Bundesamt
für Sicherheit in der
Informationstechnik

Eine Studie im Auftrag des
Bundesamtes für Sicherheit in der Informationstechnik

Projekt 154

Quellcode-basierte Untersuchung von kryptographisch relevanten Aspekten der OpenSSL-Bibliothek

Arbeitspaket 3:
Schwachstellenanalyse

Version 1.1.1 / 2015-11-03

Sirrix AG
security technologies

Zusammenfassung

Die Kryptobibliothek OpenSSL wird seit mehreren Jahren in verschiedenen kryptographischen Systemen eingesetzt. Sie ist besonders wichtig bei gesicherten Datenübertragungen in Computernetzwerken und spielt aufgrund ihrer großen Verbreitung im Internet eine besonders wichtige Rolle bei der Verwendung von HTTPS (HTTP over SSL/TLS). OpenSSL bietet eine große Zahl von kryptographischen Funktionen an; dazu gehören z.B. symmetrische und asymmetrische Datenverschlüsselung, digitale Signaturen, Authentisierung, Hashfunktionen, Erstellung und Verifikation von Zertifikaten inklusive Zertifikatsketten, Mechanismen zum Integritätsschutz, Schlüssel- und Zufallszahlenerzeugung. Die wichtigste Aufgabe von OpenSSL ist es jedoch, eine Implementierung des TLS-Protokolls (früher SSL-Protokoll) zu sein. Damit stellt die Bibliothek sowohl grundlegende Kryptofunktionen als auch das High-Level-Protokoll TLS zur Verfügung.

Dieser Bericht ist das Ergebnis der Untersuchung der OpenSSL-Bibliothek hinsichtlich Schwachstellen. Basierend auf einem Angreifermodell wird die Anfälligkeit für existierende Angriffe sowie mögliche unbekannte Schwachstellen untersucht. Wichtige Teilaspekte dieses Angriffsraums sind die Schlüsselerzeugung, die Zertifikatskettenprüfung und die Implementierung der Ciphersuites. Es wird untersucht, ob und welche Auswirkung Compiler- und Konfigurationsflags auf Schwachstellen haben. Abschließend werden die Auswirkungen der Schwachstellen und mögliche Gegenmaßnahmen analysiert.

Autoren

Wolfgang Meyer zu Bergsten (MzB), Sirrix AG
René Korthaus (RK), Sirrix AG
Juraj Somorovsky (Jso), 3curity GmbH
Christian Mainka (CM), 3curity GmbH
Jörg Schwenk (Jsc), 3curity GmbH

Quellcode-basierte Untersuchung von kryptographisch relevanten Aspekten der OpenSSL-Bibliothek

OpenSSL 1.0.1g

Inhaltsverzeichnis

1 Angreifermodell und Seitenkanalangriffe

In diesem Arbeitspaket werden Angreifermodelle für die aktuellen Angriffe gegen TLS-Implementierungen analysiert. Danach werden die Angriffe den vorgestellten Modellen zugeteilt und es werden Gegenmaßnahmen zusammengefasst und im OpenSSL Code identifiziert.

Basierend auf diesen Ergebnissen werden in AP 3.6 die identifizierten Gegenmaßnahmen näher analysiert und überprüft.

1.1 Angreifermodelle: Formal und Realistisch

Formale Modelle zur Analyse kryptographischer Primitive und Protokolle bestehen in der Regel aus drei Teilen:

1. Computational Model. In diesem Modell wird die Funktionalität des realen Protokolls möglichst genau als Kommunikation verschiedener abstrakter Rechner (z.B. Turing-Maschinen) beschrieben. Besonderen Wert muss hier auf die Unterscheidung zwischen einem Programm (i.d.R. als "party" bezeichnet) und den von diesem Programm gestarteten parallel laufenden Prozessen (i.d.R. als „(process) oracle" bezeichnet) gelegt werden.
2. Adversarial Model. Hier wird beschrieben, welche Fähigkeiten der Angreifer hat. In der Regel sind die Fähigkeiten des modellierten Angreifers deutlich größer als die jedes realen Angreifers. Wenn daher ein Protokoll im Modell sicher ist, dann ist dieses Protokoll (wenn die Modellierung korrekt war und wenn die kryptographischen Annahmen zutreffen) auch gegen jeden realen Angreifer sicher. Die Fähigkeiten des Angreifers werden in so genannten "queries" gekapselt. So wird z.B. die reale Fähigkeit eines Angreifers, einen Sitzungsschlüssel in TLS-RSA mithilfe eines Bleichenbacher-Angriffs zu berechnen, als REVEAL-Query gekapselt; sendet der Angreifer eine solche Query an ein "process oracle", so muss dieses mit dem Sitzungsschlüssel antworten.
3. Security Model. In diesem Teil wird ein Spiel zwischen einem Angreifer und einem Challenger simuliert. Der Challenger muss das Protokoll perfekt implementieren, und alle Queries des Angreifers korrekt beantworten. Nachdem durch geeignete Definitionen triviale Angriffe ausgeschlossen wurden, werden alle Ereignisse, die als erfolgreiche Angriffe auf das Protokoll angesehen werden, definiert. Ein Protokoll ist dann sicher, wenn diese Ereignisse nur mit vernachlässigbar kleiner Wahrscheinlichkeit auftreten.

Ein Angriff ist umso stärker, je schwächer das Angreifermodell ist, d.h. je weniger Fähigkeiten der Angreifer zur Durchführung des Angriffs benötigt.

Sicherheitsbeweise für TLS wurden bisher nur in einem starken Angreifermodell publiziert, aller-

dings nur für eine Teilmenge der TLS-Funktionalität, und unter Ausschluss von Seitenkanalangriffen und Implementierungsfehlern. Es scheint aufgrund der Komplexität von TLS unmöglich, die komplette Funktionalität in einem starken Angreifermodell als sicher nachzuweisen.

Praktische Modelle. Daher haben schwächere Modelle ihre Existenzberechtigung, da hier Sicherheitsnachweise ggf. einfacher geführt werden können. Diese schwächeren, praxisnahen Modelle dienen aber üblicherweise nicht dazu, Sicherheitsbeweise zu führen (denn das weist nicht die Sicherheit des gesamten Protokolls nach, sondern nur eines kleinen Teilaspekts), sondern dazu, konkrete Angriffe zu klassifizieren.

Als ein wichtiges praxisnahes Modell hat sich das "Web Attacker Model" bewährt: Ein Angreifer kann hier beliebige Datensätze an einen Serverdienst im Internet senden und die Antworten beobachten. Darüber hinaus kann er selbst einen Internetdienst betreiben und Nutzer zur Nutzung dieses Dienstes verführen.

1.1.1 Modell 1: Kryptographisches Angreifermodell (formal)

In diesem, zuerst von Bellare und Rogaway 1993 beschriebenen Modell kontrolliert der Angreifer das gesamte Kommunikationsnetz. Darüber hinaus kann er in den Besitz einiger kryptographischer Schlüssel gelangen.

Da im TLS-Handshake-Protokoll Schlüsselvereinbarung und Teilnehmer-Authentifikation realisiert sind, muss das Protokoll als gebrochen angesehen werden, wenn mindestens eine der beiden Eigenschaften gebrochen wird. Darüber hinaus sollte auch die Verschlüsselung der Daten mit dem ausgehandelten Schlüssel sicher sein.

Bis 2012 gingen alle formalen Modelle davon aus, dass dies sauber getrennt werden kann. Sie ignorierten die Tatsache, dass in diesen formalen Modellen alle praktische eingesetzten Protokolle (SSL/TLS, SSH, IPSec IKE, EMV Card2Terminal) unsicher sind. Daher wurde auf der CRYPTO 2012 das ACCE-Modell vorgestellt, in dem erstmals alle diese Protokolle sinnvoll analysierbar sind.

Beide Modelle sollen hier dargestellt werden.

1.1.1.1 Authenticated Key Exchange (AKE)

In AKE-Modellen [BR93, CK01] geht man davon aus, dass Schlüsselverenbarung und Verschlüsselung des Datentransport klar getrennt sind. Man versucht dann nachzuweisen, dass die Schlüsselvereinbarung einen Schlüssel liefert, der nicht von einem zufälligen Wert unterschieden werden kann. War dies erfolgreich, so kann die Sicherheit der Verschlüsselung unter der Annahme untersucht werden, dass ein zufälliger Schlüssel verwendet wird. Durch diese Annahmen kann man Beweise modular gestalten.

Der Angreifer wird in diesen Modellen übertrieben stark modelliert. In der Praxis möglich Angreiferszenarien sind immer deutlich schwächer. Wenn es aber möglich ist, die Sicherheit des Protokolls

im Angesicht eines solchen starken theoretischen Angreifers nachzuweisen, dann ist das Protokoll auch automaisch gegen alle praktischen Angriffe (aus derselben Angriffsklasse) sicher.

Im Folgenden sollen die Queries und ihre Berechtigung anhand aktueller Angriffe vorgestellt werden.

- SEND. Im Modell kann der Angreifer diese Query nutzen, um eine Nachricht gezielt an einen Client- oder Serverprozess zu senden. Die Antwort auf diese Nachricht wird wieder an den Angreifer zurückgesandt. In der Praxis ist dieses Szenario immer dann gegeben, wenn der Angreifer das Netzwerk kontrolliert, also z.B. in WLAN-Umgebungen, oder bei Kontrolle eines großen Internetknotens.
- REVEAL. Im Modell erhält der Angreifer als Antwort auf diese Query den Sitzungsschlüssel eines Prozesses. Bekanntestes Beispiel für ein reales Äquivalent sind Bleichenbacher-Angriffe, bei denen ein Angreifer für RSA-basierte Ciphersuites das Schlüsselmaterial einer älteren, aufgezeichneten Session ermitteln und diese somit entschlüsseln kann. Motiviert wurde diese Query durch Exhaustive Key Search Angriffe.
- CORRUPT. Diese Query ist an die Party adressiert; sie liefert den (langlebigen, statischen) privaten Schlüssel (Authentifizierung und/oder Entschlüsselung) des Programms zurück, also bei TLS den privaten Schlüssel des Servers oder ggf. auch des Clients. Sind private Schlüssel nicht hinreichend sicher abgespeichert, so können sie durch Malware oder über Angriffe wie Heartbleed gestohlen werden.

Eine besondere Rolle spielt die TEST-Query. Sie modelliert nicht eine bestimmte Fähigkeit des Angreifers, sondern dient zur Definition eines erfolgreichen Angriffs auf die Schlüsselvereinbarung. Nach Senden von TEST an einen Prozess erhält der Angreifer im Modell entweder den Sitzungsschlüssel oder eine Zufallszahl aus der gleichen Verteilung zurück (REAL-or-RANDOM-Paradigma). Kann er diese beiden Werte "unterscheiden", so ist die Schlüsselvereinbarung ggf. anfällig gegen Angriffe.

Alle praktisch eingesetzten Protokolle sind in diesem Modell absurderweise unsicher; dies soll am Beispiel TLS erläutert werden. Ein Angreifer kann die TEST-Query immer richtig beantworten, indem er wie folgt vorgeht:

- Er zeichnet einen vollständigen TLS-Handshake auf, einschließlich der verschlüsselten FINISHED-Nachrichten.
- Er stellt die TEST-Query und erhält einen Wert K zurück.
- Er versucht, mit diesem Wert K eine der verschlüsselten FINISHED-Nachrichten zu entschlüsseln. Gelingt dies, so gibt er „REAL“ aus, gelingt es nicht, „RANDOM“. Den Erfolg der Verschlüsselung kann er an den konstanten Startbytes erkennen, die in der TLS-Datenstruktur den unverschlüsselten FINISHED vorangestellt sind.

Dieser „Erfolg“ des Angreifers lässt sich in der realen Welt natürlich nicht reproduzieren, da es dort keine TEST-Query gibt. Die Analyse von TLS in einem AKE-Modell liefert also kein verwertbares Ergebnis, weder den Beweis der Sicherheit, noch den Nachweis der Unsicherheit.

1.1.1.2 Authenticated and Confidential Channel Establishment (ACCE)

Auf der CRYPTO 2012 wurde durch Jager et al. ein neues Sicherheitsmodell vorgestellt [JKSS12]. Dieses gibt die Modularität der AKE-Modelle auf und betrachtet die authentische Schlüsselvereinbarung und die nachfolgende Verschlüsselung als Einheit. Ein Protokoll ist sicher im ACCE-Modell, wenn der Angreifer weder die Authentifizierung brechen kann, noch die authentische Verschlüsselung.

Dazu wird die TEST-Query durch zwei neue Queries ersetzt:

- ENCRYPT. Mit dieser Query werden zwei Nachrichten m_0 und m_1 an den Verschlüsselungsprozess übergeben. Dieser wählt anhand eines für jeden gestarteten Prozess festen (aber zufällig gewählten) Bits b die Nachricht m_b aus und verschlüsselt diese. Der Angreifer gewinnt wenn er entscheiden kann, welche Nachricht verschlüsselt wurde. (Mit dieser Nachricht kann man CSRF-Angriffe, oder Angriffe wie CRIME oder BEAST simulieren.)
- DECRYPT. Diese Query macht nur für eine zustandsabhängige Verschlüsselung ("stateful encryption") Sinn. Mit dieser Query wird dem Prozess ein Chiffretext übergeben, der zunächst entschlüsselt wird. Kommt dabei ein "neuer" Klartext heraus, der bisher noch nicht als Chiffretext gesendet wurde, so gibt der Prozess abhängig wiederum vom festen Bit b entweder diesen Klartext oder eine Fehlermeldung zurück. Da es bei einer authentischen Verschlüsselung für den Angreifer unmöglich sein sollte, selbst einen gültigen Chiffretext zu produzieren, gibt der Prozess als "fast immer" einen Fehler aus. Nur wenn es dem Angreifer gelingt, einen neuen gültigen Chiffretext zu erzeugen, hat er eine Chance, das Bit b zu ermitteln.

Diese etwas schräge Definition dient dazu, beide Erfolgsfälle des Angreifers (erfolgreiche Entschlüsselung und erfolgreiche Verschlüsselung) als Entscheidungsprobleme zu modellieren.

1.1.2 Modell 2: Web Attacker Model - Anonymer Client (praktisch)

Dieses Modell ist ein eingeschränktes Web Attacker Model – der Angreifer ist nur in der Lage, Anfragen an einen existierenden Dienst im Internet zu stellen.

Angriffe in diesem Modell sind als äußerst schwerwiegend einzustufen, da diese Voraussetzung für jeden Serverdienst im Internet zutrifft.

1.1.3 Modell 3a: Man-in-the-Middle (praktisch)

Dieses Modell kommt dem (theoretischen) Modell 1 am nächsten:

- Der Angreifer kann alle Nachrichten lesen und modifizieren, und kann ihren Empfänger bestimmen. Dies entspricht der oben beschriebenen SEND Query.
- Im Gegensatz zum theoretischen Modell ist ein Angriff in diesem Modell aber nur dann erfolgreich, wenn der Angreifer die zwischen den beiden Parteien ausgetauschten Nachrichten auch tatsächlich lesen kann.

Die Voraussetzungen für dieses Modell sind z.B. in WLAN-Umgebungen gegeben, wenn der Angreifer den Access Point kontrolliert.

1.1.4 Modell 3b: Man-in-the-Middle - Authentifizierter Client (praktisch)

Eine spezielle Klasse von Angriffen auf TLS benötigt die Fähigkeit, von einem gegenüber einem Webserver bereits authentifizierten Client Anfragen zu senden und

1. die Klartexte der Anfragen modifizieren und den TLS-Chiffretext lesen oder
2. die Chiffretexte der Anfragen modifizieren und das Zeitverhalten der Serverantwort messen

zu können.

Diese Anforderungen lassen sich teilweise mit einem Cross-Site Request Forgery (CSRF) Angriff realisieren. CSRF-Angriffe bieten jedoch nur begrenzten Gestaltungsspielraum zur Modifikation von Anfragen: nur die aufgerufene URL kann modifiziert werden.

Cross-Site Scripting Angriffe (XSS) bieten größeren Spielraum, da man mithilfe des XMLhttpRequest-Objekts die Anfragen genau gestalten kann (direkte Anfragen werden in modernen Browsern von Cross-Origin Ressource Sharing (CORS) blockiert).

Durch das Einbetten von privilegierten Java-Applets oder die Nutzung von Malware können alle notwendigen Voraussetzungen geschaffen werden. Mit diesen Varianten kann auch eine Zeitmessung der Serverantworten und ein Mitschneiden des Netzwerkverkehrs realisiert werden.

1.1.5 Modell 4: Direkter Zugriff auf die OpenSSL-API (praktisch)

In diesem Modell ist der Angreifer in der Lage, direkt einzelne Funktionen der OpenSSL-API aufzurufen. Er kann aber nicht lesend auf existierende Schlüssel zugreifen.

1.1.6 Modell 5: Übernahme eines Prozesses durch den Angreifer (praktisch)

In diesem Modell übernimmt der Angreifer die Kontrolle über einen OpenSSL-Prozess, der eine TLS-Session verwaltet. Damit kann er das MasterSecret und alle daraus abgeleiteten Schlüssel berechnen. Andere Prozesse und die langlebigen Schlüssel sollten davon nicht betroffen sein.

Der Heartbleed-Angriff hat leider gezeigt, dass dies in der Praxis nicht immer der Fall ist.

1.2 Seitenkanäle

Je nach "Nähe" des Angreifers zur OpenSSL-Implementierung sind verschiedene Seitenkanalmessungen möglich.

1.2.1 Webanwendungen

Greift der Angreifer eine mit OpenSSL abgesicherte Webanwendung über das Internet an, so sind nur Fehlermeldungen oder Zeitmessungen als Seitenkanäle möglich.

1.2.2 Cloudumgebungen

Werden Angreifer und Ziel im selben Cloud-Rechenzentrum gehostet, so sind zunächst einmal sehr viel effizientere Zeitmessungen möglich.[1] Dieses Ziel ist relativ einfach zu erreichen [RTSS09, ZJRR12, MS14].

Werden Angreifer und Ziel sogar auf demselben physikalischen Host ausgeführt, so sind viele weitere, in der Literatur untersuchte Seitenkanäle möglich, z.B. Strommessung oder Beobachtung der CPU-Leistung.

1.2.3 Lokale Installation

Bei einer lokalen Installation kann der Angreifer nahezu alle Arten von Seitenkanälen messen. Ein mögliches Szenario wäre hier z.B. ein gestohlenes Hardware Security Modul (HSM), aus dem ein privater Schlüssel extrahiert werden soll.

1.3 Angriffsklasse Handshake

1.3.1 TLS-RSA: Bleichenbacher-Angriffe

Angreifermodell: 2 - Web Attacker

Seitenkanäle: Zeitmessung oder Fehlermeldungen in Webanwendung oder Cloudumgebung

Ziel: Berechnung des PremasterSecret

Beschreibung: Die in der Target-TLS-Session mitgeschnittene ClientKeyExchange-Nachricht wird unter Ausnutzung der Homomorphieeigenschaft der RSA-Verschlüsselung adaptiv so lange modifi-

1 Nach unserem besten Wissen existieren keine wissenschaftlichen Publikationen, die sich mit diesem Thema beschäftigt hätten und genaue Zahlen angeben würden. Unsere ersten Untersuchungen mit OpenNebula und Amazon Web Services haben ergeben, dass in unserer OpenNebula Cloud Differenzen von 10 Mikrosekunden messbar waren und in der Amazon Cloud Differenzen von 100 Mikrosekunden messbar waren. Dabei ist hat der Client mit dem Server über TCP kommuniziert.
Zeitmessungen sind aber oft von vielen Faktoren abhängig und eine erweiterte Analyse mit mehreren Cloud-Umgebungen und Hardware-Konfigurationen wäre nötig. Wir gehen davon aus, dass ein Angreifer im besten Fall Differenzen von ein paar hundert Nanosekunden bis ein paar Mikrosekunden messen könnte, wenn die Kommunikation über TCP laufen würde. Beim Einsatz von UDP könnte der Angreifer ggf. auch kleinere Zeitunterschiede messen.

ziert, bis der resultierende, dem Angreifer unbekannte Klartext PKCS#1-konform ist. Kann ein Angreifer PKCS#1-konforme von nicht konformen Chiffretexten unterscheiden (Seitenkanäle), so kann er nach Beobachtung mehrerer PKCS#1-konformer Chiffretexte das PremasterSecret berechnen [Blei].

Voraussetzungen:

1. Angreifer muss aus den Antworten des Servers oder aus Seitenkanälen der Antworten erkennen können, ob die gesendete ClientKeyExchange-Nachricht PKCS#1-konform war oder nicht.
2. PKCS#1-konforme Nachrichten müssen hinreichend oft erzeugbar sein.

Ciphersuites: Dieser Angriff ist auf alle RSA-basierten Ciphersuites anwendbar, also **TLS_RSA***

Gegenmaßnahmen OpenSSL: OpenSSL hat Maßnahmen implementiert, um Voraussetzung 2 zu verhindern. OpenSSL verifiziert dazu die PKCS#1-Konformität wie folgt:

- Prüfe, ob die ersten beiden Bytes den Wert 0x00 0x02 haben.
- Prüfe, ob das nächste 0x00 Byte auf der 49sten Position von hinten steht. Dieses 0x00 Byte ist ein Trenn-Byte zwischen dem Padding und dem PremasterSecret (damit wird auch implizit überprüft, dass das PKCS#1 Padding eine Mindestlänge von 8 Bytes erreicht hat).
- Überprüfe, ob das 48ste Byte von hinten den Wert 0x03 hat (TLS major version).
- Überprüfe, ob das 47ste Byte von hinten einen der Werte 0x01, 0x02 oder 0x03 hat (TLS minor versions für TLS 1.0, 1.1 und 1.2).

Nur wenn alle diese Überprüfungen erfolgreich sind, wird die ClientFinished-Nachricht als TLS-konform eingestuft. Damit besteht nur eine Wahrscheinlichkeit von 2^{-40}, eine TLS-konforme Nachricht zufällig zu finden.

Implementiert in:

- **ssl/s3_srvr.c** (Z. 2206 – 2252): Aufruf der PKCS#1 Entschlüsselung, Überprüfung der 48-Byte Länge des PremasterSecrets (damit wird implizit überprüft, ob das erste 0x00 Byte in der Nachricht auf der 49sten Position von hinten steht), Überprüfung der TLS-Version in der entschlüsselten PremasterSecret-Struktur, Generierung eines zufälligen PremasterSecrets (mit der Funktion RAND_pseudo_bytes aus rand_lib.c) im Falle einer invaliden Premaster-Secret-Struktur.l.
- **crypto/rsa_pk1.c: `int RSA_padding_check_PKCS1_type_2`** (Z. 181): PKCS#1

Unpadding, Überprüfung ob die Nachricht mit 0x00 0x02 anfängt und ob sie 0x00 in der Mitte beinhaltet. Zusätzlich wird explizit überprüft, ob vor dem ersten 0x00 Byte mindestens 8 Padding-Bytes stehen, wie es im PKCS#1-Standard vorgeschrieben wird.

Quellen:

- Daniel Bleichenbacher: Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1 [Blei]
- Christopher Meyer, Juraj Somorovsky, Eugen Weiss, and Jörg Schwenk, Sebastian Schinzel, Erik Tews, Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. [MSWS]

1.3.1.1 Zusammenfassung der Ergebnisse aus dem letzten USENIX-Artikel

[MSWS] analysiert mehrere TLS-Implementierungen und deren Verwundbarkeit gegen Bleichenbacher-Angriffe. Der Ausgangspunkt für unsere Analyse war der ursprüngliche Artikel von Daniel Bleichenbacher [Blei]. Die Aussage des Artikels war (kurz zusammengefasst):

1. Der beschriebene Angriff funktioniert immer, wenn der Angreifer anhand des Serververhaltens unterscheiden kann, ob ein entschlüsselter Chiffretext mit 0x00 0x02 beginnt oder nicht. Einen solchen Server nennt man ein Oracle.
2. Ein Oracle kann basierend auf direkten Server-Fehlermeldungen oder, z.B., anhand von bestimmten Zeitunterschieden konstruiert werden.
3. Wenn das konstruierte Oracle mit False Positives antwortet (eine Nachricht, die nicht mit 0x00 0x02 startet wird als valid eingestuft), führt dies zum Abbruch des Angriffs.
4. Wenn das Oracle mit False Negatives antwortet (eine Nachricht die mit 0x00 0x02 beginnt wird als invalid eingestuft), führt dies zu einer schlechteren Angriffs-Performance.

False Negatives können durch folgende Nachrichten-Validierungen eingeführt werden:

- Der Server überprüft, ob eine Nachricht mit 0x00 0x02 beginnt, ob die nächsten 8 Bytes keine 0x00 beinhalten und ob danach ein 0x00 Byte folgt. Erst wenn alles erfüllt wird, erhält der Angreifer eine valide Antwort. Anderseits antwortet der Server gleich als ob die Nachricht nicht mit 0x00 0x02 beginnen würde. Damit ist z.B. eine Nachricht, die mit 0x00 0x02 0x00 startet als invalid eingestuft, obwohl die ersten zwei Bytes korrekt sind.
- Der Server überprüft, ob die Struktur der Nachricht der TLS-Struktur entspricht. Dabei muss zusätzlich die Länge des PremasterSecrets korrekt sein, die ersten zwei Bytes des PremasterSecrets entsprechen der TLS-Version.

Während des Angriffs verändern sich die Nachrichten-Bytes hinter 0x00 0x02 "zufällig". Daher ist die Performance des Angriffs immer davon abhängig, wie groß die Wahrscheinlichkeit ist, dass ein

Oracle mit valid antwortet, wenn die Antwort mit 0x00 0x02 beginnt. Bei dem ersten Beispiel liegt die Wahrscheinlichkeit bei 60%. Bei dem zweiten Beispiel sind es aber nur $3 \cdot 10^{-6}$, was den Angriff unmöglich macht.

In [MSWS] wurden vier Side-Channels beschrieben, welche zu drei praktischen Angriffen geführt haben:

Direkter Angriff auf JSSE (Java Secure Socktest Extension)

Eine Ursache hierfür war ein INTERNAL ERROR Alert, welcher in spezifischen Fällen sichtbar war, wenn die Nachricht mit 0x00 0x02 begonnen hat (die Ursache dafür war eine interne ArrayIndexOutOfBoundsException, welche nicht korrekt abgefangen wurde). Normalerweise antwortete der Server mit einem HANDSHAKE FAILURE alert.

Die Performance war von der Schlüssellänge abhängig. Bei größeren Schlüsseln war die Chance, dass der Server mit einer INTERNAL ERROR Nachricht antwortet viel größer. Deswegen waren die Angriffe auf Server mit 1024bit Schlüsseln unmöglich. Angriffe auf Server mit 4096bit Schlüsseln nahmen ungefähr 40000 Server-Anfragen in Anspruch.

Zeitdifferenz in OpenSSL:

Eine Ursache für eine Zeitdifferenz in OpenSSL war eine zusätzliche Random-Number-Generierung, welche aufgetreten ist, wenn die entschlüsselte Nachricht eine invalide Struktur hatte. Dies hat es ermöglicht, Zeitunterschiede von ungefähr 1,5 Mikrosekunden zu messen.

Die festgestellte Zeitdifferenz hat aber zu keinem praktischen Angriff geführt. Der Grund dafür war vor allem, dass die Random-Number-Generierung nur dann entfallen ist, wenn die entschlüsselte Nachricht TLS-konform war. Wie oben beschrieben wurde, ist die Wahrscheinlichkeit dafür sehr klein, dass eine vom Angreifer zufällig generierte Nachricht TLS-konform wird ($< 2^{-40}$). Dies hat einen praktischen Angriff unmöglich gemacht.

Zusätzliche interne BadPaddingException in JSSE:

In JSSE wurde für die Unpadding-Funktionalität eine PKCS#1-Methode benutzt, die eine BadPaddingException produziert hat, wenn die PKCS#1-Struktur nicht korrekt war. Obwohl diese Exception nicht nach außen propagiert wurde, konnten wir Zeitdifferenzen von ungefähr 20 Mikrosekunden messen (alleine durch die BadPaddingException-Generierung).

Da die BadPaddingException nur in den Fällen aufgetaucht ist, wenn die PKCS#1-Struktur verletzt wurde, lag die Wahrscheinlichkeit für Generierung einer validen Nachricht bei 60%. Dies hat es erlaubt, praktische Bleichenbacher-Angriffe zu testen. Wir haben für eine Entschlüsselung ungefähr 15.000.000 Server-Anfragen in unserem LAN-Szenario gebraucht.

Unerwartetes Zeitverhalten von dem Cavium-Chip

Während unserer Forschung haben wir auch IBM Datapower untersucht, welche für TLS einen Cavium Accelerator Chip verwendet. Es hat sich herausgestellt, dass basierend auf den entschlüsselten Nachrichten unterschiedliche Antwortzeiten erkennbar sind. Die Bearbeitung einer Nachricht, die *auf der zweiten Position ein 0x02 Byte* enthalten hat, hat ungefähr 15 Mikrosekunden länger gedauert als die Bearbeitung von anderen Nachrichten.

Obwohl das Verhalten des Servers nicht "Bleichenbacher-konform" war (der Server hat Nachrichten akzeptiert, welche mit 0x01 0x02, 0x02 0x02, usw. beginnen), konnten wir in unseren Labor-Bedingungen praktische Angriffe testen. Hierfür haben wir den Bleichenbacher-Angriff angepasst. Für den Angriff wurden ungefähr 4.000.000 Anfragen erstellt.

1.3.2 TLS-DH: Kleine Untergruppen

Angreifermodell: 2 - Web Attacker

Seitenkanäle: Fehlermeldung, Timing

Ziel: Berechnung des privaten Exponenten des Servers

Beschreibung: In der ClientKeyExchange-Nachricht wird eine Zahl gesendet, die in einer kleinen Untergruppe der Ordnung r von Z_p^* liegt. Durch Testen aller r möglichen Werte für das Premaster-Secret kann der Wert des Exponenten modulo r berechnet werden.

Voraussetzungen:

1. p-1 muss kleine Primfaktoren haben.
2. Die Ordnung des Client-DH-Shares darf nicht überprüft werden.

Ciphersuites: Dieser Angriff ist auf alle DH-basierten Ciphersuites anwendbar, also **TLS_DH***

Gegenmaßnahmen OpenSSL: Der Angriff ist nicht durchführbar, wenn während jedes Handshakes DH Parameter neu generiert werden. Dies passiert in OpenSSL wenn die Option SSL_OP_SINGLE_DH_USE gesetzt ist.

Zustätzlich generiert OpenSSL für DH-Schlüssel nur solche Primzahlen *p*, bei welchen gilt dass *(p-1)/2* auch eine Primzahl ist. Diese Primzahlen werden in der OpenSSL Dokumentation auch "strong primes" genannt und ihr Einsatz verhindert die small subgroup Angriffe auch wenn während des Handshakes DH Parameter nicht neu generiert werden [OpenSSLDH]:

"If ``strong'' primes were used to generate the DH parameters, it is not strictly necessary to generate a new key for each handshake but it does improve forward secrecy. If it is not assured, that ``strong'' primes were used (see especially the section about DSA parameters below), SSL_OP_SINGLE_DH_USE must be used in order to prevent small subgroup attacks. Always using SSL_OP_SINGLE_DH_USE has an impact on the computer time needed during negotiation, but it

is not very large, so application authors/users should consider to always enable this option. The option is required to implement perfect forward secrecy (PFS)."

Implementiert in:

- **ssl/s3_srvr.c (Z. 1647 – 1656):** Aufruf der DH Schlüssel-Generierung beim ServerKeyExchange. Wenn SSL_OP_SINGLE_DH_USE nicht gesetzt ist, wird kein neuer DH Schlüssel generiert.

Quellen:

- R. Zuccherato: Methods for Avoiding the "Small-Subgroup" Attacks on the Diffie-Hellman Key Agreement Method for S/MIME [RFC2785]
- Vlastimil Klíma, Tomáš Rosa: Attack on Private Signature Keys of the OpenPGP format, PGP programs and other applications compatible with OpenPGP [KR02]
- T. Dierks, E. Rescorla, The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, 2008, http://tools.ietf.org/html/rfc5246#appendix-F.1.1.3 [TLS12]

1.3.3 TLS-ECDH: Punkte außerhalb der Kurve

Angreifermodell: 2 - Web Attacker

Seitenkanäle: Fehlermeldung, Timing

Ziel: Berechnung des privaten Exponenten des Servers

Beschreibung: In der ClientKeyExchange-Nachricht wird ein Punkt gesendet, der nicht auf der elliptischen Kurve liegt. Durch sorgfältige und adaptive Wahl dieses Punktes können aus Fehlermeldungen (oder Zeitverzögerungen) partielle Informationen über den privaten Exponenten des Servers gewonnen werden.

Voraussetzungen:

1. Es darf nicht überprüft werden, ob der Punkt auf der Kurve liegt.

Ciphersuites: Dieser Angriff ist auf alle ECDH-basierten Ciphersuites anwendbar, also **TLS_ECDH***.

Gegenmaßnahmen OpenSSL: Lage des Punktes auf der Kurve wird überprüft.

Implementiert in:

- **crypto/ecc/ecp_smpl.c:** **`int ec_GFp_simple_is_on_curve`** (Z. 940): Überprüfung ob der Punkt auf der Kurve liegt. Diese Funktion wird aufgerufen aus der Funktion **`ec_GFp_simple_oct2point`** (**crypto/ecc/ecp_oct.c ,** Z. 325).

Quellen:

- Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography. http://safecurves.cr.yp.to [SAFEECC]

1.3.4 Brumley-Boneh

Angreifermodell: 2 – Web Attacker

Seitenkanäle: Timing (lokale Installation oder Cloud-Umgebung)

Ziel: Berechnung des privaten RSA-Exponenten ***d*** des Server

Beschreibung: Die Verwendung des Chinesischen Restsatzes und der Montgomery-Arithmetik wird ausgenutzt, um aus der Messung der Zeit, die zur Entschlüsselung speziell konstruierter Nachrichten vom TLS-Server benötigt wird, auf den privaten RSA-Exponenten ***d*** zu schließen.

Voraussetzungen:

1. Angreifer kann über das Netzwerk Zeitdifferenzen messen.
2. RSA-Blinding deaktiviert in OpenSSL.
3. Montgomery-Arithmetik wird verwendet.

Ciphersuites: Dieser Angriff ist auf alle RSA-basierten Ciphersuites anwendbar, welche RSA entweder für Entschlüsselung (also **TLS_RSA***) oder Signieren (**TLS_*_RSA***) einsetzen.

Gegenmaßnahmen OpenSSL: RSA Blinding in OpenSSL ist standardmäßig aktiv [RSAblAdv]. RSA Blinding funktioniert wie folgt:

Bevor der Chiffretext c entschlüsselt wird, wird ein zufälliger Wert **`A`** generiert und ein Wert x wie folgt berechnet:

$$x = A^e \cdot c \bmod N$$

Dabei ist (e,N) der öffentliche RSA-Schlüssel.

Anschließend wird x entschlüsselt:

$$x^d \bmod N = A^{ed} \cdot c^d \bmod N = A \cdot m \bmod N$$

Schließlich wird der Klartext mit der Inverse von **`A`** berechnet:

$$m = m \cdot A \cdot A^{-1} \bmod N$$

Diese Funktionalität kann mit dem Compiler-Flag OPENSSL_NO_FORCE_RSA_BLINDING deaktiviert werden.

Implementiert in:

- **crypto/rsa/rsa_eay.c: `int RSA_eay_private_decrypt`** (Z. 492)

Quellen:

- OpenSSL Security Advisory zu RSA Blinding [RSAblAdv]
- David Brumley, Dan Boneh: Remote Timing Attacks are Practical [BB2003]
- Onur Acıiçmez, Werner Schindler, Çetin K. Koç: Improving Brumley and Boneh Timing Attack on Unprotected SSL Implementations [ASK05]

1.3.5 Timing Angriffe auf ECDSA

Angreifermodell: 2- Web Attacker

Seitenkanäle: Timing

Ziel: Berechnung des privaten Exponenten des Servers

Beschreibung: Basierend auf den Timing-Unterschieden zwischen den Berechnungen von ECDSA Signaturen mit unterschiedlichen Ephemeral Schlüsseln kann der Angreifer die Länge der Ephemeral Schlüssel abschätzen und anschließend einen Lattice-basierten Angriff durchführen. Dies führt zur Aufdeckung des privaten Exponenten des Servers.

Voraussetzungen:

1. Angreifer kann über das Netzwerk Zeitdifferenzen messen.
2. Unterschiedliche Längen von Ephemeral Schlüsseln eingesetzt bei der Berechnung von ECDSA-Signaturen.
3. Messbare Zeitdifferenzen bei den Berechnungen von ECDSA-Signaturen mit unterschiedlichen Schlüssellängen.

Ciphersuites: Dieser Angriff ist auf alle EC-basierten Ciphersuites anwendbar, welche ECDSA für Signaturerstellung verwenden (also **TLS_*_ECDSA***).

Gegenmaßnahmen OpenSSL: Timing-konstante ECDSA-Signatur Erstellung erzeugt durch eine konstante Länge von Ephemeral Schlüsseln. Dabei wird die Gegenmaßnahme von Brumley und Tuveri aus dem referenzierten Artikel eingesetzt:

```
/* We do not want timing information to leak the length of k,
 * so we compute G*k using an equivalent scalar of fixed
 * bit-length. */
if (!BN_add(k, k, order)) goto err;
if (BN_num_bits(k) <= BN_num_bits(order))
      if (!BN_add(k, k, order)) goto err;
```

Um vor Timing-Angriffen zu schützen, wird mit einem Schlüssel k' einer fixen Länge gerechnet. Dabei wird zuerst $k' = k + ord(G)$ berechnet, wobei ord(G) die Ordnung des Punktes G darstellt. Falls die Bitlänge von k' immer noch kleiner oder gleich der Bitlänge von ord(G) ist: $lg(k') \leq lg(ord(G))$, wird k' wie folgt geändert berechnet: $k' = k + 2 \cdot ord(G)$.

Die Multiplikation wird mit [k']G anstatt von [k]G durchgeführt. Dies verletzt die Signatur nicht.

Implementiert in:

- **crypto/ecdsa/ecs_ossl.c: `int ecdsa_sign_setup`** (Z. 147-160): Ephemeral Schlüssel k wird auf eine konstante Länge gesetzt wie oben beschrieben ist.

Quellen:

- Billy Bob Brumley and Nicola Tuveri: Remote Timing Attacks are Still Practical [BT2011]

1.3.6 Verfrühte ChangeCipherSpec-Nachricht

Angreifermodell: 3a – Man-in-the-middle

Seitenkanäle: keine

Ziel: Erzwingen der Verwendung schwacher Schlüssel für eine Verbindung zwischen Client und Server

Beschreibung: Es ist möglich, durch Ändern der Handshake-Nachrichten einen OpenSSL-Client (alle Versionen) und einen OpenSSL-Server (Versionen 1.01 und 1.02) dazu zu zwingen, einen Schlüssel nur anhand von öffentlichen Parametern zu erzeugen. Damit kann die Kommunikation zwischen dem Client und dem Server entschlüsselt werden oder der Angreifer kann die Nachrichten beliebig ändern.

1. Der Angreifer fügt verfrühte ChangeCipherSpec-Nachrichten an Client und Server in den Kommunikationsfluss ein.
    - Erhält der Server eine ChangeCipherSpec-Nachricht, nachdem er ServerHello gesendet hat, aber bevor er das PremasterSecret pms erhält, so berechnet er alle Schlüssel mit pms=NULL. Ab Version 1.01 wird aber die ServerFinished-Nachricht mit dem korrekten pms berechnet.
    - Erhält der Client eine ChangeCipherSpec-Nachricht, bevor das pms erzeugt wurde, so arbeitet auch er für die Schlüsselerzeugung mit pms=NULL. Es gibt aber in alle eingesetzten Versionen eine zusätzliche Überprüfung, ob eine ChangeCipherSpec-Nachricht nach Erzeugung des MasterSecret und vor dem Empfang einer ServerFinished-Nachricht empfangen wurde. Beim Client muss also zusätzlich zur verfrühten ChangeCipherSpec-Nachricht eine zweite ChangeCipherSpec-Nachricht gesendet werden, sonst bricht der Handshake ab. Diese zweite ChangeCipherSpec-Nachricht

triggert dann die Erzeugung von korrekten ClientFinished und ServerFinished-Nachrichten aus dem korrekten pms.

