio Projekts ist auf der zu dieser Bachelorarbeit zugehörigen CD zu finden. Prinzipiell liest die Anwendung zunächst die notwendigen Bytes aus [6]Primary ein. Diese Bytes werden in ein geeignetes Format gebracht, um daraufhin eine Struktur zu initialisieren, welche gebraucht wird, um den Content-Key auslesen zu können. Als nächstes werden die notwendigen Bytes aus EncryptedPackage ausgelesen, woraufhin der Inhalt mit der *IpcDecrypt()*-Funktion entschlüsselt und in eine Datei geschrieben wird.

```
int __cdecl main()
{
    IpcInitialize(); // notwendig, um Funktionen des RMS SDK 2.1 zu benutzen

    std::streampos size;
    char * license = new char[50000]; // Puffer fuer die Lizenz, die ausgelesen wird
    char xml_size_byte1; // 1. Byte, welches die Laenge der Lizenz beschreibt
    char xml_size_byte2; // 2. Byte, welches die Laenge der Lizenz beschreibt
    char content_size_byte1; // 1. Byte, welches die Laenge des Klartextes beschreibt
    char content_size_byte2; // 2. Byte, welches die Laenge des Klartextes beschreibt
    char content_size_byte3; // 3. Byte, welches die Laenge des Klartextes beschreibt
    char content_size_byte4; // 4. Byte, welches die Laenge des Klartextes beschreibt
    char byte_read; // Puffer, in welchen nacheinander die Bytes gelesen werden
    char lastbyte;
    int xml_size; // Laenge der Lizenz
    int i, nullbytecounter = 0;
    array<Byte> ^contentBytes; // Byte-Array, in welchen die zu entschlüsselnden Bytes
    // geschrieben werden
    // ...
}
```

Zunächst wird die *IpcInitialize()* – Funktion aufgerufen. Diese ist notwendig, damit überhaupt andere Funktionen des RMS SDK 2.1 verwendet werden können. Sie lokalisiert und lädt die *Msipc.dll*, diese ist die Programmbibliothek des RMS Clients. Es folgen nun einige Variablendeklarationen, welche im späteren Verlauf der *main()*-Methode Verwendung finden, daher werden sie erst an den entsprechenden Stellen näher erläutert. Im nächsten Abschnitt wird die Lizenz aus [6]Primary eingelesen.

```
// Pfad der Datei, welche die Lizenz enthaelt
std::ifstream file("[6]DataSpaces\\TransformInfo\\DRMEncryptedTransform\\[6]Primary",
    std::ios::in, std::ios::binary);
if (file.is_open())
{
    // Metadaten ueberspringen
    file.seekg(172);
    // Laenge der Lizenz einlesen (2Bytes)
}
```

```
file.read(&xml_size_byte1, 1);
file.read(&xml_size_byte2, 1);

file.seekg(0, std::ios::end);
xml_size = file.tellg();

// Die Datei mit der Lizenz kann manchmal Nullbytes enthalten
file.seekg(xml_size - 1);
file.read(&lastbyte, 1);

// Schleife, welche die letzten 10Bytes auf Nullbytes ueberprueft
for (int t = 2; t < 12; t++)
{
    if (lastbyte != '>')
    {
        nullbytecounter++;
        file.seekg(xml_size - t);
        file.read(&lastbyte, 1);
    }
    else
    {
        break;
    }
}

file.seekg(0, std::ios::end);
xml_size = file.tellg();
xml_size = xml_size - 176 - nullbytecounter;

// Beginn der eigentlichen Lizenz
file.seekg(176);
file.read(license, xml_size);
license[xml_size] = '\0';

file.close();

xml_size_byte1 = xml_size_byte1 + '\4';

initLicense(xml_size, xml_size_byte1, xml_size_byte2, license);
}
```

Am Anfang dieser Datei stehen immer Metadaten, welche für den Angriff nicht relevant sind. Diese Metadaten sind immer 172Byte groß, daher wird mit *file.seekg(172)* der Datei-Zeiger auf 172 gesetzt, damit Bytes, die von jetzt an gelesen werden, genau ab dieser Position ausgelesen werden. Mit *file.read(&xml_size_byte1, 1)* und *file.read(&xml_size_byte2, 1)* werden nun zwei Bytes eingelesen, deren Wert gibt an, wie lang die komplette Lizenz ist. Diese beiden Bytes werden im späteren Verlauf des Programms benötigt, daher werden sie hier eingelesen. Um die komplette Lizenz einzulesen, muss die Länge der Lizenz als dezimaler Wert

bekannt sein. Da sich das Konvertieren der beiden eingelesenen Bytes in einen dezimalen Wert als umständlich erwiesen hat, wird nun die komplette Dateilänge bestimmt, indem mit `file.seekg(0, std::ios::end)` der Dateizeiger auf das Ende der Datei gesetzt wird und mit `file.tellg()` nun die Anzahl der Bytes in [6]Primary bestimmt wird. Aus unbekanntem Grund kann die Datei manchmal am Ende Nullbytes enthalten, diese befinden sich hinter der Lizenz und müssen daher ignoriert werden; daher folgt im Code nun eine Schleife, welche die letzten zehn Bytes der Datei auf Nullbytes überprüft; wird ein Nullbyte gefunden, wird `nullbytecounter` um eins erhöht. In der Praxis wurden in [6]Primary allerdings nie mehr als drei Nullbytes gefunden. Mit `xml_size = xml_size - 176 - nullbytecounter` wird nun der dezimale Wert bestimmt, der die Länge der Lizenz beschreibt, `xml_size` enthielt vorher die Byteanzahl der kompletten Datei, subtrahiert werden die 176 Byte vor der eigentlichen Lizenz, die beiden Bytes, welche die Länge der Lizenz beschreiben, sowie die Anzahl der zuvor bestimmten Nullbytes. Der Dateizeiger wird auf den Wert 176 gesetzt, da hier die eigentlich Lizenz beginnt, nun kann mit `file.read(license, xml_size)` die Lizenz in ein Array eingelesen werden. Abschließend wird die Datei wieder geschlossen und es wird der hexadezimale Wert, welcher die Länge der Lizenz beschreibt, um vier erhöht, da in der aufgerufenen `initLicense`-Funktion, welche im Folgenden betrachtet wird, weitere Bytes gelesen werden.

```
System::Void initLicense(int xml_size, char xml_size_byte1, char xml_size_byte2, char
    license[])
{
    byte byte1 = 0xEF;
    byte byte2 = 0xBB;
    byte byte3 = 0xBF;
    array<Byte> ^licenseLengthBytes;
    array<Byte> ^licenseBytes;
    licenseLengthBytes = gcnew array<Byte>(4);
    licenseBytes = gcnew array<Byte>(xml_size + 4);

    licenseLengthBytes[0] = xml_size_byte1;
    licenseLengthBytes[1] = xml_size_byte2;
    licenseLengthBytes[2] = '\0';
    licenseLengthBytes[3] = '\0';
    Marshal::Copy(licenseLengthBytes, 0, (IntPtr)(&pLicense->cbBuffer), 4);
    licenseBytes[0] = byte1;
    licenseBytes[1] = byte2;
    licenseBytes[2] = byte3;

    for (int k = 0; k < xml_size; k++)
    {
        licenseBytes[k + 3] = license[k];
    }
    pLicense->pvBuffer = (LPVOID)new unsigned char[xml_size + 4];
    Marshal::Copy(licenseBytes, 0, (IntPtr)(pLicense->pvBuffer), pLicense->cbBuffer);
}
```