2. Nur in der Kombination von OpenSSL-Client (alle Versionen) und OpenSSL-Server (Versionen 1.01 und 1.02) kommt es dann zu einer kuriosen Konstellation: Beide Seiten verwenden schwache Schlüssel für den Record Layer, die aus pms=NULL generiert wurden, authentifizieren sich aber gegenseitig korrekt mit korrekten Finished-Nachrichten, die aus dem korrekten pms erzeugt wurden.

Voraussetzungen:

1. Als Client wird ein OpenSSL-Client eingesetzt, als Server eine OpenSSL-Version 1.01 oder 1.02.
2. Der Angreifer muss als Man-in-the-middle agieren.

Ciphersuites: Dieser Angriff ist Ciphersuite-unabhängig anwendbar.

Gegenmaßnahmen OpenSSL: OpenSSL 1.0.1g ist anfällig, der Fehler wurde erst in der OpenSSL 1.0.1h Version mit dem Commit bc8923b1ec9c467755cd86f7848c50ee8812e441 behoben: https://github.com/openssl/openssl/commit/bc8923b1ec9c467755cd86f7848c50ee8812e441

Implementiert in:

- **ssl/s3_clnt.c** (Z. 562-919): Es werden CCS Flags gesetzt.
- **ssl/s3_pkt.c** (Z. 1319-1327): Überprüfung, ob CCS Nachricht in korrekter Reihenfolge gesendet wurde.
- **ssl/s3_srvr.c** (Z. 676-778): Setzen der CCS Flags und Überprüfen korrekter Reihenfolge.

Quellen:

- Masashi Kikuchi. How I discovered CCS Injection Vulnerability [EarlyCCSLep]
- `OpenSSL Security Advisory EarlyCCS [EarlyCCSadv]`
- Adam Langley, Early ChangeCipherSpec Attack [EarlyCCSimp]

1.4 Angriffsklasse Record Layer

1.4.1 CRIME

Angreifermodell: 3b – Man-in-the-middle

Seitenkanäle: Länge der komprimierten und verschlüsselten Daten

Ziel: Berechnung konstanter Teile des Klartextes, z.B. HTTP Session Cookies

Beschreibung: CRIME ist kein Angriff auf OpenSSL, sondern auf die optionale Kompression im TLS Record Layer. Eine Funktionalität zur Datenkompression gibt es auch im HTTP-Protokoll, die-

se wird aber überwiegend für HTTP Responses angewandt, da diese in der Regel wesentlich größer sind als die HTTP Requests. Bei Kompression im TLS Record Layer werden beide Richtungen komprimiert.

1. Der Angreifer platziert einen String `guess` seiner Wahl so in der Nähe des Cookie-Headers, dass sich sowohl der Cookie-Wert als auch der String innerhalb des Sliding Window befinden, auf dem die Kompression durchgeführt wird.
2. Weist `guess` Ähnlichkeiten mit dem Cookie-Wert auf, so wird eine stärkere Kompression möglich. Der Angreifer kann dies anhand der Länge der von ihm beobachteten Record-Layer-Nachrichten erkennen.
3. Zahlreiche Optimierungen sind erforderlich, um Besonderheiten der Datenkompression auszugleichen und um effizient Wert und Position der einzelnen Zeichen des Cookie-Wertes zu bestimmen.

Voraussetzungen:

1. Angreifer muss HTTP-Requests triggern und adaptiv beliebige Strings in der Nähe des Cookie-Headers einfügen können. Hierzu genügt es laut der verfügbaren Dokumentation, mittels Javascript beliebige (GET oder POST) URLs aufzurufen, da der Angreifer hier seine eigenen Strings nur ganz am Anfang des Requests (im Pfad oder im Query String der URL) oder ganz am Ende (im Body der POST-Nachricht) unterbringen kann.
2. Eine weitere Möglichkeit ist XMLHttpRequest: Ein XMLHttpRequest-Objekt kann über Javascript vorbereitet werden. Ob es allerdings als HTTP-Request an den Target-Server gesendet werden darf, wird vom Browser über Cross-Origin Ressource Sharing (CORS) geregelt. Der Zugriff über CORS muss aber vom Target-Webserver für den Angreifer-Webserver explizit erlaubt werden, was wenig wahrscheinlich ist.
3. Datenkompression im Record Layer oder auf HTTP-Ebene (incl. Google SPDY) ist aktiviert.

Ciphersuites: Dieser Angriff ist Ciphersuite-unabhängig anwendbar.

Gegenmaßnahmen OpenSSL: Da Voraussetzung 1 durch die OpenSSL-Implementierung nicht kontrolliert werden kann, muss die Datenkompression (Voraussetzung 3) deaktiviert werden. Dies muss für jede OpenSSL-Installation separat erfolgen. Die Datenkompression kann in OpenSSL mit einem Compiler Flag no-comp gesteuert werden. Dies wird im AP 3.2 näher beschrieben.

Weitere Gegenmaßnahmen außerhalb von OpenSSL sind die Deaktivierung der Datenkompression in verschiedenen Webbrowsern, und eine modifizierte Datenkompression in Google's SPDY, bei der die Cookie-Header gesondert komprimiert werden.

Quellen:

- Tal Be'ery, Amichai Shulman. A perfect CRIME? Only TIME will tell [CRIME-BS13]
- Adam Langley, CRIME [CRIMEimp]
- Krzysztof Kotowicz. If it's CRIME, then I'm guilty [CRIME-koto]

1.4.2 TIME

Angreifermodell: 2 – Web Attacker

Seitenkanäle: Länge der komprimierten und verschlüsselten Daten, Timing

Ziel: Berechnung konstanter Teile des Klartextes, z.B. HTTP Session Cookies

Beschreibung: Auf der Blackhat Europe 2013 wurde von Tal Be'ery eine Variante von CRIME vorgestellt, die im schwächeren Web Attacker Modell (2) funktioniert. Der Angreifer misst hier die Zeit, die vom Server benötigt wird, um einen bestimmten HTTP-Request zu verarbeiten. Je nachdem, wie gut der Request komprimiert wird, "passt" er in n oder n+1 Datenpakete. Dies kann der Angreifer mit Javascript messen.

Der Vorteil gegenüber dem CRIME-Angriff ist es, dass der Angreifer nicht die Länge der Chiffretexte von außen beobachten muss. Stattdessen verlässt er sich lediglich auf die Dauer der Datenübertragung, welche direkt vom Javascript gemessen werden kann.

Der Autor des Angriffs hat auch eine Erweiterung beschrieben, welche die Datenkompression von HTTP-Responses behandelt: Die Dauer der Übertragung wird durch die Länge der HTTP-Responses beeinflusst. Dabei muss es dem Angreifer eine Möglichkeit gegeben werden, einen Code einzuschleusen, welcher von der Webseite reflektiert wird.

Voraussetzungen:

- Es muss eine Code-Injection-Schwachstelle (z.B. XSS) vorhanden sein, die es erlaubt, Javascript-Code in die Webseite einzuschleusen. XSS ist erforderlich, um XMLHttpRequests senden zu können, bei einem reinen CSRF-Angriff ist dies nicht möglich. Der maliziöse XSS-Code kann dann den einzufügenden Text entweder selbst generieren oder von einem Server nachladen.
- Datenkompression muss aktiviert sein.
- Der Angreifer muss eine Möglichkeit haben, seinen Code so einzuschleusen, so dass dieser zusammen mit dem Geheimnis übertragen (und damit komprimiert) wird: Entweder in dem HTTP-Request oder in der HTTP-Response.

Ciphersuites: Dieser Angriff ist Ciphersuite-unabhängig anwendbar.

Gegenmaßnahmen OpenSSL: Dieser Angriff ist unabhängig von OpenSSL.

Quellen:

- Tal Be'ery, Amichai Shulman. A perfect CRIME? Only TIME will tell [CRIME-BS13]

1.4.3 BREACH

Angreifermodell: 2 – Web Attacker

Seitenkanäle: Länge der komprimierten und verschlüsselten Daten, Timing

Ziel: Berechnung konstanter Teile des Klartextes, z.B. HTTP Session Cookies

Beschreibung: Auf der Black Hat 2013 wurde gezeigt, dass Kompressions-basierte Angriffe weiterhin möglich sind. Ziel sind diesmal Anti-CSRF-Tokens, die innerhalb des Body von HTTP-Responses gesendet werden. Der Angreifer schleust seinen String hier als User-contributed Input (z.B. Eingabe in ein Suchfeld), der in die Ergebnisseite reflektiert wird, ein. Bei eingeschalteter HTTP-Kompression war es so möglich, Anti-CSRF-Token mit 95% Wahrscheinlichkeit in ca. 30 Sekunden zu berechnen.

Voraussetzungen:

- Die Webanwendung muss eine Markup Injection-Schwachstelle (z.B. XSS) aufweisen.

Ciphersuites: Dieser Angriff ist Ciphersuite-unabhängig anwendbar.

Gegenmaßnahmen OpenSSL: Dieser Angriff ist unabhängig von OpenSSL und muss z.B. auf dem Apache Server verhindert werden.

Quellen:

- Yoel Gluck, Neal Harris, Angelo Prado, BREACH. SSL, Gone in 30 Seconds [BREACH]

1.4.4 BEAST

Angreifermodell: 3b – Man-in-the-middle

Seitenkanäle: Fehlermeldung

Ziel: Berechnung konstanter Teile des Klartextes, z.B. HTTP Session Cookies

Beschreibung: BEAST ist kein Angriff auf OpenSSL, sondern ein Angriff auf SSL-fähige Clients, die SSL 3.0 und TLS 1.0 verwenden.

1. Das Opfer besucht die Webseite angreifer.org. Dort wird eine bösartige Javascript-Datei geladen, die verschiedene Verbindungen zum Target-Server aufbaut.
2. Die Javascript-Datei versucht, eine der folgenden Verbindungen zum Target-Server aufzubauen:
    - HTML5 Websocket
    - Java URL Connection

- Silverlight WebClient

3. Sei L die Blocklänge der Blockchiffre, die im CBC-Modus im Record Layer eingesetzt wird. Der Angreifer sendet nun L-1 bekannte Bytes am Anfang jeder Nachricht, so dass das erste zu entschlüsselnde Byte sich als letztes Byte im ersten Block befindet.
4. Da SSL 3.0 und TLS 1.0 die Technik des IV-Chaining einsetzen, kann der Angreifer durch Vergleich des letzten Chiffretextblocks mit dem nächsten IV in maximal 256 Versuchen den korrekten Wert dieses Bytes bestimmen.
5. Dieser Vorgang wird durch Verkürzen des bekannten Präfixes für alle zu berechnenden Bytes wiederholt.

Voraussetzungen:

1. Angreifer muss HTML5 Websocket-, Java URL Connection- oder Silverlight WebClient-Verbindungen triggern können.
2. Target-Applikation muss HTML5 Websocket-, Java URL Connection- oder Silverlight WebClient-Verbindungen unterstützen.
3. TLS Version 1.0 oder SSL Version 3.0 sind aktiviert.
4. Angreifer ist Man-in-the-middle.

Ciphersuites: Dieser Angriff ist auf alle Ciphersuites anwendbar, welche CBC Mode-of-Operation einsetzen, also: **TLS_*_WITH_AES_128_CBC*, TLS_*_WITH_AES_256_CBC*.** Darüber hinaus kann der Angriff auf andere CBC-Ciphersuites außerhalb von TR-02102-2 anwendbar sein.

Gegenmaßnahmen OpenSSL: Dies ist kein Angriff auf OpenSSL. Generische Gegenmaßnahmen sind:

1. Server und Client: Deaktivierung aller TLS- und SSL-Versionen älter als TLS 1.1. Ab Version 1.1 ist dieser Angriff in TLS nicht mehr möglich.
2. Server: Deaktivierung von HTML5 Websockets, Java URL Connections und Silverlight WebClients.
3. Server: Deaktivierung aller CBC-basierten Ciphersuites für TLS 1.0.
4. Clients: Für Version 1.0 wurde von allen Browserherstellern außer Apple die Verschlüsselung eines Leerblocks am Anfang jedes Record Layer Paketes eingeführt ("1/n-1 Splitting").
5. Clients: Java URL Connections verwenden einen eigenen TLS-Stack. Dieser implementiert 1/n-1 Splitting und sendet die Browsercookies nicht mit.

Implementiert in: keine implementierte Gegenmaßnahme auf der Server-Seite.

Quellen:

- Thai Duong, Juliano Rizzo: Here Come The ⊕ Ninjas [BEAST]
- https://community.qualys.com/blogs/securitylabs/2013/09/10/is-beast-still-a-threat

1.4.5 Lucky13

Angreifermodell: 3b – Man-in-the-middle

Seitenkanäle: Timing

Ziel: Berechnung konstanter Teile des Klartextes, z.B. HTTP Session Cookies

Beschreibung: TLS verwendet ein MAC-then-PAD-then-ENCRYPT-Schema im Record Layer. Die Länge des Padding ist damit nicht authentifiziert. Der Angreifer ändert diese Längenangabe durch Modifikation des entsprechenden Chiffretextabschnittes.

Voraussetzungen:

1. Angreifer muss den TLS-Chiffretext abfangen und modifizieren können
2. Angreifer muss in der Lage sein, Zeitunterschiede im Antwortverhalten des Servers zu messen.

Ciphersuites: Dieser Angriff ist auf alle Ciphersuites anwendbar, welche CBC Mode-of-Operation einsetzen, also: **TLS_*_WITH_AES_128_CBC*, TLS_*_WITH_AES_256_CBC*.** Darüber hinaus kann der Angriff auf andere CBC-Ciphersuites außerhalb von TR-02102-2 anwendbar sein.

Gegenmaßnahmen OpenSSL: OpenSSL hat Maßnahmen implementiert, um Voraussetzung 2 zu verhindern. Für alle denkbaren Längenangaben wird genau die gleiche Antwortzeit des Servers garantiert.

Implementiert in:

- **ssl/s3_cbc.c**: Timing-konstantes Unpadding, Timing-konstante MAC Validierung

Quellen:

- N.J. AlFardan and K.G. Paterson, Lucky Thirteen: Breaking the TLS and DTLS Record Protocols [Lucky13]
- Adam Langley, Lucky Thirteen attack on TLS CBC [Lucky13imp]

1.4.6 Padding Oracle On Downgraded Legacy Encryption (Poodle)

Angreifermodell: 3b – Man-in-the-middle

Seitenkanäle: Verbindungsabbruch

Ziel: Berechnung konstanter Teile des Klartextes, z.B. HTTP Session Cookies

Beschreibung: TLS verwendet für Blockchiffren ein MAC-then-PAD-then-ENCRYPT-Schema im

Record Layer. Die Länge des Padding ist damit nicht authentifiziert. Der Angreifer ändert diese Längenangabe durch Modifikation des entsprechenden Chiffretextabschnittes. Der Angriff ist besonderes effizient, da in SSL 3.0 die Länge des Padding in nur einem Byte kodiert wird. Der Angriff läuft wie folgt ab:

1. Der Angreifer gestaltet die Länge des Klartextes so, dass das Padding einen volle Blocklänge der Blockchiffre einnimmt. Somit besteht der letzte Block des Klartextes aus dem Padding, und das letzte Byte hat den Wert 64 oder 128, je nach Blocklänge. Dieser feste Wert des letzten Byte sei mit BL bezeichnet.
2. Der Angreifer hat über dies den zu entschlüsselnden Wert so positioniert, dass er genau mit einem Chiffretextblock abschließt. Das letzte Byte des zu entschlüsselnden Wertes entspricht also auch dem letzten Byte des Chiffretextes. Dieser Chiffretextblock sei mit CB bezeichnet. Der Angreifer ersetzt nun den Padding-Block durch CB.
3. Bei Entschlüsselung dieser Nachricht gibt es zwei mögliche Ergebnisse:
    a) Mit Wahrscheinlichkeit 255/256 ist das letzte Byte des so erhaltenen Klartextes ungleich BL, und die Verbindung wird wegen eines fatalen Fehlers abgebrochen, da Beginn und Ende des MAC falsch berechnet wurden und daher der gefundene MAC ungültig ist.
    b) Mit Wahrscheinlichkeit 1/256 ist das letzte Byte des Klartextes gleich BL, der korrekte MAC wird gefunden, und dieser wird auch erfolgreich verifiziert, da der Chiffretext vor dem MAC nicht verändert wurde.
4. Tritt das Ereignis b) ein, so lernt der Angreifer, dass $CL \oplus PL = BL$ gilt, wobei CL das letzte Byte des vorletzten Chiffretextblocks ist, und PL das gesuchte letzte Byte des gesuchten Klartextes. Hieraus kann $PL = BL \oplus CL$ berechnet werden.
5. Nun verändert der Angreifer den Klartext derart, dass der gesuchte Klartext um 1 Byte nach hinten verschoben wird. Dadurch landet das vorletzte Byte des Klartextes auf der letzten Byteposition eines Blockes, und kann analog der Vorgehensweise in den Schritten 1 bis 4 ermittelt werden.
6. Durch weitere Verschiebungen werden alle übrigen Bytes des Klartextes berechnet.

Voraussetzungen:

1. Der Angreifer muss in der Lage sein, die Verwendung von SSL 3.0 zu erzwingen. Hierzu muss er als Man-in-the-middle im Netzwerk agieren.
2. Der Angreifer muss einzelne Bytes in den Klartext einfügen können und muss das Senden dieser Nachrichten triggern können. Hierzu reicht z.B. eine CSRF- oder XSS-Schwachstelle in der Webanwendung aus.

3. Der Angreifer muss in der Lage sein, den Verbindungsabbruch der TLS-Verbindung zu detektieren, z.B. als Man-in-the-middle.

Ciphersuites: Dieser Angriff ist auf alle Ciphersuites anwendbar, welche CBC Mode-of-Operation einsetzen, also: **TLS_*_WITH_AES_128_CBC*, TLS_*_WITH_AES_256_CBC*.** Darüber hinaus kann der Angriff auf andere CBC-Ciphersuites außerhalb von TR-02102-2 anwendbar sein.

Gegenmaßnahmen OpenSSL:

- Komplette Deaktivierung von SSL 3.0, muss manuell pro OpenSSL-Installation vorgenommen werden.
- OpenSSLVersionen 1.0.1j, 1.0.0o und 0.9.8zc, herausgegeben am 15. Oktober 2014, unterstützen TLS_FALLBACK_SCSV:
  - Moeller, Bodo; Langley, Adam (July 4, 2014). "draft-ietf-tls-downgrade-scsv-00 - TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks". *tools.ietf.org*.

Implementiert in:

- In der untersuchten OpenSSL 1.0.1g Version wurde TLS_FALLBACK_SCSV noch nicht implementiert, sie ist erst in der Version 1.0.1j vorhanden: https://www.openssl.org/news/secadv_20141015.txt
- [erst Version 1.0.1j] **ssl/ssl_lib.c** (Z. 1496-1520): Überprüfung von TLS_FALLBACK_SCSV und der genutzten TLS-Version.

Quellen:

- OpenSSL Security Advisory, POODLE und SSLv3.0 Fallback [POODLEadv]
- Bodo Möller, Thai Duong, Krzysztof Kotowicz. This POODLE bites: Exploiting the SSLv3.0 Fallback [POODLE]
- http://en.wikipedia.org/wiki/POODLE, abgerufen am 2.12.2014.

Anmerkung:

Downgrade-Angriffe auf SSL/TLS sind eigentlich seit Version 3.0 dadurch ausgeschlossen, dass die FINISHED-Nachrichten auch über die jeweils von Client und Server vorgeschlagenen Versionsnummern gebildet werden. In ihrer Veröffentlichung beschreiben Moeller, Duong und Kotowicz aber ein seltsames Verhalten von Webbrowser, das die Sicherheit des TLS-Handshakes untergräbt: Tritt ein fataler Netzwerkfehler während des Handshakes auf, so wiederholen moderne Browser den Handshake *mit der nächstkleineren TLS-Versionsnummer. Dieses undokumentierte Verhalten ermöglicht diesen Angriff erst in größerem Maßstab.*

1.5 Angriffsklasse Extension

1.5.1 TLS Renegotiation

Angreifermodell: 3a – Man-in-the-middle

Seitenkanäle: keine

Ziel: Nutzung der Authentifizierung eines Clients zum Triggern von Angreifer-initiierten Aktionen auf dem Target-Server.

Der Effekt ist vergleichbar mit einem CSRF-Angriff.

Beschreibung:

1. Der Angreifer beobachtet den TLS-Handshake zwischen dem Client und dem Target-Server und verzögert die Weitersendung des ClientHello.
2. Der Angreifer öffnet eine HTTPS-Session mit dem Target-Server und sendet eine unvollständige HTTP-Anfrage, die die Angreifer-initiierte Aktion enthält.
3. Der Angreifer sendet die originale ClientHello-Nachricht des Clients über die geöffnete HTTPS-Session an den Target-Server.
4. Dies wird vom Target-Server als eine TLS-Renegotiation-Anfrage aufgefasst, und ein vollständiger Handshake zwischen Client und Target-Server wird durchgeführt.
5. Ist diese zweite HTTPS-Session etabliert, so authentifiziert der Client sich (z.B. über ein Session Cookie), und authentifiziert somit die Angreifer-initiierte Aktion aufgrund von Besonderheiten des HTTP-Protokolls.

Voraussetzungen:

1. TLS-Renegotiation muss aktiviert sein.
2. Der Angreifer muss als Man-in-the-middle agieren können.

Ciphersuites: Dieser Angriff ist Ciphersuite-unabhängig anwendbar.

Gegenmaßnahmen OpenSSL:

- In OpenSSL 0.9.8l wurde TLS Renegotiation deaktiviert.
- Seit OpenSSL 0.9.8m wird die in RFC 5746 spezifizierte Gegenmaßnahme verwendet.

Implementiert in:

- **ssl/s3_srvr.c** (Z. 994 – 1026): Insecure Renegotiation wurde deaktiviert. Man kann sie wieder mit dem Flag SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION aktivieren.

Quellen:

- OpenSSL Security Advisory [11-Nov-2009], TLS Renegotiation Attack [RenegAdv]
- Florian Giesen, Florian Kohlar, and Douglas Stebila. 2013. On the security of TLS renegotiation [GKS13]
- E. Rescorla, M. Ray, S. Dispensa, N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension [RFC5746]
- https://www.openssl.org/docs/ssl/SSL_CTX_set_options.html
- http://www.educatedguesswork.org/2009/11/understanding_the_tls_renegoti.html

1.5.2 Triple Handshake

Angreifermodell: 3a – Man-in-the-middle

Seitenkanäle: keine

Ziel: Nutzung der Authentifizierung eines Clients zum Triggern von Angreifer-initiierten Aktionen auf dem Target-Server.

Der Effekt ist vergleichbar mit einem CSRF-Angriff.

Die in RFC 5746 beschriebenen Gegenmaßnahmen werden, wenn unvollständig implementiert, außer Kraft gesetzt. Der Angriff funktioniert nur dann, wenn die RFC 5746-Gegenmaßnahmen nur auf vollständige, und nicht auf verkürzte Handshakes angewandt werden.

Beschreibung: Der Angreifer setzt die in RFC 5746 beschriebenen Gegenmaßnahmen durch Einschaltung eines verkürzten Handshakes außer Kraft.

1. Der Angreifer baut als Man-in-the-middle zwei komplette TLS-Kanäle auf: Einen zum Client und einen zum Target-Server. Es ist wichtig, dass jeweils TLS-RSA zum Einsatz kommt, damit in beiden Verbindungen das gleiche MasterSecret verwendet werden kann.
2. Der Angreifer triggert einen verkürzten TLS-Handshake (mit TLS-Session-Resumption) direkt zwischen Client und Target-Server. Dies ist möglich, da Client und Target-Server das gleiche MasterSecret verwenden.
3. Der Angreifer initiiert so eine TLS-Renegotiation, die von den RFC 5746-Gegenmaßnahmen nicht als bösartig erkannt wird. Bei der TLS-Renegotiation leitet der Angreifer das Zertifikat des echten Servers weiter. Damit denken jetzt der Client und der Server, dass sie die ganze Zeit miteinander kommuniziert haben.

Anmerkung: Nicht nur bei TLS-RSA ist es möglich, das gleiche MasterSecret zu verwenden, siehe [BDFPS14], S. 8/9, „V-B SYNCHRONIZING DHE“:

- „Suppose that C (or S) refuses RSA ciphersuites, but accept some DHE ciphersuite. We show that A can still synchronize the two connections, because the DHE key exchange allows the server to pick and sign arbitrary Diffie-Hellman group parameters, and any client that accepts the server certificate and signature implicitly trusts those parameters." ....
- "The attack fails if C checks that p0 is prime. Yet, none of the mainstream TLS implementations perform a full primality check because it is deemed too expensive. A probabilistic primality check could help, but may not guarantee that the attacker cannot find a p0 that defeats it."
- "Even when clients and servers use known groups, care must be taken to validate the public key received from the peer."

Voraussetzungen:

1. TLS-Renegotiation muss aktiviert sein.
2. Verkürzte Handshakes mit TLS Session Resumption müssen aktiviert sein.
3. Der Client akzeptiert ein neues Zertifikat, das bei der TLS-Renegotiation transportiert wird.
4. Der Angreifer muss als Man-in-the-middle agieren können.

Ciphersuites: Dieser Angriff ist Ciphersuite-unabhängig anwendbar.

Gegenmaßnahmen OpenSSL: Deaktivierung von TLS-Renegotiation oder TLS-Session-Resumption löst das Problem. In Giesen et al. [GKS13] werden Gegenmaßnahmen beschrieben, die diesen Angriff verhindern.

Zusätzlich wurde ein neuer Extension-Draft publiziert, welcher dieses Problem behebt: Die Master-Secret-Berechnung ist so modifiziert, dass der Hashwert über alle Handshake-Nachrichten berechnet wird, anstatt nur über die Werte (ClientHello Random, ServerHello Random, PremasterSecret). Diese erweiterte Berechnung ist über eine neue Extension ausgehandelt: *Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension.*

Eine andere Gegenmaßnahme ist den Zertifikatswechsel während der TLS-Renegotiation zu verbieten.

Implementiert in: Es wurde ein Patch vorgeschlagen, es wurde aber noch in keine neuere OpenSSL Version integriert (http://openssl.6102.n7.nabble.com/PATCH-Fix-for-Triple-Handshake-attacks-via-extended-master-secret-td50058.html).

Quellen:

- Bhargavan, Delignat-Lavaud, Fournet, Pironti, Strub: Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS [BDFPS14]
- Florian Giesen, Florian Kohlar, and Douglas Stebila. 2013. On the security of TLS renegotiation [GKS13]

- K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, M. Ray: Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension [BDPLR]

1.5.3 Heartbleed

Angreifermodell: 2 - Web Attacker

Seitenkanäle: keine

Ziel: Berechnung des privaten Schlüssels des OpenSSL-Servers

Beschreibung: Der Angreifer sendet eine Heartbeat-Anfrage mit überlanger Längenangabe. Der Server schickt daraufhin einen entsprechend langen Ausschnitt seines Hauptspeichers an den Angreifer. Aus diesen Daten lässt sich im Worst Case der private Schlüssel des Servers rekonstruieren.

Voraussetzungen:

1. Heartbeat aktiviert (Default in den Versionen OpenSSL 1.0.1 bis 1.0.1f (inklusive))

Ciphersuites: Dieser Angriff ist Ciphersuite-unabhängig anwendbar.

Gegenmaßnahmen OpenSSL:

- Upgrade nach OpenSSL 1.01g, 1.02-beta2 oder höher.
- Neukompilieren des Sourcecodes mit der Option -DOPENSSL_NO_HEARTBEATS.

Implementiert in:

- **ssl/t1_lib.c** (Z. 2596 – 2632): Verifikation der Länge wurde hinzugefügt
- **ssl/d1_both.c** (Z. 1455 – 1509): Verifikation der Länge wurde hinzugefügt (nur für DTLS)

Quellen:

- R. Seggelmann, M. Tuexen, M. Williams, Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension [HeartbeatRFC]
- Nick Sullivan, Heartache and Heartbleed: The insider's perspective on the aftermath of Heartbleed [HeartbleedCloud]
- OpenSSL Security Advisory, Heartbleed [HeartbleedAdv]
- The Heartbleed Bug [Heartbleed]

1.5.4 Protocol Version Rollback über undokumentiertes Browserverhalten

In der Beschreibung des Poodle-Angriffs (s.o.) wird ein undokumentiertes Verhalten von Webbrowsern beschrieben, die die sichere Aushandlung von TLS-Versionen außer Kraft setzt und Protocol Version Rollback-Angriffe ermöglicht: Tritt während des TLS-Handshakes ein schwerer Netzwerkfehler auf, so versuchen moderne Webbrowser einen ernueten Verbindungsaufbau mit der nächst-

niedrigen TLS-Versionsnummer.

1.6 Angriffsklasse Zertifikatskettenprüfung

1.6.1 Incorrect Checks for Malformed DSA / ECDSA Signatures

Angreifermodell: 3a – Man-in-the-middle

Seitenkanäle: keine

Ziel: Nachahmen eines TLS-Servers um an Geheimnisse eines Clients zu gelangen (bspw. Cookies, Anmeldedaten).

Beschreibung:

1. Der Angreifer fängt die ClientHello-Nachricht des Clients ab antwortet mit einem eigenen ServerHello.
2. Der Angreifer öffnet parellel im Hintergrund eine HTTPS-Session mit dem Target-Server.
3. Der Angreifer sendet eine Certificate-Nachricht mit einem speziell preparierten Zertifikat, das einen DSA oder ECDSA-Schlüssel enthält.
4. Der Client verifiziert das preparierte Zertifikat erfolgreich und schließt den Handshake mit dem Angreifer erfolgreich ab.
5. Der Angreifer leitet fortan alle Anfragen zwischen Client und Target-Server weiter.

Voraussetzungen:

1. Der Angreifer muss als Man-in-the-middle agieren können.

Ciphersuites: Dieser Angriff ist auf alle DSA- und ECDSA-basierten Ciphersuites anwendbar, also **TLS_*_DSA*** und **TLS_*_ECDSA**.

Gegenmaßnahmen OpenSSL:

- Seit OpenSSL 0.9.8j werden die Rückgabewerte von EVP_VerifyFinal() und ssl_verify_cert_chain() korrekt ausgewertet.

Implementiert in:

- **ssl/s2_clnt.c** (Z. 1044): Korrekte Überprüfung des Rückgabewertes von ssl_verify_cert_chain().
- **ssl/s2_srvr.c** (Z. 1054, 1083): Korrekte Überprüfung des Rückgabewertes von ssl_verify_cert_chain().
- **ssl/s3_clnt.c** (Z. 972, 1459, 1477): Korrekte Überprüfung der Rückgabewerte von ssl_ve-

rify_cert_chain() und EVP_VerifyFinal().

- **ssl/s3_srvr.c** (Z. 2566): Korrekte Überprüfung des Rückgabewertes von ssl_verify_cert_chain().

Quellen:

- OpenSSL Security Advisory, Incorrect checks for malformed signatures [mitmAdv]
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5077

2 Kompilierungs- und Compiler-Flags

Das Ziel dieses Unterarbeitspakets ist die Dokumentation der von OpenSSL unterstützten Kompilierungsflags, welche sicherheitsrelevante Änderungen am Objektcode ergeben.

2.1 Konfiguration und Funktionsweise von Compiler-Flags in OpenSSL

Die Konfiguration der Compiler-Flags erfolgt über den Befehl `./config`. Das explizite Aktivieren der Flags erfolgt über das Argument `enable-optionname`, das explizite Deaktivieren über `no-optionname`. Beispiele hierfür wären die Aufrufe `./config no-ssl2` oder `./config enable-gmp`.

Intern führt der `./config` Befehl dabei eine Anpassung der relevanten `Makefiles` im OpenSSL Projekt aus. Das `Makefile` speichert dabei die gesetzten Flags in der Variable `OPTIONS` ab. Diese Variable dient allerdings nur zur Dokumentation der genutzten Compiler-Flags, die als Argument für das `./config` gesetzt wurden und wird im weiteren nicht mehr verwendet.

Die eigentlichen Auswirkungen der konfigurierten Flags befinden sich in der Variable DEPFLAG und beinhalten die eigentlichen Flags, die später innerhalb des OpenSSL Source Codes genutzt werden, z.B. `DEPFLAG= -DOPENSSL_NO_SSL2`.

Die Übersetzung der `./config` Flags hin zu den Makefile `DEPFLAGS` erfolgt über das `./Configure` Skript, welches von `./config` aufgerufen wird. Dieses Perl-Skript enthält in Zeilen 1079f folgenden Code:

```
my ($ALGO, $algo);
($ALGO = $algo = $_) =~ tr/[\-a-z]/[_A-Z]/;
```

Dieser bewirkt, dass die klein geschriebenen `./config` Flags, z.B. „no-ssl2“, in Großbuchstaben umgewandelt werden. Anschließend werden in Zeile 1104 die Flags für das Makefile definiert:

```
$depflags .= " -DOPENSSL_NO_$ALGO";
```

Die Flags werden zusätzlich auch noch in der von `./Configure` automatisch generierten Datei `./crypto/opensslconf.h` abgespeichert.

2.2 Eigenheiten der OpenSSL Konfiguration

Die Konfiguration über ./config besitzt einige Besonderheiten die im folgenden aufgeführt werden.

2.2.1 Keine Warnungen bei nicht existenten Flags

Wird `./config` mit einem nicht existierenden Flag aufgerufen, z.B. `./config no-ssl5`, so wird keine Warnung oder Fehlermeldung ausgegeben. Die Flag wird an `./Configure` übergeben, welches seinerseits `DEPFLAGS = OPENSSL_NO_SSL5` setzt.

Dieses Verhalten kann also sehr leicht dazu führen, dass bei der Konfiguration von OpenSSL Compiler-Flags unerwünschtes Verhalten auftritt.

Beispielsweise würde `./config -no-ssl-2` zu `DEPFLAGS = OPENSSL_NO_SSL_2` führen, und die SSLv2 Version bleibt davon unangetastet, d.h. SSLv2 ist weiterhin aktiv.

Erwähnenswert ist dies z.B., da selbst das OpenSSL Wiki nicht existierende Compiler-Flags erwähnt. So listet das OpenSSL Wiki[2] ein Flag `no-dtls` auf, welches aber nicht existiert (auch nicht im aktuellen GIT branch). Das korrekte Flag würde `no-dtls1` lauten. Siehe dazu auch die Liste der verfügbaren Compiler-Flags im nächsten Kapitel.

Lediglich kleinere „Fehlerkorrekturen“ fängt das Skript ab: `./config -no-ssl2` (man beachte den führenden Bindestrich vor „no“) wird intern vom `./Configure` Skript korrekt verarbeitet.

2.2.2 Flags werden von links nach rechts geparsed

Die Compiler-Flags für ./config werden von links nach rechts verarbeitet. Somit bewirkt beispielsweise ein Aufruf von

```
./config no-ssl2 enable-ssl2
```

dass SSLv2 weiterhin aktiv bleibt, während der Aufruf von

```
./config enable-ssl2 no-ssl2
```

dazu führt, dass SSLv2 deaktiviert wird.