Prinzipiell wird in dieser Methode die eine Instanz der IPC_BUFFER Struktur namens `pLicense` initiali-

siert, wobei *pLicense* global definiert ist. Diese wird benötigt, um im späteren Verlauf des Codes den Content-Key aus der Lizenz auszulesen.

In der `IPC_BUFFER` Struktur sind zwei Puffer zu initialisieren. Es werden zunächst drei Bytes *byte1* bis

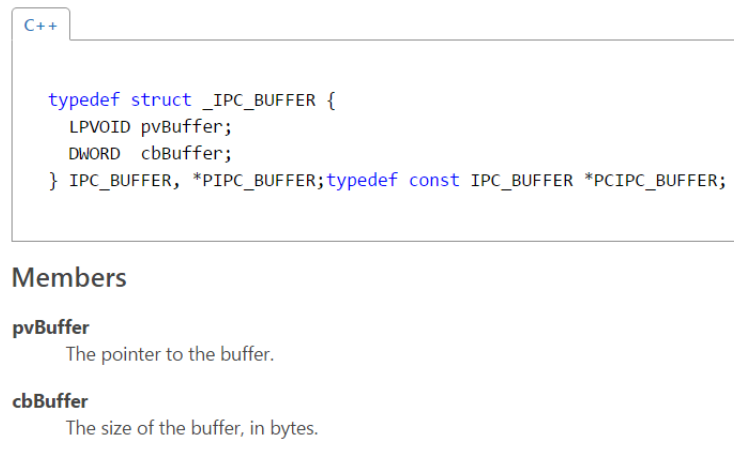


Abbildung 4.9: Syntax der `IPC_BUFFER` Struktur
[12]

byte3 definiert, danach werden zwei Byte-Arrays erzeugt. In das erste Array *licenseLengthBytes* werden die beiden Bytes, welche die Länge der Lizenz beschreiben und zwei Nullbytes geschrieben. Nun kann *cbBuffer* von *pLicense* initialisiert werden, indem der Inhalt von *licenseLengthBytes* in diesen Puffer kopiert wird. Die zuvor definierten Bytes werden nun inklusive der Lizenz-Bytes in das *licenseBytes*-Array geschrieben, damit schließlich auch *pvBuffer* initialisiert werden kann. Warum *byte1* bis *byte3* mit diesen speziellen Werten benötigt wird, konnte nicht herausgefunden werden. Dies wurde herausgefunden, indem eine beliebige Datei mit *IPCNotepad*[13], einer Bespielanwendung von Microsoft, welche Funktionen des RMS SDK 2.1 verwendet, verschlüsselt wurde. Wurde diese Datei im Hex-Editor betrachtet, so waren in jedem Fall genau diese Bytes vor der Lizenz zu finden. Es hat sich herausgestellt, dass genau diese Bytes wieder bei der Entschlüsselung benötigt werden, indem sie beim initialisieren von *pvBuffer* verwendet werden. Als nächstes wird der Codeausschnitt betrachtet, in welchem die zu entschlüsselnden Bytes in *EncryptedPackage* eingelesen werden.

```

// Datei, die die Laenge des Klartextes und den verschluesselten Inhalt enthaelt
char* path = "EncryptedPackage";
std::ifstream file2(path, std::ios::binary);
if (file2.is_open())
{

    file2.seekg(0, std::ios::end);
    size = file2.tellg();
    file2.seekg(0, std::ios::beg);
    file2.read(&content_size_byte1, 1);
    file2.read(&content_size_byte2, 1);
    file2.read(&content_size_byte3, 1);

```

```

file2.read(&content_size_byte4, 1);
file2.seekg(8);
contentBytes = gnew array<Byte>(4 + (int) size);

content_size_byte1 = content_size_byte1 + '\4' + '\4' + '\4' + '\4' + '\4' + '\4';

contentBytes[0] = content_size_byte1;
contentBytes[1] = content_size_byte2;
contentBytes[2] = content_size_byte3;
contentBytes[3] = content_size_byte4;

for (i = 0; i <= (int) size - 9; i++)
{
    file2.read(&byte_read, 1);
    contentBytes[i + 4] = byte_read;
}
contentBytes[i + 5] = '\0';

// vorbereitete Bytes entschlüsseln
ReadKeyAndDecrypt(contentBytes);
delete[] license;
file2.close();
}

```

Die *EncryptedPackage* Datei wird zunächst geöffnet, da hier, wie in vorherigen Abschnitten beschrieben, der eigentliche verschlüsselte Inhalt enthalten ist. Unter *size* wird die Anzahl der Bytes abgespeichert, die in *EncryptedPackage* enthalten ist. Nachdem der Datei-Zeiger wieder auf den Anfang der Datei gesetzt wird, werden vier Bytes eingelesen. Diese Bytes bestimmen zusammen die Anzahl der Bytes des Klartextes, sie werden in den Variablen *content_size_byte1* bis *content_size_byte4* abgespeichert. Byte fünf bis Byte acht sind Nullbytes, da in der Praxis Dateien nicht so groß sind, dass diese Bytes ebenfalls belegt werden. Der Datei-Zeiger wird mit *file2.seekg(8)* so gesetzt, dass diese Nullbytes übersprungen werden und damit später die verschlüsselten Bytes eingelesen werden können, welche ab Byte neun beginnen. *contentBytes = gnew array<Byte>(4 + (int)size)* erzeugt nun ein Byte-Array mit benötigter Größe. In dieses Array werden nun alle benötigten Bytes geschrieben, welche später für die Entschlüsselung benötigt werden. Zuvor wird allerdings der Wert von *content_size_byte1* erhöht, da es beim Entwickeln des Programms vorkam, dass der Wert, der die Anzahl der Bytes des Klartextes beschreibt, für manche geschützte Word-Dokumente kleiner war als die tatsächliche Anzahl des Klartextes. Die Erhöhung dieses Wertes stellt also sicher, dass alle Bytes entschlüsseln werden. Allerdings kann es so auch vorkommen, dass so ein Block *zu viel* entschlüsseln wird. Für den Angriff ist dies aber nicht fatal, da alle Blöcke entschlüsselt werden, der letzte Block ist dann lediglich doppelt vorhanden. Den Wert sechs mal nacheinander jeweils um vier zu erhöhen, hat hierbei durch Ausprobieren zum Erfolg geführt. Für alle während dieser Bachelorarbeit untersuchten Word-Dokumente war dieser Wert ausreichend. Bevor die geöffnete Datei wieder geschlossen und das Array gelöscht wird, wird Das *contentBytes*-Array der *ReadKeyAndDecrypt*-Methode übergeben, welche im Folgenden erläutert

wird.

```
System::Void ReadKeyAndDecrypt(array<Byte> ^encryptedBytes)
{
    MemoryStream ^encryptedStream;
    pin_ptr<IPC_KEY_HANDLE> pContentKey;

    // Contentkey aus der Lizenz holen
    pContentKey = &contentKey;
    IpcGetKey(pLicense, 0, nullptr, nullptr, pContentKey);

    encryptedStream = gnew MemoryStream(encryptedBytes);
    readDecryptedContent(encryptedStream);

    encryptedStream->Close();
}
```

Die *ReadKeyAndDecrypt*-Methode liest zunächst den Content-Key aus der Lizenz, dafür benötigt wird *pLicense*, welche zuvor initialisiert wurde. Danach wird ein neuer *MemoryStream* namens *encryptedStream* erzeugt, dieser enthält das *encryptedBytes*-Array als Eingabe. Schließlich wird die *readDecryptedContent*-Methode aufgerufen, in welcher die Entschlüsselung stattfindet. Sobald die Methode durchlaufen ist, wird der *MemoryStream* wieder geschlossen. Die genaue Funktionsweise der *readDecryptedContent*-Methode¹ wird im Folgenden betrachtet:

```
System::Void readDecryptedContent(Stream ^encryptedStream)
{
    DWORD          cbReadRemaining;
    DWORD          cbRead;
    DWORD          cbOutputBuffer;
    DWORD          cbDecrypted;
    array<Byte>    ^readBuffer;
    array<Byte>    ^writeBuffer;
    pin_ptr<Byte>  pbReadBuffer;
    pin_ptr<unsigned char> pbWriteBuffer;
    int            cBlock;
    std::ofstream output_stream;
    std::string    filename;
    filename = "decrypted.docx";

    output_stream.open(filename.c_str(), std::ios_base::binary);

    try
    {
        readBuffer = gnew array<Byte>(16);
```

¹diese Methode ist angelehnt an die *readBody*()-Methode aus *IPCNotepad*[13], sie wurde hier entsprechend angepasst


```
pbReadBuffer = &readBuffer[0];

cbOutputBuffer = 16;
writeBuffer = gcnew array<Byte>(cbOutputBuffer);
pbWriteBuffer = &writeBuffer[0];

cbRead = encryptedStream->Read(readBuffer, 0, 4);

Marshal::Copy(readBuffer, 0, (IntPtr)&cbDecrypted, 4);

cBlock = 0;
cbReadRemaining = (DWORD)(encryptedStream->Length - encryptedStream->Position);

while (cbReadRemaining > 16)
{
    cbRead = 0;
    encryptedStream->Read(readBuffer, 0, 16);
    IpcDecrypt(contentKey,
        cBlock,
        false,
        pbReadBuffer,
        16,
        pbWriteBuffer,
        cbOutputBuffer,
        &cbRead);

    for (int i = 0; i < 16; i++) // die Bloecke sind immer 16Byte gro
    {
        output_stream << pbWriteBuffer[i];
    }

    cBlock++;
    cbReadRemaining -= cbRead;
}

cbRead = 0;
encryptedStream->Read(readBuffer, 0, cbReadRemaining);
IpcDecrypt(contentKey,
    cBlock,
    true,
    pbReadBuffer,
    cbReadRemaining,
    pbWriteBuffer,
    cbOutputBuffer,
    &cbRead);

// letzten Block bytewise in eine Datei schreiben
```

```
for (int i = 0; i < 16; i++)
{
    output_stream << pbWriteBuffer[i];
}

}
finally
{
    readBuffer = nullptr;
    writeBuffer = nullptr;
    output_stream.close();
}
}
```

Nach einigen Variablendeklarationen wird zunächst eine Datei namens *decrypted.docx* angelegt und geöffnet; in diese Datei werden später die entschlüsselten Bytes geschrieben. Danach werden vier Bytes aus *encryptedStream* ausgelesen, welche in *readBuffer* abgespeichert werden, es handelt sich bei den ausgelesenen Bytes um die Anzahl der Bytes des Klartextes. Mittels $cbReadRemaining = (DWORD)(encryptedStream->Length - encryptedStream->Position)$ wird nun die Anzahl an Bytes bestimmt, die entschlüsselt werden müssen, indem die Anzahl der Bytes in *encryptedStream* von der aktuellen Position dieses Stream subtrahiert wird. Nun werden in einer Schleife die 16Byte großen Blöcke nach und nach entschlüsselt, indem jeweils 16Byte aus *encryptedStream* eingelesen werden, woraufhin diese eingelesenen Bytes der *IpcDecrypt()*-Funktion übergeben werden. Für den Angriff relevant ist nach erfolgreichen Ausführen der *IpcDecrypt()*-Funktion der Inhalt von *pbWriteBuffer*, da in diesen Puffer der entschlüsselte Klartext gespeichert wird. Für jeden Block wird also eine Schleife durchlaufen, in welcher der Inhalt von *pbWriteBuffer* in die zuvor geöffnete Datei namens *decrypted.docx* geschrieben wird. Die Schleife durchläuft alle 16Byte des entsprechenden Blocks einzeln, da die entschlüsselten Bytes Nullbytes sein können. Ein einfaches Ausgeben mittels $output_stream \ll pbWriteBuffer$ würde nicht genügen, da der \ll Operator bei Nullbytes terminieren würde und so nur die Bytes bis zum ersten Nullbytes ausgeben würde. Nachdem die Schleife durchlaufen ist, wird der letzte Block eingelesen. Dieser letzte Block ist 16Byte groß oder kleiner als 16Byte und wurde in der Schleife nicht behandelt. Auch auf diesen Block wird die *IpcDecrypt()*-Funktion angewendet, woraufhin der Inhalt von *pbWriteBuffer* wieder byteweise in die Datei geschrieben wird. Schlussendlich werden die initialisierten Arrays wieder auf null gesetzt und die Datei wird geschlossen. Die Datei *decrypted.docx* repräsentiert nun das ungeschützte Word-Dokument.