2.3 Ermittlung der verfügbaren Compiler-Flags

Die offizielle Dokumentation der existierenden Compiler-Flags ist sehr dürftig. Dem OpenSSL Wiki konnte folgende Methode zur Detektierung der Flags entnommen werden:

grep -r '^#if.*OPENSSL_NO' . | grep -o 'OPENSSL_NO_[a-zA-Z0-9_]*' | sort -u | sed 's/OPENSSL_//' | tr '[A-Z_]' '[a-z-]'

Der Befehl bewirkt dabei, dass alle vorhanden Source Code Dateien nach Präprozessordirektiven durchsucht werden, die OPENSSL_NO_* Flags benutzen. Die folgenden ./config Flags konnten somit identifiziert werden:

2 http://wiki.openssl.org/index.php/Compilation_and_Installation#Configure_Options

```
no-aes, no-algorithms, no-asm, no-bf, no-bio, no-buffer, no-buf-freelists, no-camellia, no-capieng, no-cast, no-chain-verify, no-cms, no-comp, no-decc-init, no-deprecated, no-des, no-descbcm, no-dgram, no-dh, no-dsa, no-dtls1, no-dynamic-engine, no-ec, no-ec2m, no-ecdh, no-ecdsa, no-ec-nistp-64-gcc-128, no-engine, no-err, no-evp, no-fp-api, no-gmp, no-gost, no-hash-comp, no-heartbeats, no-hmac, no-hw, no-hw-4758-cca, no-hw-aep, no-hw-atalla, no-hw-chil, no-hw-cswift, no-hw-ibmca, no-hw-ncipher, no-hw-nuron, no-hw-padlock, no-hw-sureware, no-hw-ubsec, no-hw-zencod, no-idea, no-inline-asm, no-jpake, no-krb5, no-lhash, no-locking, no-md2, no-md4, no-md5, no-mdc2, no-multibyte, no-nextprotoneg, no-object, no-ocsp, no-posix-io, no-psk, no-rc2, no-rc4, no-rc5, no-rdrand, no-rfc3779, no-ripemd, no-ripemd160, no-rmd160, no-rsa, no-rsax, no-sctp, no-seed, no-setvbuf-ionbf, no-sha, no-sha0, no-sha1, no-sha256, no-sha512, no-sock, no-speed, no-srp, no-srtp, no-ssl2, no-ssl3, no-ssl-intern, no-stack, no-static-engine, no-stdio, no-store, no-tls, no-tls1, no-tls1-2-client, no-tlsext, no-whirlpool, no-x509, no-x509-verify
```

2.4 Verwendeter Compiler

Der für die Untersuchung verwendete Compiler entspricht dem standardmäßig in Ubuntu (64bit) vorhanden Compiler:

```
$ gcc --version
gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Die aktuellste Version von GCC ist 4.9.1 (Stand: 10.09.2014).

2.5 Default Compiler-Flags

2.5.1 Default Compiler-Flags im offiziellen OpenSSL Projekt

Die standardmäßig für das OpenSSL Testsystem gesetzten Flags sehen wie folgt aus:

```
OPTIONS=-Wa,--noexecstack no-ec_nistp_64_gcc_128 no-gmp no-jpake no-krb5 no-md2 no-rc5 no-rfc3779 no-sctp no-shared no-store no-zlib no-zlib-dynamic static-engine
```

Diese werden zu folgenden Flags übersetzt:

Flag	Betroffene Dateien	Beschreibung
OPENSSL_NO_EC_NISTP_64_GCC_128	crypto/opensslconf.h crypto/ec/ecp_nistp521.c crypto/ec/ectest.c crypto/ec/ecp_nistputil.c crypto/ec/ecp_nistp256.c crypto/ec/ec.h crypto/ec/ec_curve.c crypto/ec/ecp_nistp224.c	Deaktiviert die NIST Kurven. Sie werden zum einen nicht in die Liste der verfügbaren Kurven in ec_curve.c aufgenommen. Zum anderen werden die ecp_nistp*.c Dateien durch eine Präprozessordirektive inhaltslos, d.h. es enthält keinen Quelltext, der noch compiliert wird. Betroffen sind jeweils nur die NIST Kurven. Andere Kurven bleiben weiterhin aktiv. Die Liste der betroffenen Dateien bedeutet lediglich, dass das Flag dort überhaupt verwendet wird. Es wird nicht (wie bei vielen anderen Flags) der vollständige Dateiinhalt inhaltslos. OpenSSL 1.0.1 unterstützt nicht die Brainpool-Kurven.
OPENSSL_NO_GMP	crypto/opensslconf.h crypto/ec/ecp_nistp521.c crypto/ec/ectest.c crypto/ec/ecp_nistputil.c crypto/ec/ecp_nistp256.c crypto/ec/ec.h crypto/ec/ec_curve.c crypto/ec/ecp_nistp224.c	Deaktiviert die GMP Unterstützung. Das Flag bewirkt hauptsächlich, dass die Datei e_gmp.c durch eine Präprozessordirektive inhaltslos wird.
OPENSSL_NO_JPAKE	apps/apps.h apps/s_client.c apps/apps.c apps/s_server.c util/mk1mf.pl crypto/opensslconf.h crypto/err/err_all.c crypto/jpake/jpake.h crypto/jpake/jpaketest.c	J-PAKE (Password Authenticated Key Exchange by Juggling) ist per default deaktivert. Das Feature ist experimentell in OpenSSL. Um es zu aktivieren, reicht ein enable-jpake nicht aus. Es muss explizit mit experimental-jpake aktiviert werden.

Flag	Betroffene Dateien	Beschreibung
OPENSSL_NO_MD2	apps/version.c apps/progs.h apps/speed.c util/mk1mf.pl crypto/opensslconf.h crypto/md2/md2.h crypto/md2/md2test.c crypto/evp/evp.h crypto/evp/m_md2.c crypto/pkcs7/sign.c crypto/pkcs7/verify.c crypto/rand/rand_lcl.h demos/engines/rsaref/rsaref.c	Deaktiviert die MD2 Hash Funktionalität. Maßgeblich wird md2.h durch eine Präprozessordirektive inhaltslos.
OPENSSL_NO_RC5	apps/progs.h apps/progs.pl apps/speed.c util/mk1mf.pl crypto/opensslconf.h crypto/rc5/rc5test.c crypto/rc5/rc5.h crypto/evp/e_old.c crypto/evp/e_rc5.c crypto/evp/evp.h crypto/evp/c_allc.c	Deaktiviert die RC5 Funktionalität. Anders als bei den obigen Flags wird die rc5.h allerdings nicht inhaltslos, sondern wirft einen Fehler über die Präprozessordirektive #error falls das Flag gesetzt ist.
OPENSSL_NO_RFC3779	crypto/opensslconf.h crypto/asn1/x_x509.c crypto/x509v3/v3_addr.c crypto/x509v3/v3_asid.c crypto/x509v3/x509v3.h crypto/x509v3/ext_dat.h crypto/x509v3/v3_purp.c crypto/x509/x509.h crypto/x509/x509_vfy.c	RFC3779 definiert *X.509 Extensions for IP Addresses and AS Identifiers*. Maßgeblich ist die Datei x509v3.h von diesem Flag betroffen, worin die entsprechende Funktionalität definiert ist.

Flag	Betroffene Dateien	Beschreibung
OPENSSL_NO_SCTP	ssl/d1_both.c ssl/d1_srvr.c ssl/d1_pkt.c ssl/dtls1.h ssl/d1_lib.c ssl/ssl3.h ssl/d1_clnt.c makevms.com crypto/opensslconf.h crypto/bio/bss_dgram.c crypto/bio/bio.h	Deaktiviert die Unterstützung für SCTP (Stream Control Transmission Protocol).
OPENSSL_NO_STORE	crypto/opensslconf.h crypto/store/store.h	Dieses Feature ist experimentell und muss explizit über `experimental-store` aktiviert werden. Maßgeblich ist die Datei store.h von diesem Flag betroffen. Bei gesetzten Flag wird ein #error geworfen.
OPENSSL_NO_KRB5	apps/s_client.c:3 apps/s_server.c:10 ssl/kssl.h:2 ssl/d1_srvr.c:2 ssl/kssl_lcl.h:2 ssl/kssl.c:3 ssl/ssl_asn1.c:10 ssl/ssl.h:4 ssl/ssltest.c:2 ssl/ssl_lib.c:9 ssl/s3_clnt.c:5 ssl/s3_srvr.c:7 ssl/s3_lib.c:4 ssl/ssl_ciph.c:1 ssl/ssl_sess.c:2 ssl/d1_clnt.c:4 ssl/ssl_txt.c:2	Dieses Flag deaktiviert die Kerberos 5 Unterstützung.

2.5.2 Default Compiler-Flags in Ubuntu

Zum Vergleich zu den OpenSSL Flags, die das OpenSSL-Projekt als Default vorgibt, wird hier ein kurzer Vergleich zu den standardmäßig in Ubuntu verwendeten Flags vorgestellt. Allerdings wurde für Ubuntu die zum Untersuchungszeitpunkt aktuellste Version 1.0.1f von OpenSSL mit der zu un-

tersuchenden Version 1.0.1g verglichen.

Flag in Default OpenSSL 1.0.1g	Flag in Default Ubuntu OpenSSL 1.0.1f
-DOPENSSL_NO_EC_NISTP_64_GCC_128	
-DOPENSSL_NO_GMP	-DOPENSSL_NO_GMP
	-DOPENSSL_NO_IDEA
-DOPENSSL_NO_JPAKE	-DOPENSSL_NO_JPAKE
-DOPENSSL_NO_MD2	-DOPENSSL_NO_MD2
	-DOPENSSL_NO_MDC2
-DOPENSSL_NO_RC5	-DOPENSSL_NO_RC5
-DOPENSSL_NO_RFC3779	-DOPENSSL_NO_RFC3779
-DOPENSSL_NO_SCTP	-DOPENSSL_NO_SCTP
	-DOPENSSL_NO_SSL2
-DOPENSSL_NO_STORE	-DOPENSSL_NO_STORE

Man kann der Tabelle entnehmen, dass Ubuntu die Elliptische Kurven der NIST zulässt, dafür aber zusätzlich die IDEA Chiffre, die MDC-2 Hashfunktion, sowie die SSLv2 Protokoll Version verbietet.

2.5.3 Default Compiler-Flags in Archlinux

Es wurde ebenfalls ein kurzer Vergleich der Default OpenSSL Flags mit den Flags gemacht, die in der weit verbreiteten Distribution Archlinux gesetzt sind. Archlinux nutzt dabei fast alle Default OpenSSL Flags. Als einziger Unterschied werden ebenfalls wie in Ubuntu die Elliptischen Kurven der NIST erlaubt.

2.6 Nicht funktionierende Flags

OpenSSL lässt sich nicht mit beliebigen Compiler-Flags auf dem Testsystem kompilieren. Beispielsweise führt das Flag `no-tls` sowie `no-tls1` zu folgender Fehlermeldung:

```
d1_srtp.c: In function ‘SSL_CTX_set_tlsext_use_srtp’:
d1_srtp.c:230:44: error: ‘SSL_CTX’ has no member named ‘srtp_pro-
files’
  return ssl_ctx_make_profiles(profiles,&ctx->srtp_profiles);
  ...
d1_srtp.c: In function ‘SSL_get_selected_srtp_profile’:
d1_srtp.c:260:2: warning: control reaches end of non-void function
[-Wreturn-type]
  }
  ^
make[1]: *** [d1_srtp.o] Fehler 1
make[1]: Verlasse Verzeichnis '/home/christian/openssl/ssl'
make: *** [build_ssl] Fehler 1
```

Ebenso ist es auch nicht möglich, OpenSSL mit `no-aes, no-hmac` oder `no-sha1` zu bauen. Dieses Verhalten trifft vermutlich auf weitere Flags zu und wurde aus Zeitgründen nicht tiefgründiger untersucht.

Es wurde untersucht, ob OpenSSL in den folgenden Konfigurationen gebaut werden kann:

1. nur TLSv1.1 und TLSv1.2 mit / ohne Kompression: Nicht möglich.
   - Es konnte nur ein Kompilat erzeugt werden, welches TLSv1.0 – TLSv1.2 erlaubt (siehe Empfehlung). Die explizite Deaktivierung von TLSv1.0 ist über Compiler-Flags nicht möglich.
   - Die TLS-Version scheint nur über Quelltext festlegbar zu sein (include/openssl/tls1.h):
     - Z. 171f:

       ```
       #define TLS1_VERSION_MAJOR         0x03 ←TLS1.2
       #define TLS1_VERSION_MINOR         0x02 ←TLS1.1
       ```
     - Eine kurze Evaluation mittels s_client zeigte, dass damit TLS1.0 erfolgreich deaktiviert wurde.
2. nur TLSv1.1 und TLSv1.2, ausschließlich mit Ciphersuites, die AES-CBC, AES-GCM, HMAC-SHA2 bzw. HMAC-SHA1 nutzen, sowie nur RSA und ECC (NIST-Kurven) für Authentisierung und Schlüsselvereinbarung (also kein Kerberos, keine Passwort-Authentisierung, etc.): Über Compiler-Flags lassen sich solch fein granularen Einstellungen nicht vornehmen.
3. 1. und 2., aber ohne Key-Renegotiation: Es scheint kein Compilerflag zu geben, welches die Renegotiation beeinflusst.

2.7 Evaluierte Compiler-Flags

In diesem Kapitel werden einige relevante Flags überprüft. Ziel ist es zu verifizieren, dass die Auswahl der Flags auch zu dem gewünschten Ergebnis führt.

2.7.1 Test Setup

OpenSSL mit dem Codestand der Version 1.0.1g wurde auf einer virtuellen Maschine mit Ubuntu 14.04 kompiliert und installiert. Dabei wurden unterschiedliche Flags an `./config` übergeben. Anschließend wurde der OpenSSL Server wie folgt gestartet:

```
OPENSSL_CONF=/path/to/openssl.cnf \
            openssl s_server -cert server.pem -www
```

Anschließend wurde die Konfiguration mithilfe einer weiteren Virtuellen Maschine (Kali Linux)

und Tools wie `sslscan` verifiziert.

Es wurde hierbei darauf geachtet, dass das Setup möglich weit automatisert erstellt werden kann. So werden beispielsweise mithilfe eines Skriptes verschiedene OpenSSL Flags nacheinander gebaut, in separate Verzeichnisse installiert und entsprechende Start-Skripte erstellt.

2.7.2 Default Flags

Bei den default Flags wurde ermittelt, dass OpenSSL eine schwache Export Ciphersuite anbietet. Dies wurde ermittelt durch:

```
openssl s_client -cipher NULL,EXPORT,LOW,3DES -connect site:port
```

Das Ergebnis zeigt:

```
Cipher    : EXP-EDH-RSA-DES-CBC-SHA
```

Diese Ciphersuite sollte aufgrund ihrer schwachen kryptographischen Eigenschaften nicht eingesetzt werden, z.B. indem `no-des` als Flag gesetzt wird.

Die Kompression ist ausgeschaltet wodurch CRIME unterbunden ist.

Das Ergebnis von sslscan lautet wie folgt:

```
  Supported Server Cipher(s):
    Accepted  SSLv3  256 bits  ECDHE-RSA-AES256-SHA
    Accepted  SSLv3  256 bits  DHE-RSA-AES256-SHA
    Accepted  SSLv3  256 bits  DHE-RSA-CAMELLIA256-SHA
    Accepted  SSLv3  256 bits  AES256-SHA
    Accepted  SSLv3  256 bits  CAMELLIA256-SHA
    Accepted  SSLv3  168 bits  ECDHE-RSA-DES-CBC3-SHA
    Accepted  SSLv3  168 bits  EDH-RSA-DES-CBC3-SHA
    Accepted  SSLv3  168 bits  DES-CBC3-SHA
    Accepted  SSLv3  128 bits  ECDHE-RSA-AES128-SHA
    Accepted  SSLv3  128 bits  DHE-RSA-AES128-SHA
    Accepted  SSLv3  128 bits  DHE-RSA-SEED-SHA
    Accepted  SSLv3  128 bits  DHE-RSA-CAMELLIA128-SHA
    Accepted  SSLv3  128 bits  AES128-SHA
    Accepted  SSLv3  128 bits  SEED-SHA
    Accepted  SSLv3  128 bits  CAMELLIA128-SHA
    Accepted  SSLv3  128 bits  ECDHE-RSA-RC4-SHA
    Accepted  SSLv3  128 bits  RC4-SHA
    Accepted  SSLv3  128 bits  RC4-MD5
    Accepted  SSLv3  56 bits   EDH-RSA-DES-CBC-SHA
    Accepted  SSLv3  56 bits   DES-CBC-SHA
    Accepted  SSLv3  40 bits   EXP-EDH-RSA-DES-CBC-SHA
```

```
    Accepted  SSLv3  40 bits   EXP-DES-CBC-SHA
    Accepted  SSLv3  40 bits   EXP-RC2-CBC-MD5
    Accepted  SSLv3  40 bits   EXP-RC4-MD5
    Accepted  TLSv1  256 bits  ECDHE-RSA-AES256-SHA
    Accepted  TLSv1  256 bits  DHE-RSA-AES256-SHA
    Accepted  TLSv1  256 bits  DHE-RSA-CAMELLIA256-SHA
    Accepted  TLSv1  256 bits  AES256-SHA
    Accepted  TLSv1  256 bits  CAMELLIA256-SHA
    Accepted  TLSv1  168 bits  ECDHE-RSA-DES-CBC3-SHA
    Accepted  TLSv1  168 bits  EDH-RSA-DES-CBC3-SHA
    Accepted  TLSv1  168 bits  DES-CBC3-SHA
    Accepted  TLSv1  128 bits  ECDHE-RSA-AES128-SHA
    Accepted  TLSv1  128 bits  DHE-RSA-AES128-SHA
    Accepted  TLSv1  128 bits  DHE-RSA-SEED-SHA
    Accepted  TLSv1  128 bits  DHE-RSA-CAMELLIA128-SHA
    Accepted  TLSv1  128 bits  AES128-SHA
    Accepted  TLSv1  128 bits  SEED-SHA
    Accepted  TLSv1  128 bits  CAMELLIA128-SHA
    Accepted  TLSv1  128 bits  ECDHE-RSA-RC4-SHA
    Accepted  TLSv1  128 bits  RC4-SHA
    Accepted  TLSv1  128 bits  RC4-MD5
    Accepted  TLSv1  56 bits   EDH-RSA-DES-CBC-SHA
    Accepted  TLSv1  56 bits   DES-CBC-SHA
    Accepted  TLSv1  40 bits   EXP-EDH-RSA-DES-CBC-SHA
    Accepted  TLSv1  40 bits   EXP-DES-CBC-SHA
    Accepted  TLSv1  40 bits   EXP-RC2-CBC-MD5
    Accepted  TLSv1  40 bits   EXP-RC4-MD5

  Prefered Server Cipher(s):
    SSLv3  256 bits  ECDHE-RSA-AES256-SHA
    TLSv1  256 bits  ECDHE-RSA-AES256-SHA
```

Die genutzte sslscan-Version (v1.8.2, openssl 1.0.1e, Kali Linux) unterstüzt lediglich TLSv1.0. TLSv1.x Tests wurden mithilfe von s_client manuell durchgeführt.

2.7.3 Weitere Ergebnisse

Flag	Ergebnis
no-des	Alle Ciphersuites, die DES enthielten, schlugen fehl. Dieses Flag deaktiviert ebenfalls 3DES.
no-dh	Alle Ciphersuites, die Diffie-Hellman benutzen (DHE-*), schlugen fehl. Dies betrifft kein Elliptische Kurven Diffie Hellman Ciphersuites, siehe no-ec. DH-* Tests schlagen ebenfalls fehl.

Flag	Ergebnis
no-ec	Alle Ciphersuites, die Elliptische Kurven Diffie-Hellman benutzen (ECDHE-*), schlugen fehl. ECDH-* Suites werden ebenfalls nicht mehr unterstützt. Der Server lässt sich z.B. nicht mehr starten, wenn mittels -cipher eine konkrete ECDH-* Cipher ausgewählt wird.
no-heartbeats	`openssl s_client -tlsextdebug -connect 192.168.254.102:4433` (gestartet vom exterenen Kali Linux) zeigt an, dass die Heartbeat Extension nicht verfügbar ist.
no-rc4	Alle Ciphersuites, die RC4 enthielten, schlugen fehl.
no-ssl2	Alle Versuche eine Verbindung mittels SSLv2 aufzubauen schlugen fehl.
no-ssl3	Alle Versuche eine Verbindung mittels SSLv3 aufzubauen schlugen fehl.
no-tls	Das Flag no-tls kann in dem vorliegenden Code nicht genutzt werden. Der Code wird nicht erfolgreich übersetzt.

2.8 Deaktivierung der Hardwareunterstützung

2.8.1 no-hw

Die Hardwareunterstützung in OpenSSL wird durch verschiedene Flags gesteuert.

Mit dem Flag no-hw wird die Hardwareunterstützung für alle Hardware Geräte deaktiviert. Spezifische Geräte können hingegen mit `no-hw-xxx` deaktivert werden, z.B. `no-hw-padlock`. Die verfügbaren Geräte können über den Engine Support von OpenSSL wie folgt eingesehen werden:

```
$ ./openssl engine -t
(rsax) RSAX engine support
     [ available ]
(dynamic) Dynamic engine loading support
     [ unavailable ]
(4758cca) IBM 4758 CCA hardware engine support
     [ unavailable ]
(aep) Aep hardware engine support
     [ unavailable ]
(atalla) Atalla hardware engine support
```

```
     [ unavailable ]
(cswift) CryptoSwift hardware engine support
     [ unavailable ]
(chil) CHIL hardware engine support
     [ unavailable ]
(nuron) Nuron hardware engine support
     [ unavailable ]
(sureware) SureWare hardware engine support
     [ unavailable ]
(ubsec) UBSEC hardware engine support
     [ unavailable ]
(gost) Reference implementation of GOST engine
     [ available ]
```

Die obige Ausgabe wurde durch eine OpenSSL Konfiguration mit Default Flags auf dem Testsystem erzeugt.

2.8.2 no-engine

Der Engine Support wurde mit OpenSSL 0.9.6 eingeführt. Die Idee dahinter ist ein einfacher Austausch von Krypto-Komponenten wie beispielsweise die Implementierung von Algorithmen. Genau durch diesen Ansatz werden in OpenSSL auch Hardwarebeschleuniger, z.B. Intel rdrand, eingebunden.

Der Engine Support kann durch das Flag `no-engine` vollständig deaktiviert werden. Dadurch wird die Datei `apps/engine.c` dürch eine Präprozessordirektive inhaltslos, es werden keine Engines aktiviert, und somit ist auch keine Hardwareunterstützung verfügbar.

No-hw deaktiviert den Hardware-Support vollständig. Mit no-engine wird lediglich der Engine support deaktiviert. Engines können z.B. auch alternative (Software) Implementierungen von (Crypto) Algorithmen enthalten. Sie müssen nicht zwangsläufig Hardware mit einbeziehen (obwohl sie laut Dokumentation dafür sehr gut geeignet sind/das die empfohlene Einbedingung von Hardware ist).

2.8.3 no-rdrand

Durch Setzen des Flags `no-rdrand` wird in der Datei `crypto/engine/eng_all.c` durch eine Präprozessor Direktive verhindert, dass die rdrand Funktionalität geladen wird. Die entsprechende Funktionalität für x86_64 Prozessoren befindet sich in der Datei `crypto/x86_64cpuid.s`.

Es ist allerdings zu beachten, dass das vorliegende Testsystem eine virtuelle Maschine ist. VirtualBox unterstützt bis heute noch keine Durchreichung des rdrand Gerätes, so dass dieses nicht unterstützt werden kann.

no-hw hat keinen Einfluss auf rdrand. Nach der Kompilation mit diesem Flag existiert weiterhin eine engine rdrand. Das Flag no-engine deaktiviert den Engine Support vollständig, so dass auch das rdrand Modul nicht mehr geladen werden kann.

Anscheinend bezieht sich no-hw nur auf die Einbindung von externer Hardware, nicht auf spezielle (interne) Hardware-Features. Allerdings würde die Verifikation dieser Aussage den Umfang dieses APs sprengen.

2.8.4 aesni

Mit OpenSSL Version 0.9.8 wurde der Support für AESNI als Engine in den Code eingepflegt. Ab Version 1.0.1 ist AESNI nicht mehr als Engine verfügbar (siehe `openssl engine` Befehl). Stattdessen ist AESNI fest integriert in OpenSSL.

Verantwortlich für den Assembler Code sind dabei die folgenden Dateien:

./crypto/aes/asm/aesni-sha1-x86_64.pl
./crypto/aes/asm/aesni-x86.pl
./crypto/aes/asm/aesni-x86_64.pl

Diese Perl Skripte generieren dabei den entsprechenden Assembler Code (*.s Dateien) der die AES-NI Befehle enthält. Der Code wird dann direkt über das EVP Interface angesprochen.

Um AESNI zu deaktivieren, wird oftmals empfohlen das entsprechende Kernelmodul (`modprobe -r aesni_intel`) zu deaktivieren. Dies funktioniert nicht, da der entsprechende Code wie oben gezeigt fester Bestandteil von OpenSSL ist und unabhängig vom Kernelmodul funktioniert.

Eine Deaktivierung des Moduls ist nach tiefen Recherchen nicht einfach über Compiler-Flags möglich. Stattdessen empfehlen wir in folgenden Dateien:

```
crypto/evp/e_aes.c
174: #define   AESNI_CAPABLE  (OPENSSL_ia32cap_P[1]&(1<<(57-32)))
crypto/evp/e_aes_cbc_hmac_sha1.c
98: #define AESNI_CAPABLE   (1<<(57-32))
```

Die entsprechende Definition von `AESNI_CAPABLE` auf `0` zu setzen. Dadurch wird in den Dateien (`crypto/evp/e_aes.c`, Zeilen 435 und 457 bzw. `crypto/evp/e_aes_cbc_hmac_sha1.c`, Zeilen 562 und 568) nicht die AESNI Implementierung eingebunden.

2.9 Empfehlungen

Als Grundlage für die Auswahl der Compiler-Flags empfehlen wir hierbei die von Ubuntu gewählten Flags, welche im Grunde die OpenSSL Default Flags sinnvoll erweitern. Zusätzlich würden wir

vorschlagen, nicht benötigte Features zu deaktivieren, so dass mindestens folgende Flags genutzt werden sollten:

Flag	Erklärung

`no-gmp`
`no-idea`
`no-jpake`
`no-md2`
`no-mdc2`
`no-rc5`
`no-rfc3779`
`no-sctp`
`no-ssl2`
`no-store`
	Default Ubuntu Flags

`no-heartbeats` `no-dtls1`	Nicht benötigte Features
`no-ssl3`	Nicht benötigte Protokoll Version.

`no-rc4`
`no-des`
`no-sha1`
	Deaktivierung von schwachen bzw. abgekündigten Algorithmen. Hinweis zu no-sha1: TLSv1.0 benötigt SHA1, allerdings bewirkt das Flag die vollständige Deaktivierung. Damit also erfolgreich SHA1 deaktiviert werden kann, müsste ebenfalls SSLv2 bis TLSv1.1 deaktiviert werden. Allerdings funktionieren die dazugehörigen TLS Flags wie oben aufgeführt leider nicht. Der Compiler-Fehler ist unten dargestellt.

-DDEVRANDOM_EGD=NULL	Deaktivierung des Entropy Gathering Daemons als Entropiequelle für den RNG, s. [AP2], Abs. 2.6.7

Diese Konfiguration entspricht somit auch den technischen Richtlinien des BSI (TR-02102-2).

Allerdings scheint es mit dem Flag no-sha1 zu Problemen beim Kompilieren zu kommen:

```
-DWHIRLPOOL_ASM -DGHASH_ASM   -c -o s3_cbc.o s3_cbc.c
s3_cbc.c: In function ‘ssl3_cbc_digest_record’:
s3_cbc.c:484:4:   warning:   implicit   declaration   of   function
‘SHA1_Init’ [-Wimplicit-function-declaration]
    SHA1_Init((SHA_CTX*)md_state.c);
    ^
s3_cbc.c:486:68: error: ‘SHA1_Transform’ undeclared (first use in
this function)
       md_transform  =  (void(*)(void  *ctx,  const  unsigned  char
*block)) SHA1_Transform;
```

```
^
s3_cbc.c:486:68: note: each undeclared identifier is reported only
once for each function it appears in
```

In der entsprechenden `s3_cbc.c` Datei wird trotz gesetztem Flag versucht, SHA1 zu benutzen. Da die entsprechende Implementierung allerdings durch das `no-sha1` Flag nicht vorhanden ist, kommt es zum Kompilier-Fehler. Es wurde nicht geprüft, ob das Flag in der aktuellsten OpenSSL Version korrekt implementiert wurde.

3 Analyse der RSA-Schlüsselerzeugung

Untersucht wird die Funktion zur Schlüsselerzeugung der Standard-RSA-Implementierung crypto/rsa/rsa_eay.c: rsa_pkcs1_eay_meth.

3.1 Primzahlerzeugung

Die Primzahlerzeugung erfolgt mit dem Algorithmus 4.44: „Random search for a prime using the Miller-Rabin test“ mit inkrementeller Suche gemäß 4.51(i,ii): „incremental search“ aus [HAC].

Funktion BN_generate_prime_ex() (crypto/bn/bn_prime.c):

1. Erzeugen einer zufälligen Zahl a der Länge „B“ (in Bits) (mit BN_rand(a, B, 1, 1)). B ist durch den verwendeten Datentypen für den Parameter „int“ plattformabhängig auf INT_MAX beschränkt,

   a) Entnahme von rng-Daten aus dem konfigurierten RNG (sofern nicht explizit anders konfiguriert, ist dies der OpenSSL Standard-RNG) für eine Zahl der Länge b Bits. Es werden immer ganze Bytes entnommen (mit RAND_bytes()). Wenn die gewünschte Bitlänge nicht durch 8 teilbar ist, werden die nicht verwendeten Bits in Unterschritt c) auf 0 gesetzt.

   b) Setzen der beiden höchstwertigsten Bits „11“ und des niedrigwertigsten Bit „1“. Dies hat zur Folge, dass ein Produkt zweier solcher Zahlen immer die Länge der Summe ihre Länge in Bit hat sowie, dass die Zahl ungerade ist. Das höchstwertigste Bit einer solchen Zahl muss immer 1 sein, da sonst die Bitlänge nicht erreicht wird. Das niedrigste Bit einer Primzahl ist ebenfalls immer 1. Die Kenntnis des zweithöchsten Bits reduziert den Aufwand des Angreifers um maximal ein Bit.

   c) Setzen der führenden, leeren Bits auf „0“. Dies ist dem Umstand geschuldet, dass evtl. nicht alle Bits des führenden Bytes benötigt werden.

   d) Konvertierung in BN (BN_bin2bn(), diese Funktion wurde nicht geprüft)

2. Die so erzeugte Zahl wird zwei Tests unterzogen. Wenn ein Test fehlschlägt, setze a=a+2 und teste erneut. Die Erhöhung findet maximal $2^{64}-1-17863$ mal statt, dann wird wieder mit Schritt 1. gestartet.

   a) Teilbarkeit durch die ersten 2048 Primzahlen p_i außer 2 (3...17863). Es wird der Reihe nach, mit der kleinsten Zahl angefangen, durch die Primzahlen geteilt. Sobald Teilbarkeit gegeben ist wird der Kandidat sofort um 2 erhöht und erneut getestet.

   b) gcd(a-1, p_i) ==1. Aus dem Quelltext geht nicht hervor, warum dieser Test durchgeführt wird. Eventuell ist das Ziel, Cycling Attacks auszuschließen. Derzeit wird aber davon

ausgegangen, dass Cycling Attacks nicht praktikabel sind.

3. Miller-Rabin-Primzahltest von a mit 2 Durchläufen, sofern a mehr als 1300 Bits hat (BN_is_prime_fasttest_ex()). Die Basis für den MRPT wird im Interval [1...a] mittels BN_pseudo_rand_range() gewählt. Die Worst-Case-Wahrscheinlichkeit beträgt somit für Zahlen mit mehr als 1300 Bits 2^{-4}, für Zahlen zwischen 1000 und 1300 Bits 2^{-6}. Realistische Wahrscheinlichkeiten ergeben sich nach [HAC]: Fact 4.48 zu $<2^{-90}$ für 1024 und 1536 Bits, sowie $<2^{-106}$ für 2048 Bits. Für Zahlen mit mehr als 1854 Bits ist die Wahrscheinlichkeit für Zusammengesetztheit gemäß [HAC]: Fact 4.48(ii) immer geringer als 2^{-100}, da der Test dann konstant mit 2 MRPT-Durchläufen durchgeführt wird und die Wahrscheinlichkeit für Zusammengesetztheit mit größer werdenden Kandidaten geringer wird.

Bemerkung 1: Die Schritte in 1. und 2. werden in der Funktion crypto/bn/bn_prime.c:probable_prime() durchgeführt. Diese Funktion erzeugt evtl. eine Zahl, die mehr Bits hat als angefordert. Dies ist z.B. der Fall, wenn das erste gewählte $a=2^{bits}-1$ ist und bei der inkrementellen Suche dann größer wird als 2^{bits}.

Empfehlung 1: Die Wahrscheinlichkeit ist bei großen Zahlen vernachlässigbar klein (bei n Bits: $p<2^{-(n-18)}$).

Bemerkung 2: Die Kenntnis des zweiten Bits in Schritt 1b) reduziert den Aufwand des Angreifers um maximal 50%. Sollte es Angriffe geben, die einen größeren Vorteil aus Kenntnis des Bits erzielen können, so können diese durch Raten dieses Bits ebenfalls durchgeführt werden, und die Kenntnis des Bits verringert den Aufwand ebenfalls um maximal 50%. Das zweithöchste Bit wird gesetzt, damit das Produkt zweier Primzahlen garantiert die Länge der Summe der Primzahllänge entspricht.

Empfehlung 2: Das Setzen des zweiten Bits sollte unterbleiben, und stattdessen bei der Schlüsselerzeugung später überprüft werden, ob die erzeugten Primzahlen notwendige Anforderungen erfüllen (Wie z.B. bei RSA, dass der erzeugte Modulus pq die geforderte Länge hat).

Bemerkung 3: Die Anzahl von Miller-Rabin-Tests ist deutlich geringer als die in TR-02102-1 für besonders sicherheitskritische Schlüssel notwendigen 50 Iterationen.

Empfehlung 3: Erhöhung der Iterationen des Miller-Rabin-Tests gemäß der Anforderungen für den Fall, dass die Primzahlkandidaten gleichverteilt sind, d.h. 4 Durchläufe bis 1232 Bits, 3 Durchläufe bis 1853 Bits und für größere Zahlen 2 Durchläufe. Für Langzeitschlüssel sollten 50 Durchläufe erfolgen. Dies kann am einfachsten in der Datei crypto/bn/bn.h in der Definition BN_prime_checks_for_size() erfolgen.

Es wird nicht überprüft, ob die erzeugten Zahlen „sichere“ Primzahlen sind. Sofern die Primzahl aber zufällig gewählt wurde, kann angenommen werden, dass die Bedingung erfüllt ist. (s. [HAC] Note 8.8iii, sowie [BNetzA] Abs. 3.1, Bemerkung 1).

3.2 Schlüsselerzeugung

Die Schlüsselerzeugung für RSA-Schlüssel erfolgt nach folgendem Schema (crypto/rsa/rsa_gen.c: rsa_builtin_keygen()):

Eingabe:

- Größe des öffentlichen Modulus in Bits B
- Verschlüsselungs-Exponent e. Die Größe und der Wert von e kann beliebig vom Benutzer gewählt werden.

Algorithmus:

1. Erzeuge Primzahl a gemäß Abs. 3.1. Die Länge von a in Bits wird auf $\lfloor (B+1)/2 \rfloor$ gesetzt.
2. Teste, ob a-1 teilerfremd zu e ist, sonst zurück zu 1). Der Sonderfall gemäß Bemerkung 1 wird nicht geprüft.
3. Erzeuge Primzahl b gemäß Abs. 3.1. Die Länge von b in Bits wird auf $\lfloor B/2 \rfloor$ gesetzt. b wird unabhängig von a gewählt.
4. Test, ob $b \neq a$ ist, sonst zurück zu 3)
5. Test, ob b-1 teilerfremd zu e ist, sonst zurück zu 3)
6. Speichern der größeren Zahl von (a,b) als p, die kleinere Zahl als q
7. Berechne öffentlichen Modulus $n = p \cdot q$
8. Berechne $d = e^{-1} \bmod (p-1)(q-1)$ mit Erweitertem Euklidischem Algorithmus (BN_mod_inverse())
9. Erzeuge Werte für spätere Berechnungen mit dem Chinese Remainder Theorem:
   - $dmp1 = d \bmod (p-1)$
   - $dmq1 = d \bmod (q-1)$
   - $iqmp = q^{-1} \bmod p$

   Diese Werte werden genau so geschützt wie der geheime Exponent d. Berechnungen mit dem Chinese Remainder Theorem erfolgen in crypto/rsa/rsa_eay.c:RSA_eay_mod_exp().

Da Seitenkanalangriffe auf RSA mit dem CRT-Algorithmus existieren. Es muss untersucht werden, ob OpenSSL RSA mit dem CRT-Algorithmus in einer angreifbaren Variante nutzt. Diese Untersuchung ist Teil von Kap. 6.

3.2.1 Kriterien nach [TR021021] Kap. 3.5: Schlüsselgenerierung

1. Wähle zwei Primzahlen p und q zufällig und unabhängig voneinander unter der Nebenbedingung $0.1<|\log_2 p-\log_2 q|<30p$ und q werden zufällig und unabhängig voneinander gesucht. Die Nebenbedingung wird mit großer Wahrscheinlichkeit erfüllt, aber nicht garantiert.
   - $2^{n+1}+2^n+1<p<2^{n+1}-1$

     $$0<|\log_2 p-\log_2 q|<\log_2\left(\frac{2^{n+1}-1}{2^n+2^{n-1}+1}\right)\approx\log_2\left(\frac{2^{n+1}}{2^n+2^{n-1}}\right)=\log_2\left(\frac{4}{3}\right)\approx 0{,}415$$

2. Bei der empfohlenen Schlüssellänge von 2000 Bit (s.u.), wähle den öffentlichen Exponenten $e\in\mathbb{N}$ unter den Nebenbedingungen

   $$\mathrm{ggT}(e,(p-1)\cdot(q-1))=1 \quad \text{und} \quad 2^{16}+1\le e\le 2^{1824}-1.$$

   - ggT wird erfüllt, ansonsten kann der Entschlüsselungsexponent nicht gefunden werden. Der Nutzer legt vor der Erzeugung des Schlüssels den Verschlüsselungsexponenten e fest. Die Bedingung für den Wertebereich von e wird von OpenSSL nicht erzwungen.

3. Berechne den geheimen Exponenten $d\in\mathbb{N}$ in Abhängigkeit von e unter der Nebenbedingung

   $$e\cdot d=1 \bmod \mathrm{kgV}(p-1,q-1)$$

   - Diese Bedingung wird erfüllt.

3.2.2 Kriterien nach [BNetzA]

1. Absatz 3.1: Die Primfaktoren p und q von n sollten die gleiche Größenordnung haben, aber nicht zu dicht beieinander liegen.
   - s. voriger Abschnitt
2. Aus Absatz 7.3: Es ist vorgesehen, beim Einsatz von RSA-Signaturen die Verwendung von öffentlichen Exponenten, die kleiner als 2^{16} oder größer als 2^{256} sind, ab dem Jahr 2021 nicht mehr zu gestatten.
   - s. voriger Abschnitt
3. Insbesondere muss bei Nutzung eines probabilistischen Primzahltests mit hinreichender Wahrscheinlichkeit ausgeschlossen sein, dass p oder q in Wirklichkeit zusammengesetzte Zahlen sind. Als obere Schranke für diese Wahrscheinlichkeit wird der Wert 2^{-100} empfohlen.
   - s. voriger Abschnitt

Während der Schlüsselerzeugung wird der RNG nicht mit frischen Daten geseedet.

3.3 Cleanup/Sicheres Löschen

RSA_free()

- Bei RSA Objekten wird für die Speicherverwaltung ein Referenz-Zähler genutzt. Beim Erzeugen eines RSA-Objekts wird dieser mit 1 initialisiert. Wenn beispielsweise ein Schlüssel kopiert werden soll, wird eine Referenz auf das Objekt erzeugt, statt alle Objekteigenschaften zu kopieren. Beim Aufruf von RSA_free() wird dieser Referenzzähler dekrementiert, und wenn der Referenzzähler 0 ist, werden die Daten entsprechend dem nächsten Punkt freigegeben.
- BN_clear_free führt für die Werte n,e,d,p,q,dmp1, dmq1, iqmp des RSA-Schlüssels das sichere Überschreiben mittels OPENSSL_cleanse aus

Bemerkung 4: bn_expand2() überschreibt alte Daten nicht, wenn die Zahl größer wird.

Empfehlung 4: Dies stellt kein Problem dar, da die geheimen Daten (d, p, q, dmp1, dmq1, iqmp) während deren Lebenszeit nicht in der Größe verändert werden.

3.4 Fazit

Unter der Annahme, dass RAND_bytes() sichere Zufallsdaten liefert, wird der grundsätzliche Algorithmus für die Primzahlerzeugung von uns als sicher angesehen. Das Erzeugen der Schlüssel ist nicht zeitkonstant, da die Primzahlkandidaten zufällig gewählt werden und daher keine Garantien gegeben werden können, wie lange die Schlüsselerzeugung dauert. Allerdings sehen wir dieses Problem nicht kritisch, da die Primzahlerzeugung in TLSv1.2 (RSA Schlüsselerzeugung, DH Parametererzeugung) normalerweise offline erfolgt, und ein Angreifer keinen Zugriff auf den Erzeugungsprozeß hat. Die Anzahl der Miller-Rabin-Testläufe ist niedrig gehalten. Gerade weil die Schlüssel offline erstellt werden, wäre eine höhere Sicherheit an dieser Stelle dem Geschwindigkeitsverlust bei der Erzeugung vorzuziehen, um zusätzliche Sicherheit zu erlangen (z.B. gemäß [TR021021]).

4 Zertifikatskettenprüfung bei TLS-Handshake

In diesem Arbeitspaket wird die Zertifikatskettenprüfung in OpenSSL analysiert. Dazu wird zunächst erläutert, mit welchen Methoden ein TLS-Handshake konfiguriert und initiiert werden kann. Im weiteren wird vertiefend auf die Konfigurierbarkeit der Zertifikatskettenprüfung eingegangen. Bevor dann die Zertifikatskettenprüfung in OpenSSL analysiert wird, wird zunächst der Algorithmus aus RFC 5280 [RFC5280] näher erläutert. Im Anschluss wird die Konformität des in OpenSSL implementierten Algorithmus mit dem Algorithmus aus RFC 5280 bewertet. Testergebnisse einer Analyse mittels x509test werden daraufhin erläutert. Abschließend wird ein Fazit der Analyse der Zertifikatskettenprüfung in OpenSSL gezogen.

4.1 Peer Authentication

Die TLS Peer Authentication in OpenSSL beinhaltet das Behandeln von Trusted Certificates, der Verifikation von Zertifikatsketten, der Nutzung von Zertifikatssperrlisten und von Post-connection Verifikation [NETSEC]. Die Zertifikatsprüfung in OpenSSL erfolgt, wie in AP1 genannt, grundsätzlich gegen eine Zertifikatsdatenbank, die entsprechende Struktur dafür ist X509_STORE_CTX.

Vor der Verifikation von Zertifikatsketten müssen zunächst Trusted Certificates festgelegt werden. Dafür gibt es in OpenSSL mehrere Funktionen: SSL_CTX_load_verify_locations() lädt ein oder mehrere Trusted Certificates aus einer angegeben Datei im PEM-Format oder einem angegebenen Verzeichnis, während SSL_CTX_set_default_verify_paths() alle Zertifikate aus den OpenSSL Default Certificate Locations zu Trusted Certificates macht. In Ubuntu 14.04 sind das alle Zertifikate im Verzeichnis /etc/ssl/certs, insgesamt 493 Dateien.

Die Verifikation einer Zertifikatskette findet im TLS-Handshake im Falle eines TLS-Servers beim Aufruf von SSL_accept() und im Falle eines TLS-Clients beim Aufruf von SSL_connect() statt. Im Umfeld eines Webservers kommt üblicherweise nur eine TLS-Serverauthentifizierung zum Einsatz. Eine TLS-Clientauthentifizierung findet nur bei bestimmten Webanwendungen statt. OpenSSL bietet eine eingebaute Routine zur Verifikation von Zertifikatsketten während des TLS-Handshakes, die im Folgenden Gegenstand der Untersuchung sein wird, jedoch durch Aufruf von SSL_CTX_set_cert_verify_callback() durch eine eigene Routine ersetzt werden kann. Dies wird jedoch ausdrücklich nicht empfohlen. Mittels SSL_CTX_set_verify() kann jedoch eine Filterfunktion registriert werden, die den Rückgabewert der eingebauten Verifikationsroutine filtert und einen neuen Verifikationsstatuswert zurückgibt [VFY]. Außerdem wird SSL_CTX_set_verify() dazu genutzt den Verifikationsmodus festzulegen. Dabei können vier Flags per bitweisem Oder miteinander kombiniert werden:

SSL_VERIFY_NONE

Bedeutung für TLS-Server: Es wird kein Zertifikat beim TLS-Client angefordert und TLS-Clients sollten kein Zertifikat senden.

Bedeutung für TLS-Client: Das Zertifikat des TLS-Servers wird verifiziert. Ein Fehler bei der Verifikation führt jedoch nicht zum Abbruch des Handshakes. Ein Clientprogramm würde über die Filterfunktion über den Fehler informiert, sollte es eine Filterfunktion registriert haben.

Die Dokumentation weist darauf hin das alle anderen Flags Vorrang vor diesem Flag haben, daher sollte dieses Flag nicht mit anderen Flags kombiniert werden.

SSL_VERIFY_PEER

Bedeutung für TLS-Server: Es wird ein Zertifikat beim TLS-Client angefordert. Der TLS-Client kann die Anforderung ignorieren. Wenn er jedoch ein Zertifikat sendet wird es verifiziert. Schlägt die Verifikation fehl, führt dies zum Abbruch des Handshakes.

Bedeutung für TLS-Client: Sendet der TLS-Server ein Zertifikat, wird es verifiziert. Schlägt die Verifikation fehl, führt dies zum Abbruch des Handshakes. Sendet der TLS-Server kein Zertifikat, weil eine Anonymous Cipher eingesetzt wird, wird dieses Flag ignoriert. Bei einer Anonymous Cipher findet keine TLS-Authentisierung statt.

SSL_VERIFY_FAIL_IF_NO_PEER_CERT

Bedeutung für TLS-Server: Es wird ein Zertifikat beim TLS-Client angefordert. Sendet der TLS-Client kein Zertifikat, führt dies zum Abbruch des Handshakes. Dieses Flag kann nur zusammen mit dem Flag *SSL_VERIFY_PEER* eingesetzt werden.

Bedeutung für TLS-Client: ignoriert

SSL_VERIFY_CLIENT_ONCE

Bedeutung für TLS-Server: Es wird ein Zertifikat beim TLS-Client nur während des initialen Verbindungsaufbaus angefordert. Im Falle einer TLS Renegotiation wird kein Zertifikat beim TLS-Client angefordert. Dieses Flag kann nur zusammen mit dem Flag *SSL_VERIFY_PEER* eingesetzt werden.

Bedeutung für TLS-Client: ignoriert

Mit SSL_CTX_set_verify_depth() kann die maximal erlaubte Länge einer Zertifikatskette konfiguriert werden. Diese gibt an, wie viele Zertifikate, ausgehend vom Peer-Zertifikat, verifiziert werden dürfen bis ein Trusted Certificate erreicht ist. Wird dieser Wert bei der Verifikation überschritten, schlägt die Verifikation fehl.

Das Prüfen des Zertifikats der Gegenstelle gegen eine Zertifikatssperrliste (Certificate Revocation List – CRL) muss explizit aktiviert werden. Dazu wird X509_VERIFY_PARAM_set_flags() mit dem Flag X509_V_FLAG_CRL_CHECK aufgerufen und dann mittels SSL_CTX_set1_param() an

den TLS-Handshake gebunden.

Nach der Zertifikatskettenprüfung im Rahmen von SSL_connect() bzw. SSL_accept() können Post-connection Checks durchgeführt werden, die über die reine Zertifikatskettenprüfung hinausgehen. Sendet eine Gegenstelle beispielsweise kein Zertifikat, während zwar eines angefordert aber nicht verpflichtend gefordert war, wird die Verifikationsroutine keinen Fehler zurückgeben. In diesem Fall liefert die Funktion SSL_get_peer_certificate() NULL zurück, was dann als Abbruchkriterium genutzt werden kann. Weiterhin sollte beispielsweise nach Aufruf von SSL_connect() bzw. SSL_accept() eine Verifikation des Hostnamens (Hostname Verification) durchgeführt werden. Andernfalls sind einfache Man-in-the-middle Angriffe möglich, da einfach jedes Peer-Zertifikat einer gültigen Zertifikatskette akzeptiert wird. Empfehlungen zur Umsetzung von Hostname Verification werden in RFC 6125 [RFC6125] gegeben. Sind alle Checks durchgeführt worden, muss mittels SSL_get_verify_result() das Ergebnis der Zertifikatskettenprüfung abgefragt werden (wenn eine Filterfunktion gesetzt worden ist, geht auch ihr Rückgabewert in diesen Ergebniswert ein). Im Erfolgsfall liefert die Funktion X509_V_OK zurück. Damit ist die TLS-Verbindung aufgebaut und bereit zur Übertragung der verschlüsselten Nutzdaten.

Bemerkung: Hostname Verification ist ein kritischer Bestandteil einer Zertifikatskettenprüfung. Das Unterlassen der Prüfung des Hostnames ist ein verbreitetes Problem [ZDNET]. In OpenSSL findet jedoch keine Hostname Verification statt. Es wird derzeit auch keine API angeboten, mit der ein Anwendungsentwickler eine RFC 6125 konforme Hostname Verification durchführen kann, obwohl die Dokumentation [HOST] eine solche API ausweist. Einträge im Changelog von OpenSSL legen nahe, dass diese API in Version 1..0.2 eingeführt werden soll.

4.2 Konfiguration

Wie bereits beschrieben, erfolgt die Zertifikatsprüfung in OpenSSL grundsätzlich gegen eine Zertifikatsdatenbank. Eine Zertifikatsdatenbank ist in OpenSSL durch die Struktur X509_STORE_CTX abgebildet. Auch die Routine die die eigentliche Prüfung durchführt erhält als einzigen Parameter eine Zertifikatsdatenbank und arbeitet anschließend auf dieser [VFY-CERT]:

int X509_verify_cert(X509_STORE_CTX*ctx)

Member von X509_STORE_CTX	**Beschreibung (falls vorhanden)**
X509_STORE *ctx	Enthält Tabellen und weiteres zur Verifikation
int current_method	Zum Nachschlagen von Zertifikaten benutzt
X509 *cert	Das zu prüfende Zertifikat, siehe X509_STORE_CTX_set_cert()
STACK_OF(X509) *untrusted	Eine Kette von Untrusted Certificates, die zum Zusammensetzen der Zertifikatskette benötigt werden, siehe X509_STORE_CTX_set_chain()
STACK_OF(X509_CRL) *crls	Eine Menge von CRLs, siehe X509_STORE_CTX_set0_crls()
X509_VERIFY_PARAM *param	Parameter für die Verifikation
void *other_ctx	Eine Menge von Trusted Certificates, siehe X509_STORE_CTX_trusted_stack()
Callbacks für verschiedene Operationen	
int (*verify)(X509_STORE_CTX *ctx);	Funktion, die zur Verifikation eines Zertifikats aufgerufen wird
int (*verify_cb)(int ok,X509_STORE_CTX *ctx)	Callback-Funktion, die während der Verifikation eines Zertifikats aufgerufen wird (Filterfunktion), siehe Mittels SSL_CTX_set_verify()
int (*get_issuer)(X509 **issuer, X509_STORE_CTX *ctx, X509 *x)	Funktion, die das Issuer Zertifikat zu Zertifikat *x* in der Datenbank bestimmt
int (*check_issued)(X509_STORE_CTX *ctx, X509 *x, X509 *issuer)	Funktion, die bestimmt ob das Zertifikat *x* von *issuer* ausgestellt wurde
int (*check_revocation)(X509_STORE_CTX *ctx)	Funktion, die den Revocation Status der Zertifikatskette bestimmt
int (*get_crl)(X509_STORE_CTX *ctx, X509_CRL **crl, X509 *x)	Funktion, die die zu Zertifikat *x* gehörende CRL bestimmt
int (*check_crl)(X509_STORE_CTX *ctx, X509_CRL *crl)	Funktion, die die CRL *crl* auf Gültigkeit prüft
int (*cert_crl)(X509_STORE_CTX *ctx, X509_CRL *crl, X509 *x)	Funktion, die Zertifikat *x* gegen die CRL *crl* prüft
int (*check_policy)(X509_STORE_CTX *ctx)	Funktion, die die Certificate Policy prüft

STACK_OF(X509) * (*lookup_certs) (X509_STORE_CTX *ctx, X509_NAME *nm)	Funktion, die Zertifikate auf die *nm* passt bestimmt
STACK_OF(X509_CRL) * (*lookup_crls) (X509_STORE_CTX *ctx, X509_NAME *nm)	Funktion, die CRLs auf die *nm* passt bestimmt
int (*cleanup)(X509_STORE_CTX *ctx)	Funktion, die die Datenstruktur freigibt
Die folgenden Werte werden zusammengesetzt	
int valid	Bedeutung unklar
int last_untrusted	Index des letzten Untrusted Certificate in der Kette
STACK_OF(X509) *chain	Zertifikatskette – wird zusammengesetzt und dann verifiziert
X509_POLICY_TREE *tree	Valid Policy Tree
int explicit_policy	Wert von requireExplicitPolicy
Im Fehlerfall gesetzte Werte	
int error_depth	
int error	Fehlercode
X509 *current_cert	Zertifikat, für das die Verifikation fehlgeschlagen ist
X509 *current_issuer	Issuer Zertifikat des Zertifikats, für das die Verifikation fehlgeschlagen ist
X509_CRL *current_crl	CRL des Zertifikats, für das die Verifikation fehlgeschlagen ist
int current_crl_score	Score der CRL des Zertifikats, für das die Verifikation fehlgeschlagen ist
unsigned int current_reasons	CRL Reason Mask
X509_STORE_CTX *parent	Für CRL Path Validation: Parent Context
CRYPTO_EX_DATA ex_data	Anwendungsspezifische Daten, siehe X509_STORE_CTX_set_ex_data()

Tabelle 1: Beschreibung der Struktur X509_STORE_CTX

Mit den in AP1, Abschnitt 1.4.6.1 genannten Funktionen kann die Zertifikatsdatenbank konfiguriert werden. Die interne Struktur der Zertifikatsdatenbank ist in Tabelle 1 dargestellt. X509_STORE_CTX_init() initialisiert eine Zertifikatsdatenbank mit einer Menge von Trusted Certificates, dem zu prüfenden Zertifikat und einer Menge von zusätzlichen Zertifikaten, die unter Umständen benötigt werden um eine Zertifikatskette zu erzeugen. X509_STORE_CTX_trusted_stack() legt eine Menge von Zertifikaten als Trusted Certificates fest. X509_STORE_CTX_set_cert() legt das zu prüfende Zertifikat fest. X509_STORE_CTX_set_chain() legt eine Menge von zusätzlichen Zertifikaten, die unter Umständen benötigt werden um eine Zertifikatskette zu erzeugen, fest. Mit X509_STORE_CTX_set0_crls() werden eine Menge von Zertifikatssperrlisten (Certificate Revocation Lists – CRLs) zur Datenbank hinzugefügt. Diese werden aber nur ausgewertet wenn mittels einer X509_VERIFY_PARAM Struktur und der Funktion X509_STORE_CTX_set0_param() die CRL-Prüfung explizit aktiviert worden ist.

Weitere Parameter für die Zertifikatskettenprüfung können mittels eines X509_VERIFY_PARAM gesetzt werden [FLAGS]. Die Funktion X509_VERIFY_PARAM_set_flags() setzt dabei bestimmte Verifikationsflags, die dann mit den bereits gesetzten Flags per bitweisem Oder verknüpft werden. Die unterstützten Flags sind in Tabelle 2 aufgeführt.

Flag	Bedeutung
X509_V_FLAG_CRL_CHECK	Aktiviert die CRL-Prüfung für das zu prüfende Zertifikat
X509_V_FLAG_CRL_CHECK_ALL	Aktiviert die CRL-Prüfung für die gesamte Zertifikatskette
X509_V_FLAG_IGNORE_CRITICAL	Deaktiviert die Prüfung von X.509 Critical Extensions
X509_V_FLAG_X509_STRICT	Deaktiviert Workarounds für einige defekte Zertifikate und wendet eine strikte Prüfung nach X.509 an
X509_V_FLAG_ALLOW_PROXY_CERTS	Aktiviert die Prüfung von Proxy-Zertifikaten
X509_V_FLAG_POLICY_CHECK	Aktiviert die Prüfung von Certificate Policies
X509_V_FLAG_EXPLICIT_POLICY	Setzt das „require explicit policy“ Flag nach RFC 3280 [RFC3280] (aktiviert implizit die Prüfung von Certificate Policies)
X509_V_FLAG_INHIBIT_ANY	Setzt das „inhibit any policy“ Flag nach RFC 3280 (aktiviert implizit die Prüfung von Certificate Policies)
X509_V_FLAG_INHIBIT_MAP	Setzt das „inhibit policy mapping“ Flag nach RFC 3280 (aktiviert implizit die Prüfung von Certificate Policies)
X509_V_FLAG_NOTIFY_POLICY	Ist die Prüfung von Certificate Policies erfolgreich, wird der Callback-Funktion (Filterfunktion) ein spezieller Statuscode übergeben, so dass dort ggf. weitere Policy Checks durchgeführt werden können
X509_V_FLAG_EXTENDED_CRL_SUPPORT	Aktiviert zusätzliche Funktionen wie Indirect CRLs und CRLs die mit anderen Schlüsseln unterschrieben wurden
X509_V_FLAG_USE_DELTAS	Aktiviert die Prüfung gegen Delta CRLs
X509_V_FLAG_CHECK_SS_SIGNATURE	Aktiviert die Prüfung der Signatur des Root CA Zertifikats
X509_V_FLAG_CB_ISSUER_CHECK	Aktiviert das Debuggen von Certificate Issuer Checks, bspw. zur Protokollierung der Prüfung

Tabelle 2: Unterstützte Flags bei Aufruf von X509_VERIFY_PARAM_set_flags()

Mit der Funktion X509_VERIFY_PARAM_get_flags() können die gesetzten Flags abgerufen werden, während X509_VERIFY_PARAM_clear_flags() alle zuvor gesetzten Flags zurücksetzt.

X509_VERIFY_PARAM_set_purpose() setzt den erwarteten Key Purpose des zu prüfenden Zertifi-

kats auf den angegebenen Wert, bspw. SSL Client oder SSL Server (id-kp-serverAuth oder id-kp-clientAuth). X509_VERIFY_PARAM_set_trust() setzt die Trust Settings der Root CA. Trust Settings sind weder Teil des RFC 5280 noch der darunterliegenden Standards, sondern sind ein OpenSSL-eigenes Konstrukt [TRUST]. Mit X509_VERIFY_PARAM_set_time() kann die Systemzeit zum Zeitpunkt der Verifikation gesetzt werden, falls das Zertifikat nicht zum aktuellen Zeitpunkt geprüft werden soll. X509_VERIFY_PARAM_add0_policy() aktiviert die Prüfung von Certificate Policies und fügt die angegebene Policy zur Menge von akzeptierten Policies hinzu. Alle zuvor hinzugefügten Policies werden dabei ersetzt. X509_VERIFY_PARAM_set_depth() setzt die maximal erlaubte Länge einer Zertifikatskette. Diese gibt an, wie viele Zertifikate, ausgehend vom Peer-Zertifikat, verifiziert werden dürfen bis ein Trusted Certificate erreicht ist (siehe auch SSL_CTX_set_verify_depth(), Abschnitt 4.1).

4.3 Zertifikatskettenprüfung nach RFC 5280

RFC 5280 spezifiziert neben dem Zertifikatsprofil und dem CRL-Profil für die Internet X.509 PKI (PKIX) auch einen Algorithmus zur Zertifikatskettenprüfung solcher Zertifikate (s.RFC 5280, Kap. 6). Dieser Algorithmus wird im Folgenden beschrieben. RFC 5280 erfordert nicht, dass konforme Implementierungen *diesen* Algorithmus implementieren, sie müssen aber einen Algorithmus implementieren der am Ende das korrekte Resultat liefert (s. RFC 5280, Kap. 6, Absatz 2).

Eingabe:

(a) Zertifikatskette der Länge n – wenn der Trust Anchor ein self-signed Zertifikat ist, dann ist es nicht Teil der Zertifikatskette

(b) Aktuelles Datum/Uhrzeit

(c) Eine Menge von Certificate Policy Identifiern, die die akzeptierten Policies des Nutzers des Zertifikats angeben

(d) Informationen über den Trust Anchor – eine CA die als Trust Anchor für die gegebene Zertifikatskette gilt (kann ein self-signed Zertifikat sein)

(e) Die Angabe ob Policy Mapping in der Zertifikatskette erlaubt ist

(f) Die Angabe ob die Zertifikatskette für mindestens eine der Policies aus (c) gültig sein muss

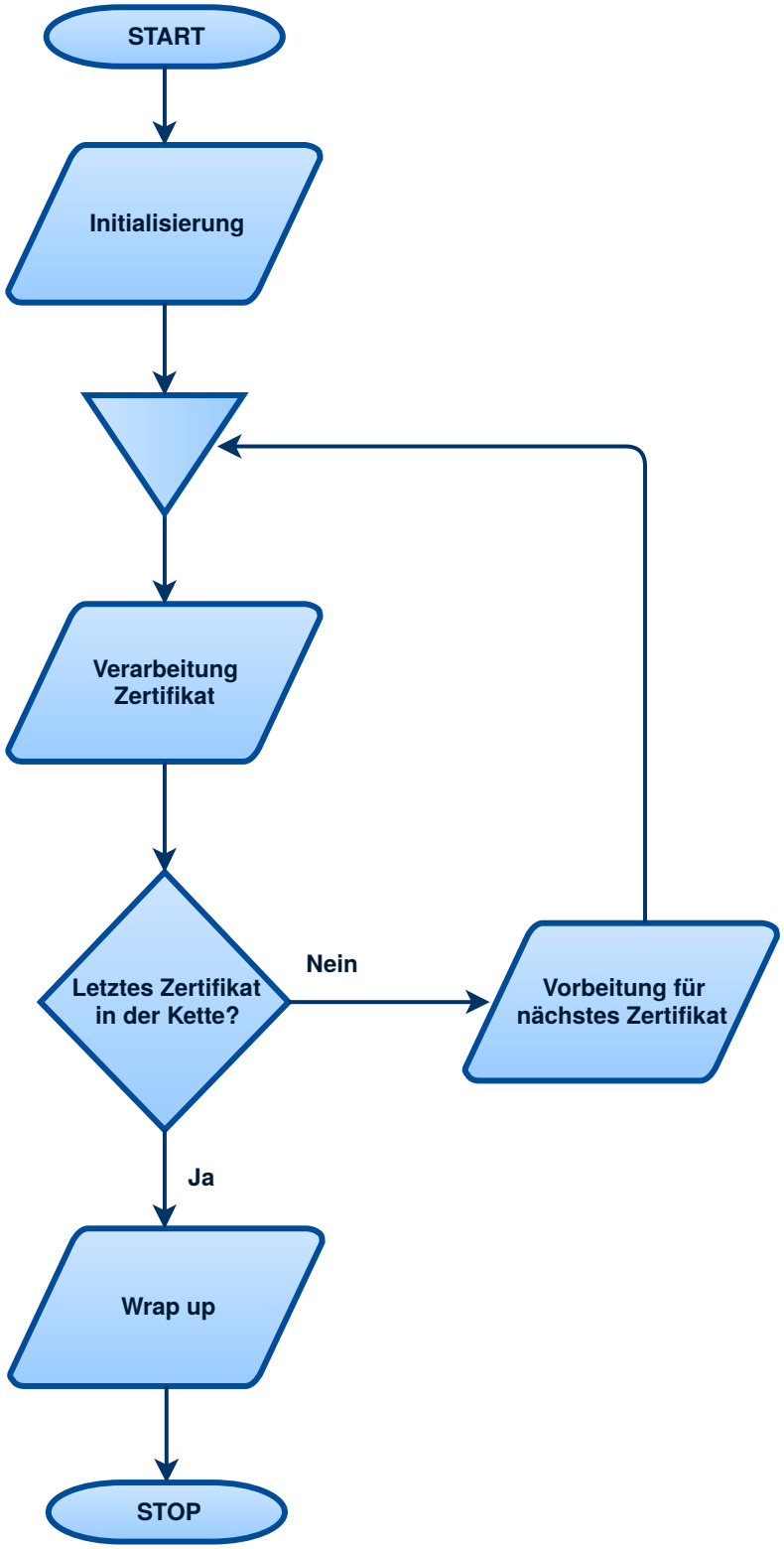
(g) Die Angabe ob die OID *anyPolicy* bei der Prüfung berücksichtigt werden soll

(h) Eine Menge von Subtrees in die alle Subject Names in allen Zertifikaten der Zertifikatskette fallen müssen (Permitted Subtrees)

(i) Eine Menge von Subtrees in die alle Subject Names in allen Zertifikaten der Zertifikatskette nicht fallen dürfen (Excluded Subtrees)

Ausgabe:

SUCCESS, im Falle einer erfolgreichen Zertifikatskettenprüfung, sonst FAIL



Der Algorithmus besteht im Wesentlichen aus vier Schritten, wie in Abbildung 1 dargestellt. Im ers-

ten Schritt erfolgt die Initialisierung des Algorithmus. Im zweiten Schritt folgt die grundlegende Zertifikatsverarbeitung (Basic Certificate Processing). Im dritten Schritt folgen die Vorbereitungen zur Verarbeitung des jeweils nächsten Zertifikats. Im vierten und letzten Schritt werden abschließende Arbeiten durchgeführt (Wrap up). Schritt 2 wird für jedes Zertifikat in der Kette durchgeführt, während Schritt 3 für jedes Zertifikat in der Kette außer dem letzten Zertifikat durchgeführt wird. Die Schritte 1 und 4 werden nur genau einmal durchgeführt.

Die Schritte Initialisierung, Verarbeitung, Vorbereitungen zur Verarbeitung des jeweils nächsten Zertifikats und Wrap up werden im Folgenden beschrieben.

4.3.1 Initialisierung

In der Initialisierungsphase des Algorithmus werden sieben Zustandsvariablen gesetzt, basierend auf den gegebenen Eingabewerten. Die Initialisierung der Zustandsvariablen wird hier nicht näher erläutert, stattdessen wird im nächsten Abschnitt *Verarbeitung* immer auch der erste Durchlauf des Algorithmus betrachtet (*i=1*).

4.3.2 Verarbeitung

Bei der Verarbeitung eines Zertifikats *i* aus der Menge der Zertifikate *[1..n]* werden folgende Schritte durchgeführt:

(a) Grundlegende Zertifikatsinformationen werden geprüft. Es muss die folgenden Kriterien erfüllen:

  (1) Die Signatur des Zertifikats *i* kann erfolgreich verifiziert werden. Der zur Verifikation genutzte Schlüssel ist beim ersten zu prüfenden Zertifikat (*i=1*) der Schlüssel des Trust Anchors, sonst der des vorherigen Zertifikats *i-1*.

  (2) Der Gültigkeitszeitraum des Zertifikats enthält den aktuellen Zeitpunkt.

  (3) Das Zertifikat wurde nicht zurückgezogen. Dies kann durch Heranziehen der jeweiligen CRL oder durch andere Mechanismen (Out of Band) bestimmt werden.

  (4) Der Herausgeber (Issuer Name) des Zertifikats *i* stimmt mit dem Antragssteller (Subject Name) des vorherigen Zertifikats *i-1* überein. Im Falle *i=1* wird der Antragssteller des Trust Anchors verwendet.

(b) Ist das Zertifikat ein self-issued Zertifikat und ist es nicht das letzte Zertifikat in der Kette (*i=n*), überspringe diesen Schritt. Sonst stelle sicher, dass der Antragssteller (Subject Name und Alternative Names) innerhalb der Permitted Subtrees für dieses Zertifikat liegt (falls Name Constraints Extension vorhanden und Feld permittedSubtrees gesetzt). Falls das vorherige Zertifikat *i-1* ebenfalls permittedSubtrees gesetzt hatte, stelle stattdessen sicher, dass der Antragssteller (Subject Name) innerhalb der Schnittmenge der Permitted Subtrees für Zertifikat *i* und für Zertifikat *i-1* liegt. Im Falle *i=1* wird der Eingabewert (h) zum Vergleich

herangezogen.

(c) Ist das Zertifikat ein self-issued Zertifikat und ist es nicht das letzte Zertifikat in der Kette (*i=n*), überspringe diesen Schritt. Sonst stelle sicher, dass der Antragssteller (Subject Name und Alternative Names) nicht innerhalb der Excluded Subtrees für dieses Zertifikat liegt (falls Name Constraints Extension vorhanden und Feld excludedSubtrees gesetzt). Falls das vorherige Zertifikat *i-1* ebenfalls excludedSubtrees gesetzt hatte, stelle stattdessen sicher, dass der Antragssteller (Subject Name) nicht innerhalb der Schnittmenge der Excluded Subtrees für alle Zertifikate *i..i-1* mit dem Zertifikat *i* liegt. Im Falle *i=1* wird der Eingabewert (i) zum Vergleich herangezogen.

(d) Ist die Certificate Policy Extension vorhanden und ist die Zustandsvariable valid_policy_tree nicht NULL, verarbeite die Certificate Policy.

(e) Ist die Certificate Policy Extension nicht vorhanden, setze die Zustandsvariable valid_policy_tree zu NULL (überspringt die Überprüfung für das Einhalten der Certificate Policy - Schritt (d) - für das nächste Zertifikat *i+1)*.

(f) Prüfe ob der Wert der Zustandsvariablen explicit_policy größer ist als 0 oder der Wert von valid_policy_tree nicht NULL ist.

Schlägt einer der Schritte (a), (b), (c) oder (f) fehl, so gilt die Zertifikatskettenprüfung als fehlgeschlagen.

Ist Zertifikat *i* nicht das letzte Zertifikat (*i!=n*), treffe Vorbereitungen für das nächste Zertifikat *i+1*. Sonst, schließe die Prüfung mit dem Schritt Wrap up ab.

4.3.3 Vorbereitung für nächstes Zertifikat

Zur Vorbereitung der Verarbeitung des nächsten Zertifikats *i+1* sind für Zertifikat *i* noch die folgenden Schritte nötig:

(a) Ist die Certificate Policy Mappings Extension vorhanden, prüfe, dass weder die issuerDomainPolicy noch die subjectDomainPolicy dem speziellen Wert anyPolicy entspricht.

(b) Ist die Certificate Policy Mappings Extension vorhanden, prüfe die issuerDomainPolicy.

(c) Setze den Subject Name zum zu erwartenden Issuer Name des nächsten Zertifikats *i+1*.

(d) Setze den subjectPublicKey zum zu erwartenden korrekten Signaturschlüssel des nächsten Zertifikats *i+1*.

(e) Enthält das subjectPublicKeyInfo-Feld einen Eintrag für algorithm mit Parametern die nicht NULL sind, setze die Parameter des zu erwartenden korrekten Signaturschlüssel für das nächste Zertifikat *i+1*. Enthält das subjectPublicKeyInfo-Feld einen Eintrag für algorithm mit NULL-Parametern oder keinen Parametern, vergleiche das algorithm-Feld mit dem des

Zertifikats *i-1*. Im Falle *i=1*, vergleiche mit dem des Trust Anchors. Unterscheiden sie sich, so setze die Parameter des zu erwartenden korrekten Signaturschlüssel für das nächste Zertifikat *i+1* zu NULL.

(f) Setze den subjectPublicKey algorithm zum zu erwartenden korrekten Signaturschlüssel für das nächste Zertifikat *i+1*.

(g) Ist die Name Constraints Extension vorhanden, so beachte die Felder permittedSubtrees und excludedSubtrees. Die genaue Vorgehensweise ist bereits im Abschnitt *Verarbeitung*, (b) und (c) beschrieben.

(h) Ist das Zertifikat *i* nicht self-issued, dekrementiere die Zustandsvariablen explicit_policy, policy_mapping und inhibit_anyPolicy, wenn sie jeweils nicht Null sind.

(i) Ist die Certificate Policy Constraints Extension vorhanden:

(1) Ist das Feld requireExplicitPolicy vorhanden und enthält einen Wert kleiner als den Wert der Zustandsvariable explicit_policy, so setze explicit_policy auf den Wert von requireExplicitPolicy.

(2) Ist das Feld inhibitPolicyMapping vorhanden und enthält einen Wert kleiner als den Wert der Zustandsvariable policy_mapping, so setze policy_ mapping auf den Wert von inhibitPolicyMapping.

(j) Ist die inhibitAnyPolicy Extension vorhanden und ist ihr Wert kleiner als der Wert der Zustandsvariablen inhibit_anyPolicy, so setze inhibit_anyPolicy auf den Wert von inhibitAnyPolicy.

(k) Ist das Zertifikat *i* ein Version 3 Zertifikat, prüfe ob die Basic Constraints Extension vorhanden ist und ob das Feld cA auf TRUE gesetzt ist. Für Version 1 oder Version 2 Zertifikate muss auf andere Weise sichergestellt werden das das Zertifikat ein CA-Zertifikat ist oder das Zertifikat muss abgelehnt werden. Version 1 oder Version 2 Intermediate-Zertifikate dürfen abgelehnt werden.

(l) Ist das Zertifikat nicht self-issued, prüfe ob max_path_length > 0 und dekrementiere dann max_path_length. Diese Zustandsvariable wird in der Initialisierungsphase mit dem Wert *n* initialisiert.

(m) Ist die Basic Constraints Extension vorhanden, ist das Feld pathLenConstraint gesetzt und weist einen kleineren Wert auf als max_path_length, so setze max_path_ length auf den Wert von pathLenConstraint.

(n) Ist eine Key Usage Extension vorhanden, prüfe ob das keyCertSign-Bit gesetzt ist.

(o) Verarbeite alle weiteren Critical Extensions die im Zertifikat vorhanden sind. Verarbeite alle weiteren bekannten Non-Critical Extensions die im Zertifikat vorhanden sind und die für die

Zertifikatskettenverarbeitung relevant sind.

Schlägt einer der Schritte (a), (k), (l), (n) oder (o) fehl, so gilt die Zertifikatskettenprüfung als fehlgeschlagen.

Sind die Schritte (a), (k), (l), (n) und (o) erfolgreich, so setze *i=i+1* und kehre zurück zu *Verarbeitung*.

4.3.4 Wrap up

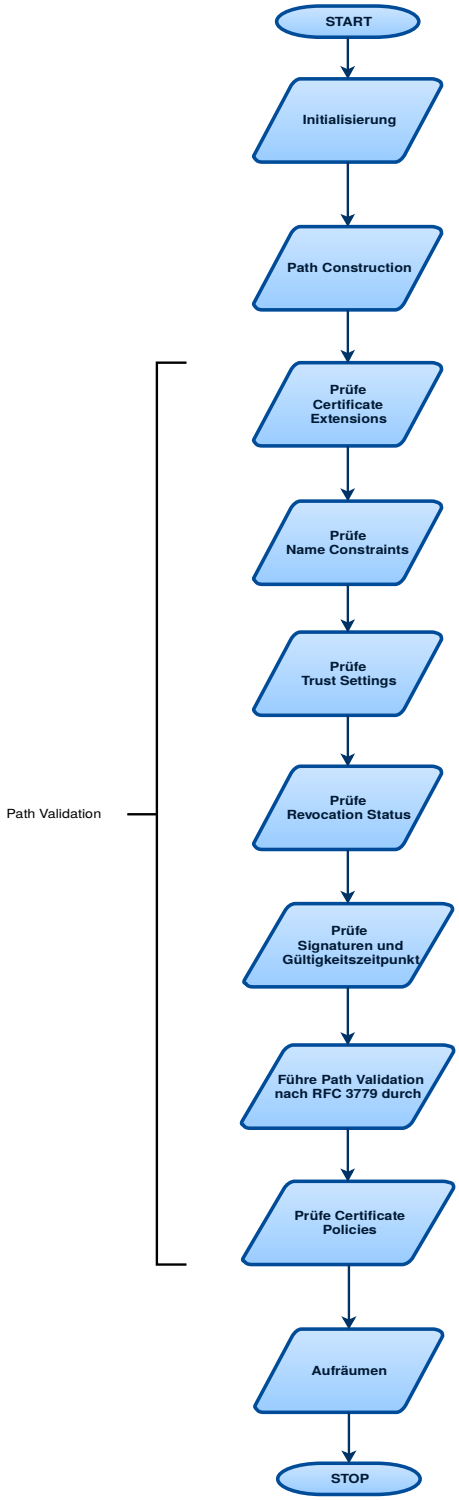
Um die Verarbeitung des zu prüfenden Zertifikats *n* abzuschließen müssen noch einige Schritte ausgeführt werden. Dazu gehört zum einen die Verarbeitung aller weiteren Critical Extensions die im Zertifikat vorhanden sind. Außerdem müssen alle weiteren bekannten Non-Critical Extensions die im Zertifikat *n* vorhanden sind und die für die Zertifikatskettenverarbeitung relevant sind verarbeitet werden. Zuletzt werden die Zustandsvariablen explicit_policy auf > 0 und valid_policy_tree auf nicht NULL geprüft. Sind beide Bedingungen erfüllt, so gilt die Zertifikatskettenprüfung als erfolgreich.

4.4 Zertifikatskettenprüfung durch OpenSSL

Die Zertifikatskettenprüfung in OpenSSL findet in der Funktion

int X509_verify_cert(X509_STORE_CTX*ctx)

statt (Datei crypto/x509/x509_vfy.c). Wie in Abschnitt 4.2 beschrieben, erhält die Funktion als einzigen Parameter eine Zertifikatsdatenbank und arbeitet anschließend auf dieser. Die Datenbank enthält unter anderem den Trust Anchor und das Peer-Zertifikat. Im Erfolgsfall liefert die Funktion den Wert 1 zurück, im Fehlerfall 0. In Ausnahmefällen kann die Funktion auch einen negativen Fehlercode zurückgeben. Im Fehlerfall können weitergehende Informationen zur Fehlerdiagnose bspw. mittels X509_STORE_CTX_get_error() abgerufen werden. Der Algorithmus besteht im Wesentlichen aus zehn Schritten, die grob in drei Phasen eingeteilt werden können: Initialisierung, Path Construction und Path Validation [PKI]. Der Algorithmus ist in Abbildung 2 dargestellt.


START
Initialisierung
Path Construction
Prüfe Certificate Extensions
Prüfe Name Constraints
Prüfe Trust Settings
Prüfe Revocation Status
Prüfe Signaturen und Gültigkeitszeitpunkt
Führe Path Validation nach RFC 3779 durch
Prüfe Certificate Policies
Aufräumen
STOP
Path Validation

4.4.1 Initialisierung

In der Initialisierungsphase werden Variablen, die im Laufe der Funktion verwendet werden, deklariert und initialisiert (Z. 155-161). Abschließend wird geprüft, ob in der übergebenen Zertifikatsdatenbank ein Peer-Zertifikat vorhanden ist (Z. 162-166). Ist dies nicht der Fall, so kehrt die Funktion mit dem Rückgabewert -1 zurück.

4.4.2 Path Construction

In dieser Phase wird aus dem gegebenen Peer-Zertifikat, gegebenen Trusted Certificates und möglicherweise weiteren Untrusted Certificates eine vollständige Zertifikatskette konstruiert, die in der nächsten Phase zur Prüfung gelangt (Z. 172-336). Die Details des Algorithmus werden hier nicht weiter beschrieben.

4.4.3 Path Validation

In dieser finalen Phase wird die eigentliche Zertifikatskettenprüfung durchgeführt. Die Prüfung erfolgt, wie in Abbildung 2 dargestellt, in mehreren Schritten.

Certificate Extensions

Die Extensions aller Zertifikate in der Zertifikatskette werden geprüft (Z. 339-341, 449-605):

1. Gibt es Critical Extensions, die nicht behandelt werden können?
2. Stimmt der Key Purpose überein (siehe X509_VERIFY_PARAM_set_purpose())?
3. Ist die maximal erlaubte Länge der Zertifikatskette überschritten (siehe X509_VERIFY_PARAM_set_depth())?

Ist mindestens eines der Bedingungen nicht erfüllt, so kehrt die Funktion mit dem Rückgabewert 0 zurück.

4.4.3.1 Name Constraints

Die Name Constraints aller Zertifikate in der Zertifikatskette werden geprüft (Z. 345-347, 607-641). Für jedes Zertifikat in der Zertifikatskette wird geprüft, ob die Name Constraints aller in der Zertifikatshierarchie darüber liegenden Zertifikate, einschließlich der des Trusted Certificate, eingehalten werden (Felder permittedSubtrees und excludedSubtrees). Schlägt die Prüfung der Name Constraints fehl, so kehrt die Funktion mit dem Rückgabewert 0 zurück.

4.4.3.2 Trust Settings

Die Trust Settings des Peer-Zertifikats werden geprüft. (Z. 351-353, 643-667). Trust Settings sind weder Teil des RFC 5280 noch der darunterliegenden Standards, sondern sind ein OpenSSL-eigenes Konstrukt. Auf die Beschreibung der Prüfung der Trust Settings wird daher an dieser Stelle verzich-

tet.

4.4.3.3 Revocation Status

Der Revocation Status aller Zertifikate in der Zertifikatskette wird geprüft (Z. 362-363, 669-690). Alle CRLs müssen zum Zeitpunkt der Prüfung bereits vorliegen. Die CRL-Prüfung muss mittels X509_VERIFY_PARAM_set_flags() aktiviert worden sein. Schlägt eine CRL-Prüfung fehl, so kehrt die Funktion mit dem Rückgabewert 0 zurück.

4.4.3.4 Signaturen und Gültigkeitszeitpunkt

Die Signaturen aller Zertifikate und ihre Gültigkeit zum aktuellen Zeitpunkt werden geprüft (Z. 366-370). Ist eine eigene Callback-Funktion mittels SSL_CTX_set_cert_verify_callback() registriert, so wird diese aufgerufen. Anderenfalls kommt die eingebaute Routine zur Prüfung von Zertifikatsketten zum Einsatz (Z. 1586-1674). Diese Routine verifiziert zum einen die Signaturen aller Zertfikate in der Zertifikatskette mit dem Public Key des jeweils nächsten Zertifikats in der Zertifikatskette mittels X509_verify() und prüft zum anderen mittels check_cert_time() (Z. 1539-1584), ob der aktuelle Zeitpunkt im Gültigkeitsbereich des jeweiligen Zertifikats liegt (Felder notBefore und notAfter). Schlägt eine Signaturprüfung oder eine zeitliche Gültigkeitsprüfung fehl, so kehrt die Funktion mit dem Rückgabewert 0 zurück.

4.4.4 Path Validation nach RFC 3779

Ist die Path Validation nicht mittels Compilerdirektive OPENSSL_NO_RFC3779 deaktiviert worden, so wird eine Path Validation nach RFC 3779 [RFC3779] durchgeführt (Z. 372-378). Schlägt die Prüfung fehl, so kehrt die Funktion mit dem Rückgabewert 0 zurück.

4.4.5 Certificate Policy

Ist Certificate Policy Checking mittels X509_VERIFY_PARAM_add0_policy() aktiviert worden, dann wird die Certificate Policy geprüft (Z. 380-383, 1489-1537). Schlägt die Prüfung fehl, so kehrt die Funktion mit dem Rückgabewert 0 zurück.

Zuletzt wird zur Laufzeit der Funktion reservierter Speicher freigegeben. Das Ergebnis der Zertifikatskettenprüfung wird an die aufrufende Funktion zurückgegeben (Z. 386-391).

4.5 Konformität zu RFC 5280

In RFC 5280 wird ein Algorithmus zur Verifikation von Zertifikatsketten beschrieben. Dieser Algorithmus ist für Anwendungen, die sich zu RFC 5280 konform verhalten wollen nicht verpflichtend. Konforme Anwendungen müssen jedoch einen Algorithmus implementieren, der am Ende das gleiche Ergebnis wie der in RFC 5280 beschriebene Algorithmus liefert.

Conforming implementations of this specification are not required to implement this algorithm, but

MUST provide functionality equivalent to the external behavior resulting from this procedure. Any algorithm may be used by a particular implementation so long as it derives the correct result. – RFC 5280, Kapitel 6, Absatz 2.

Im Wesentlichen enthält der in RFC 5280 beschriebene Algorithmus die Prüfung der Signatur, des Gültigkeitszeitpunktes, des Revocation Status, der Name Constraints, der Certificate Policies, der Certificate Extensions und der Basic Constraints.

OpenSSL implementiert einen Algorithmus, der im Wesentlichen die Prüfung der Certificate Extensions, der Name Constraints, der Trust Settings, des Revocation Status, der Signatur und des Gültigkeitszeitpunktes, die Path Validation nach RFC 3779 und die Prüfung der Certificate Policy enthält. Damit erfüllt der in OpenSSL implementierte Algorithmus im Wesentlichen die Anforderungen an einen zu RFC 5280 konformen Algorithmus. Die Vollständigkeit oder gar Korrektheit im Einzelnen des in OpenSSL implementierten Algorithmus kann daraus jedoch nicht hergeleitet werden. Um die Korrektheit des implementierten Algorithmus zu bewerten erscheint es sinnvoll, die Implementierung mittels einer Testsuite auf Konformität zu überprüfen.

4.5.1 Analyse mittels x509test

Es wurde eine Analyse der Zertifikatskettenprüfung mittels des Tools *x509test* [x509test] in Version 02187cb59 durchgeführt. x509test ist ein in Python 3 geschriebenes Tool zum Testen der Zertifikatskettenprüfung eines SSL/TLS-Clients. Ziel des Tools ist es, die Sicherheit eines SSL/TLS-Clients zu erhöhen, indem den Entwicklern negatives Feedback gegeben wird.

Um einen SSL/TLS-Client zu testen, stellt x509test einen TLS-Server bereit, zu dem sich der SSL/TLS-Client verbinden soll. Als TLS-Server kommt OpenSSL zum Einsatz. Zum Starten des TLS-Servers erwartet x509test zunächst einen Domainnamen (Fully Qualified Domain Name – FQDN) und ein Root CA Zertifikat. Wird kein Root CA Zertifikat angegeben, generiert x509test ein self-signed Zertifikat das als Root CA Zertifikat zum Einsatz kommt. Im nächsten Schritt generiert x509test eine Menge von Test-Zertifikaten. Einige der Zertifikate sind direkt von der Root CA signiert, andere von zusätzlich erzeugten Intermediate CAs. Die meisten der erzeugten Test-Zertifikate enthalten Fehler.

x509test wartet dann auf den Verbindungsaufbau des zu testenden SSL/TLS-Clients. In jeder Session mit dem Client wird eine andere Zertifikatskette präsentiert. Akzeptiert der Client eine fehlerhafte Zertifikatskette im Handshake oder bricht der Handshake bei einer gültigen Zertifikatskette ab, so wird dies als potentieller Fehler des Clients aufgezeichnet. Enthalten sind Postiv- wie Negativtests zu Basic Certificate Fields wie Name und Valitidy, Certificate Extensions wie Name Constraints, KeyUsage und ExtendedKeyUsage, zur Hostname Verification, auch mittels Wildcards, und zum Handling von Self-signed Zertifikaten. Eine Liste aller Tests erhält man mit dem Aufruf von:

python3 x509test.py –list

Tests für Revocation Status, Path Validation nach RFC 3779 und Certificate Policy existieren derzeit nicht.

x509test wurde mit den folgenden Parametern gestartet:

python3 x509test.py www.tls.org -p 4433

Die Tests wurden mit dem OpenSSL Kommandozeilentool *openssl s_client* durchgeführt. Das Tool wurde mit den folgenden Parametern gestartet:

openssl s_client -verify -showcerts -debug -CAfile x509test/ca/ca.pem -verify_return_error

Im Test mit dem OpenSSL TLS-Testclient schlugen 19 der 45 durchgeführten Tests fehl. Vier Negativtests wurden nicht ausgeführt, da bereits der zugehörige Positivtest fehlschlug. 17 der fehlgeschlagenen Tests betreffen die Hostname Verification nach RFC 6125. Wie bereits in Abschnitt 4.1 erläutert, führt OpenSSL derzeit keine Hostname Verification durch. Daher werden die betreffenden fehlgeschlagenen Tests hier nicht weiter betrachtet. Die übrigen zwei fehlgeschlagenen Tests betreffen die Prüfungen der Certificate Extensions und der Name Constraints. x509test gibt für jeden fehlgeschlagenen Test die Referenz zum betreffenden Standarddokument und den Schweregrad des Fehlverhaltens an.

4.5.2 Certificate Extensions

Bei der Prüfung von Certificate Extensions schlug ein Test fehl.

Name	Typ	Referenz	Verhalten OpenSSL	Schwere
MissingIntCABasicConstraintWithCertSign	Negativ	RFC 5280, 4.2.1.9	Akzeptiert	Hoch

Tabelle 3: Fehlgeschlagene Tests bei Prüfung Certificate Extensions

Im fehlgeschlagenen Test MissingIntCABasicConstraintWithCertSign enthält ein Intermediate CA Zertifikat in der Zertifikatskette zwar im Feld KeyUsage den Wert keyCertSign, es fehlt jedoch die BasicConstraints Extension. In diesem Fall muss OpenSSL das Peer-Zertifikat per RFC 5280 ablehnen.

The cA boolean indicates whether the certified public key may be used to verify certificate signatures. If the cA boolean is not asserted, then the keyCertSign bit in the key usage extension MUST NOT be asserted. If the basic constraints extension is not present in a version 3 certificate, or the extension is present but the cA boolean is not asserted, then the certified public key MUST NOT be used to verify certificate signatures. – RFC 5280, Abschnitt 4.2.1.9 Basic Constraints.

4.5.3 Name Constraints

Bei der Prüfung von Name Constraints schlug ein Test fehl. Vier weitere, verwandte Tests kamen deshalb nicht zur Ausführung.

Name	Typ	Referenz	Verhalten OpenSSL	Schwere
ValidNameConstraint	Positiv	RFC 5280, 4.2.1.10	Abgelehnt: permitted subtree violation	k.A.
InvalidNameConstraint Exclude	Negativ	RFC 5280, 4.2.1.10	Nicht ausgeführt	Mittel
InvalidNameConstraint PermitRight	Negativ	RFC 5280, 4.2.1.10	Nicht ausgeführt	Mittel
InvalidNameConstraint PermitThenExclude	Negativ	RFC 5280, 4.2.1.10	Nicht ausgeführt	Mittel
InvalidNameConstraint Permit	Negativ	RFC 5280, 4.2.1.10	Nicht ausgeführt	Mittel

Tabelle 4: Fehlgeschlagene bzw. nicht ausgeführte Tests bei Prüfung Name Constraints

Im Test ValidNameConstraint enthält das erste Intermediate CA Zertifikat im Feld permittedSubtree den Wert „.org“ und das zweite Intermediate CA Zertifikat im Feld excludedSubtrees den Wert „www.tls.org.invalid.org“. In diesem Fall muss OpenSSL das Peer-Zertifikat per RFC 5280 akzeptieren.

Im Test InvalidNameConstraintExclude enthält das erste Intermediate CA Zertifikat im Feld excludedSubtree den Wert „.org“. In diesem Fall muss OpenSSL das Peer-Zertifikat per RFC 5280 ablehnen.

Im Test InvalidNameConstraintPermitRight enthält das erste Intermediate CA Zertifikat im Feld permittedSubtree den Wert „.tls“.

Im Test InvalidNameConstraintPermitThenExclude enthält das erste Intermediate CA Zertifikat im Feld permittedSubtree den Wert „.tls“, während das zweite Intermediate CA Zertifikat im Feld excludedSubtree auch den Wert „.tls“ enthält.

Im Test InvalidNameConstraintPermit enthält das erste Intermediate CA Zertifikat im Feld permittedSubtree den Wert „.tlsx“.

4.6 Fazit

In diesem AP wurde die Zertifikatskettenprüfung in OpenSSL untersucht. Es wurde zunächst erläutert, wie der TLS-Handshake in OpenSSL konfiguriert werden kann. Dabei konnte beispielsweise herausgestellt werden, dass eine Hostname Verification nach RFC 6125, eigentlich ein zentraler Bestandteil einer Zertifikatskettenprüfung, nicht Bestandteil der untersuchten OpenSSL-Version ist. Im nächsten Abschnitt wurde vertiefend auf die Konfiguration der Zertifikatskettenprüfung eingegangen. Dabei wurde beispielsweise herausgearbeitet, dass die CRL-Prüfung des Peer-Zertifikats oder auch der ganzen Zertifikatskette standardmäßig deaktiviert sind und erst mittels bestimmter Flags aktiviert werden müssen.

Als Referenzalgorithmus zur Zertifikatskettenprüfung wurde der Algorithmus aus RFC 5280 ausführlich beschrieben. Der in OpenSSL implementierte Algorithmus wurde anschließend näher erläutert und mit dem Algorithmus aus RFC 5280 verglichen. Dabei wurde herausgestellt, dass der in OpenSSL implementierte Algorithmus alle wesentlichen Schritte einer Zertifikatskettenprüfung nach RFC 5280 implementiert. Die Vollständigkeit oder gar Korrektheit im Einzelnen des in OpenSSL implementierten Algorithmus kann aus dieser Erkenntnis jedoch nicht hergeleitet werden.

Um die Korrektheit des Algorithmus zu verifizieren erschien es sinnvoll, die Implementierung mittels einer Testsuite gezielt zu testen. Die Testsuite „x509test“ wurde dafür verwendet. Die Testsuite enthält 45 Tests, die die Basisfunktionen Prüfung von Name und Validity, Certificate Extensions wie Name Constraints, KeyUsage und ExtendedKeyUsage, Hostname Verification und Handling von Self-signed Zertifikaten abdecken. Bei zwei Tests wurde ein vom Standard abweichendes Verhalten festgestellt. Der erste fehlgeschlagene Test betrifft das Zusammenspiel von ExtendedKeyUsage und BasicConstraints in einem CA-Zertifikat. Hier akzeptiert OpenSSL ein nach RFC 5280 ungültiges Zertifikat. Der zweite fehlgeschlagene Test betrifft die Prüfung der Name Constraints. Hier lehnt OpenSSL ein nach RFC 5280 gültiges Zertifikat ab mit einem Fehler bei der Prüfung der permittedSubtrees ab. Auch da beide Felder in der Internet PKI zum Einsatz kommen [CAB] und daher möglicherweise kritisch sind, wird empfohlen mit den Entwicklern von OpenSSL in Kontakt zu treten um die Fehler genauer zu untersuchen und wenn möglich zu beheben. Anschließend sollten die im Rahmen der Prüfung der Name Constraints nicht ausgeführten Tests ausgeführt werden.

Um eine präzisere Aussage über die Korrektheit des in OpenSSL implementierten Algorithmus zur Zertifikatskettenprüfung zu erhalten, erscheint es sinnvoll die Testabdeckung durch Erweiterung mit zusätzlichen Tests in x509test zu erhöhen. Auch Tests für bisher nicht abgedeckte Bereiche der Prüfung erscheinen in diesem Zusammenhang wünschenswert.

5 Diffie-Hellman-Verfahren

5.1 DH in Z_p^*

Es wird basierend auf den Ergebnissen von AP1.2 die OpenSSL-Implementierung des Diffie-Hellman-Verfahrens (DH) analysiert und in Pseudo-Code spezifiziert. Basierend auf dem Pseudocode wird die Wahrscheinlichkeit berechnet, mit der die jeweiligen Moduli zusammengesetzt sind. Es wird betrachtet, ob der Algorithmus korrekt implementiert ist und die Compileroptionen werden ggf. besonders untersucht.

Die Analyse bezieht sich dabei schwerpunktmäßig auf das DH-Verfahren in Primzahlgruppen. Aufgrund der Komplexität der Fragestellungen rund um die Auswahl sicherer EC-Gruppen kann dieser Aspekt nur stichpunktartig beleuchtet werden.

5.1.1 DH Parametererzeugung

Für das Diffie-Hellman-Verfahren werden als Parameter eine Primzahl p und ein Erzeuger g einer zyklischen Untergruppe von Z_p^* von beiden Parteien geteilt sowie von beiden Parteien je ein Geheimnis x_A und x_B . Für die Werte 2 und 5 überprüft OpenSSL, ob sie tatsächlich Erzeuger für Z_p sind. Für andere Werte gelten die Garantien wie in Abs. 5.1.1.2 dargestellt. Der Benutzer muss den Erzeuger stets selbst wählen.

5.1.1.1 Erzeugen einer Primzahl p (crypto/dh/dh_gen.c:dh_builtin_genparams())

Eingabe:

- Länge der Primzahl p in Bits b
- Erzeuger einer Untergruppe g

Algorithmus:

1. wenn $g \leq 1$: Abbruch
2. Erzeuge sichere Primzahl p (crypto/bn/bn_prime.c:BN_generate_prime_ex()), d.h. q=(p-1)/2 prim. Die Werte padd und rem in Abhängigkeit von g werden gemäß [RFC4419] gewählt.
    1. Für Erzeuger g=2 oder g=5 stelle sicher, dass $rem \equiv p \bmod padd$ mit (g, padd, rem) = (2, 24, 11) oder (g, padd, rem) = (5, 10, 3). Diese Werte stammen aus [RFC4419].
    2. Lese Zufallszahl r der Form: 1???....???1, b-1 Bits aus dem RNG (mit RAND_bytes() [AP2])

3. Berechne $q = r - (r \bmod \frac{padd}{2}) + \lfloor \frac{\text{rem}}{2} \rfloor$. Für Werte aus Schritt 2.1 gilt: $\lfloor \frac{rem}{2} \rfloor = \frac{rem-1}{2,}$, und damit ergibt sich $p = 2 \cdot q + 1$ zu:

   $$p = 2 \cdot q + 1 = 2r - 2(r \bmod \frac{padd}{2}) + 2\frac{rem-1}{2} + 1 = 2r - (2r \bmod padd) + rem \quad .$$

   Hierdurch wird die Bedingung aus 2.1 erfüllt.

4. $p = 2 \cdot q + 1$
5. Wiederhole:
   1. Test, ob p oder q durch eine Primzahlen T, $3 \leq T \leq 17863$, teilbar sind. Wenn nicht: Schritt 6
   2. $p = p + padd, q = q + \frac{padd}{2}$. Die Addition von $\frac{padd}{2}$ garantiert, dass weiterhin die Bedingung aus 2.1 erfüllt wird.
6. Miller-Rabin Primzahltest für p und $q = \frac{p-1}{2}$, 2 Durchläufe (für bits >=1300) (s. Abs. 3.3.1).

3. return p

Bemerkung 5: Während der Schlüsselerzeugung wird der RNG nicht reseedet.

Empfehlung 5: Um Enhanced Forward Secrecy sicherzustellen, muss der RNG reseedet werden.

5.1.1.2 Garantien bzgl. der Ordnung q von g

Voraussetzung:

Primzahlen p, q mit p = 2q+1

Multiplikative Gruppe G

Nach dem Satz von Lagrange teilt die Ordnung |H| der Untergruppe H von G die Ordnung der Gruppe G. Die Ordnung der multiplikativen Gruppe Z_p^* ist in diesem Fall $\varphi(p)$ = p-1 = 2q, mit q prim. Die Ordnung von g ist 2, q oder 2q. Die Elemente 1 und p-1 haben Ordnung 2. Da g ungleich 1 ist und p-1 nicht „klein“ (2 oder 5) ist, muss g die Ordnung q=(p-1)/2 oder 2q=p-1 haben..

5.1.2 Kriterien nach BSI TR-02102-1 Kap. 7.2.1

Um die Eignung der Parametererzeugung für sicherheitskritische Komponenten zu beurteilen, vergleichen wir hier die Eigenschaften der erzeugten Parameter mit den Anforderungen der Empfeh-

lungen des BSI für kryptografische Verfahren [TR021021]. Es werden die einzelnen Kriterien zitiert und die Umsetzung dieser in OpenSSL analysiert.

1. Wähle eine Primzahl p.
    - Dieses Kriterium wird durch den Primzahltest erfüllt. Die Wahrscheinlichkeit, dass p zusammengesetzt ist, ist für Schlüssellängen ab 1854 Bits (allgemein) geringer als 2^{-100} .
2. Wähle ein Element $g \in F_p^*$ mit $\mathrm{ord}(g)$ prim und $\mathrm{ord}(g) \geq 2^{224}$ (es wird $\mathrm{ord}(g) \geq 2^{250}$ empfohlen im Falle eines Einsatzzeitraumes nach 2015).
    - Aufgrund der verwendeten sicheren Primzahlen haben Elemente einer Multiplikativen Untergruppe entweder die Ordnung 2, q oder 2q. Sofern g nicht gleich p-1 oder 1 ist kann die Ordnung 2 ausgeschlossen werden.

Fazit: Wir stellen fest, dass OpenSSL die Kriterien gemäß [TR021021] für Diffie-Hellman Parameter ab einer Länge von 1854 Bits erfüllt.

5.1.3 DH Schlüsselerzeugung

Hier wird die Schlüsselerzeugung auf der Seite von A mit dem zugehörigen privaten Schlüssel a betrachtet. Für den Kommunikationspartner B ist die Erzeugung äquivalent.

Erzeuge privaten und öffentlichen Schlüssel (crypto/dh/dh_key.c:generate_key())

1. x = rand(), b-1 Bits der Form: 1???...???. (mit RAND_bytes() [AP2]) oder x<q, falls q bekannt (mit BN_rand_range())
2. $k_A = g^x \bmod p$ (Diese Operation wird optional in konstanter Zeit ausgeführt)

Erzeugen des gemeinsamen Schlüssels k mit dem öffentlichen Schlüssel von B k_B (crypto/dh/dh_key.c:compute_key()).

1. Wenn $k_B \leq 1$: Abbruch
2. Wenn $k_B \geq p-1$: Abbruch
3. $k_{AB} = (k_B)^a \bmod p$

5.1.3.1 Kriterien nach BSI TR-02102-1 Kap. 7.2.1

Um die Eignung der Schlüsselerzeugung für sicherheitskritische Komponenten zu beurteilen, vergleichen wir hier die Eigenschaften der erzeugten Schlüssel mit den Anforderungen der Empfehlungen des BSI für kryptografische Verfahren [TR021021]. Es werden die einzelnen Kriterien zitiert und die Umsetzung dieser in OpenSSL analysiert.

1. A wählt gleichverteilt einen Zufallswert $x \in \{1, \ldots, q-1\}$ und sendet $Q_A := g^x \bmod p$ an B
    - Dieses Kriterium ist erfüllt.
2. B wählt gleichverteilt einen Zufallswert $y \in \{1, \ldots, q-1\}$ und sendet $Q_B := g^y \bmod p$ an A
    - Identisch mit 1.
3. A berechnet $(g^y)^x = g^{xy} \bmod p$.
    - Die Berechnung erfolgt auf diese Weise.
4. B berechnet $(g^x)^y = g^{xy} \bmod p$.
    - Identisch mit 3.

Fazit: Wir stellen fest, dass OpenSSL die Kriterien gemäß [TR021021] für den Diffie-Hellman Schlüsselaustausch erfüllt.

Darüber hinaus überprüfen wir die Kriterien nach [RS2000], Sec. 7.1, in gleicher Weise:

1. Spot unconventional Messages
    - Es wird nicht überprüft, ob der erzeugte Schlüssel $g^{xy} = 1 \bmod p$. Die anderen Kriterien werden erfüllt.
2. Be Careful About g's order
    - Aufgrund der Verwendung von sicheren Primzahlen werden die Kriterien erfüllt.
3. Make Sure the DH Public Keys Received Have the Correct Order
    - Da $p = 2q + 1$, treffen diese Kriterien für selbst erzeugte Schlüssel nicht zu.
    - Für extern bereitgestellte Parameter werden diese Tests nicht durchgeführt.
4. Make Sure the System Parameters Are Not Chosen Maliciously
    - Extern zur Verfügung gestellte Parameter können mit DH_check() darauf geprüft werden, ob p eine sichere Primzahl ist und ob g ein geeigneter Erzeuger ist.
5. Choose Secure Parameters
    - Die Größe der von OpenSSL gewählten Exponenten sind sicher.
    - Die weiteren Parameter sind Teil des Protokolldesigns und müssen entsprechend berücksichtigt werden.

Fazit: Wir stellen fest, dass OpenSSL die Kriterien gemäß [RS2000] für den Diffie-Hellman Schlüsselaustausch nur teilweise erfüllt.

5.2 Elliptic Curve Diffie Hellman

Für den Schlüsselaustausch mit dem Elliptic Curve Diffie Hellman Verfahren wird hier der Schlüsselaustausch auf für TLS1.2 definierten Kurven betrachtet. Daher wurde die Erzeugung von Parametern für EC-Kryptosysteme nicht untersucht.

5.2.1 Erzeugung des öffentlichen Schlüssels

Eingabe

- Elliptische Kurve E mit Generator P

Algorithmus (crypto/ec/ec_key.c:EC_KEY_generate_key())

1. Feststellen der Ordnung der Elliptischen Kurve ord(E). Für die hier betrachteten Named Curves ist die Ordnung bekannt und wurde statisch in OpenSSL gespeichert.
2. Erzeugen einer Zufallszahl x, $1 \leq x < ord(E)$. (mit BN_rand_range() und RAND_bytes() [AP2])
3. Berechnen des öffentlichen Schlüssels $Q = x \cdot P$ in E.

5.2.2 Berechnen des gemeinsamen Schlüssels

Eingabe

- Elliptische Kurve E
- privater Schlüssel x
- öffentlicher Schlüssel des Kommunikationspartners Q_B

Algorithmus (crypto/ecdh/ech_ossl.c:ecdh_compute_key())

1. Berechne $K = x \cdot Q_B$.
2. optional wird eine KDF angewandt.

Bemerkung 6: Tests auf Zugehörigkeit des Punktes zur Kurve werden nicht automatisch durchgeführt, sondern müssen explizit mittels EC_KEY_check_key() vor deren Verwendung durchgeführt werden.

Empfehlung 6: Um Invalid-Curve Angriffen zu widerstehen muss bei der Nutzung der OpenSSL-Funktionen für Elliptischen Kurven mittels EC_KEY_check_key() sichergestellt werden, dass Punkte von Kommunikationspartnern auf der Kurve liegen.

Diese Tests werden von der SSL-Implementierung beim importieren von Punkten in EC_KEY_set_public_key_affine_coordinates() durchgeführt.

Bemerkung 7: Es werden keine Maßnahmen ergriffen, um Small Subgroup Angriffe zu verhindern.

Empfehlung 7: Bei Kurven mit einer nicht-primen Ordnung ord(E) sollten vor der Verwendung von Punkten Tests gemäß http://safecurves.cr.yp.to/twist.html durchgeführt werden.

5.2.3 Kriterien nach BSI TR-02102-1 Kap. 7.2.2

Um die Eignung der Schlüsselerzeugung für sicherheitskritische Komponenten zu beurteilen, vergleichen wir hier die Eigenschaften der erzeugten Schlüssel mit den Anforderungen der Empfehlungen des BSI für kryptografische Verfahren [TR021021]. Es werden die einzelnen Kriterien zitiert und die Umsetzung dieser in OpenSSL analysiert.

1. A wählt gleichverteilt einen Zufallswert $x \in \{1, \ldots, q-1\}$ und sendet $Q_A := x \cdot P$ an B.
    - Dieses Kriterium ist erfüllt
2. B wählt gleichverteilt einen Zufallswert $y \in \{1, \ldots, q-1\}$ und sendet $Q_B := y \cdot P$ an A.
    - Identisch mit 1.
3. A berechnet $x \cdot Q_B = xy \cdot P$.
    - Die Berechnung erfolgt auf diese Weise.
4. B berechnet $y \cdot Q_A = xy \cdot P$.
    - Identisch mit 3.

Um die Anforderungen bezüglich der Schlüssellänge zu erfüllen muss eine geeignete Kurve aus dem TLS1.2-Pool ausgewählt werden.

Fazit: Wir stellen fest, dass OpenSSL die Kriterien gemäß [TR021021] für den Elliptic-Curve-Diffie-Hellman Schlüsselaustausch erfüllt, sofern eine geeignete Kurve aus dem TLS1.2-Pool ausgewählt wurde.

6 Implementierungsfehler in Cipher-Suites

Dieses Kapitel untersucht die Verwundbarkeit der Implementierung der Cipher-Suites. Wir beginnen mit einer Top-Down Analyse des TLS-Stacks und stellen daran anschließend die Ergebnisse der Untersuchung der Low-Level Kryptoalgorithmen dar.

6.1 Top-Down Analyse des TLS-Stacks

In diesem Kapitel werden Angriffe aus AP 3.1 analysiert, die praktische Gegenmaßnahmen in der Implementierung des TLS-Servers erfordern. Des Weiteren werden Standardeinstellungen des TLS-Servers untersucht.

Für die Analyse wurden unter anderem zwei am Lehrstuhl für Netz- und Datensicherheit der Ruhr-Universität Bochum entwickelte Frameworks: T.I.M.E. (TIME) und TLS-Attacker (zur Zeit in Entwicklung). Als Resultat der Analyse wird eine virtuelle Maschine zur Verfügung gestellt, in welcher beide Frameworks vorhanden sind.

In diesem Teil wird ein Top-Down-Ansatz für die Analyse gewählt. Wir beginnen mit der Untersuchung am Client-Sichtbaren Interface und arbeiten uns von dort ausgehend in die Details der Implementierung vor, um mögliche Schwachstellen und deren Ursache zu finden.

6.1.1 Identifikation der relevanten Code-Stellen mit Debugging

Um den OpenSSL Code-Ablauf zu verfolgen und die relevanten Code-Stellen finden zu können, wurde Debugging eingesetzt. Dafür wurde OpenSSL wie folgt kompiliert:

```
./config –d
make
make test
```

Anschließend wurde Netbeans 8.0.1 in Kombination mit dem GDB Debugger (Version 7.7) eingesetzt.

6.1.2 Timing-basierte Angriffe – Genutzte Umgebung

In diesem AP werden auch Timing-basierte Angriffe betrachtet. Timing-Angriffe basieren allgemein auf Zeitunterschieden, welche von einer Implementierung produziert werden, wenn sie vertrauliche Werte verarbeitet. Anhand von den Zeitunterschieden kann der Angreifer Rückschlüsse auf geheime Werte ziehen.

Bei unserer Untersuchung haben wir uns auf lokale Timing-basierte Angriffe konzentriert. Dabei haben wir zuerst Stellen im OpenSSL Code gesucht, welche für den Angreifer relevant sind (z.B. bei den Bleichenbacher-Angriffen zählt die PKCS#1-Entschlüsselung zu den relevanten Stellen).

Danach haben wir Timing-basierte Tests von den aufgefundenen Funktionen direkt in OpenSSL Code eingebaut und evaluiert. Dabei wurde die relevante Funktion zwischen zwei Assembler-Aufrufe gesetzt, und es wurde die Anzahl der Prozessor-Ticks gemessen:

```
 asm volatile(
        "cpuid \n"
        "rdtsc"
        : "=a"(minor),
        "=d"(mayor)
        : "a" (0)
        : "%ebx", "%ecx"
        );

start = ((((ticks) mayor) << 32) | ((ticks) minor));

 Timing-Relevant Function

asm volatile(
        "cpuid \n"
        "rdtsc"
        : "=a"(minor),
        "=d"(mayor)
        : "a" (0)
        : "%ebx", "%ecx"
        );

end = ((((ticks) mayor) << 32) | ((ticks) minor));
result = end - start;
```

Dabei wird der Prozessor-Timestamp-Counter (TSC) über die Assembler-Instruktion **RDTSC** ausgelesen. Die Werte werden in den Variablen **mayor** und **minor** gespeichert, welche zur Berechnung der Zeitstempel **start** und **end** eingesetzt werden. Schließlich wird der Unterschied zwischen **start** und **end** als Resultat genommen.

Dies hat uns ermöglicht, Zeitunterschiede im Nanosekundenbereich zu messen. Die Messungen wurden anschließend mit validen und invaliden Werten abwechselnd in der Schleife (1000, 5000, 10000 und 20000 mal) wiederholt. Die Messungen wurden dann mit der Fau-Timer Library (FAUT) evaluiert.

Gemessen wurde auf einer Workstation mit Xubuntu 14.04 und folgender Konfiguration:

- Model name: AMD Athlon(tm) X2 340 Dual Core Processor
- CPU MHz: 1400.000
- Cache size: 1024 KB

Um Rauschen zu vermeiden, wurden die Timing-Analysen *nicht* in einer virtuellen Umgebung durchgeführt. Debugging-Option wurde bei der Kompilierung rausgenommen.

6.1.3 Bleichenbacher-Angriff

Wie im Kapitel 3.1 beschrieben wurde, gehören Bleichenbacher-Angriffe (Blei) zu einer Klasse von adaptiven Chosen-Ciphertext-Angriffen. Dabei sendet der Angreifer veränderte Chiffretexte an den Server und beobachtet die Server-Antworten, welche es erlauben, Rückschlüsse auf die Nachrichtenvalidität zu ziehen. Bei jeder Server-Antwort lernt der Angreifer etwas über den Chiffretext.

Wenn der Angreifer keine Möglichkeit bekommt, valide von invaliden Nachrichten zu unterscheiden, kann er diesen Angriff nicht durchführen. Daher ist es nötig, dass der Server alle Anfragen mit gleichen Antworten bearbeitet, welche in konstanter Zeit produziert werden.

Um konkret den Bleichenbacher-Angriff zu verhindern, wurde folgender Abschnitt im TLS 1.2-Standard [RFC5246] hinzugefügt:

```
   1. Generate a string R of 46 random bytes
   2. Decrypt the message to recover the plaintext M
   3. If the PKCS#1 padding is not correct, or the length of
      message M is not exactly 48 bytes:
             pre_master_secret = ClientHello.client_version || R
          else If ClientHello.client_version <= TLS 1.0, and
version number check is explicitly disabled:
             pre_master_secret = M
          else:
             pre_master_secret = ClientHello.client_version ||
M[2..47]
```

Damit ist gewährleistet, dass ein PremasterSecret immer in konstanter Zeit generiert wird. Wenn die Struktur der Nachricht nach der Entschlüsselung inkorrekt ist, wird als PremasterSecret eine zufällig gezogene Zahl verwendet. Diese muss immer erzeugt werden, unabhängig davon ob die entschlüsselte Struktur korrekt war (vgl. Zeile 1).

Im Folgenden wird analysiert, ob diese Gegenmaßnahme in OpenSSL korrekt implementiert wurde.

6.1.3.1 Identifikation Relevanter Funktionen

Die relevanten Funktionen befinden sich in folgenden Dateien:

- **ssl/s3_srvr.c** (Z. 2157 – 2259): ClientKeyExchange-Bearbeitung, Aufruf der PKCS#1-Entschlüsselung, Überprüfung der TLS-Version in der entschlüsselten Struktur, Generierung einer zufälligen Zahl.
- **crypto/rsa/rsa_eay.c: `int RSA_eay_private_decrypt`** (Z. 492): RSA-Blinding, Entschlüsselung und Aufruf der PKCS#1-Unpadding-Funktion
- **crypto/rsa_pk1.c: `int RSA_padding_check_PKCS1_type_2`** (Z. 181): PKCS#1-Unpadding

- **test/rsa_test.c**: Tests für RSA-OAEP und PKCS#1 v1.5 Verschlüsselung/Entschlüsselung

6.1.3.2 Test Unterschiedlicher Fehlermeldungen

Mithilfe des T.I.M.E.-Tools haben wir versucht, verschiedene Fehlermeldungen nach der Entschlüsselung von invaliden PKCS#1-/TLS-Nachrichten zu provozieren.

Eine valid-formatierte PKCS#1-Nachricht hat im TLS-Protokoll folgende Struktur:

0x00 0x02 padding 0x00 pms

Dabei ist das Padding so lang, dass das PremasterSecret pms eine Länge von 48 Bytes erreicht. Das Padding beinhaltet kein 0x00 Byte. Die ersten zwei Bytes des PremasterSecrets entsprechen der TLS-Version. Folgende Änderungen können anhand von dieser Struktur berücksichtigt werden:

- Die ersten zwei Bytes werden geändert, so dass die Nachricht nicht mit 0x00 0x02 anfängt. Wenn der Server in solchem Fall mit einem Fehler antwortet, führt es zu einem direkten Bleichenbacher-Angriff, so wie in dem ursprünglichen Artikel beschrieben wurde (Blei).
- In das Padding wird an verschiedenen Stellen ein 0x00 Byte eingesetzt. Dies führt dazu, dass das PremasterSecret verlängert wird und damit eine invalide Länge bekommt.
- Die Länge des PremasterSecrets wird verkürzt, in dem das 0x00 Byte in der Mitte verschoben wird.
- Das 0x00 Byte in der Mitte wird gelöscht (zusammen mit anderen 0x00 Bytes im PremasterSecret), so dass der Server die Grenze zwischen dem PremasterSecret und dem Padding nicht erkennen kann.
- Die TLS-Version in dem PremasterSecret wird invalidiert.

Alle diese Veränderungen könnten ein invalides Server-Verhalten provozieren – und damit auch eine Server-Alert-Nachricht, welche zu einem Bleichenbacher-Angriff führen kann.

Unsere Tests haben ergeben, dass der Server *nicht* mit einer Alert-Nachricht antwortet und damit kein direkter Bleichenbacher-Angriff möglich ist. Dies hat unsere Ergebnisse aus dem Usenix-Artikel bestätigt (MSWS), welche mit einer älteren OpenSSL-Version erreicht wurden.

6.1.3.3 Timing-basierte Tests: Zusätzliche Zufallszahlgenerierung

Bei der Analyse der Datei **ssl/s3_srvr.c** wurde sichtbar, dass der Server bei der Entschlüsselung des PremasterSecrets wie folgt vorgeht:

```
1. Decrypt the message to recover the plaintext M
2. If the PKCS#1 padding is not correct, or the length of
   message M is not exactly 48 bytes or TLS Version is
   incorrect:
          Generate a string R of 46 random bytes
```

```
            pre_master_secret = ClientHello.client_version || R
      else:
            pre_master_secret = M
```

Wenn also invalide Nachrichten bearbeitet werden, wird eine zusätzliche zufällige Zahl generiert. Anderseits wird diese nicht generiert und es wird mit dem extrahierten PremasterSecret weiter gerechnet.

Wie in unserem Artikel analysiert wurde (MSWS), bringt die zusätzliche Zufallszahlgenerierung *keine* Vorteile für den Angreifer:

1. Die Unterschiede zwischen den validen und invaliden Nachrichten sind zu klein (im Bereich von 1-3 Mikrosekunden), sie lassen sich nicht klar erkennen.
2. Die Struktur der Nachricht wird strikt überprüft. Um eine valide Nachricht zufällig zu generieren, müsste der Angreifer viele Anfragen an den Server schicken. Der Bleichenbacher-Angriff ist damit nicht praktikabel.

Weitere Untersuchung hat ergeben, dass der Source-Code an den relevanten Stellen seit unserer Analyse nicht geändert wurde.

6.1.3.4 Timing-basierte Tests: Nicht Konstante PKCS#1-Bearbeitung

Die Analyse der **crypto/rsa_pk1.c** Datei hat ergeben, dass die Funktion **RSA_padding_check_PKCS1_type_2** nicht in konstanter Zeit Ergebnisse liefert:

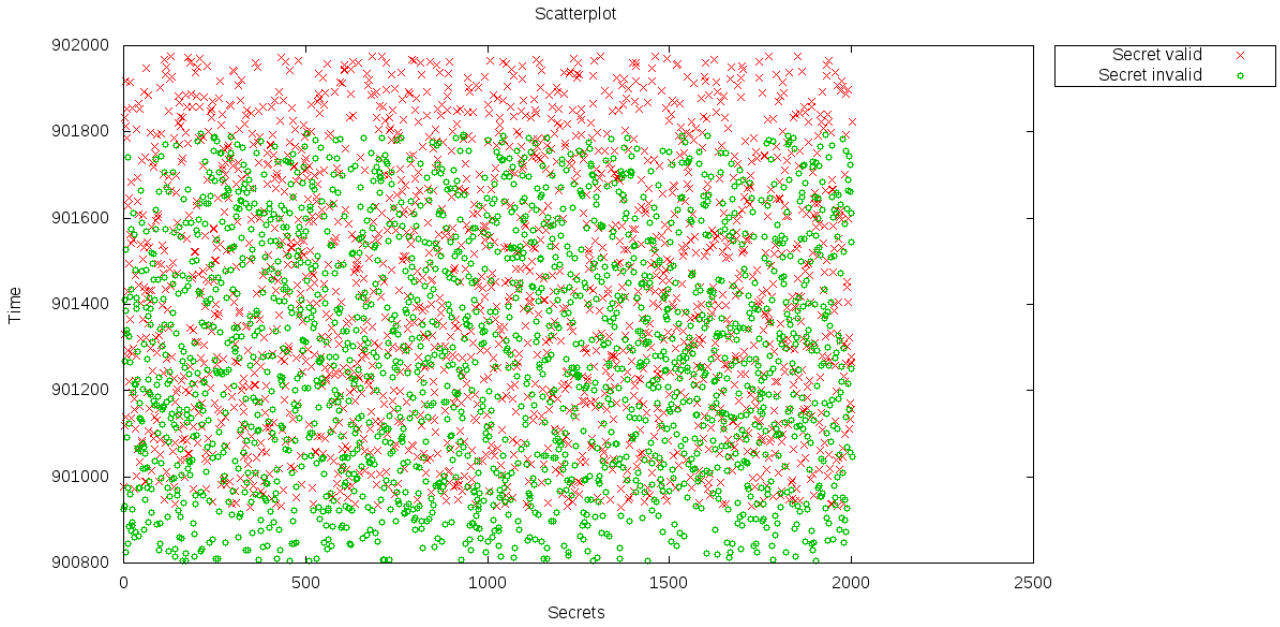
```
int RSA_padding_check_PKCS1_type_2(unsigned char *to, int tlen,
             const unsigned char *from, int flen, int num)
       {
       int i,j;
       const unsigned char *p;
       p=from;
       if ((num != (flen+1)) || (*(p++) != 02))
              {
              RSAerr(RSA_F_RSA_PADDING_CHECK_PKCS1_TYPE_2,RSA_R_BLOCK_TYPE_IS_NOT_02);
              return(-1);
              }
#ifdef PKCS1_CHECK
       return(num-11);
#endif
       /* scan over padding data */
       j=flen-1; /* one for type. */
       for (i=0; i<j; i++)
```

```
            if (*(p++) == 0) break;


      if (i == j)
            {
            RSAerr(RSA_F_RSA_PADDING_CHECK_PKCS1_TYPE_2,RSA_R_NULL_BEFORE_BLOCK_MISS-
ING);
            return(-1);
            }
      if (i < 8)
            {
            RSAerr(RSA_F_RSA_PADDING_CHECK_PKCS1_TYPE_2,RSA_R_BAD_PAD_BYTE_COUNT);
            return(-1);
            }
      i++; /* Skip over the '\0' */
      j-=i;
      if (j > tlen)
            {
            RSAerr(RSA_F_RSA_PADDING_CHECK_PKCS1_TYPE_2,RSA_R_DATA_TOO_LARGE);
            return(-1);
            }
      memcpy(to,p,(unsigned int)j);
      return(j);
      }
```

Wie in dem Code zu sehen ist, bricht der Server früher ab, wenn z.B. das zweite Byte der Nachricht kein 0x02 Byte an der zweiten Position enthält oder wenn die Nachricht kein 0x00 Byte hat.

Wir haben die Testdatei **test/rsa_test.c** erweitert und eine Funktion hinzugefügt, welche Zeitunterschiede zwischen der Entschlüsselung von validen und invaliden PKCS#1-Nachrichten misst. Dabei wurde bei der invaliden Nachricht das zweite Byte (0x02) verändert, so dass die PKCS#1-Bearbeitung hier früher abbricht.



Es wurden 20000 Nachrichten mit der RSA-PKCS#1-Funktion entschlüsselt und anschließend wurden die Zeiten mit der Fau-Timer Library evaluiert. Die Library hat die Zeitunterschiede sortiert und bestimmte Bereiche zwischeneinander verglichen. Das Ergebnis der Evaluierung in Grafik 1 zeigt die Zeitmessungen von validen und invaliden Nachrichten zwischen dem 35. und 45. Perzentil. Auf der linken Achse ist die Anzahl der Ticks, welche eine Nachrichtenbearbeitung in Anspruch nimmt. Wie aus der Grafik sichtbar ist, dauert die Bearbeitung einer invaliden Nachricht ungefähr 200 Ticks weniger als die Bearbeitung einer validen Nachricht. Dies bestätigte unsere Vermutung, dass die Funktion nicht konstante Zeit dauern wird. Jedoch ist der Unterschied von 200 Ticks – in unserem Fall ca. 0,1 Mikrosekunden – so klein, dass ein praktischer Angriff damit *unmöglich* ist (im Vergleich wurden praktische Angriffe durchgeführt, wenn die Unterschiede ca. zehn Mikrosekunden betrugen (MSWS)).

6.1.3.5 Identifikation der Compiler-Flags

In dem OpenSSL Code wurden in den relevanten Funktionen keine Compiler Flags gefunden, welche eine Auswirkung auf die Gegenmaßnahmen gegen Bleichenbacher-Angriffe hätten.

6.1.3.6 Ausblick

Unsere Untersuchung hat keine praktischen Timing-Angriffsmöglichkeiten gefunden. Auf der CCS 2014 haben aber Zhang et al. neue Bleichenbacher-Angriffe in PaaS Clouds vorgestellt. Dabei haben die Forscher Flush-Reload Angriffe gegen OpenSSL genutzt. Eine Voraussetzung für diese Angriffe ist es, dass der Angreifer auf der selben CPU wie sein Opfer Berechnungen durchführen kann:

- Yinqian Zhang, Ari Juels, Michael K. Reiter, Thomas Ristenpart: Cross-Tenant Side-Channel Attacks in PaaS Clouds (ZJRR)

6.1.4 TLS-DH: Kleine Untergruppen

Wie im AP 3.1 beschrieben wurde, sendet der Angreifer bei dem Angriff mit kleinen Untergruppen an den Server ClientKeyExchange-Nachricht mit Zahlen, die in einer kleinen Untergruppe liegen. Wenn der Server die Ordnung dieser Zahl nicht überprüft, kann der Angreifer den privaten DH-Schlüssel extrahieren.

Eine effektive Gegenmaßnahme gegen diese Angriffe ist es, für jede Verbindung einen neuen ephemeral DH-Schlüssel zu benutzen. Dies kann in OpenSSL mit der SSL_OP_SINGLE_DH_USE Option erzwungen werden.

6.1.4.1 Identifikation relevanter Funktionen

Die relevanten Funktionen befinden sich in folgenden Dateien:

- **`ssl/s3_srvr.c: ssl3_send_server_key_exchange`** (Z. 1621 – 1673): ServerKeyExchange Behandlung, Aufruf der DH-Schlüssel-Generierung, wenn noch kein DH-Schlüssel vorhanden ist (passiert beim Start des Servers mit Anonymous DH) oder wenn kein SSL_OP_SINGLE_DH_USE gesetzt ist.
- **`crypto/dh/dh_key.c: generate_key`** (Z. 117 – 200): DH-Schlüssel-Generierung.

6.1.4.2 Analyse

Für diese Analyse wurde ein SSL-Client in Java mit SSLSocket geschrieben. Dieser führt einen SSL-Handshake aus. Der Client wurde in eine ausführbare JAR-Datei gepackt, die mit folgendem Kommando gestartet wird:

```
java -Djavax.net.debug=ssl -jar TLS-Client-1.0-SNAPSHOT.jar
```

Mit **`-Djavax.net.debug=ssl`** wird SSL-Debugging aktiviert. Für diese Variante haben wir uns deswegen entschieden, weil der Java-Client mehr Ausgaben als der OpenSSL-Client zur Verfügung stellt, die bei der Analyse von ServerKeyExchange-Nachrichten notwendig sind.

Der SSL-Server wurde mit folgender Konfiguration gestartet, welche als Eingaben einen DH-Schlüssel und ein DSA-Zertifikat nimmt:

```
openssl s_server -accept 51624 -key /home/developer/TLS-Attacker/resources/dsa-private.pem -cert /home/developer/TLS-Attacker/resources/dsa-cert.pem -dhparam /home/developer/TLS-Attacker/resources/dhparam.pem -dkey /home/developer/TLS-Attacker/resources/dh-private.pem -debug
```

Der SSL-Client verbindet sich mit dem OpenSSL-Server mit TLS 1.2 und mit der DHE-DSS-

AES256-GCM-SHA384 Ciphersuite.

Nach der Durchführung von mehrmaligen Verbindungen wurde festgestellt, dass obwohl der Server mit einer DHE-Ciphersuite arbeiten sollte, die DH-Werte in der ServerKeyExchange-Nachricht nicht verändert werden. Der SSL-Client gibt im Debug-Modus bei jedem TLS-Handshake folgende Werte aus:

```
*** Diffie-Hellman ServerKeyExchange

DH Modulus:  { 219, 205, 204, 42, 195, 179, 204, 110, 146, 37, 37, 195, 2, 77, 115, 85,
58, 42, 53, 16, 54, 198, 16, 58, 224, 108, 108, 91, 217, 245, 7, 137, 121, 63, 71, 77,
18, 111, 27, 34, 134, 146, 176, 231, 103, 91, 17, 241, 124, 184, 132, 215, 192, 241, 250,
197, 189, 147, 120, 69, 177, 169, 148, 227, 232, 127, 115, 172, 27, 160, 236, 234, 128,
37, 241, 245, 174, 41, 147, 220, 180, 150, 241, 238, 236, 70, 116, 171, 212, 198, 252,
55, 20, 227, 172, 96, 77, 44, 31, 219, 56, 117, 225, 2, 45, 114, 56, 160, 211, 51, 202,
229, 184, 246, 37, 128, 14, 187, 5, 56, 103, 164, 147, 35, 154, 145, 89, 3 }

DH Base:  { 2 }

Server DH Public Key:  { 14, 171, 165, 166, 140, 187, 207, 111, 14, 103, 178, 57, 238,
187, 252, 63, 11, 7, 29, 62, 195, 94, 133, 169, 217, 5, 178, 120, 58, 56, 146, 200, 193,
118, 211, 244, 190, 29, 55, 12, 67, 201, 62, 151, 34, 91, 42, 64, 111, 26, 242, 195, 53,
182, 60, 52, 76, 210, 129, 179, 213, 179, 136, 192, 217, 91, 146, 180, 62, 160, 255, 241,
200, 107, 54, 110, 193, 220, 9, 89, 136, 131, 43, 82, 76, 198, 99, 86, 28, 160, 172, 187,
254, 96, 220, 106, 254, 151, 64, 58, 16, 115, 105, 118, 103, 138, 127, 128, 74, 48, 197,
81, 221, 83, 166, 172, 99, 49, 225, 201, 75, 138, 176, 46, 121, 52, 217, 201 }
```

Nachdem wir im SSL-Context von OpenSSL den Wert von **SSL_OP_SINGLE_DH_USE** gesetzt haben (beispielhaft in der Zeile 1880, apps/s_server.c):

```
SSL_CTX_set_options(ctx, SSL_OP_SINGLE_DH_USE);
```

wurden die DH-Schlüssel bei jedem neuen TLS-Handshake neu generiert (Funktion **generate_key**). Das hat dazu geführt, dass der Server mit unterschiedlichen ServerKeyExchange-Nachrichten geantwortet hat:

1. TLS-Handshake:

```
*** Diffie-Hellman ServerKeyExchange

DH Modulus:  { 219, 205, 204, 42, 195, 179, 204, 110, 146, 37, 37, 195, 2, 77, 115, 85,
58, 42, 53, 16, 54, 198, 16, 58, 224, 108, 108, 91, 217, 245, 7, 137, 121, 63, 71, 77,
18, 111, 27, 34, 134, 146, 176, 231, 103, 91, 17, 241, 124, 184, 132, 215, 192, 241, 250,
197, 189, 147, 120, 69, 177, 169, 148, 227, 232, 127, 115, 172, 27, 160, 236, 234, 128,
37, 241, 245, 174, 41, 147, 220, 180, 150, 241, 238, 236, 70, 116, 171, 212, 198, 252,
55, 20, 227, 172, 96, 77, 44, 31, 219, 56, 117, 225, 2, 45, 114, 56, 160, 211, 51, 202,
229, 184, 246, 37, 128, 14, 187, 5, 56, 103, 164, 147, 35, 154, 145, 89, 3 }

DH Base:  { 2 }

Server DH Public Key:  { 158, 126, 252, 38, 232, 228, 213, 73, 118, 2, 12, 102, 225, 141,
27, 85, 255, 157, 223, 32, 190, 216, 195, 90, 144, 233, 245, 97, 30, 53, 60, 137, 216,
211, 185, 73, 241, 178, 238, 80, 6, 245, 243, 174, 123, 124, 44, 115, 218, 246, 83, 59,
234, 251, 114, 29, 37, 197, 198, 196, 10, 122, 0, 172, 68, 63, 36, 94, 215, 41, 204, 5,
64, 249, 41, 164, 92, 159, 17, 18, 13, 202, 33, 177, 186, 134, 120, 104, 72, 206, 93,
255, 131, 66, 20, 158, 43, 227, 59, 84, 191, 78, 63, 238, 97, 235, 254, 156, 29, 0, 255,
```

```
241, 21, 41, 195, 236, 89, 141, 160, 123, 215, 226, 4, 94, 216, 105, 1, 11 }
```

2. TLS-Handshake:

*** Diffie-Hellman ServerKeyExchange

```
DH Modulus:  { 219, 205, 204, 42, 195, 179, 204, 110, 146, 37, 37, 195, 2, 77, 115, 85,
58, 42, 53, 16, 54, 198, 16, 58, 224, 108, 108, 91, 217, 245, 7, 137, 121, 63, 71, 77,
18, 111, 27, 34, 134, 146, 176, 231, 103, 91, 17, 241, 124, 184, 132, 215, 192, 241, 250,
197, 189, 147, 120, 69, 177, 169, 148, 227, 232, 127, 115, 172, 27, 160, 236, 234, 128,
37, 241, 245, 174, 41, 147, 220, 180, 150, 241, 238, 236, 70, 116, 171, 212, 198, 252,
55, 20, 227, 172, 96, 77, 44, 31, 219, 56, 117, 225, 2, 45, 114, 56, 160, 211, 51, 202,
229, 184, 246, 37, 128, 14, 187, 5, 56, 103, 164, 147, 35, 154, 145, 89, 3 }

DH Base:  { 2 }

Server DH Public Key:  { 39, 172, 216, 150, 20, 181, 157, 19, 62, 76, 14, 67, 30, 52, 77,
101, 35, 135, 167, 214, 26, 227, 96, 245, 97, 200, 176, 253, 214, 173, 77, 231, 110, 237,
232, 189, 195, 218, 58, 60, 105, 106, 114, 39, 120, 246, 25, 83, 203, 203, 19, 213, 168,
113, 106, 29, 119, 252, 62, 221, 162, 164, 44, 61, 139, 44, 234, 226, 255, 109, 152, 246,
35, 35, 15, 175, 131, 225, 163, 203, 229, 154, 205, 13, 24, 54, 90, 194, 189, 89, 98,
211, 244, 208, 6, 181, 42, 150, 126, 13, 147, 151, 169, 32, 127, 30, 16, 85, 216, 38,
219, 103, 234, 59, 122, 69, 228, 73, 189, 138, 37, 145, 136, 87, 153, 29, 195, 165 }
```

Wir haben überprüft, dass durch das Einschalten des **SSL_OP_SINGLE_DH_USE** Flags nicht nur das Verhalten bei den Ephemeral DH-Ciphersuites geändert wird. Der Server hat neue DH-Schlüssel bei jedem TLS-Handshake erzeugt, z.B. bei der ADH-AES256-SHA256 Ciphersuite.

Unsere Analyse hat damit gezeigt, dass Perfect-Forward-Secrecy (und daneben auch eine erweiterte Gegenmaßnahme gegen Angriffe mit kleinen DH-Untergruppen) nur dann korrekt eingesetzt wird, wenn es auf dem OpenSSL-Server mit dem Befehl

```
SSL_CTX_set_options(ctx, SSL_OP_SINGLE_DH_USE);
```

erzwungen wird. Sonst benutzt der Server für die Kommunikation nur einen statischen DH-Schlüssel, der bei dem Server-Start einmalig generiert wird.

6.1.4.3 Identifikation der Compiler-Flags

In dem OpenSSL Code wurden in den relevanten Funktionen keine Compiler Flags gefunden, welche eine Auswirkung auf die Gegenmaßnahmen gegen Angriffe mit kleinen DH-Untergruppen hätten.

6.1.5 Elliptische Kurven

Eine Voraussetzung für die Ausführung der Angriffe auf elliptische Kurven aus AP 3.1 ist, dass der Server nicht überprüft, ob die gesendeten Punkte auf der Kurve liegen. Dies sollte ein Teil der Untersuchung in diesem Abschnitt sein.

Laut BSI-TR-02102-2 müssen beim Einsatz von elliptischen Kurven stets kryptographisch starke Kurven über endlichen Körpern der Form F_p (p prim) verwendet werden. Zusätzlich wird empfoh-

len, nur Named curves einzusetzen, um Angriffe über nicht verifizierte schwache Domainparameter zu verhindern. Es werden folgende Named Curves empfohlen:

- brainpoolP256r1, brainpoolP384r1, brainpoolP512r1, secp224r1, secp256r1, secp384r1

OpenSSL 1.0.1 unterstützt nicht die Brainpool-Kurven. Daher sind nur die Kurven secp224r1, secp256r1, secp384r1 ein Teil der Untersuchung.

Ab Version 1.0.2 werden auch die Brainpool-Kurven unterstützt.

Die OpenSSL TLS-Implementierung unterstützt *nur* Named Curves, obwohl Funktionen zur Bearbeitung von beliebigen elliptischen Kurven im OpenSSL-Code teilweise vorhanden sind (siehe **`crypto/ec/ec_asn1.ccrypto/ec/ec_asn1.c`**) (NCsupport).

6.1.5.1 Identifikation Relevanter Funktionen

Die relevanten Funktionen für die Bearbeitung von Kurven über endlichen Körpern der Form F_p befinden sich in folgenden Dateien:

- **ssl/s3_srvr.c** (Z. 2519 – 2674): ClientKeyExchange-Bearbeitung, EC-Punkt-Dekodierung, Aufruf der PremasterSecret- und MasterSecret-Berechnung.
- **crypto/ecc/ec_oct.c: `int EC_POINT_oct2point`** (Z. 167): Überprüfung, ob die Gruppe des eingehenden Punktes in ClientKeyExchange gleich ist der benutzten Gruppe, Aufruf der Dekodierungsfunktion.
- **crypto/ecc/ecp_oct.c: `int ec_GFp_simple_oct2point`** (Z. 325): Punkt-Dekodierung, Aufruf der Überprüfung ob der eingehende Punkt auf der Kurve liegt.
- **crypto/ecc/ec_lib.c: `int EC_POINT_is_on_curve`** (Z. 975): Aufruf der Überprüfung, ob der eingehende Punkt auf der Kurve liegt.
- **crypto/ecc/ecp_smpl.c: `int ec_GFp_simple_is_on_curve`** (Z. 940): Überprüfung, ob der Punkt auf der Kurve liegt.
- **test/ectest.c: `void prime_field_tests`** (Z. 310): Test für kryptographische Operationen auf elliptischen Kurven. Der Test beinhaltet explizite Überprüfung, ob ein Punkt auf der Kurve liegt.

Ähnliche Funktionen können für Kurven der Form F_2^m gefunden werden.

6.1.5.2 Analyse

Unsere automatisierte Analyse mit dem T.I.M.E. Framework als auch unsere manuelle Source-Code Analyse haben ergeben, dass der Server während des Handshakes immer überprüft, ob der eingehende Punkt auf der Kurve liegt. Damit ist es für den Angreifer *nicht* möglich, den Server dazu zu zwingen, mit einem falschen Punkt eine Berechnung zu starten und das MasterSecret zu berechnen.

Im Folgenden ist noch die Funktion **ec_GFp_simple_is_on_curve** aus der Datei **rypto/ecc/ecp_smpl.c** dargestellt, welche für die Überprüfung des Punktes zuständig ist:

```
int ec_GFp_simple_is_on_curve(const EC_GROUP *group, const EC_POINT *point, BN_CTX *ctx)
	{
	int (*field_mul)(const EC_GROUP *, BIGNUM *, const BIGNUM *, const BIGNUM *,
BN_CTX *);
	int (*field_sqr)(const EC_GROUP *, BIGNUM *, const BIGNUM *, BN_CTX *);
	const BIGNUM *p;
	BN_CTX *new_ctx = NULL;
	BIGNUM *rh, *tmp, *Z4, *Z6;
	int ret = -1;
	if (EC_POINT_is_at_infinity(group, point))
		return 1;
	field_mul = group->meth->field_mul;
	field_sqr = group->meth->field_sqr;
	p = &group->field;

	if (ctx == NULL)
		{
		ctx = new_ctx = BN_CTX_new();
		if (ctx == NULL)
			return -1;
		}
	BN_CTX_start(ctx);
	rh = BN_CTX_get(ctx);
	tmp = BN_CTX_get(ctx);
	Z4 = BN_CTX_get(ctx);
	Z6 = BN_CTX_get(ctx);
	if (Z6 == NULL) goto err;

	/* We have a curve defined by a Weierstrass equation
```

```
 *      y^2 = x^3 + a*x + b.
 * The point to consider is given in Jacobian projective coordinates
 * where  (X, Y, Z)  represents  (x, y) = (X/Z^2, Y/Z^3).
 * Substituting this and multiplying by  Z^6  transforms the above equation into
 *      Y^2 = X^3 + a*X*Z^4 + b*Z^6.
 * To test this, we add up the right-hand side in 'rh'.
 */
/* rh := X^2 */
if (!field_sqr(group, rh, &point->X, ctx)) goto err;
if (!point->Z_is_one)
       {
       if (!field_sqr(group, tmp, &point->Z, ctx)) goto err;
       if (!field_sqr(group, Z4, tmp, ctx)) goto err;
       if (!field_mul(group, Z6, Z4, tmp, ctx)) goto err;

       /* rh := (rh + a*Z^4)*X */
       if (group->a_is_minus3)
              {
              if (!BN_mod_lshift1_quick(tmp, Z4, p)) goto err;
              if (!BN_mod_add_quick(tmp, tmp, Z4, p)) goto err;
              if (!BN_mod_sub_quick(rh, rh, tmp, p)) goto err;
              if (!field_mul(group, rh, rh, &point->X, ctx)) goto err;
              }
       else
              {
              if (!field_mul(group, tmp, Z4, &group->a, ctx)) goto err;
              if (!BN_mod_add_quick(rh, rh, tmp, p)) goto err;
              if (!field_mul(group, rh, rh, &point->X, ctx)) goto err;
              }
       /* rh := rh + b*Z^6 */
       if (!field_mul(group, tmp, &group->b, Z6, ctx)) goto err;
       if (!BN_mod_add_quick(rh, rh, tmp, p)) goto err;
       }
else
       {
       /* point->Z_is_one */
```

```
            /* rh := (rh + a)*X */
            if (!BN_mod_add_quick(rh, rh, &group->a, p)) goto err;
            if (!field_mul(group, rh, rh, &point->X, ctx)) goto err;
            /* rh := rh + b */
            if (!BN_mod_add_quick(rh, rh, &group->b, p)) goto err;
            }
      /* 'lh' := Y^2 */
      if (!field_sqr(group, tmp, &point->Y, ctx)) goto err;
      ret = (0 == BN_ucmp(tmp, rh));
 err:
      BN_CTX_end(ctx);
      if (new_ctx != NULL)
            BN_CTX_free(new_ctx);
      return ret;
      }
```

Eine ähnliche Funktion (**ec_GF2m_simple_is_on_curve**) kann in der Datei **crypto/ecc/ec2_smpl.c** gefunden werden. Diese wird für die Überprüfung der Punkte auf Kurven über endlichen Körper der Form F_2^m eingesetzt.

6.1.5.3 Analyse der Perfect Forward Secrecy

Bei der Analyse von DH-basierten Ciphersuites wurde festgestellt, dass Perfect Forward Secrecy nur dann angewandt wird, wenn explizit **SSL_OP_SINGLE_DH_USE** gesetzt wird. ECDH-basierte Ciphersuites bieten eine ähnliche Einstellung an: **SSL_OP_SINGLE_ECDH_USE**

Mit dem Java-basierten SSL-Client haben wir getestet, ob die Perfect Forward Secrecy benutzt wird, auch wenn diese Einstellung nicht explizit eingeschaltet wird.

Der OpenSSL-Server wurde dabei wie folgt gestartet:

```
openssl s_server -accept 51624 -key /home/developer/TLS-Attacker/resources/ec256-private.pem -cert /home/developer/TLS-Attacker/resources/ec256-cert.pem -debug
```

Zwischen dem Client und dem Server wurde in diesem Fall die Ciphersuite TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA ausgehandelt.

Unsere Analyse hat gezeigt, dass auch hier der Server die gleichen Werte im ServerKeyExchange zur Verfügung stellt, in unserem Fall:

```
*** ECDH ServerKeyExchange
Server key: SunPKCS11-NSS EC public key, 256 bits (id 2, session object)
```

```
                                    public                  x                   coord:
26748954278827642868231577030487080474942228497987864730128793043715839160132

                                    public                  y                   coord:
33700388554067854212161888279433212173484625275918030855184458500512122783 27

  parameters: secp256r1 [NIST P-256, X9.62 prime256v1] (1.2.840.10045.3.1.7)
```

Diesen Test haben wir nach dem Einschalten des Flags noch einmal gestartet, dabei haben wir beobachtet, dass sich die Werte im ServerKeyExchange bei jedem TLS-Handshake ändern:

1. TLS-Handshake:

```
*** ECDH ServerKeyExchange

Server key: SunPKCS11-NSS EC public key, 256 bits (id 2, session object)

                                    public                  x                   coord:
11260214399022885671884644412121454344276138709393725288127695941098673389881 2

                                    public                  y                   coord:
24369394455487105988508222509293144535960989083856760298990887749250773426746
```

2. TLS-Handshake:

```
*** ECDH ServerKeyExchange

Server key: SunPKCS11-NSS EC public key, 256 bits (id 2, session object)

                                    public                  x                   coord:
25752815458804979310798596058706666074825032542156599911185391978729102878684

                                    public                  y                   coord:
66184603332114782298453477003872770055665476808282428562271705163385624394531
```

Perfect Forward Secrecy ist also bei ECDHE-Ciphersuites nur dann gewährleistet, wenn der **SSL_OP_SINGLE_ECDH_USE** Flag gesetzt ist. Dies wird mit dem folgenden Befehl erzwungen:

```
SSL_CTX_set_options(ctx, SSL_OP_SINGLE_ECDH_USE);
```

Dieses Verhalten wurde schon in der Publikation von Brumley et al. beobachtet [BBPV2012].

6.1.5.4 Identifikation der Compiler-Flags

In dem OpenSSL Code wurden in den relevanten Funktionen keine Compiler Flags gefunden, welche eine Auswirkung auf die Gegenmaßnahmen gegen diese Angriffe hätten.

6.1.6 Brumley-Boneh

Brumley und Boneh haben im Jahr 2003 neue Timing-basierte Seitenkanalangriffe gegen die RSA-Verschlüsselung vorgestellt [BB2003]. Das Ziel der Angriffe war die Berechnung des privaten RSA-Schlüssels anhand der Dauer von RSA-Exponentiationen mit privatem Schlüssel (Entschlüsselung oder Signaturerstellung).

Als Gegenmaßnahme wurde RSA-Blinding für private RSA-Operationen vorgestellt. RSA-Blinding ist seit 2003 standardmäßig in OpenSSL aktiviert. Man kann diesen Mechanismus mit einem Compiler-Flag ausschalten. Ursprünglich wurde das Compiler-Flag OPENSSL_NO_FORCE_RSA_BLINDING genannt (RSAblinding). In der jetzigen OpenSSL-Version wird für diese Funktionalität RSA_FLAG_NO_BLINDING Macro eingesetzt.

6.1.6.1 Identifikation Relevanter Funktionen

Die relevanten Funktionen befinden sich in folgenden Dateien:

- **`crypto/rsa/rsa_eay.c: RSA_eay_private_encrypt, RSA_eay_private_decrypt`** (Z. 350 – 630): RSA-Operationen mit privatem Exponenten (RSA-Signaturerstellung und -Entschlüsselung).
- **`crypto/rsa/rsa_crpt.c: RSA_setup_blinding`** (Z. 190 – 255): Einrichten der Blinding-Parameter, falls diese noch nicht vorhanden sind.
- **`crypto/bn/bn_blind.c: BN_BLINDING_create_param`** (Z. 306 – 385): Aufruf der Parameter-Generierung, erzeugt neue Elemente **`A`** und $\mathbf{A_{inv}}$, wobei $A \cdot A_{inv} \equiv 1 \, mod \, N$ mit der Funktion BN_rand_range, so dass $0 \leq A < N$. Wenn **`A`** falsch generiert wurde (es existiert keine Inverse), wird die Generierung wierholt und **`A`** noch einmal generiert.
- **`crypto/bn/bn_blind.c: BN_BLINDING_convert_ex`** (Z. 224 – 250): Modulare Multiplikation mit den Blinding-Parametern.
- **`crypto/bn/bn_blind.c: BN_BLINDING_update`** (Z. 186 – 216): Die Blinding-Parameter können bis zu 32 mal aktualisiert werden, so dass jede RSA-Operation mit unterschiedlichen Blinding-Parametern ausgeführt wird. Bei der Aktualisierung werden **`A`** und $\mathbf{A_{inv}}$ neu gesetzt: $A' \equiv A \cdot A \, mod \, N$ und $A'_{inv} \equiv A_{inv} \cdot A_{inv} \, mod \, N$. Nach 32 Aktualisierungen wird wieder Funktion **`BN_BLINDING_create_param`** ausgeführt, die komplett neue Elemente **`A`** und $\mathbf{A_{inv}}$ erzeugt.
- **`crypto/bn/bn_rand.c: BN_rand_range`** (Z. 230 – 294): Generierung einer zufälligen Zahl (in diesem Zusammenhang **`A`**, so dass $0 \leq A < N$). Es wird ein echter Random Number Generator eingesetzt.
- **`crypto/bn/bn_gcd.c: BN_mod_inverse`** (Z. 209 – 500): Generierung einer Inverse aus einer gegebenen Zahl.

6.1.6.2 Analyse

Während unserer Analyse haben wir einen OpenSSL-Server gestartet und diesen mit einem RSA-basierten Handshake getestet. Dabei haben wir den Control-Flow beobachtet. Bei dem *ersten* RSA-basierten Handshake wird vor der Entschlüsselung die Funktion

`blinding = rsa_get_blinding(rsa, &local_blinding, ctx)` (Z.538)

ausgeführt. Dann wird die Funktion **`RSA_setup_blinding`** ausgeführt, welche die Funktion **`BN_BLINDING_create_param`** anspricht. **`BN_BLINDING_create_param`** generiert eine zufällige Zahl und ihre Inverse für die Blinding-Parameter (Funktionen **`BN_rand_range`** und **`BN_mod_inverse`**). Damit sind die Blinding-Parameter initialisiert. Anschließend kann die Funktion

`rsa_blinding_convert(blinding, f, unblind, ctx)` (Z. 553)

vor der Entschlüsselung und die Funktion

`rsa_blinding_invert(blinding, ret, unblind, ctx)` (Z. 589)

nach der Entschlüsselung ausgeführt werden.

Bei jeder neuen Blinding-Operation (in der Funktion **`BN_BLINDING_convert_ex`**) werden die Blinding-Parameter aktualisiert (Funktion **`BN_BLINDING_update`**). Bei der Aktualisierung werden die Elemente **A** und $\mathbf{A_{inv}}$ nicht komplett neu generiert, sie werden anhand von den bestehenden Elementen wie folgt berechnet: $A' \equiv A \cdot A \bmod N$ und $A'_{inv} \equiv A_{inv} \cdot A_{inv} \bmod N$. Damit wird gewährleistet, dass $A' \cdot A'_{inv} \equiv 1 \bmod N$. Jede neue RSA-Operation wird somit mit unterschiedlichen Blinding-Parametern ausgeführt. Nach 32 RSA-Operationen (die Zahl 32 wird mit **`BN_BLINDING_COUNTER`** definiert) werden mit der Funktion **`RSA_setup_blinding`** die Blinding-Parameter komplett neu generiert.

Ein ähnliches Verhalten war auch bei der Signaturerstellung zu sehen.

6.1.6.3 Identifikation der Compiler-Flags

In dem OpenSSL Code wurden in den relevanten Funktionen keine Compiler Flags gefunden, welche eine Auswirkung auf die Gegenmaßnahmen gegen diese Angriffe hätten.

Die RSA-Blinding-Funktionalität kann über ein Macro gesteuert werden. Mit dem Flag RSA_FLAG_NO_BLINDING kann das RSA-Blinding ausgeschaltet werden. Standardmäßig ist RSA-Blinding eingeschaltet.

6.1.7 Verfrühte ChangeCipherSpec-Nachricht

In dem OpenSSL-Security-Advisory vom Juni 2014 (EarlyCCSadv) wurde ein neuer Angriff gegen OpenSSL beschrieben, welcher auf einer fehlerhaften Verifikation der Nachrichten-Reihenfolge basiert. Ein Web-Angreifer kann damit den OpenSSL-Client und OpenSSL-Server dazu bringen, bekannte Schlüssel (welche auf einem MasterSecret=NULL basieren) für den Record-Layer auszuhandeln. Damit kann der Angreifer die gesamte Kommunikation entschlüsseln oder die Nachrichten fälschen. Verbundbar sind dabei die nachfolgend aufgeführten Versionen:

- Client: OpenSSL kleiner oder gleich 0.9.8y, 1.0.0l, 1.0.1g
- Server: OpenSSL kleiner oder gleich 1.0.1g, Beta-Versionen von 1.0.2

Nur eine Kombination aus diesen Versionen kann zu einem erfolgreichen Angriff führen.

6.1.7.1 Identifikation Relevanter Funktionen

Die relevanten Funktionen befinden sich in folgenden Dateien:

- **ssl/s3_pkt.c: `int ssl3_do_change_cipher_spec`** (Z. 1421 – 1471): Bearbeitung der ChangeCipherSpec-Nachricht und Berechnung von verifyData für die Verifikation der Finished-Nachricht.
- **ssl/s3_pkt.c: `int ssl3_read_bytes`** (Z.1280 – 1310): Allgemeine Bearbeitung von SSL/TLS-Nachrichten.
- **ssl/s3_srvr.c: `int ssl3_accept`**: SLL/TLS-Zustandsautomat, spezifisch für die Server-Seite.

6.1.7.2 Analyse

Als Gegenmaßnahme gegen diese Angriffe auf der Server-Seite wurde in der Funktion **`ssl3_accept`** ein neues Flag SSL3_FLAGS_CCS_OK definiert, welcher an bestimmten Stellen gesetzt ist, ab wann eine ChangeCipherSpec-Nachricht akzeptiert werden. Wenn die ChangeCipherSpec-Nachricht in einem späteren Protokollverlauf empfangen wird, wird überprüft ob das Flag gesetzt worden ist.

Die Funktion **`ssl3_accept`** wurde wie folgt erweitert:

```
int ssl3_do_change_cipher_spec(SSL *s)
	{
	int i;
	const char *sender;
	int slen;

	if (s->state & SSL_ST_ACCEPT)
		i=SSL3_CHANGE_CIPHER_SERVER_READ;
	else
		i=SSL3_CHANGE_CIPHER_CLIENT_READ;

	if (s->s3->tmp.key_block == NULL)
		{
+		if (s->session->master_key_length == 0)
```

```
+                       {
+                       SSLerr(SSL_F_SSL3_DO_CHANGE_CIPHER_SPEC,SSL_R_UNEXPECTED_CCS);
+                       return (0);
+                       }
                if (s->session == NULL)
                        {
                        /* might happen if dtls1_read_bytes() calls this */
                        SSLerr(SSL_F_SSL3_DO_CHANGE_CIPHER_SPEC,SSL_R_CCS_RECEIVED_EARLY);
                        return (0);
                        }

                s->session->cipher=s->s3->tmp.new_cipher;
                if (!s->method->ssl3_enc->setup_key_block(s)) return(0);
                }
        ...
```

In der Funktion wurde eine zusätzliche Überprüfung der MasterSecret-Länge hinzugefügt: Wenn ein uninitialisiertes MasterSecret erkannt wird, so wird die weitere Bearbeitung abgebrochen.

In der Funktion **`ssl3_read_bytes`** wurde eine Überprüfung mit dem SSL3_FLAGS_CCS_OK gesetzt.

```
        ...
        if (rr->type == SSL3_RT_CHANGE_CIPHER_SPEC)
                {
                /* 'Change Cipher Spec' is just a single byte, so we know
                 * exactly what the record payload has to look like */
                if (   (rr->length != 1) || (rr->off != 0) ||
                        (rr->data[0] != SSL3_MT_CCS))
                        {
                        al=SSL_AD_ILLEGAL_PARAMETER;
                        SSLerr(SSL_F_SSL3_READ_BYTES,SSL_R_BAD_CHANGE_CIPHER_SPEC);
                        goto f_err;
                        }

                /* Check we have a cipher to change to */
                if (s->s3->tmp.new_cipher == NULL)
                        {
                        al=SSL_AD_UNEXPECTED_MESSAGE;
```

```
                    SSLerr(SSL_F_SSL3_READ_BYTES,SSL_R_CCS_RECEIVED_EARLY);
                    goto f_err;
                    }
+           if (!(s->s3->flags & SSL3_FLAGS_CCS_OK))
+                   {
+                   al=SSL_AD_UNEXPECTED_MESSAGE;
+                   SSLerr(SSL_F_SSL3_READ_BYTES,SSL_R_UNEXPECTED_CCS);
+                   goto f_err;
+                   }

            rr->length=0;
            …
```

6.1.7.3 Technische Angriffsbeschreibung

Mit dem TLS-Attacker Framework haben wir den Angriff implementiert und analysiert. Folgender Nachrichten-Ablauf führt zu einem korrekten RSA-Handshake, wenn ein OpenSSL 1.0.1g Server eingesetzt wird:

1. ClientHello
2. ServerHello, ServerCertificate, ServerHelloDone
3. ChangeCipherSpec
4. ClientKeyExchange: Im Vergleich zu einem validen Handshake muss bei dem Angriff die ClientKeyExchange-Nachricht im Record-Layer verschlüsselt werden. Für die Schlüssel-Initialisierung wird ein MasterSecret mit 48 Null-Bytes benutzt.
5. ClientFinished: Für die Verschlüsselung im Record-Layer wird weiterhin der selbe Schlüssel benutzt. Für die Berechnung von VerifyData in der Finished-Nachricht wird ein neuer Schlüssel eingesetzt, welcher auf dem PremasterSecret aus der ClientKeyExchange-Nachricht basiert.

In der virtuellen Maschine befindet sich ein Client, welcher diesen Handshake bis zur valid-verschlüsselten Finished-Nachricht ausführt, womit der komplette Handshake-Ablauf mit einem anfälligen OpenSSL-Server (Version 1.0.1g) nachvollzogen werden kann.

Wir haben weiterhin verifiziert, dass die Gegenmaßnahme in der neueren OpenSSL-Version 1.0.1h korrekt implementiert wurde und dass der Angriff nicht mehr möglich ist. Eine verfrühte ChangeCipherSpec-Nachricht wird von dem Server abgelehnt.

6.1.8 Lucky 13

Der Lucky 13-Angriff (Lucky13) hat gezeigt, dass das MAC-then-PAD-then-ENCRYPT-Schema im Record-Layer zu Timing-Problemen führen kann. Konkret werden bei diesem Angriff Zeitdifferenzen bei der MAC-Validierung von unterschiedlich langen Nachrichten gemessen, wodurch die Chiffretexte entschlüsselt werden können.

Eine detaillierte Analyse von Lucky 13 wurde von Adam Langley durchgeführt (Lucky13imp). In seiner Analyse berichtet er von seinem Patch in OpenSSL, welcher eine konstante CBC-Entschlüsselung und MAC-Überprüfung erzwingt. Damit werden Zeitunterschiede von höchstens 200 CPU-Ticks gemessen, was praktische Angriffe sehr schwer umzusetzen macht. Sein Code ist direkt in der OpenSSL-Implementierung zu finden (**ssl/s3_cbc.c**).

Aus zeitlichen Gründen und wegen der detaillierten Analyse von Adam Langley wurde es auf diese Angriffe nicht tiefer eingegangen.

6.1.8.1 Identifikation der Compiler-Flags

In dem OpenSSL Code wurden in den relevanten Funktionen der Datei **ssl/s3_cbc.c** keine Compiler Flags gefunden, welche eine Auswirkung auf die Gegenmaßnahmen gegen diese Angriffe hätten.

6.1.9 Heartbleed

Der Heartbleed-Angriff gehört zu den jüngsten Angriffen auf OpenSSL. Die Schwachstelle wurde durch eine fehlerhafte Längenüberprüfung der Heartbeat-Requests eingeführt. Der Angriff wurde in OpenSSL 1.0.1g im April 2014 verhindert.

Die Heartbeat-(D)TLS-Extension (HeartbeatRFC) (welche die Heartbeat-Nachrichten definiert) wurde spezifiziert um das Aufrechterhalten einer Verbindung zwischen dem Client und dem Server auf der (D)TLS-Ebene überprüfen zu können. Dabei sendet ein Kommunikationsteilnehmer eine bis zu 16 kByte große Anfrage mit beliebigen Daten an die Gegenseite. Die Gegenseite sendet die Daten zurück. Somit kann zusätzlich zum Aufrechterhalten der Verbindung auch die Response-Performanz des TLS-Peers getestet werden.

Eine Heartbeat-Nachricht hat folgende Struktur:

- HeartbeatMessageType (1 Byte): identifiziert den Typ der Heartbeat-Nachricht, 1 steht für HeartbeatRequest, 2 für HeartbeatResponse.
- PayloadLength (2 Bytes): definiert die Länge des Payloads.
- Payload: Daten, welche reflektiert werden sollten.
- Padding (16-256 Bytes): Zufälliges Padding, welches ignoriert wird.

Wenn ein Heartbeat-Request an den Server geschickt wird, bestimmt dieser zuerst die Payload-Län-

ge. Danach liest der Server die Payload-Daten aus (anhand der Payload-Länge). Anschließend antwortet der Server mit einer Heartbeat-Response, welche die ausgelesenen Payload-Daten und ein neues Padding beinhaltet.

OpenSSL Version 1.0.1 (bis zur Version 1.0.1f) hat nicht überprüft, ob die Payload-Länge korrekt angegeben ist und ob diese nicht größer als die Anzahl der Bytes im Payload ist. Dies hat zu einem Buffer-Overread geführt: da im Payload nicht genug Daten liegen, werden die Daten aus einem zufälligen Adressraum ausgelesen und an den Client reflektiert. Die Daten konnten auch vertrauliches Schlüsselmaterial enthalten (HeartbleedCloud).

Der Angriff funktioniert nur für den Fall, dass der angegriffene TLS-Kommunikationsteilnehmer die Heartbleed-Extension aktiviert hat. In OpenSSL ist diese Extension standardmäßig aktiviert.

Identifikation Relevanter Funktionen

Die relevante Funktion befindet sich in folgender Datei:

- **ssl/t1_lib.c: `tls1_process_heartbeat`** (Z. 2583 – 2654): Bearbeitung eines Heartbeat-Request und Antwort mit einer Heartbeat-Response.

Analyse

Die Schwachstelle wurde mit einer Längenüberprüfung behoben (im Source-Code Zeile 2601):

```
int tls1_process_heartbeat(SSL *s)
	{
	unsigned char *p = &s->s3->rrec.data[0], *pl;
	unsigned short hbtype;
	unsigned int payload;
	unsigned int padding = 16; /* Use minimum padding */

	if (s->msg_callback)
		s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
			&s->s3->rrec.data[0], s->s3->rrec.length,
			s, s->msg_callback_arg);

	/* Read type and payload length first */
	if (1 + 2 + 16 > s->s3->rrec.length)
		return 0; /* silently discard */
	hbtype = *p++;
	n2s(p, payload);
	if (1 + 2 + payload + 16 > s->s3->rrec.length)
```