4.6 Fazit zum ersten Angriff

In diesem Kapitel wurde eine in der Praxis existierende Schwachstelle in Microsofts ERM-System analysiert. Es wurde gezeigt, dass ein Benutzer in einer Domäne Vollzugriff auf ein mit RMS geschütztes Word-Dokument erhalten kann, wenn dieser Benutzer eigentlich nur das Leserecht für das Dokument besitzt. In der Praxis kann dies für Unternehmen fatale Folgen haben, wenn Mitarbeiter geschützte firmeninter-

ne Dokumente entschlüsseln können. Denn wenn Mitarbeiter das Unternehmen verlassen, können sie die entschlüsselten Dokumente auf dem Schwarzmarkt oder an die Konkurrenz verkaufen. In dem hier vorgestellten Beispiel ging es um ein Word-Dokument; da jedoch andere Microsoft Office Formate, wie *.xlsx* für *Microsoft Excel* oder *.pptx* für *Microsoft Powerpoint* ebenfalls Open Office XML Formate sind, ist die hier vorgestellte Anwendung bei diesen Formaten eins zu eins anwendbar. Es müssen keine Änderungen im Quelltext vorgenommen werden, da die XML-Dateien immer den gleichen Aufbau haben. Es müsste lediglich das Format der Ausgabedatei geändert werden, wobei das Dateiformat streng genommen auch nach Ausführen des Programms geändert werden kann und die Datei trotzdem geöffnet werden könnte. Theoretisch sind jedoch nicht nur Microsoft Office Dokumente angreifbar, sondern alle mit RMS geschützten Dateien, welche zB mit der RMS Freigabeanwendung[14] geschützt worden sind. In diesen Fällen müsste das Programm jedoch angepasst werden, da die Lizenz und die verschlüsselten Bytes aus der geschützten Datei selbst eingelesen werden müssten, da es sich beispielsweise bei einer *.pdf*-Datei nicht um ein Open Office XML Format handelt.

5 Weitere Angriffsszenarien

In diesem Kapitel werden zwei weitere Angriffsszenarien betrachtet. Es soll untersucht werden, ob diese ebenfalls ausnutzbar sind. Da es sich um eine ähnliche Vorgehensweise handelt, werden beide in einem Kapitel zusammen behandelt. Am Ende des Kapitels wird ein Fazit gezogen.

Angriffsszenario 2:

Im zweiten Angriffsszenario ist das Ziel ein Dokument im Namen eines anderen Benutzers aus der Domäne zu erstellen. Da bei der Erstellung eines Dokuments immer eine PL erstellt wird, welche mit dem privaten Schlüssel des CLC signiert ist, ist es für diesen Angriff notwendig den privaten Schlüssel des CLCs des Opfers zu übernehmen. Um diesen Angriff durchzuführen wird angenommen, dass der Angreifer für einen kurzen Zeitraum physischen Zugriff auf den Computer des Opfers im eingeloggt Zustand hat, sodass er dort ein Programm ausführen kann, welches das CLC bzw. dessen privaten Schlüssel und gegebenenfalls weitere benötigte Dateien aus dem System extrahiert.

Angriffsszenario 3:

Beim nächsten Angriff wird versucht, Zugriff auf ein mit RMS geschütztes Dokument zu erlangen, auf welches ein Benutzer in einer Domäne nicht einmal das Leserecht besitzt. Es muss versucht werden, an die Rechte eines anderen Benutzers aus der Domäne für dieses Dokument zu gelangen. Die Angriffs-idee besteht also darin, an den privaten Schlüssel des RACs eines Nutzers, welcher für das Dokument Rechte besitzt, zu gelangen, da mit dem RAC der Content-Key aus der Use-License extrahiert werden kann. Auch in diesem Szenario wird angenommen, dass der Angreifer für einen kurzen Augenblick physischen Zugang auf den Computer seiner Opfers hat, sodass er das RAC bzw. dessen privaten Schlüssel kopieren kann.

Die beiden beschriebenen Angriffe sind sehr ähnlich, da es jeweils darum geht zu versuchen, an die Benutzerzertifikate des Opfers bzw. an die privaten Schlüssel dieser Zertifikate zu gelangen.

5.1 Übernehmen der Benutzeridentität des Opfers

Als erstes wurde getestet, alle Zertifikate (CLC, RAC, SPC) sowie den mit der DPAPI geschützten privaten Schlüssel des SPCs von der Maschine des Opfers zu kopieren und mit denen des Angreifers zu ersetzen, um somit die Benutzeridentität des Opfers zu übernehmen. Das Übernehmen der Benutzeridentität hat das Ziel, das somit beide zuvor beschriebenen Angriffe durchgeführt werden könnten, da sich der RMS Client um die Entschlüsselung des privaten Schlüssels des SPCs, des RACs und des CLCs kümmert. Das SPC muss

mitkopiert werden, da die Maschinenidentität an die Benutzeridentität gebunden ist (der private Schlüssel des RAC ist mit dem öffentlichen Schlüssel des SPC verschlüsselt). Da das SPC an die Hardware gebunden ist[15], wurde dieser Angriff zunächst auf dem gleichen Client getestet, indem die Zertifikate und der geschützte private Schlüssel des SPC von Erika Musterfrau mit den Zertifikaten und dem geschützten privaten Schlüssel von Max Mustermann ersetzt wurde.

Beobachtung:

Nachdem die Zertifikate und der geschützte private Schlüssel des SPC ersetzt worden waren, wurde getestet, ob ein Dokument geöffnet werden konnte, auf welches Max Mustermann nicht einmal das Leserecht hatte, Erika Musterfrau allerdings schon. Das Ergebnis war, dass Word eine Fehlermeldung ausgab, in der es hieß, dass dem Benutzer nicht die entsprechenden Rechte eingeräumt wurden. Des Weiteren war zu beobachten, dass die zuvor ersetzten Zertifikate gelöscht und neue angelegt wurden. Auch wurden die ersetzten Registry Einträge, welche für den privaten Schlüssel des SPC verantwortlich sind, erneuert. Der RMS Client hat somit erkannt, dass die Benutzer- und die Maschinenidentität verändert worden sind.