```
            return 0; /* silently discard per RFC 6520 sec. 4 */
        ...
```

Dabei wurden folgende Variablen benutzt:

- payload: Die Payload-Länge aus dem Heartbeat-Request.
- s->s3->rrec.length: Die Länge des gesamten entschlüsselten TLS-Records (also die gesamte Heartbeat-Nachricht)
- 1 + 2 + 16: Längen der Pflichtfelder: HeartbeatMessageType, Payload-Länge und minimales Padding.

Bevor der Server antwortet, wird also überprüft, ob die Payload-Länge und Längen der Pflichtfelder nicht die Länge der gesamten Nachricht überschreiten.

Mit dem TLS-Attacker-Framework haben wir den Heartbleed-Angriff nachgebaut und die Korrektheit der Längen-Überprüfung verifiziert. Darüber hinaus wurden folgende Angriffe getestet:

- Da die Record-Länge bei der Überprüfung eine wichtige Rolle spielt, könnte es zu Problemen kommen, wenn ein TLS-Record zwei Heartbeat-Requests beinhalten würde (dies ist z.B. üblich bei den Handshake-Nachrichten). Dies ist aber nicht möglich. Der Server bearbeitet in diesem Fall nur den ersten Heartbeat-Request. Den zweiten Heartbeat-Request betrachtet der Server als ein Padding und deswegen wird dieser ignoriert.
- Payload-Länge-Overflow: Da die Payload-Länge lediglich aus zwei Bytes besteht (maximum 0xffff=65535) und da die Payload-Länge auf dem Server als unsigned integer gespeichert wird, ist ein Overflow nicht möglich.

Die Testvektoren sind in der virtuellen Maschine vorhanden.

Es ist uns mit den erzeugten Vektoren nicht gelungen, gegen die getestete OpenSSL-Server-Version den Angriff erfolgreich durchzuführen. Den Patch gegen diesen Angriff beurteilen wir hiermit als korrekt.

Identifikation der Compiler-Flags

In dem OpenSSL Code wurden in den relevanten Funktionen keine Compiler Flags gefunden, welche eine Auswirkung auf die Gegenmaßnahmen gegen diese Angriffe hätten.

6.1.9.1 Fazit

In diesem Kapitel wurden die im AP 3.1 vorgestellten Angriffe getestet. Unsere Tests haben gezeigt, dass der OpenSSL-Server in der Version 1.0.1g gegen die vorgestellten Angriffe nicht anfällig ist (im Falle des Early-CCS-Angriffs wurde die OpenSSL-Nachfolgerversion 1.0.1h überprüft).

Während unserer Untersuchung wurde allerdings festgestellt, dass die Perfect-Forward-Secrecy

nicht standardmäßig aktiviert ist. Der Server generiert einen ephemeral Schlüssel (für DHE und ECDHE Ciphersuites) während des Server-Starts. Dieser Schlüssel wird nachfolgend für alle TLS-Verbindungen eingesetzt, bis der Server nicht neu gestartet wird. Generierung der ephemeral Schlüssel bei jeder TLS-Verbindung muss explizit mit den Flags **`SSL_OP_SINGLE_DH_USE`** und **`SSL_OP_SINGLE_ECDH_USE`** erzwungen werden. Im AP 3.8 werden wir näher drauf eingehen, ob diese Flags korrekt in nginx und Apache httpd eingesetzt werden.

6.2 Low-Level Analyse der Chiffren

In diesem Kapitel überprüfen wir die Implementierungen der für TLSv1.2 relevanten Algorithmen. Es werden Testvektoren untersucht, eventuelle Selbsttests[3] zur Laufzeit, und in verschiedenen Fällen das Timing-Verhalten.

6.2.1 AES[128|256] in Betriebsmodus [CBC|GCM]

Verwendet in: Record Layer Encryption Algorithm

6.2.1.1 Testsuite

OpenSSL bringt Tests für die AES-Chiffre im ECB und CBC Betriebsmodus mit. Diese Tests testen die Verschlüsselung und Entschlüsselung anhand von öffentlich verfügbaren Testvektoren. Für den GCM-Betriebsmodus hat OpenSSL keine Testvektoren. Um die Korrektheit dieser Implementierung nachzuweisen, wurde eine neue Testsuite mit Testvektoren aus der GCM-Spezifikation [GCM] erstellt.

Chiffre	Testprogramm	Quelle der Testvektoren	Verifikation erfolgreich
AES128-ECB Enc	test/evp-test evptest.txt	1 aus FIPS197, C.1 4 aus SP800-38A, F.1.1	Ja
AES128-ECB Dec	test/evp-test evptest.txt	4 aus SP800-38A, F.1.1	Ja
AES256-ECB Enc	test/evp-test evptest.txt	1 aus FIPS197, C.3 4 aus SP800-38A, F.1.5	Ja
AES256-ECB Dec	test/evp-test evptest.txt	4 aus SP800-38A, F.1.5	Ja
AES128-CBC Enc	test/evp-test evptest.txt	4 aus SP800-38A, F.2.1	Ja
AES128-CBC Dec	test/evp-test evptest.txt	4 aus SP800-38A, F.2.2	Ja
AES256-CBC Enc	test/evp-test evptest.txt	4 aus SP800-38A, F.2.5	Ja
AES256-CBC Dec	test/evp-test evptest.txt	4 aus SP800-38A, F.2.6	Ja
AES128-GCM Enc	aesgcm[4]	6 aus [GCM]	Ja
AES128-GCM Dec	aesgcm[4]	6 aus [GCM]	Ja
AES256-GCM Enc	aesgcm[4]	6 aus [GCM]	Ja
AES256-GCM Dec	aesgcm[4]	6 aus [GCM]	Ja

Aufgrund der erfolgreichen Ausführung der Testsuite gehen wir davon aus, dass die genannten Funktionen spezifikationskonform implementiert sind.

3 Ein Selbsttest überprüft die Funktionalität des Programms, indem es überprüft, ob bekannte Ergebnisse vom Programm korrekt berechnet werden.

4 Das Testprogramm wurde im Rahmen dieses Projekts entwickelt.

6.2.1.2 Selbsttests

OpenSSL führt keine Selbsttests der AES-Funktionen und deren Betriebsmodi durch. Untersucht wurde die Instantiierung einer Chiffre-Instanz, die Initialisierung der SSL-Bibliothek (SSL_library_init()) und die Initialisierung der Kernbibliothek (OPENSSL_init()).

6.2.2 SHA-1

Verwendet in: HMAC.

6.2.2.1 Testsuite

Die SHA1 Hashfunktion wird wie folgt getestet:

Hashfunktion	Testprogramm	Quelle der Testvektoren	Verifikation erfolgreich
SHA-1	test/sha1test	3 aus FIPS 180-2 A.1, A.2, A.3	Ja

Aufgrund der erfolgreichen Ausführung der Testsuite gehen wir davon aus, dass SHA-1 spezifikationskonform implementiert ist.

6.2.2.2 Selbsttests

OpenSSL führt keine Selbsttests der SHA-1-Funktion durch. Untersucht wurde die Instantiierung einer Hash-Instanz, die Initialisierung der SSL-Bibliothek (SSL_library_init()) und die Initialisierung der Kernbibliothek (OPENSSL_init()).

6.2.2.3 „Malicious SHA-1“

Im August 2014 wurde ein Angriff namens „Malicious SHA-1“ veröffentlicht [MALSHA1]. Bei diesem Angriff wurden vier in SHA-1 genutzte Rundenkonstanten K_t so verändert, dass es einfach möglich ist, Kollisionen zu erzeugen. Hieraus ergeben sich zwei Angriffsmöglichkeiten auf die Hashfunktion:

- Direkter Angriff auf SHA-1, so wie er standardisiert und implementiert ist
- Die Frage, ob die Konstanten in SHA-1 schon während der Standardisierung böswillig gewählt wurden

Ein Angriff auf den existierenden SHA-1 Algorithmus ist nicht möglich, da Malicious SHA-1 die Veränderung der Rundenkonstanten voraussetzt. Damit wäre der implementierte Algorithmus nicht mehr kompatibel zum Standard und eine solche Implementierung könnte einfach über Testvektoren entlarvt werden.

Die Frage, ob die Konstanten in SHA-1 bereits böswillig gewählt wurden, lässt sich nicht mit Sicherheit beantworten. Allerdings gibt es verschiedene Punkte, die dafür sprechen, dass dies nicht

der Fall ist (s. [MALSHA1] FAQ):

- Die Konstanten wurden als nothing-up-my-sleeve Zahlen gewählt: Sie sind die Quadratwurzeln von 2, 3, 5 und 10, multipliziert mit 2^{30} .
- Die Implementierung von SHA-1 setzt Techniken der Kryptoanalyse voraus, die (öffentlich) erst seit 2004 entwickelt wurden.
- Kurz vor SHA-1 wurde von der NSA SHA-0 entwickelt. In SHA-0 wurden schnell Schwächen gefunden, so dass die Vermutung nahe liegt, dass die NSA (zumindest damals) keinen großen Vorsprung in der Kryptanalyse von SHA-1-ähnlichen Hashfunktionen hat.

Da böswillig gewählte Konstanten aber nicht generell schwächer sind als die in SHA-1 gewählten, kann nicht mit Sicherheit ausgeschlossen werden, dass die Konstanten des SHA-1 Standards eine Hintertür ermöglichen.

6.2.3 SHA-2

Verwendet in: HMAC

6.2.3.1 Testsuite

Hashfunktion	Testprogramm	Quelle der Testvektoren	Verifikation erfolgreich
SHA-256	test/sha256t	3 aus FIPS 180-2 B.1, B.2, B.3	Ja
SHA-384	test/sha512t	3 aus FIPS 180-2 D.1, D.2, D.3	Ja
SHA-512	test/sha512t	3 aus FIPS 180-2 C.1, C.2, C.3	Ja

Aufgrund der erfolgreichen Ausführung der Testsuite gehen wir davon aus, dass die genannten Funktionen spezifikationskonform implementiert sind.

6.2.3.2 Selbsttests

OpenSSL führt keine Selbsttests der SHA-256, SHA-384 und SHA-512-Funktion durch. Untersucht wurde die Instantiierung einer Hash-Instanz, die Initialisierung der SSL-Bibliothek (SSL_library_init()) und die Initialisierung der Kernbibliothek (OPENSSL_init()).

6.2.4 HMAC

Verwendet in

- TLS Record Layer als Message Authentication Code (MAC) (HMAC mit SHA-1, SHA-256, SHA-384)

- PRF (HMAC-SHA256)

6.2.4.1 Testsuite

Funktion	Testprogramm	Quelle der Testvektoren	Verifikation erfolgreich
HMAC-MD5	test/hmactest	3 aus RFC2104 Appendix 1 OpenSSL-spezifisch	Ja
HMAC-SHA1	hmactest	3 aus IETF IPSec Mailing List[5]	Ja
HMAC-SHA256	hmactest	7 aus RFC4231	Ja
HMAC-SHA384	hmactest	7 aus RFC4231	Ja

Aufgrund der erfolgreichen Ausführung der Testsuite gehen wir davon aus, dass die genannten Funktionen spezifikationskonform implementiert sind.

6.2.4.2 Selbsttests

OpenSSL führt keine Selbsttests der HMAC-Funktionen durch. Untersucht wurde die Instantiierung einer HMAC-Instanz, die Initialisierung der SSL-Bibliothek (SSL_library_init()) und die Initialisierung der Kernbibliothek (OPENSSL_init()).

6.2.5 Datenabhängige Ausführungszeit von arithmetischen Operationen

Die Ausführungszeit von Arithmetikoperationen können abhängig von den verarbeiteten Daten sein. Erste praktische Angriffe sind in [PCK1996] aufgeführt, und Timing-Angriffe sind weiterhin ein häufig genutzter Angriffsvektor. Daher untersuchen wir, vorbereitend für die Untersuchung der asymmetrischen Primitiven, das BigNum-Modul (Unterverzeichnis crypto/bn/) auf relevante Schwächen.

Für die Messung wurde ein HP 8460p Notebook mit „Intel Core i7-2620M, 2,7 GHz“-CPU verwendet. Die Tests wurden nativ durchgeführt.

6.2.5.1 Speicherung von BigNum-Zahlen in OpenSSL

OpenSSL speichert BigNum-Zahlen in der folgenden Struktur:

5 http://www.vpnc.org/ietf-ipsec/97.ipsec/msg00096.html

```
struct bignum_st
	{
	BN_ULONG *d; /* Pointer to an array of 'BN_BITS2' bit chunks. */
	int top;     /* Index of last used d +1. */
	/* The next are internal book keeping for bn_expand. */
	int dma^;    /* Size of the d array. */
	int neg;     /* one if the number is negative */
	int flags;
	};
```

6.2.5.2 BN_FLG_CONSTTIME

Die BigNum-Zahlenstruktur unterstützt verschiedene Flags. Von besonderer Bedeutung ist das Flag BN_FLG_CONSTTIME. Durch dieses Flag wird signalisiert, dass Operationen mit dieser Zahl nur mit Algorithmen durchgeführt werden sollen, die eine datenunabhängige, konstante Laufzeit aufweisen. Es kann mit dem Makro „BN_set_flags()“ (cryto/bn/bn.h) gesetzt werden.

Die folgende Tabelle beschreibt, in welchen Situationen das Flag für konstante Ausführungszeit gesetzt wird.

Funktion	Wert	Wann?	Bemerkung
crypto/bn/bn_blind.c: BN_BLINDING_new()	Modulus	falls verarbeitete Zahlen CONSTTIME gesetzt haben	Blinding
crypto/bn/bn_gcd.c: BN_mod_inverse_nobranch()	A und N	falls verarbeitete Zahlen CONSTTIME gesetzt haben	Invertierung von A mod N
crypto/dh/dh_key.c: generate_key()	privater Schlüssel	Sofern DH_FLAG_NO_EXP_CONSTTIME nicht gesetzt	DH Schlüsselerzeugung: Privater und öffentlicher Schlüssel
crypto/dh/dh_key.c: compute_key()	privater Schlüssel	Sofern DH_FLAG_NO_EXP_CONSTTIME nicht gesetzt	DH Schlüsselberechnung mit öffentlichem und privatem Schlüssel
crypto/dsa/dsa_key.c: dsa_builtin_keygen()	privater Schlüssel	Sofern DSA_FLAG_NO_EXP_CONSTTIME nicht gesetzt	DSA Schlüsselerzeugung: Privater und öffentlicher Schlüssel
crypto/dsa/dsa_ossl.c: dsa_sign_setup()	Paramter k	Sofern DSA_FLAG_NO_EXP_CONSTTIME nicht gesetzt	DSA Signatur-Setup

Funktion	Wert	Wann?	Bemerkung
crypto/rsa/rsa_crpt.c: RSA_setup_blinding()	Modulus n	Sofern RSA_FLAG_NO_CONSTTIME nicht gesetzt	RSA Blinding
crypto/rsa/rsa_eay.c: RSA_eay_private_encrypt()	Privater Schlüssel d	Sofern RSA_FLAG_NO_CONSTTIME nicht gesetzt	RSA Signatur
crypto/rsa/rsa_eay.c: RSA_eay_private_decrypt	Privater Schlüssel d	Sofern RSA_FLAG_NO_CONSTTIME nicht gesetzt	RSA Entschlüsselung
crypto/rsa/rsa_eay.c: RSA_eay_mod_exp()	d, p, q, I, dmq1, dmp1	Sofern RSA_FLAG_NO_CONSTTIME nicht gesetzt	RSA Potenzierung, u.U. mit CRT (s. Abs. 6.2.8.4)
crypto/rsa/rsa_gen.c: rsa_builtin_keygen()	pr, d, p	Sofern RSA_FLAG_NO_CONSTTIME nicht gesetzt	RSA Schlüsselerzeugung mit: pr = (p-1)(q-1), mit Erzeugung der Werte für CRT (s. Abs. 6.2.8.4)

Somit wird klar, dass OpenSSL für die sicherheitskritischen Daten bei Diffie-Hellman Schlüsselaustausch, DSA-Signatur, RSA-Verschlüsselung und -Signatur standardmäßig die zeitkonstante Implementierung der Algorithmen auswählt.

6.2.5.3 BN_is_zero()

Die Ausführungszeit ist nicht datenabhängig, da nur überprüft wird, ob das Element „top“ von der „bignum_st“-Sturktur ungleich „0“ ist. Das Feld wird bei der Durchführung von Rechenoperationen, wo der Wert ohnehin vorliegt, mitgepflegt.

6.2.5.4 BN_is_negative()

Die Ausführungszeit ist nicht datenabhängig, da nur überprüft wird, ob das Element „neg“ von der „bignum_st“-Struktur ungleich „0“ ist.

6.2.5.5 BN_add()

Die Addition ist abhängig von der tatsächlichen Länge der verarbeiteten Zahlen. So werden, sofern kein Übertrag entsteht, nur die genutzten Worte einer Zahl addiert, alle weiteren Worte werden von der größeren Zahl kopiert. Die Addition zweier großer Zahlen benötigt etwa 1,5-mal mal so lange wie die Addition einer kleinen zu einer großen Zahl.

Werte	Operation	Zeit
$a=1 \quad , \quad b=2^{2048}-1$	$r=a+b$	$45\,s \cdot 10^{-9}$
$a=b=2^{2048}-1$	$r=a+b$	$72\,s \cdot 10^{-9}$

In der Addition wird das BN_FLG_CONSTTIME Flag einer BigNum-Zahl nicht berücksichtigt.

6.2.5.6 BN_sub()

Die Subtraktion ist in der gleichen Art von der Länge der verarbeiteten Zahlen abhängig wie die Addition. Allerdings ist hier der Einfluss noch deutlich größer: So benötigt die Subtraktion zweier großer Zahlen im getesteten Fall 2,5-mal so lange wie die Subtraktion einer kleinen Zahl.

Werte	Operation	Zeit
$a=1, b=2^{2048}-1$	$r=b-a$	$26\,s \cdot 10^{-9}$
$a=2^{2048}-1, b=2^{2048}-1$	$r=b-a$	$73\,s \cdot 10^{-9}$

In der Subtraktion wird das BN_FLG_CONSTTIME Flag einer BigNum-Zahl ebenfalls nicht berücksichtigt.

6.2.5.7 BN_mul()

Die Zeit für Multiplikation ist ebenfalls von der Länge der verarbeiteten Zahlen abhängig.

Für die Multiplikation nutzt OpenSSL verschiedene Algorithmen, abhängig von der Größe der Faktoren und deren Verhältnisse zueinander. Für ähnlich große Zahlen wird die Karatsuba-Comba-Methode (s. [KKO63], [COMBA]), genutzt, und ansonsten eine normale Multiplikation (Schulbuchmethode). Wir haben die Ausführungszeiten von Multiplikationen von Zahlen verschiedener Bitlängen gemessen. Die Schritte wurden zu 64 Bit gewählt, da dies die Wortgröße des verwendeten Computers (x86-64 Architektur) und damit auch die Größe der von der BigNum-Bibliothek verarbeiteten Einheiten ist. Gemessen wurde die Ausführungszeit für 1000000 Multiplikationen. Die Beispiele 1 und 5 werden mit normaler Multiplikation durchgeführt, die Beispiele 2-4 mit der Karatsuba-Comba-Methode.

Beispiel	Länge von a, b	Operation	Zeit
1	len(a) = 1920, len(b) = 2048	$r=a \cdot b$	1,7s
2	len(a) = 1984, len(b) = 2048	$r=a \cdot b$	1,8s
3	len(a) = 2048, len(b) = 2048	$r=a \cdot b$	0,9s
4	len(a) = 2112, len(b) = 2048	$r=a \cdot b$	3,0s
5	len(a) = 2176, len(b) = 2048	$r=a \cdot b$	1,9s

Es wird deutlich, dass die Ausführungszeit der Karatsuba-Comba-Methode bei gleicher Größe der Faktoren (Beispiel 3) deutlich geringer als bei der normalen Multiplikation, bei Beispiel 4 allerdings deutlich länger. Hierüber lassen sich Schlüsse auf die Länge der multiplizierten Zahlen ziehen. Ob und wie sich diese Information ausnutzen lässt konnte im Rahmen des Projektes nicht festgestellt werden.

Bei der Multiplikation wird das BN_FLG_CONSTTIME Flag einer BigNum-Zahl ebenfalls nicht berücksichtigt.

6.2.5.8 BN_sqr()

Die Zeit für Quadrierung ist ebenfalls von der Länge der verarbeiteten Zahlen abhängig. Der Algorithmus ist analog zu BN_mul() implementiert.

Beispiel	Länge von a, b	Operation	Zeit
1	len(a) = 1920	$r = a^2$	1,20s
2	len(a) = 1984	$r = a^2$	1,26s
3	len(a) = 2048	$r = a^2$	0,91s
4	len(a) = 2112	$r = a^2$	1,40s
5	len(a) = 2176	$r = a^2$	1,45s

Hier fällt lediglich auf, dass die Zeit für die Quadrierung einer 2048 Bit Zahl aus deutlich niedriger ist als bei den anderen Bitlängen.

6.2.5.9 BN_mod()

BN_mod() ruft für die Implementierung der modularen Reduktion BN_div() auf.

6.2.5.10 BN_div()

BN_div() berechnet das Ganzzahlergebnis von d = a / b sowie den Rest der Division. Wenn die zu verarbeitenden Zahlen das BN_FLG_CONSTTIME Flag gesetzt haben, erfolgt die Operation in konstanter Zeit (nur abhängig von der Länge der zu verarbeitenden Zahlen).

6.2.5.11 BN_mod_mul()

Die Zeit für modulare Multiplikation ist ebenfalls von der Länge der verarbeiteten Zahlen abhängig.

Für die modulare Multiplikation nutzt OpenSSL intern BN_mul()/BN_sqr() und BN_mod(). Daher ist es nicht verwunderlich, dass die Ausführungszeit ähnlich der von BN_mul() schwankt. Es werden wieder 1000000 Multiplikation durchgeführt, diesmal allerdings modulo einer zufällig erzeug-

ten Primzahl m der Länge 2048 Bits.

Beispiel	Länge von a, b	Operation	Zeit
1	len(a) = 1920, len(b) = 2048	$r = a \cdot b \bmod m$	6,42s
2	len(a) = 1984, len(b) = 2048	$r = a \cdot b \bmod m$	6,75s
3	len(a) = 2048, len(b) = 2048	$r = a \cdot b \bmod m$	6,22s
4	len(a) = 2112, len(b) = 2048	$r = a \cdot b \bmod m$	8,33s
5	len(a) = 2176, len(b) = 2048	$r = a \cdot b \bmod m$	7,44s

Bei der modularen Multiplikation wird das BN_FLG_CONSTTIME Flag einer BigNum-Zahl ebenfalls nicht berücksichtigt.

6.2.5.12 BN_exp_mod()

Diese Funktion zur Potenzierung von Zahlen nutzt intern die BN_mod_exp_mont()-Funktion.

6.2.5.13 BN_mod_exp_mont()

Dies ist die normale Variante der Montgomery-Potenzierung (s. [HAC], Algorithm 14.94) mit Sliding Window (s. [HAC], Algorithm 14.85). Wenn der Exponent das BN_FLG_CONSTTIME Flag gesetzt hat, wird die BN_mod_exp_mont_consttime()-Methode genutzt.

Wir haben die Ausführungszeit von BN_mod_exp_mont() für Zahlen gleicher Größenordnung gemessen. Getestet wurde mit Zahlen der Länge 2048 Bit, wobei die Basis b mit dem Exponenten e modulo einer zufällig erzeugte Primzahl p potenziert und die Zeit für 1000 Berechnungen gemessen wurde.

Die zufälligen Zahlen A und B wurden einmalig erzeugt und die Berechnungen mit der gleichen Zahl immer wieder durchgeführt.

Variable	Wert
b	0xFFFF000000000000FFFF000000000000....FFFF000000000000FFFF
p	0xF1CA6AF51A2B1D0A7CA93A3244976C26815F9CDBD577960192933B5C31D4233B14BD08236FF37EDD969B3BAE80DED2576815DD945A7F814BF19FB7BAA13E71F45D927516D4BB7B5F3406EEE99069C43E14D0739C7B9A625B41083FF4320B4B68A8C05A572EB7989EC93646D5847DBC1E3F4E8B78B28A0EC6C0447E53DDA176970F0AAE68495F1DCFCBB09DAD44AC876207A34C71D82C32C549F7699BAF26359D0E26FC72C1BFF5EA97510B30FF8FA314DE26FC08EA85D8BA08A01428277573D172A223F784EC2DCA12F10F93B6042949AD90CCF1369B0B8E4E6F5D6E90116CF64205DCE3182392133F7CF6A958CBEE8BD15B92E42D58B75BD77E9685F014F081

Exponent e	Hamming-Gewicht von e	Operation	Zeit
0x80000...000	1	$b^e \bmod p$	4,8s
0xFFFF...FFF	2048	$b^e \bmod p$	5,6s
0xAAA...AAA	1024	$b^e \bmod p$	5,6s
0x8C6318C...318C	411	$b^e \bmod p$	5,5s
0x84210842...1084	206	$b^e \bmod p$	5,3s
0x800000008000...80000000	32	$b^e \bmod p$	4,9s
zufällige Zahl A	1008	$b^e \bmod p$	5,5s
zufällige Zahl B	1009	$b^e \bmod p$	5,6s

Es ist offensichtlich, dass die Operationen eine Abhängigkeit von den verarbeiteten Daten haben. Da OpenSSL für sicherheitskritische Daten aber die Constant-Time-Variante der Potenzierung nutzt, stellt dies kein Problem dar (s. Abs. 6.2.5.2, 6.2.5.14).

6.2.5.14 BN_mod_exp_mont_consttime()

Für modulare Potenzierung in konstanter Zeit nutzt OpenSSL den Montgomery-Algorithmus mit Fenster[6]. Die Fenstergröße beträgt 5 Bits für Zahlen mit mehr als 306 Bits. Die Montgomery-Konstante ist $2^{(\lceil \lceil \log_2(mod) \rceil / 64 \rceil) \cdot 64}$ (d.h. sie hat die doppelte Länge der Länge des Modulus, mit einer Granularität von 64-Bit-Blöcken. So ergibt sich z.B. bei einem 2048-Bit Moduls die Montgomery-Konstante 2^{4096} . Bei einem 2000-Bit Modulus wird aufgrund der Granularität von 64-Bit-Blöcken die gleiche Montgomery-Konstante genutzt). Um die zeitliche Messbarkeit eines möglichen zusätzlicher Reduktionsschritt bei der Rückwandlung von der Montgomery-Darstellung in das Ergebnis zu verhindern, wurde diese Schritt ohne Verzweigungen implementiert (s. crypto/bn/bn_mont.c:240-267).

Wenn die gleichen Messungen wie für BN_mod_exp_mont() mit Zahlen durchgeführt werden, die das CONSTTIME Flag gesetzt haben, ergeben sich folgende Zeiten:

6 Es handelt sich hier um die normale Fenster-Methode, auch bekannt als „Left-to-right k-ary exponentiation“, s. [HAC], Alg. 14.82.

Exponent e	Operation	Zeit
0x80000...000	$b^e \bmod p$	5,9s
0xFFFF...FFF	$b^e \bmod p$	5,9s
0xAAA...AAA	$b^e \bmod p$	5,9s
0x8C6318C...318C	$b^e \bmod p$	5,9s
0x84210842...1084	$b^e \bmod p$	5,9s
0x800000008000...80000000	$b^e \bmod p$	5,9s
zufällige Zahl A	$b^e \bmod p$	5,9s
zufällige Zahl B	$b^e \bmod p$	5,9s

Die Messungen lassen darauf schließen, dass die Potenzierung in konstanter Zeit ausgeführt wird, sofern die verarbeiteten Zahlen das entsprechende Flag gesetzt haben.

6.2.6 Diffie Hellman in Z_p^*

Verwendet in: Key Exchange Algorithm

6.2.6.1 Testsuite

OpenSSL hat in „test/dhtest“ einen Test für den Diffie Hellman Schlüsselaustausch. Es werden die Funktionen aus Kapitel 3.5, in dem auch die Parameter- und Schlüsselerzeugung genau beschrieben wird, getestet. Dieser Test führt die folgenden Schritte durch:

1. Festlegen des Erzeugers g = 5
2. Erzeugen einer zufälligen Primzahl p für eine zyklischen Gruppe F_p , für die g ein Erzeuger ist.
3. Erzeugen eines zufälligen privaten Schlüssels a_{priv} für Partei A, daraus Berechnung des öffentlichen Schlüssel $a_{pub} = g^{a_{priv}} \bmod p$.
4. Erzeugen eines privaten Schlüssels b_{priv} für Partei B, daraus Berechnung des öffentlichen Schlüssel b_{pub}
5. Berechnen des gemeinsamen Schlüssel $k_{ab} = (b_{pub})^{a_{priv}} \bmod p$ aus dem privaten Schlüssel von A und dem öffentlichen Schlüssel von B.
6. Berechnen des gemeinsamen Schlüssel $k_{ba} = (a_{pub})^{b_{priv}} \bmod p$ aus dem privaten Schlüssel von B und dem öffentlichen Schlüssel von A.

7. Vergleich von k_{ab} und k_{ba} . Bei Identität ist der Test erfolgreich, ansonsten fehlgeschlagen.

Die Schlüssellänge beträgt bei dem OpenSSL-Test 64 bit. Der Test wird einmal ausgeführt. Für einen Kommunikationspartner werden die Constant-Time Algorithmen genutzt, für den anderen Partner die normalen Algorithmen.

Um eine größere Sicherheit für die korrekte Funktion des DH-Schlüsseleinigungsverfahrens zu erlangen, wurde der Test um die folgenden Komponenten erweitert:

- Nutzung unterschiedlicher Erzeuger (2, 3, 5)
- Erzeugung von Schlüsseln mit unterschiedlichen Schlüssellängen (64, 128, 256, 512 und 1024 Bits)
- Wiederholte Erzeugung von Schlüsseln (64 Bits: 256000, 128 Bits: 64000, 256 Bits: 16000, 512 Bits: 4000, 1024 Bits: 1000 Durchläufe)

Diese Tests sind erfolgreich durchgelaufen. Daher gehen wir davon aus, dass die Implementierung von Diffie-Hellman in OpenSSL korrekt ist.

6.2.6.2 Selbsttests

OpenSSL führt keine Selbsttests der DH-Funktionen durch. Untersucht wurde die Instantiierung einer DH-Instanz, die Initialisierung der SSL-Bibliothek (SSL_library_init()) und die Initialisierung der Kernbibliothek (OPENSSL_init()).

6.2.6.3 Berechnung des öffentlichen und gemeinsamen Schlüssels

Die OpenSSL-Default-Implementierung markiert, sofern nicht explizit das Flag DH_FLAG_NO_EXP_CONSTTIME gesetzt ist, bei der Schlüsselerzeugung den privaten Schlüssel mit dem BN_FLG_CONSTTIME Flag. Dies hat Auswirkung auf

- die Berechnung des öffentlichen Schlüssels $a_{pub} = g^{a_{priv}} \bmod p$,
- die Berechnung des gemeinsamen Schlüssel $k_{ab} = (b_{pub})^{a_{priv}} \bmod p$.

Für die Berechnung wird die Potenzierung gemäß Abs. 6.2.5.14 durchgeführt. Daher ist Resistenz gegen Timing-Angriffe gegeben.

6.2.7 DSA

Verwendet in: Authentication Algorithm (Signature), Zertifikatsprüfung (Signature)

6.2.7.1 Testsuite

OpenSSL stellt in „test/dsatest“ einen Test für die DSA-Signatur bereit. Hier wird zunächst die Pa-

rametererzeugung (P, Q und G) anhand der in [FIPS186-1], Appendix 5 vorgegebenen Testvektoren durchgeführt und auf Korrektheit überprüft. Dann wird eine OpenSSL-spezifische Nachricht signiert und diese Signatur verifiziert. Hierbei wird direkt ein String („12345678901234567890“) ohne die Anwendung einer Hashfunktion signiert. Es wird sowohl die normale Implementierung als auch die Constant-Time Implementierung von DSA getestet. Es ist nicht trivial, die Signatur und Verifikation des Beispiels in FIPS 186-1 in OpenSSL durchzuführen, da hierfür eine bestimmte Zufallszahl k benötigt wird, die aber im OpenSSL-DSA-Interface nicht spezifiziert werden kann.

Bemerkung 8: Damit DSA-Signaturen sicher sind, muss für jede Signatur ein frisches k erzeugt werden. Daher erfolgt die Erzeugung dieses Parameters innerhalb der OpenSSL-Bibliothek und ist für den Programmierer nicht explizit setzbar und sichtbar. Die Parametererzeugung wird mit Hilfe des OpenSSL-RNGs (s. [AP2]) durchgeführt. Durch diese Vorgehensweise werden sicherheitsrelevante Programmierfehler in einer OpenSSL-nutzenden Anwendung verhindert.

Empfehlung 8: Diese implizite Erzeugung des Parameters k ist zur Verhinderung von Programmierfehlern sinnvoll und daher zu begrüßen. Dennoch wäre eine Möglichkeit, die DSA-Implementierung reproduzierbar gegen Testvektoren zu verifizieren, wünschenswert.

Wir haben mit vertretbarem Aufwand versucht, den OpenSSL-Quelltext so zu ändern, dass die Verifikation des Testvektors aus FIPS 186-1 ermöglicht wird. Dies ist uns in der zur Verfügung stehenden Zeit jedoch nicht gelungen.

Der Test wurde von uns so erweitert, dass viele DSA Signaturen (und nicht nur eine) mit dem gleichen Schlüssel erzeugt werden. Diese Tests sind erfolgreich.

Daher gehen wir davon aus, dass die Implementierung von DSA in OpenSSL korrekt ist.

6.2.7.2 Selbsttests

OpenSSL führt keine Selbsttests der DSA-Funktionen durch. Untersucht wurde die Instantiierung einer DSA-Instanz, die Initialisierung der SSL-Bibliothek (SSL_library_init()) und die Initialisierung der Kernbibliothek (OPENSSL_init()).

6.2.7.3 Implementierung der Signatur

Die Signatur mit DSA ist in zwei Teile aufgeteilt. Zum einen werden in crypto/dsa/dsa_ossl.c:dsa_sign_setup() der Parameter k erzeugt und die von der Nachricht unabhängigen Parameter für die Signatur berechnet, zum anderen wird in crypto/dsa/dsa_ossl.c:dsa_do_sign() die Signatur (r, s) erzeugt. Die Parameter der Signatur sind die DSA-Systemparamter p, q und g, der private Schlüssel x und der zugehörige öffentliche Schlüssel y. Signiert wird der Digest m einer Nachricht. Die Hashfunktion zur Erzeugung des Digests kann vom Programmierer frei gewählt werden.

dsa_sign_setup()

Hier wird der geheime Paramter k , $0 \leq k < q$, erzeugt (zufällig mittels crypto/bn/bn_rand.c: BN_rand_range(), s. [AP2]). Sofern der Programmierer nicht explizit das Flag DSA_FLAG_NO_EXP_CONSTTIME gesetzt hat, wird dieser geheime Parameter mit dem Flag BN_FLG_CONSTTIME geschützt. Um k bei der Berechnung von $t = (g^k \bmod p)$ zu maskieren wird die äquivalente Berechnung $t = (g^{k_m} \bmod p)$ durchgeführt, wobei $k_m = k + q$ oder $k_m = k + 2q$, wenn $k + q$ nicht die gleiche Länge in Bits hat wie q . Diese Berechnung wird mit BN_mod_exp_mont() (s. Abs. 6.2.5.13) durchgeführt. Der erste Teil der Signatur wird zu $r = t \bmod q$ berechnet (BN_mod()). Abschließend wird $k_{inv} = k^{-1} \bmod q$ berechnet.

dsa_do_sign()

Bei der weiteren Signaturberechnung wird zunächst xr berechnet (BN_mod_mul()) und dann der zu signierende Digest m addiert (mit BN_add()). Die Reduktion modulo q wird erreicht, indem q subtrahiert wird, wenn $(m + xr) > q$. Das Ergebnis ist äquivalent zu $l = m + xr \bmod q$ Abschließend wird $s = k_{inv}\, l \bmod q$ berechnet (BN_mod_mul()). Es wird sichergestellt, dass r und s größer als 0 sind. Die DSA-Signatur ist (r, s). Der Digest der zu signierenden Nachricht wird unabhängig von dessen Größe und der Ordnung des Basispunktes auf der Kurve immer direkt verwendet, ohne explizit gekürzt oder gepaddet zu werden.

6.2.7.4 Implementierung der Verifikation

Bei der Signaturüberprüfung wird sichergestellt, dass die r und s größer 0 sind.

6.2.8 RSA

Verwendet in: Authentication Algorithm (Signature), Key Exchange Algorithm (Encryption, Signature), Zertifikatsprüfung (Signature)

6.2.8.1 Testsuite

OpenSSL stellt in „test/rsa_test“ einen Test für das RSA Verschlüsselungssystem bereit. Getestet wird mit Testvektoren aus „p1ovect1.txt“ [P1OVECT1], die von RSA Security, Inc. für PKCS#1 [PKCS1] veröffentlicht wurden. Es werden die folgenden Schlüssellängen und öffentliche Exponenten getestet:

Schlüssellänge	Öffentlicher Exponent
512	17
400	3
1024	17

Hierbei fällt auf, dass der wohl am weitesten verbreitet Exponent 0x1001 (65537) nicht getestet wird. Ebenso sind die getesteten Schlüssellängen nicht zeitgemäß. Die Funktionen werden mit PKCS#1-1.5 und PKCS#1-OAEP Padding getestet. Es werden eine normale und eine Constant-Time Implementierung getestet, jeweils ausschließlich mit Blinding. Da diese Tests erfolgreich ausgeführt werden, gehen wir davon aus, dass das RSA Verschlüsselungssystem in OpenSSL korrekt implementiert ist.

Für den Test der RSA-Signatur wurde ein neuer Test implementiert. Dieser testet die Schlüsselerzeugung, die Signaturerstellung mit PKCS#1v1.5-Padding und deren Verifikation, sowohl in der Constant-Time Variante als auch normal. Darüber hinaus werden die Operationen mit und ohne Blinding getestet. Als Schlüssellänge werden 512, 1024, 2048 und 4096 Bit gewählt. Da diese Tests erfolgreich ausgeführt werden, gehen wir davon aus, dass das RSA-Signatursystem in OpenSSL korrekt implementiert ist.

6.2.8.2 Selbsttests

OpenSSL führt keine Selbsttests der RSA-Funktionen durch. Untersucht wurde die Instantiierung einer RSA-Instanz, die Initialisierung der SSL-Bibliothek (SSL_library_init()) und die Initialisierung der Kernbibliothek (OPENSSL_init()).

6.2.8.3 Padding-Verfahren

Bei der Verwendung von RSA für die Verschlüsselung und Signaturen muss ein Paddingverfahren explizit spezifiziert werden.

Padding für RSA Verschlüsselung

OpenSSL unterstützt für RSA-Signatur die Verfahren PKCS#1-v1.5 [PKCS1], OAEP [RFC3447], SSLv23 [SSL3] und direkte Verschlüsselung ohne Padding. Das zu verwendende Verfahren muss über einen Parameter beim Aufruf von crypto/rsa/rsa_eay.c:RSA_eay_public_encrypt() spezifiziert werden. Beim OAEP-Padding wird der Seed mit dem Standard-RNG von OpenSSL erzeugt (RAND_bytes(), s. [AP2]).

Padding für RSA Signatur

OpenSSL unterstützt für RSA-Signatur die Verfahren PKCS#1-v1.5, PSS, X9.31, SSLv23 und direkte Signatur ohne Padding. PKCS#1-v1.5 Padding kann über einen Parameter der RSA_private_encrypt() Funktion hinzugefügt werden. PSS Padding wird durch die Funktion RSA_padding_add_PKCS1_PSS() zu Daten hinzugefügt, und die resultierenden Daten werden dann mit crypto/rsa/rsa_eay.c:RSA_eay_private_encrypt() direkt (ohne zusätzliches Padding) signiert. Das Salt für die PSS-Signatur wird mit dem Standard-RNG von OpenSSL erzeugt (RAND_bytes(), s. [AP2]).

6.2.8.4 RSA mit Chinese Remainder Theorem

Die RSA-Implementierung in OpenSSL nutzt das Chinese Remainder Theorem, um Berechnungen mit öffentlichen und privaten Schlüsseln zu beschleunigen.

Aus dem Entschlüsselungsexponentent d und den Faktoren p und q von n werden die dafür notwendigen Werte $dmq1 = d \bmod (q-1)$, $dmp1 = d \bmod (q-1)$ und $iqmp = q^{-1} \bmod p$ erzeugt in:

- rsa_buildin_keygen

Sie werden in crypto/rsa/rsa_eay.c: RSA_eay_mod_exp() genutzt.

- CRT wird immer genutzt, auch wenn Time Constant Ausführung verlangt wird
- Montgomery Multiplikation und Reduktion wird immer genutzt

Der Algorithmus für die Entschlüsselung mit privatem Schlüssel in RSA_eay_mod_exp() ist wie folgt implementiert:

Input: I, d, dmp1, dmq1, iqmp, n, p, q

Output: $r = I^d \bmod n$

1. Potenzierung von I mit d mit Hilfe von CRT
   a) $r1 = I \bmod q$
   b) T $m1 = r1^{dmq1} \bmod q$
   c) $r1 = I \bmod p$
   d) T $r0 = r1^{dmp1} \bmod p$
   e) $r0 = r0 - m1$
   f) T wenn $r0 < 0$: $r0 = r0 + p$
   g) T $r1 = r0 \cdot iqmp$
   h) $r0 = r1 \bmod p$
   i) T wenn $r0 < 0$: r $0 = r0 + p$
   j) T $r1 = r0 \cdot q$
   k) $r0 = r1 + m1$
2. Verifikation der Berechnung
   a) $v = r0^e \bmod n$
   b) $vrfy = vrfy - I \bmod n$

c) wenn $vrfy\,!=0$: $r0=I^d \bmod n$ (Berechnung ohne Verwendung des CRT)

3. Rückgabe von $r=r0$

Von diesen Operationen ist die Ausführungszeit der mit einem T markierten Schritte anfällig für Timing-Angriffe. Die Schritte 1f) und 1i) und sind durch die datenabhängig bedingte Ausführung der Operation nicht zeitkonstant. Darüber hinaus sind die Multiplikation, Addition und Subtraktion nicht zeitkonstant. Die Potenzierung hingegen ist in OpenSSL mit konstanter Ausführungszeit implementiert, sofern die verarbeiteten Zahlen das entsprechende Flag gesetzt haben (s. Abs 6.2.5.2). Wir konnten leider nicht feststellen, ob die Zeitabhängigkeit ausgenutzt werden kann, um das Kryptosystem anzugreifen.

Der Schritt 2c) ist ebenfalls nicht zeitkonstant. Allerdings wird hier nur die Information geleakt, ob die Entschlüsselung mit CRT erfolgreich war, was bei korrekten Eingabewerten stets der Fall ist. Die im Fehlerfall durchgeführte Potenzierung ohne CRT wird mit BN_exp_mod() durchgeführt, wobei durch Setzen des CONSTTIME-Flags die zeitkonstante Variante genutzt wird. In dieser Variante haben wir keine Timing-Leaks gefunden.

Boneh und Brumley haben in [BB2003] gezeigt, dass Remote Timing Attacks gegen OpenSSL möglich sind. Der dort skizzierte Angriff basiert auf zwei Timing-Leaks in OpenSSL: und zwar bei der Multiplikation in BN_mul() sowie der Potenzierung mit BN_exp_mod(). Die dort genutzte Schwäche in BN_mul() existiert weiterhin, die Schwäche in BN_exp_mod() hingegen ist behoben. Da OpenSSL aber standardmäßig Blinding für die Entschlüsselung einsetzt, wird der Angriff vereitelt.

Ein weiterer Timing-Angriff auf die modulare Potenzierung wurde von Colin Percival in [CP2005] vorgestellt. Dieser Angriff wird nach Einschätzung des Autors durch die zeitkonstante Montgomery-Potenzierung vereitelt.

Bemerkung 9: Die RSA-Entschlüsselung hat Timing Leaks in der Multiplikation und in der bedingten Ausführung von Programmpfaden.

Empfehlung 9: Auch wenn wir keinen Angriff aus den Timing-Leaks entwickeln konnten, kann nicht ausgeschlossen werden, dass die fortschreitende Entwicklung erfolgreiche Angriffe ermöglicht. Daher sollten die Addition, Subtraktion und Multiplikation um eine Variante erweitert werden, die konstante Zeit benötigt. Die bedingte Ausführung der Operationen in Schritt 1f) und 1i) sollten ebenfalls durch ein zeitkonstantes Konstrukt ersetzt werden.

Für RSA-Entschlüsselung werden die Funktion BN_is_zero, BN_is_negative, BN_add, BN_sub, BN_mul und BN_mod aus dem BigNum-Modul genutzt.

6.2.9 Eigenschaften der Elliptischen Kurven

In diesem Abschnitt betrachten wir allgemeine Eigenschaften der verwendeten Elliptischen Kurven.

Die gemäß [TR021022] erlaubten elliptischen Kurven, die in OpenSSL implementiert sind, sind secp224r1, secp256r1 und secp384r1. Die empfohlenen Brainpool-Kurven (brainpoolP256r1, brainpoolP384r1, brainpoolP512r1) sind seit OpenSSL 1.0.2 vom 22. Januar 2015 enthalten, in der untersuchten OpenSSL-Version allerdings noch nicht.

Für die verschiedenen unterstützen Elliptischen Kurven werden unterschiedliche Implementierungen genutzt. So gibt es eine generische Implementierung sowie gesonderte Implementierungen für nistp256, nistp224 und nisp521.

Nach Berechnungen auf den Kurven wird nicht mittels des Kofaktors überprüft, ob das Ergebnis in der korrekten Untergruppe liegt.

Die detaillierte Untersuchung dieser Implementierungen wurde in diesem Projekt ausgeklammert. Daher werden insbesondere keine Aussagen auf Basis der Implementierung zu Geschwindigkeit, Beschleunigungsmaßnahmen und Timing-Seitenkanälen gemacht.

6.2.9.1 Algorithmen für die Berechnungen auf EC

Für die betrachteten Kurven werden die Operationen aus der EC_METHOD EC_GFp_mont_method() (crypto/ec/ecp_mont.c) genutzt. Es werden mit der Montgomery-Multiplikation optimierte Algorithmen für die Multiplikation und Quadrierung genutzt. Für die Addition, Verdopplung und Invertierung sowie den Test eines Punktes auf Unendlich, Kurvenzugehörigkeit eines Punktes, Punktvergleich und die Koordinatentransformation werden die Standard-Funktionen aus ecp_smpl.c genutzt.

Für manche Kurven, die aber nicht Teil der Betrachtung sind, existieren andere Implementierungen.

6.2.9.2 Kofaktor

Die von OpenSSL implementierten erlaubten Kurven haben den Kofaktor 1.

6.2.10 ECDH

Verwendet in: Key Exchange Algorithm

6.2.10.1 Testsuite

Die Testsuite (test/ecdhtest) für Elliptic Curve Diffie Hellman Key Exchange testet ausschließlich Named Curves (s. [IANAECC]). Es werden verschiedene NIST-Kurven getestet. Für diese Kurven werden die folgenden Operationen getestet:

1. Erzeugen zweier Schlüsselpaare für die Parteien A und B.
2. Export der Koordinaten des öffentlichen Schlüssels a_{pub} , b_{pub}

3. Berechnen des gemeinsamen Schlüssel k_{ab} aus dem privaten Schlüssel von A und dem öffentlichen Schlüssel von B. Als Schlüsselableitungsfunktion wird SHA-1 genutzt.
4. Berechnen des gemeinsamen Schlüssel k_{ba} aus dem privaten Schlüssel von B und dem öffentlichen Schlüssel von A. Als Schlüsselableitungsfunktion wird SHA-1 genutzt.
5. Vergleich von k_{ab} und k_{ba} . Bei Identität ist der Test erfolgreich, ansonsten fehlgeschlagen.

Es werden keine speziellen Testvektoren getestet. Der OpenSSL Code überprüft die folgenden Kurven:

secp192r1	secp224r1	secp256r1	secp384r1
secp521r1	sect163k1	sect163r2	sect233k1
sect233r1	sect283k1	sect283r1	sect409k1
sect409r1	sect571k1	sect571r1	

Im Rahmen des Projektes haben wir die Tests um die fehlenden Kurven aus RFC4492: „ECC Cipher Suites for TLS“ erweitert:

sect163r1	sect193r1	sect193r2	sect239k1
secp160k1	secp160r1	secp160r2	secp192k1
secp224k1	secp256k1		

Die Tests werden erfolgreich ausgeführt. Daher gehen wir davon aus, dass der Elliptic Curve Diffie Hellman Schlüsselaustausch in OpenSSL korrekt implementiert ist.

6.2.10.2 Selbsttests

OpenSSL führt keine Selbsttests der ECDH-Funktionen durch. Untersucht wurde die Instantiierung einer ECDH-Instanz, die Initialisierung der SSL-Bibliothek (SSL_library_init()) und die Initialisierung der Kernbibliothek (OPENSSL_init()).

6.2.10.3 Implementierung

Der private Schlüssel wird in der Funktion crypto/ecdh/ec_key.c:EC_KEY_generate_key() mit der Funktion BN_rand_range() (s. [AP2]) erzeugt. Der RNG wird weder vor noch nach der Erzeugung des Schlüssels reseedet. Der öffentliche Schlüssel wird durch Multiplikation des privaten Schlüssels mit dem Basispunkt berechnet.

6.2.11 ECDSA

Verwendet in: Authentication Algorithm (Signature), Zertifikatsprüfung (Signature)

6.2.11.1 Testsuite

Die Testsuite (test/ecdsatest) für Elliptic Curve DSA testet ausschließlich Named Curves. Es werden sämtliche in RFC4492 genannten Kurven getestet Kurven getestet. Für diese Kurven werden die folgenden Operationen getestet:

1. Erzeugen zweier Schlüssel K_1 und K_2
2. Erzeugung zweier Message Digests MD_a und MD_b , beide mit der Länge 20 Bytes.
3. Erstellen einer Signatur $S_{a,1}$ eines Message Digest MD_a mit Schlüssel K_1
4. Verifikation der Signatur $S_{a,1}$ mit dem korrekten Schlüssel K_1 gegen MD_a .
5. Negativtest: Verifikation der Signatur $S_{a,1}$ mit K_2 gegen MD_a . Wenn die Verfikation erfolgreich ist, bricht der Test mit einem Fehler ab.
6. Negativtest: Verifikation der Signatur $S_{a,1}$ mit K_1 gegen MD_b . Wenn die Verfikation erfolgreich ist, bricht der Test mit einem Fehler ab.
7. Negativtest: Verifikation der Signatur $S_{a,1}$ mit K_1 gegen MD_a , aber mit falscher Längenangabe beim Funktionsaufruf. Wenn die Verifikation erfolgreich ist, bricht der Test mit einem Fehler ab.
8. Negativtest: Veränderung eines Bytes in $S_{a,1}$ und Verifikation der so veränderten Signatur mit K_1 gegen MD_a . Wenn die Verifikation erfolgreich ist, bricht der Test mit einem Fehler ab.

Diese Testsuite ist äußerst genau. So wird nicht nur die Implementierung des ECDSA-Algorithmus geprüft, sondern auch die korrekte Funktion der zugehörigen Bibliotheksschnittstellen.

Zusätzlich wird ein Testvektor aus [ANSIX9.26] getestet.

	Testprogramm	Quelle der Testvektoren	Verifikation erfolgreich
ECDSA mit secp192r	test/ecdsatest	1 Vektor aus X9.62	Ja

Aufgrund der erfolgreichen Verifikation der Testsuite gehen wir davon aus, dass der ECDSA-Algorithmus korrekt implementiert ist.

6.2.11.2 Selbsttests

OpenSSL führt keine Selbsttests der ECDSA-Funktionen durch. Untersucht wurde die Instantiierung einer ECDSA-Instanz, die Initialisierung der SSL-Bibliothek (SSL_library_init()) und die In-

itialisierung der Kernbibliothek (OPENSSL_init()).

6.2.11.3 Implementierung

Vor der Signaturerstellung wird der RNG mit dem Digest der Nachricht geseedet, wobei die Entropieschätzung gleich der Länge des Digests ist. Dies kann, wie in [AP2] Abs. 3.1.3 bereits angeführt, zu einer zu hohen Schätzung der enthaltenen Entropie führen. Eine Gegenmaßnahme wird an gleicher Stelle diskutiert. Das weitere Vorgehen der Signatur erfolgt analog zu Abs. 6.2.7.3 in zwei Teilen: crypto/ecdsa/ecs_ossl.c:ecdsa_sign_setup() und crypto/ecdsa/ecs_ossl.c:ecdsa_do_sign(). Die Parameter der Signatur sind die Elliptische Kurve E, die Ordnung des Basispunktes auf E $ord(E)$, der geheime Schlüssel d und der Digest m der zu signierenden Nachricht. Die Hashfunktion zur Erzeugung des Digests kann vom Programmierer frei gewählt werden. Der Digest der zu signierenden Nachricht wird unabhängig von dessen Größe und der Ordnung des Basispunktes auf der Kurve immer direkt verwendet, ohne explizit gekürzt oder gepaddet zu werden.

ecdsa_sign_setup()

Hier wird der geheime Parameter k, $0 \le k < ord(E)$, erzeugt (zufällig mittels crypto/bn/bn_rand.c: BN_rand_range() s. [AP2]). Um vor Timing-Leaks zu schützen wird dann $k_m = k + ord(E)$ berechnet, oder $k_m = k + 2 \cdot ord(E)$, falls $k + ord(E)$ eine geringere oder gleiche Bitlänge hat wie $ord(E)$. Hiermit wird dann die zu $k \cdot G$ äquivalente Multiplikation $k_m \cdot G$ berechnet, wobei G den Basispunkt der zugrunde liegenden Kurve E darstellt. Der erste Teil der Signatur wird aus der affinen x-Koordinate x von $k_m \cdot G$ durch Reduktion modulo der Gruppenordnung berechnet $r = x \bmod ord(E)$. Falls $r = 0$, wird ein neuer Parameter k erzeugt und der Prozess wiederholt.

Abschließend wird $k_{inv} = k^{-1} \bmod ord(E)$ berechnet.

ecdsa_do_sign()

Bei der weiteren Signaturberechnung wird zunächst dr berechnet (BN_mod_mul()) und dann der zu signierende Digest m addiert und modulo q reduziert (mit BN_mod_add_quick()). Das Ergebnis entspricht dann $l = m + dr \bmod q$. Abschließend wird $s = k_{inv}\, l \bmod q$ berechnet (BN_mod_mul()). Wenn $s = 0$, dann wird die Signaturerzeugung erneut durchgeführt oder eine Fehlermeldung erzeugt (wenn k_{inv} oder r vom Benutzer vorgegeben wurden). Die DSA-Signatur ist (r, s).

6.2.11.4 Implementierung der Verifikation

Bei der Signaturüberprüfung wird sichergestellt, dass die r und s größer 0 sind.

6.2.11.5 Timing-Angriffe

2011 haben Brumley und Tuveri [BT2011] einen Angriff auf die ECDSA-Implementierung in

OpenSSL vorgestellt. Dieser Angriff basiert auf dem Umstand, dass ein Bias bei der Auswahl des Signaturparameters k zu einem Key Recovery Angriff genutzt werden kann. Die Autoren haben einen Angriff auf OpenSSL 0.9.8o durchgeführt, da sich in dieser Version die Länge von k auf die Ausführungszeit der Signatur auswirkt und dadurch Signaturen für die Analyse ausgewählt werden können, bei denen k weniger als n Bits hat und damit biased ist. Diese Eigenschaft wird ausgenutzt, um einen Lattice-Angriff auf den privaten Schlüssel durchzuführen. Der beschriebene Angriff betrachtet GF2-Kurven, bei denen die Montgomery-Ladder für die Skalarmultiplikation genutzt wird. Auch wenn diese Kurven hier nicht betrachtet werden, ist der Angriff relevant, falls auch die GFp-Implementierung in OpenSSL Informationen über die Länge von k leakt.

Der Angriff wurde am 25.05.2011 in OpenSSL mit einem Patch gefixt, der von den Autoren des Artikels vorgeschlagen wurde. Der Fix ist ab OpenSSL 1.1.0 und 1.0.1e enthalten (OpenSSL 1.0.1g: crypto/ecdsa/ecs_ossl.c:ecdsa_signsetup(), ll. 152-153):