Erklärung:

Beim Öffnen der geschützten Datei hat der RMS Client versucht den geschützten privaten Schlüssel des SPC zu entschlüsseln. Dieser lag geschützt in der Registry und konnte nicht erfolgreich entschlüsselt werden. Der Grund ist die Funktionsweise der DPAPI. Wird ein Klartext mit der DPAPI verschlüsselt, so hängt der entstandene Data-Blob vom Passwort des jeweiligen Benutzers ab. Da die Verschlüsselung auf der Maschine des Opfers stattfand und die Entschlüsselung auf der Maschine des Angreifers, kann der Angriff so nicht funktionieren, denn beide Nutzer verwenden ein anderes Passwort.

5.2 Durchführung der Angriffe, wenn privater Schlüssel des SPC des Opfers bekannt ist

Das Übernehmen der Benutzeridentität hat wie zuvor beschrieben nicht funktioniert, um die Angriffe durchzuführen, da der RMS Client aufgrund des nicht bekannten Passworthashes den geschützten privaten Schlüssel des SPCs nicht entschlüsseln kann. Eine Idee, um die Angriffe dennoch durchführen zu können, besteht darin, den geschützten privaten Schlüssel des SPC selbst zu entschlüsseln. Dies wäre theoretisch möglich, wenn auf der Maschine des Opfers ein vom Angreifer erstelltes Programm ausgeführt würde, welches die Entschlüsselungsfunktion der DPAPI benutzt, um damit den geschützten privaten Schlüssel des SPC zu entschlüsseln, welcher daraufhin an den Angreifer gesendet wird. Der Angreifer kann dann folgendermaßen vorgehen:

In Angriffsszenario 2:

Wenn der private Schlüssel des SPCs also bekannt ist, kann damit auch der private Schlüssel des RACs und des CLCs entschlüsselt werden, so wie es an anderen Stellen dieser Arbeit beschrieben worden ist. Das

Erstellen eines mit RMS geschützten Dokuments läuft offline ab. Somit kann der Angreifer ein beliebiges Dokument erstellen und mit RMS schützen. Die PL und die UL, welche der Autor beim Schützen einer Datei für sich selbst ausstellt könne nun manipuliert werden, indem der Name des Autors in der UL und der PL verändert wird. Die Signatur über die UL wird daraufhin mit dem privaten Schlüssel des RAC und die Signatur über die PL mit dem privaten Schlüssel des CLC neu erstellt. Die Manipulation der UL ist deshalb notwendig, da beim Öffnen einer mit RMS geschützten Datei eine Verifikation dieser beiden Lizenzen stattfindet.[1]

In Angriffsszenario 3:

Für das dritte Angriffsszenario wird nur der private Schlüssel des RAC benötigt. Dieser kann entschlüsselt werden, wenn der private Schlüssel des SPC bekannt ist. Mit dem privaten Schlüssel des RAC des Opfers kann der Angreifer aus der UL den Content-Key extrahieren. Mit diesem Content-Key kann der Angreifer Vollzugriff auf das Dokument erlangen, so wie es im letzten Kapitel beschrieben wurde. Voraussetzung wäre hier, dass der Benutzer, dessen privater Schlüssel kopiert wird, der Autor der Datei ist, da die UL des Autors immer im geschützten Dokument selbst steht.

5.3 Extrahieren des privaten Schlüssels des SPC

Im Folgenden wird untersucht, ob es möglich ist den privaten Schlüssel des SPC zu extrahieren, um die Angriffe, so wie zuvor beschrieben, durchzuführen. Bei der Erstellung des SPC wird laut einer Anleitung von Microsoft der private Schlüssel des SPC sowohl mit der DPAPI, als auch mit RSAVault geschützt.[15] Der geschützte private Schlüssel wird daraufhin in der Registry des jeweiligen Benutzers abgelegt. Obwohl im Internet keine Beschreibung von RSAVault zu finden ist, muss dies bei der Entschlüsselung des privaten Schlüssels berücksichtigt werden.

5.3.1 Analyse des geschützten privaten Schlüssels des SPC

HKEY_CURRENT_USER\Software\Classes\LocalSettings\Software\Microsoft\uDRM ist laut eine Veröffentlichung von Microsoft der Pfad des geschützten privaten Schlüssels des SPCs.[15] Dieser Pfad ist in der Registry allerdings nicht vorhanden, was daran liegt dass sich die Anleitung auf eine veraltete Version des AD RMS Clients bezieht. In dieser Arbeit ist AD RMS Client 2.1 verwendet worden, sodass der Pfad HKEY_CURRENT_USER\Software\Classes\LocalSettings\Software\Microsoft\MSIPC sein muss. Dort zu finden sind vier Binäreinträge, wie die folgende Abbildung zeigt.

Name	Typ	Daten
(Standard)	REG_SZ	(Wert nicht festgelegt)
AC	REG_BINARY	71 00 33 00 d8 58 5e 42 cc 9a 0c d6 66 c3 7a db 87 a2 26 42 a2 17 1b 43 c
DefaultIdentityServer	REG_SZ	dc1.test-domain.com
MSIPP-MK	REG_BINARY	01 00 00 00 d0 8c 9d df 01 15 d1 11 8c 7a 00 c0 4f c2 97 eb 01 00 00 00 8f
MSIPP-SK	REG_BINARY	01 00 00 00 d0 8c 9d df 01 15 d1 11 8c 7a 00 c0 4f c2 97 eb 01 00 00 00 8f
RK	REG_BINARY	a1 f8 b2 0e a1 b0 60 9b c9 c2 1e 9b 63 a4 3e 92 16 8f 76 74 95 43 ea 96 24

Abbildung 5.1: vier Registry-Einträge

Die Binäreinträge haben folgende Größen:

AC 4096 Byte

RK 4096 Byte

MSIPP-MK 1324 Byte

MSIPP-SK 276 Byte

Kopiert man die Binäreinträge in eine Datei und öffnet diese mit einem Hex-Editor, so fällt auf, dass MSIPP-MK und MSIPP-SK an die Strukturen von Data-Blobs der DPAPI erinnern, weshalb als nächstes versucht wird, diese Data-Blobs zu entschlüsseln. AC und RK haben keine erkennbare Struktur. Vom Namen her deuten MSIPP-MK und MSIPP-SK auf die von der DPAPI generierten Schlüssel hin (Master-Key und Session-Key). Ob dies tatsächlich der Fall ist kann nicht gesagt werden; dagegen spricht beispielsweise, dass der Session-Key normalerweise gar nicht abgespeichert wird.

5.4 Verwenden der DPAPI zum Entschlüsseln der Registry-Einträge

Im Folgenden wird ein in Visual C++ entwickeltes Programm¹ vorgestellt, welches den Inhalt von einer ausgewählten Datei ausliest und diesen der DPAPI-Entschlüsselungsfunktion übergibt. Die Ausgabe dieser Funktion (also der Klartext) wird in einer Datei namens [?] abgespeichert. Mit diesem Programm wird daraufhin nacheinander versucht, die Binäreinträge zu entschlüsseln. Der Einfachheit halber wurden für die Analyse die Registry-Einträge in Dateien kopiert.

```
void main()
{
    DATA_BLOB DataOut;
    DATA_BLOB DataVerify;
    BYTE pbDataOutput[5000];
    DWORD cbDataOutput;
```

¹Das Programm ist auch auf der zu dieser Arbeit zugehörigen CD zu finden

```
std::ofstream output_file;
std::string filename;

char readbyte;
int file_length;

// DATA BLOB einlesen
std::ifstream file("BLOB", std::ios::in, std::ios::binary);
if (file.is_open())
{
    file.seekg(0, std::ios::end);
    file_length = (int)file.tellg();
    file.seekg(0, std::ios::beg);
    printf("File length: %d\n", file_length);
    for (int i = 0; i <= file_length; i++)
    {
        file.read(&readbyte, 1);
        pbDataOutput[i] = readbyte;
    }
    pbDataOutput[file_length] = '\0';
    cbDataOutput = file_length;
    DataOut.pbData = pbDataOutput;
    DataOut.cbData = cbDataOutput;
    file.close();
}
else
{
    printf("Could not open file!\n");
}

// DPAPI Entschluesslungsfunktion aufrufen

if (CryptUnprotectData(
    &DataOut, // zu entschlüsselnder Data-Blob
    NULL, // Beschreibung des Data-Blobs
    NULL, // Sekundaere Entropie
    NULL, // Reserviert
    NULL, // Optionale PromptStruct
    NULL, // Optionale Flags
    &DataVerify)) // enthaelt die entschlüsselten Daten
{
    printf("The decrypted data is: %s\n", DataVerify.pbData);
    filename = "decrypted.txt";
    output_file.open(filename.c_str(), std::ios_base::binary);

    for (int i = 0; i < DataVerify.cbData; i++)
    {
```



```
        output_file << DataVerify.pbData[i];
    }
}
else
{
    std::cerr << "CryptUnprotectData() failed with error number: " << GetLastError() <<
        std::endl;
    exit(1);
}
}
```

Das Programm öffnet zunächst eine Datei namens *BLOB*. In dieser Datei muss der gewünschte zu entschlüsselnde Data-Blob stehen. Es wird daraufhin die Länge der Datei bestimmt, welche auch auf der Konsole ausgegeben wird. Danach wird byteweise der Inhalt der Datei ausgelesen und in *pbDataOutput* abgespeichert. Eine Struktur mit dem Namen *DataOut* vom Typ *DATA_BLOB* wird daraufhin initialisiert, indem in *pbData* die eingelesenen Bytes und in *cbData* die Länge der eingelesenen Bytes geschrieben werden. Die Datei wird daraufhin wieder geschlossen. Der Data-Blob kann nun der DPAPI-Entschlüsselungsfunktion *CryptUnprotectData()* übergeben werden. Sofern das Ausführen der Funktion erfolgreich war, wird der entschlüsselte Inhalt sowohl auf der Konsole ausgegeben als auch in eine Datei namens *decrypted.txt* geschrieben. Falls die Funktion fehlschlagen sollte wird die *GetLastError()*-Funktion aufgerufen, welche eine Fehlernummer der zuletzt fehlgeschlagenen Funktion ausgibt.

Beobachtung:

Für alle Binäreinträge liefert das Programm als Ausgabe: *CryptUnprotectData() failed with error number: 13*. Fehlercode Nummer 13 ist hierbei *ERROR_INVALID_DATA*[16]. Das Programm funktioniert mit selbsterstellten DPAPI-Blobs, welche mit der *CryptProtectData()*-Funktion erstellt werden und in eine Datei geschrieben werden, somit ist ein Implementierungsfehler auszuschließen.

Erklärung:

Das Fehlschlagen der *CryptUnprotectData()*-Funktion kann auf das Vorhandensein von *RSVault* hindeuten, über welches Microsoft keine Auskunft gibt. Hierbei könnten bestimmte Bytes des Data-Blobs verändert worden sein. Andererseits ist es möglich, dass bei der Verschlüsselung des privaten Schlüssels sekundäre Entropie verwendet worden ist. Sollte dies der Fall sein, muss für die Entschlüsselung die gleiche Entropie verwendet werden. Da diese Entropie irgendwo abgespeichert werden muss, ist es möglich, dass einer der vier Binäreinträge die sekundäre Entropie ist. Aufgrund dieser Annahme ist das Programm erweitert worden, sodass es zusätzlich zum Data-Blob eine sekundäre Entropie aus einer Datei einliest und diese bei der *CryptUnprotectData()*-Funktion verwendet. Der dritte Parameter der *CryptUnprotectData()*-Funktion muss dann von *NULL* auf Entropie gesetzt werden, da dieser Parameter für die optionale Entropie verantwortlich ist. Auch so meldet das Programm die Fehlermeldung 13.

5.5 Weitere Untersuchungen

Die *CryptProtectData()*- und die *CryptUnprotectData()*-Funktion haben, wie im Codeabschnitt zu erkennen, noch weitere Parameter. Parameter Nummer 6 soll in diesem Abschnitt noch näher untersucht werden, da dort FLAGS definiert werden können, welche bei der Entschlüsselung eine Rolle spielen. Beispielsweise kann ein mit dem *CRYPTPROTECT_SYSTEM* Flag geschützter Data-Blob nur entschlüsselt werden, wenn bei der Entschlüsselung dieses Flag wieder gesetzt wird. Der folgende Ausschnitt von MSIPP-SK im Hex-Editor zeigt allerdings, dass nach der von Passcape entdeckten Struktur eines Data-Blobs keines dieser Flags gesetzt ist:

```

0x000: 01 00 00 00 D0 8C 9D DF 01 15 D1 11 8C 7A 00 C0   ....DI B..N.Iz.Ä
0x010: 4F C2 97 EB 01 00 00 00 85 6B 94 D4 7A CF 23 43   OÄIe....IkIÖzi#C
0x020: AF 70 76 78 A3 D3 FC F7 00 00 00 04 00 00 00   pvxſÖü+.....
0x030: 20 00 00 00 03 66 00 00 C0 00 00 00 10 00 00   ....f..Ä.....
0x040: 23 50 59 CD 65 E4 BF EA 3A C7 44 25 C6 2A AB 50   #PYIeäçè:ÇD%#*«P
0x050: 00 00 00 00 04 80 00 00 A0 00 00 00 10 00 00   ....I.....
0x060: 92 50 5B D2 C2 67 50 73 1A 5A A8 9C FA E8 26 10   ^P[ÖÄgPs.Z"lIeS.
0x070: 88 00 00 00 60 2F 86 45 AF 36 A0 0C BE 6E 18 BA   I...^/IE76 .#n.9
0x080: E2 B1 09 C9 5D B5 FA B4 8E BC 9D A4 A8 AE 8B AD   ä±.É]µü'Im *®I-
0x090: 33 92 BD EB F3 83 56 92 88 E9 CC FE B7 91 8B 48   3'‰eöIV'Ieİp.'IH
0x0A0: 08 B2 DA 3A B7 99 35 29 23 02 AC 24 AB 9C 45 93   .²Ü:·I5)#,~$«IEI
0x0B0: 32 E9 SE BA D1 1F DE C1 6B FA C0 07 09 DC 63 91   2é^9Ñ.ÉÁküÀ..Üc^
0x0C0: E6 B0 E2 34 DC 6C 25 77 53 B0 0F 9D 8F 4C F7 75   æ*ä4Ül%wS°. L=u
0x0D0: 39 16 FD BB 31 0C 8B D7 8A 3C 94 A4 29 E8 2F 6E   9.y»1.IxI(*)è/n
0x0E0: B4 8A D0 91 D5 F2 6F 32 96 73 65 18 99 9B 3F 26   'ID'Ööo2Ise.II?S
0x0F0: 16 35 39 ED F6 DA B8 FF 33 41 8A 89 14 00 00 00   .59iöU.y3AII....
0x100: 8A 65 24 97 03 7F 91 88 72 3C 68 DB 3B F5 B9 AD   Ie$I.I'IrchÜ;ö+-
0x110: F8 0A 18 C8   e..È

```

Abbildung 5.2: Ausschnitt von MSIPP-SK im Hex-Editor; kein Flag ist gesetzt

Die Position der Bytes, welche die Flags beschreiben, konnte durch eine Analyse bestätigt werden. Wurde ein Klartext mit der *CryptProtectData()*-Funktion mit beliebig gesetztem Flag geschützt, so war Byte Nummer 40 das Byte, welches das gesetzte Flag gekennzeichnet hat. Auch für MSIPP-MK ist kein Flag gesetzt.

5.6 Fazit

In diesem Kapitel wurden zwei weitere Angriffe auf RMS untersucht. Für beide Angriffe hat man physischen Zugriff auf die Maschine des Opfers angenommen, um dort ein Programm ausführen zu können, welches die Zertifikate des Opfers kopiert bzw. den privaten Schlüssel des SPCs entschlüsseln sollte. Das Übernehmen der Benutzeridentität hat nicht funktioniert, da hierfür die Login-Informationen des Opfers notwendig wären. Als nächstes wurde versucht, den privaten Schlüssel des SPCs zu entschlüsseln; dafür wurde ein Programm vorgestellt, welches die DPAPI-Entschlüsselungsfunktion verwendet und Data-Blobs aus einer Datei einliest. Eine Erklärung, warum das Programm für den geschützten Data-Blob nicht funktioniert hat, ist das entweder unbekannte sekundäre Entropie verwendet worden ist oder die originale Struktur des Data-Blobs ist mit RSAVault, über welches Microsoft keine Angaben macht, verschleiert worden.

6 Zusammenfassung

Im Folgenden werden die Kapitel der Bachelorarbeit kurz zusammengefasst.

Zunächst wurde das Thema motiviert. Es wurde erwähnt, dass potentielle Schwachstellen in Microsofts AD RMS entdeckt wurden, die es einem Benutzer innerhalb einer AD RMS Domäne erlauben, Rechte für Dokumente zu erlangen, welche der Administrator bzw. der Autor für diesen Benutzer so nicht vorgesehen hat. Es wurde daraufhin eine kurze Einführung in die Funktionsweise der AD RMS gegeben, um die internen Abläufe zu verstehen. Auch wurde die Funktionsweise der DPAPI erläutert, da mit dieser die Maschinen- und somit auch die Benutzeridentität geschützt ist. Daraufhin wurde eine mit dem RMS-SDK 2.1 entwickelte Anwendung vorgestellt, durch welche ein Nutzer einer Domäne auf ein mit RMS geschütztes Word-Dokument Vollzugriff erlangen kann, sofern das Leserecht für diesen Benutzer gesetzt ist. Durch die aufgeführten Voraussetzungen für diesen Angriff wurde deutlich, dass es sich um eine in der Praxis ausnutzbare Sicherheitslücke handelt. Mitarbeiter von Unternehmen könnten eine solche Anwendung verwenden, um damit firmeninterne Dokumente zu entschlüsseln, welche sie dann auf dem Schwarzmarkt oder an die Konkurrenz verkaufen können. Im nächsten Kapitel dieser Arbeit wurden zwei weitere Angriffsszenarien diskutiert. Der erste bestand darin, ein Dokument im Namen eines anderen Benutzers aus der Domäne erstellen zu können; der zweite war darauf ausgelegt, Rechte für ein Dokument zu erlangen, auf welches ein Nutzer nicht einmal das Leserecht besitzt. Es wurde jeweils angenommen, dass der Angreifer für einen kurzen Zeitraum physischen Zugriff auf den Computer seines Opfers im eingeloggten Zustand hat. Zunächst wurde in diesem Kapitel versucht, die Benutzeridentität des Opfers zu übernehmen, um die beschriebenen Angriffe durchzuführen. Dies ist bei der Entschlüsselung des privaten Schlüssels des SPCs gescheitert, da durch die Funktionsweise der DPAPI die Login-Informationen des Benutzers in den DPAPI-Blob des geschützten Schlüssels mit einfließen. Der RMS Client hat erkannt, dass die Maschinen- und Benutzeridentität verändert worden sind und hat daraufhin neue Zertifikate erstellt. Als nächstes wurde versucht, den geschützten privaten Schlüssel des SPC mit den Funktionalitäten der DPAPI zu entschlüsseln. Für diesen Zweck wurde ein Programm, welches die DPAPI-Entschlüsselungsfunktion verwendet, erstellt. Das Programm ist für die entsprechenden Einträge in der Registry fehlgeschlagen, eine mögliche Begründung dafür ist, dass entweder bei der Erstellung des Data-Blobs unbekannte sekundäre Entropie verwendet worden ist oder RSAVault, über welches Microsoft keine Auskunft gibt, hat die Bytes des Data-Blobs verändert.

6.1 Ausblick

Eventuell lassen sich die weiteren beiden Angriffen mit Methoden lösen, die im Rahmen dieser Bachelorarbeit nicht untersucht worden sind. So könnten möglicherweise weitere Untersuchungen über RSAVault

angestellt werden, um herauszufinden, ob und wie Microsoft auf diese Weise den privaten Schlüssel des SPCs zusätzlich zur DPAPI schützt. Würden diese Angriffe gelingen, hätte dies wiederum fatale Folgen in echten Unternehmenssituationen. So könnte ein Mitarbeiter Dokumente im Namen seines Vorgesetzten erstellen, bzw. er könnte firmeninterne Dokumente entschlüsseln, auf die er nicht einmal das Leserecht besitzt. In einem in der Praxis eingesetzten Rechteverwaltungssystem dürfen solche Angriffe, besonders in größeren Unternehmen, nicht möglich sein.

Abbildungsverzeichnis

2.1	Abhängigkeiten der Zertifikate und Lizenzen	4
2.2	Funktionsweise der DPAPI	6
2.3	Struktur der Data-Blobs nach <i>passcape</i> [5]	7
3.1	Aufbau der Testumgebung	8
4.1	Max öffnet die Datei	10
4.2	Berechtigungen von Max	10
4.3	Fehlermeldung 1 beim Versuch die Datei zu öffnen	12
4.4	Fehlermeldung 2 beim Versuch die Datei zu öffnen	12
4.5	entschlüsselter Inhalt mit aktivierter Kopieren-Funktion in Word	13
4.6	XML-Dateien des verschlüsselten Word-Dokuments	13
4.7	Ausschnitt von [6]Primary im Hex-Editor	14
4.8	Ausschnitt von EncryptedPackage im Hex-Editor	14
4.9	Syntax der IPC_BUFFER Struktur	18
5.1	vier Registry-Einträge	27
5.2	Ausschnitt von MSIPP-SK im Hex-Editor; kein Flag ist gesetzt	30

Literaturverzeichnis

- [1] KAISER, Jan: *Analyse von Microsofts Rights Management Services in Windows Server 2012 R2*. August 2015
- [2] 7-ZIP: *Willkommen auf der offiziellen 7-Zip Webseite!* <http://www.7-zip.de/>. – Online: 08. März 2016
- [3] MICROSOFT: *Licenses and Certificates, and how AD RMS protects and consumes documents*. https://blogs.technet.microsoft.com/information_protection/2011/03/13/licenses-and-certificates-and-how-ad-rms-protects-and-consumes-documents/. – Online: 08. März 2016
- [4] MICROSOFT: *Windows Data Protection*. <https://msdn.microsoft.com/en-us/library/ms995355.aspx>. – Online: 08. März 2016
- [5] PASSCAPE: *DPAPI Secrets*. <http://www.passcape.com/index.php?section=docsys&cmd=details&id=28#23>. – Online: 08. März 2016
- [6] MICROSOFT: *IpcfReadFile function*. [https://msdn.microsoft.com/de-de/library/windows/desktop/dn771753\(v=vs.85\).aspx](https://msdn.microsoft.com/de-de/library/windows/desktop/dn771753(v=vs.85).aspx). – Online: 08. März 2016
- [7] MICROSOFT: *IpcfDecryptFile function*. [https://msdn.microsoft.com/de-de/library/windows/desktop/dn133058\(v=vs.85\).aspx](https://msdn.microsoft.com/de-de/library/windows/desktop/dn133058(v=vs.85).aspx). – Online: 08. März 2016
- [8] MICROSOFT: *IpcfDecryptFileStream function*. [https://msdn.microsoft.com/de-de/library/windows/desktop/dn456837\(v=vs.85\).aspx](https://msdn.microsoft.com/de-de/library/windows/desktop/dn456837(v=vs.85).aspx). – Online: 08. März 2016
- [9] MICROSOFT: *IpcDecrypt function*. [https://msdn.microsoft.com/en-us/library/windows/desktop/hh535258\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/hh535258(v=vs.85).aspx). – Online: 08. März 2016
- [10] MICROSOFT: *IpcGetKey function*. [https://msdn.microsoft.com/de-de/library/windows/desktop/hh535263\(v=vs.85\).aspx](https://msdn.microsoft.com/de-de/library/windows/desktop/hh535263(v=vs.85).aspx). – Online: 08. März 2016
- [11] MICROSOFT: *Protected Content Stream*. [https://msdn.microsoft.com/en-us/library/dd944643\(v=office.12\).aspx](https://msdn.microsoft.com/en-us/library/dd944643(v=office.12).aspx). – Online: 08. März 2016
- [12] MICROSOFT: *IPC BUFFER structure*. [https://msdn.microsoft.com/de-de/library/windows/desktop/hh535273\(v=vs.85\).aspx](https://msdn.microsoft.com/de-de/library/windows/desktop/hh535273(v=vs.85).aspx). – Online: 08. März 2016
- [13] MICROSOFT: *IPCNotepad Sample*. <https://code.msdn.microsoft.com/windowsdesktop/IPCNotepad-Sample-f67dae80>. – Online: 08. März 2016
- [14] MICROSOFT: *Rights Management-Freigabeanwendung für Windows*. [https://technet.microsoft.com/de-de/library/dn919648\(v=ws.10\).aspx](https://technet.microsoft.com/de-de/library/dn919648(v=ws.10).aspx). – Online: 08. März 2016

-
- [15] MICROSOFT: *AD RMS under the hood: Client bootstrapping*. <http://blogs.technet.com/b/rms/archive/2012/08/17/ad-rms-under-the-hood-client-bootstrapping-part-1-of-2.aspx>. – Online: 08. März 2016
- [16] MICROSOFT: *System Error Codes*. [https://msdn.microsoft.com/de-de/library/windows/desktop/ms681382\(v=vs.85\).aspx](https://msdn.microsoft.com/de-de/library/windows/desktop/ms681382(v=vs.85).aspx). – Online: 08. März 2016