```
if (!BN_add(k, k, order)) goto err;
if (BN_num_bits(k) <= BN_num_bits(order))
      if (!BN_add(k, k, order)) goto err;
```

In unseren Tests, in denen wir die Länge des Parameters k begrenzt haben, konnten wir keinen signifikanten Unterschied in der Ausführungszeit der Signaturerstellung feststellen. Der Patch ist somit wirkungsvoll.

6.2.12 Fazit

In diesem Kapitel wurden die Low-Level Implementierungen der in TLS1.2 verwendeten Chiffren untersucht. Es wurden jeweils untersucht ob Tests der Funktionen existieren, wobei zwischen Offline-Tests und automatischen Selbsttests zur Laufzeit unterschieden wurde. Wir haben festgestellt, dass automatische Selbsttests zur Laufzeit generell nicht implementiert sind. Offline-Tests stehen für die meisten Algorithmen innerhalb von OpenSSL zur Verfügung. Sofern solche Tests nicht existierten, wurden sie im Rahmen dieses Projektes implementiert. Die Offline-Tests haben die korrekte Funktionsweise der Algorithmen bezüglich Konformität zu Standards belegt.

Als weiterer Aspekt wurden die Algorithmen hinsichtlich Timing-Seitenkanalangriffen untersucht. Es ist aufgefallen, dass für die auf endlichen Körpern basierenden Algorithmen zum einen schnelle Implementierungen existieren, die Timing-Seitenkanalangriffe ermöglichen könnten. Zum anderen gibt es aber auch Implementierungen mit konstanter Ausführungszeit, die gegen Timing-Seitenkanalangriffe resistent sind. Für die Verarbeitung mit sicherheitskritischen Daten wird standardmäßig die Variante mit konstanter Ausführungszeit genutzt, sofern diese nicht explizit deaktiviert wurde.

Die Verarbeitung sicherheitskritischer Daten mit einer Implementierung, die Informationen leakt, ist höchst bedenklich. Daher sollte aus Sicherheitsgründen die Flags RSA_FLAG_NO_CONSTTIME, DSA_FLAG_NO_EXP_CONSTTIME und DH_FLAG_NO_EXP_CONSTTIME niemals gesetzt werden, oder sogar dem Nutzer von OpenSSL vorenthalten werden. Darüber hinaus wäre es wünschenswert, wenn generell nur die nicht angreifbare Version in OpenSSL enthalten wäre, um die fälschliche Nutzung der angreifbaren Implementierung auszuschließen. Allerdings muss hierfür zunächst überprüft werden, ob der zusätzliche Rechenaufwand, der auch bei Verarbeitung von nichtsicherheitskritischen Daten entsteht, vertretbar ist.

Die von OpenSSL in den untersuchten Krypto-Algorithmen implementierten Maßnahmen zur Vermeidung von Timing-Leaks sind wirkungsvoll. Wir haben Timing-Leaks in den Basisoperationen BN_add(), BN_sub(), BN_mul(), BN_mod_mul() und BN_sqr() gefundenen, aber keinen Weg, diese auszunutzen. Dennoch sollte über eine Implementierung mit konstanter Ausführungszeit nachgedacht werden.

7 Konfiguration der Ciphersuites und Nachweis der Wirksamkeit

In diesem Kapitel werden Einstellungen des OpenSSL-Stacks betrachtet und ihre Wirksamkeit analysiert. Dabei dient als Grundlage das von OpenSSL zur Verfügung gestellte Kommandozeilenprogramm **s_server**, das es ermöglicht bestimmte TLS-Version und Ciphersuites zu wählen oder bestimmte Zertifikatsüberprüfungen explizit zu setzen. Basierend auf diesen Ergebnissen werden im AP 3.8 weitere Webanwendungen wie Apache httpd (mit dem mod_ssl Modul), nginx und OpenSSH getestet.

Der Grund warum der OpenSSL Test-Server als Grundlage der Untersuchung genommen wurde ist, dass die auf OpenSSL basierenden Frameworks gleich konfiguriert werden wie der OpenSSL Test-Server. Dabei wird der OpenSSL-Context mit den gewählten Flags initialisiert. Der Context wird später für die Steuerung des Protokoll-Verlaufs genutzt.

7.1 Relevante Konfigurations-Optionen

OpenSSL Server bietet mehrere Konfigurations-Optionen an, mit welchen man den OpenSSL Test-Server starten kann. In diesem AP werden folgende Optionen überprüft (gefilterte Ausgabe von **`openssl s_server –help`**):

Konfigurations-Option	Beschreibung
-named_curve arg	Elliptic curve name to use for ephemeral ECDH keys
-cipher arg	Play with 'openssl ciphers' to see what goes here
-serverpref	Use server's cipher preferences
-ssl3	Just talk SSLv3
-tls1_2	Just talk TLSv1.2
-tls1_1	Just talk TLSv1.1
-tls1	Just talk TLSv1
-no_ssl2	Just disable SSLv2
-no_ssl3	Just disable SSLv3
-no_tls1	Just disable TLSv1
-no_tls1_1	Just disable TLSv1.1
-no_tls1_2	Just disable TLSv1.2
-no_dhe	Disable ephemeral DH
-no_ecdhe	Disable ephemeral ECDH

Wenn der Test-Server mit diesen Optionen gestartet wird, werden diese in den SSL-Context gesetzt, z.B.:

`SSL_CTX_set_options(ctx,off);` (Zeile 1546, **`apps/s_server.c`**)

Diese Optionen werden dann in allen weiteren TLS-Verbindungen benutzt.

7.2 Genutzte Umgebung und Tools

Für die Bearbeitung dieses APs wurde OpenSSL mit dem Debug-Flag kompiliert (siehe AP 3.6), um mit Debugging bestimmte Pfade nachvollziehen zu können. Für die Überprüfung der genutzten Einstellungen wurden folgende Tools eingesetzt:

- Wireshark: für die Verfolgung der TLS-Kommunikation
- Netbeans: Source-Code Analyse
- OpenSSL s_client, testssl.sh (https://testssl.sh/), Java TLS Client: Überprüfung der Server-Konfiguration

Alle für diese Analyse relevanten Dateien befinden sich in der virtuellen Maschine in dem Ordner **/home/developer/openssl-config-analysis,** darunter kryptographische Schlüssel und Scripts für den Start des Servers / Clients.

7.3 TLS Version

Um die korrekte Auswahl der TLS-Version zu überprüfen, wurde der TLS-Server mehrmals wie folgt gestartet:

`/home/developer/openssl-1.0.1g/apps/openssl s_server -accept 51624 -key ../keys/privkey1024NoPass.pem -cert ../keys/server-cert1024.pem -[protocol-version]`

Dabei definiert **`[protocol-version]`** die Protokollversion und kann mit folgenden Werten initialisiert werden: **`ssl3, tls1, tls1_1, tls1_2`** (**`ssl2`** ist standardmäßig nach dem Kompillieren ausgeschlossen).

Der TLS Client wurde mit dem folgenden Befehl beim Einsatz von unterschiedlichen Protokoll-Versionen gestartet:

`openssl s_client -connect localhost:51624 -[protocol-version]`

Unsere Tests haben gezeigt, dass nach der Protokoll-Version-Einstellung der Server nur TLS-Handshakes mit korrekter Version erfolgreich durchführt. Eine invalide Version im ClientHello führt zum folgenden Fehler:

`error:1408A10B:SSL routines:SSL3_GET_CLIENT_HELLO:wrong version`

number:s3_srvr.c:960

Wenn mehrere Protokoll-Versionen beim Server-Start explizit definiert werden, wird immer die letzte Option genommen.

Definierung von mehreren Protokoll-Versionen erlauben folgende Argumente: **no_sl3, no_tls1, no_tls1_1, no_tls1_2** (**no_ssl2** ist auch hier standardmäßig nach dem Kompilieren ausgeschlossen).

Unsere Tests haben auch hier gezeigt, dass die Einstellungen korrekt übernommen werden. Wenn z.B. der Server mit **no_ssl3** und **no_tls1** initialisiert wird, werden nur TLS-Handshakes mit TLS1.1 und TLS1.2 Versionen erfolgreich durchgeführt. Befehl:

/home/developer/openssl-1.0.1g/apps/openssl s_server -accept 51625 -key ../keys/privkey1024NoPass.pem -cert ../keys/server-cert1024.pem -no_tls1 -no_ssl3

7.3.1 Identifikation relevanter Funktionen

Mit Hilfe von Debugging wurden folgende Funktionen als relevant identifiziert:

- **ssl/s3_srvr.c** (Z. 954 – 970): Überprüfung der TLS-Version aus der ClientHello-Nachricht (für TLS1.1 und TLS1.2 Version).
- **ssl/s23_srvr.c** (Z. 238 – 631): Überprüfung der TLS-Version aus der ClientHello-Nachricht (für SSL3.0 und TLS1.0 Version).

7.4 Einschränkung der TLS-Ciphersuites

Die Auswahl der TLS Ciphersuites kann auf der Server-Seite (wie auch auf der Client-Seite im s_client) mit dem Argument -**cipher** gesteuert werden. Um die Richtigkeit dieser Option zu testen, wurde ein Skript geschrieben, welches den TLS-Server mit unterschiedlichen Ciphersuites startet. Anschließend wurde das testssl Tool ausgeführt und die Ciphersuites ausgegeben, die zu einem validen Handshake geführt haben. Die Resultate wurden manuell untersucht.

Das Skript für die Ciphersuites-Überprüfung:

```
#!/bin/bash
trap "kill 0" SIGINT SIGTERM EXIT

SERVER="/home/developer/openssl-1.0.1g/apps/openssl s_server -tls1_2 -accept"
PORT=51624
#KEY=" -key ../keys/privkey1024NoPass.pem -cert ../keys/server-cert1024.pem"
KEY=" -key ../keys/ec192-private.pem -cert ../keys/ec192-cert.pem"
TEST_SSL="/home/developer/testssl.sh-master/testssl.sh -e localhost:"
```

```
#array of the available ciphers (see the output of: openssl ciphers)

CIPHER_ARRAY=(ECDHE-RSA-AES256-GCM-SHA384 ECDHE-ECDSA-AES256-GCM-SHA384 ECDHE-RSA-AES256-
SHA384 ECDHE-ECDSA-AES256-SHA384 ECDHE-RSA-AES256-SHA ECDHE-ECDSA-AES256-SHA SRP-DSS-AES-
256-CBC-SHA SRP-RSA-AES-256-CBC-SHA SRP-AES-256-CBC-SHA DHE-DSS-AES256-GCM-SHA384 DHE-
RSA-AES256-GCM-SHA384 DHE-RSA-AES256-SHA256 DHE-DSS-AES256-SHA256 DHE-RSA-AES256-SHA DHE-
DSS-AES256-SHA DHE-RSA-CAMELLIA256-SHA DHE-DSS-CAMELLIA256-SHA ECDH-RSA-AES256-GCM-SHA384
ECDH-ECDSA-AES256-GCM-SHA384 ECDH-RSA-AES256-SHA384 ECDH-ECDSA-AES256-SHA384 ECDH-RSA-
AES256-SHA ECDH-ECDSA-AES256-SHA AES256-GCM-SHA384 AES256-SHA256 AES256-SHA CAMELLIA256-
SHA PSK-AES256-CBC-SHA ECDHE-RSA-DES-CBC3-SHA ECDHE-ECDSA-DES-CBC3-SHA SRP-DSS-3DES-EDE-
CBC-SHA SRP-RSA-3DES-EDE-CBC-SHA SRP-3DES-EDE-CBC-SHA EDH-RSA-DES-CBC3-SHA EDH-DSS-DES-
CBC3-SHA ECDH-RSA-DES-CBC3-SHA ECDH-ECDSA-DES-CBC3-SHA DES-CBC3-SHA PSK-3DES-EDE-CBC-SHA
ECDHE-RSA-AES128-GCM-SHA256 ECDHE-ECDSA-AES128-GCM-SHA256 ECDHE-RSA-AES128-SHA256 ECDHE-
ECDSA-AES128-SHA256 ECDHE-RSA-AES128-SHA ECDHE-ECDSA-AES128-SHA SRP-DSS-AES-128-CBC-SHA
SRP-RSA-AES-128-CBC-SHA SRP-AES-128-CBC-SHA DHE-DSS-AES128-GCM-SHA256 DHE-RSA-AES128-GCM-
SHA256 DHE-RSA-AES128-SHA256 DHE-DSS-AES128-SHA256 DHE-RSA-AES128-SHA DHE-DSS-AES128-SHA
DHE-RSA-SEED-SHA DHE-DSS-SEED-SHA DHE-RSA-CAMELLIA128-SHA DHE-DSS-CAMELLIA128-SHA ECDH-
RSA-AES128-GCM-SHA256 ECDH-ECDSA-AES128-GCM-SHA256 ECDH-RSA-AES128-SHA256 ECDH-ECDSA-
AES128-SHA256 ECDH-RSA-AES128-SHA ECDH-ECDSA-AES128-SHA AES128-GCM-SHA256 AES128-SHA256
AES128-SHA SEED-SHA CAMELLIA128-SHA PSK-AES128-CBC-SHA ECDHE-RSA-RC4-SHA ECDHE-ECDSA-RC4-
SHA ECDH-RSA-RC4-SHA ECDH-ECDSA-RC4-SHA RC4-SHA RC4-MD5 PSK-RC4-SHA EDH-RSA-DES-CBC-SHA
EDH-DSS-DES-CBC-SHA DES-CBC-SHA EXP-EDH-RSA-DES-CBC-SHA EXP-EDH-DSS-DES-CBC-SHA EXP-DES-
CBC-SHA EXP-RC2-CBC-MD5 EXP-RC4-MD5)


for (( i = 0 ; i < ${#CIPHER_ARRAY[@]} ; i++ )) do

    echo Testing option: ${CIPHER_ARRAY[$i]}

    port=$((PORT+i))

    # create a start command for the server on a given port with a given cipher

    server="$SERVER $port $KEY -cipher ${CIPHER_ARRAY[$i]}"

    echo Starting server with $server

    # start the server with a coproc command (needed to handle input/output of the opens-
sl server, otherwise the server crashes)

    coproc $server &> log

    last_pid=$!

    sleep $[1]s

    # command for starting a testssl client on a given port

    client="$TEST_SSL$port"

    # start the client and output only the 3 lines after the Hexcode word appears (these
lines contain available ciphersuites)

    $client | grep -A3 Hexcode

    # kill server

    kill -9 $last_pid

done
```

Nach der Ausführung sieht die Ausgabe dieses Skripts wie folgt aus:

```
Testing option: ECDHE-RSA-AES256-GCM-SHA384
```

```
Starting server with /home/developer/openssl-1.0.1g/apps/openssl s_server -tls1_2 -accept
51624  -key  ../keys/privkey1024NoPass.pem  -cert  ../keys/server-cert1024.pem  -key2
../keys/ec192-private.pem  -cert2  ../keys/ec192-cert.pem  -cipher  ECDHE-RSA-AES256-GCM-
SHA384

Hexcode  Cipher Suite Name (OpenSSL)    KeyExch.   Encryption Bits

-------------------------------------------------------------------------

 xc030   ECDHE-RSA-AES256-GCM-SHA384    ECDH       AESGCM     256


Testing option: ECDHE-RSA-AES256-SHA384

Starting server with /home/developer/openssl-1.0.1g/apps/openssl s_server -tls1_2 -accept
51626  -key  ../keys/privkey1024NoPass.pem  -cert  ../keys/server-cert1024.pem  -key2
../keys/ec192-private.pem -cert2 ../keys/ec192-cert.pem -cipher ECDHE-RSA-AES256-SHA384

Hexcode  Cipher Suite Name (OpenSSL)    KeyExch.   Encryption Bits

-------------------------------------------------------------------------

 xc028   ECDHE-RSA-AES256-SHA384        ECDH       AES        256
```

Da der Server immer mit nur einer Ciphersuite gestartet wurde, konnten wir verifizieren, dass immer nur ein Handshake zum Erfolg geführt hat (Mapping von den TLS-Ciphersuites zu OpenSSL -Ciphersuites kann z.B. hier eingesehen werden: https://testssl.sh/openssl-rfc.mappping.html).

Für die Korrektheit der Resultate mussten wir das Skript mehrmals ausführen, jeweils mit Schlüsseln für unterschiedliche asymmetrische Algorithmen: RSA, EC und DH.[7] Wenn ein TLS-Client nur Ciphersuites vorschlägt, für die keine Schlüssel vorhanden sind, antwortet der Server sofort mit einem TLS HandshakeFailure Alert. Dies passiert unabhängig davon, ob der Server mit dieser Ciphersuite gestartet wurde oder nicht.[8]

Zusätzlich zu den oben genannten Eigenschaften, die von dem testssl.sh automatisiert überprüft wurden, haben wir manuell die Auswahl der Authentifizierungs- und MAC-Algorithmen überprüft. Dies haben wir erzielt indem wir zusätzliche Debug-Ausgaben in den OpenSSL-Source Code eingebaut haben. Die Ausgaben haben bestätigt, dass die Auswahl der Authentifizierungs- und MAC-Algorithmen korrekt ist.

Wenn mehrere Ciphersuites definiert werden sollen, kann dies der Anwender wie folgt tun:

-cipher [cipher1],[cipher2],...

Die Korrektheit dieser Einstellung konnte auch erfolgreich überprüft werden und TLS-Handshakes mit den eingestellten Ciphersuites haben zum Erfolg geführt. Die Präferenz der eingestellten Ciphersuites wird im folgenden Abschnitt betrachtet.

7 Obwohl es der s_server erlaubt, mit den Argumenten **-key2** und **-cert2** einen weiteren Schlüssel zu benutzen, war der Versuch mit diesen Argumenten zu arbeiten nicht erfolgreich (das zweite Schlüsselpaar wurde nicht angenommen).

8 Als Beispiel kann man einen Server betrachten, der mit einem RSA-Schlüssel initialisiert wird. Wenn der TLS Client dann versucht mit einem ECDH-ECDSA Algorithmus den Server zu kontaktieren, wird die Verbindung mit einem TLS Handshake Failure Alert abgeschlossen.

7.4.1 Identifikation relevanter Funktionen

Mit Hilfe von Debugging wurden folgende Funktion als relevant identifiziert:

- **ssl/s3_srvr.c** (Z. 1347): Aufruf der Funktion **`ssl3_choose_cipher`**. Als Parameter werden die vom Client vorgeschlagenen Ciphersuites benutzt.
- **ssl/s3_lib.c** (Z. 3757): Funktion **`ssl3_choose_cipher`**. Sorgt für die Überprüfung der eingestellten Ciphersuites und deren Präferenzen, und um die Auswahl der elliptischen Kurven (falls diese zum Einsatz kommen). Wenn ein falscher Schlüssel auf dem Server hinterlegt wird (falscher asymmetrischer Algorithmus oder falsche elliptische Kurve), wird keine Ciphersuite ausgewählt.

7.5 Präferenzeinstellung bei TLS-Ciphersuites

Der Server kann mit dem Argument -**serverpref** gestartet werden, womit eine Präferenz für bestimmte TLS-Ciphersuites definiert wird.

Für die Testzwecke wurde zuerst der Server mit den folgenden zwei Ciphersuites gestartet:

`-cipher ECDH-ECDSA-AES256-SHA,ECDHE-ECDSA-AES256-SHA384`

Anschließend wurde der Client mit

`-cipher ECDHE-ECDSA-AES256-SHA384,ECDH-ECDSA-AES256-SHA`

und

`-cipher ECDH-ECDSA-AES256-SHA,ECDHE-ECDSA-AES256-SHA384`

gestartet.

In dem ersten TLS-Handshake wurde dabei **`ECDHE-ECDSA-AES256-SHA384`** ausgewählt, bei dem zweiten TLS-Handshake wurde **`ECDH-ECDSA-AES256-SHA`** ausgewählt.

Nach dem Server-Start mit der -serverpref Option

`-cipher ECDH-ECDSA-AES256-SHA,ECDHE-ECDSA-AES256-SHA384 -serverpref`

wurde bei jedem TLS-Handshake immer die **`ECDH-ECDSA-AES256-SHA`** Ciphersuite ausgewählt, unabhängig davon, ob der TLS-Client diese auf der ersten oder späteren Stelle definiert hat. Dies wurde mit den -cipher Parametern erzielt.

7.5.1 Identifikation relevanter Funktionen

Für die Ciphersuite-Auswahl ist (ähnlich wie im vorherigen Kapitel beschrieben) die Funktion **`ssl3_choose_cipher`** zuständig (**ssl/s3_lib.c**).

7.6 Sperren von Ephemeral Ciphersuites

Mit den Parametern **-no_dhe** und **-no_ecdhe** können die Ephemeral Ciphersuites ausgeschaltet werden. Diese Funktionalität konnten wir nachweisen, in dem wir einen Server mit

```
-cipher ECDHE-ECDSA-AES256-SHA384 -no_ecdhe
```

gestartet haben (also nur die **ECDHE-ECDSA-AES256-SHA384** Ciphersuite wurde erlaubt) und anschließend versucht haben mit dem Server eine Verbindung herzustellen. Der TLS-Client war dabei so konfiguriert, dass dieser **ECDHE-ECDSA-AES256-SHA384** akzeptieren sollte. Die Verbindung wurde aber während des Handshakes abgebrochen, der Server hat mit einem TLS-Alert geantwortet.

Eine ähnliche Funktionalität konnten wir beobachten, als wir **-no_dhe** Attribut gesetzt haben.

7.6.1 Identifikation relevanter Funktionen

Für die Ciphersuite-Auswahl ist die Funktion **`ssl3_choose_cipher`** zuständig (**ssl/s3_lib.c**). Diese Funktion gibt einen Fehler zurück, da es zur Zeit ihrer Ausführung keinen Ephemeral ECDH Schlüssel gibt. Dieser wird auf dem Server nicht erzeugt, weil bei dem Server-Start die Funktion **`EC_KEY_new_by_curve_name`** nicht ausgeführt wird. Diese wird normalerweise ausgeführt, direkt in der MAIN-Methode (**apps/s_server.c**, Z. 1722).

7.7 Explizite Definition von Elliptischen Kurven für Ephemeral Schlüsselaustausch

Die Elliptische Kurve für Ephemeral Schlüsselaustausch kann mit dem Argument **-named_curve** explizit definiert werden (siehe **`openssl ecparam -list_curves`** für die Liste von allen elliptischen Kurven). Standardmäßig wird für die Ephemeral Schlüssel **secp256k1** benutzt.

Die korrekte Funktionalität dieses Attributes konnten wir im Debug-Modus nachweisen. Dabei wurde der Code-Verlauf verfolgt und die die Debug-Ausgaben betrachtet.

7.7.1 Identifikation relevanter Funktionen

Anhand von dem verwendeten named_curve Attribut wird in der Datei ssl/s_server.c ein neuer Ephemeral ECDH Schlüssel generiert und im SSL-Kontext gesetzt (**`ssl/s_server.c`** Z. 1731):

```
SSL_CTX_set_tmp_ecdh(ctx,ecdh);
```

7.8 Konfiguration von komplexen Ciphersuite-Listen

Im Folgenden wird diskutiert, wie man komplexe Ciphersuite-Listen erstellt. Dies ist ermöglicht durch spezifische Keywords, welche für diesen Zweck eingesetzt werden können.

Dieser Abschnitt basiert zu großem Teil auf dem Buch von Ivan Ristić [BPSSL2014].

7.8.1 Einleitung in die OpenSSL Ciphersuite Keywords

Eine Liste aller unterstützten Ciphersuites kann mit dem folgenden Befehl ausgegeben werden:

```
$ openssl ciphers
```

Eine Einschränkung der Ciphersuite-Liste kann z.B. wie folgt vorgenommen werden:

```
$ openssl ciphers 'SHA'
```

Damit werden nur die Ciphersuites genommen, welche den SHA-1-Algorithmus einsetzen. Auf der anderen Seite können auch nur Ciphersuites dargestellt werden, welche keinen SHA-1-Algorithmus benutzen:

```
$ openssl ciphers '!SHA'
```

Um die Ciphersuites beliebig kombinieren zu können, können mehrere Keywords auf einmal benutzt werden. Folgender Befehl stellt nur Ciphersuites dar, welche die ECDH und DH Algorithmen, aber keinen RSA Algorithmus einsetzen:

```
$ openssl ciphers 'ECDH DH !RSA'
```

Anstatt von Leerzeichen kann zwischen den Keywords auch ein Doppelpunkt benutzt werden:

```
$ openssl ciphers 'ECDH:DH:!RSA'
```

Bei der Bearbeitung werden die Keywords von links nach rechts geparst und die Ciphersuite-Reihenfolge so ausgegeben.

Für die komplette Liste der Keywords verweisen wir auf [BPSSL2014].

7.8.2 Konfiguration der Ciphersuites aus [TR021022]

Im Folgenden wird ein beispielhafter Prozess einer Ciphersuite-Definition nach [TR021022] beschrieben. Diesen Prozess fangen wir damit an, dass wir alle Ciphersuites verbieten, welche SHA1, MD5, RC4 oder keine Verschlüsselung / Authentikation einsetzen:

```
$ openssl ciphers '!NULL !SHA !MD5 !RC4'
```

Wir interessieren uns vor allem für Ciphersuites, welche Perfect Forward Secrecy anbieten:

```
$ openssl ciphers 'kEECDH kEDH'
```

Bei der Ciphersuite-Reihenfolge wollen wir aber Ciphersuites mit ECDSA und DSS Authentikationsalgorithmen bevorzugen (anstatt von RSA), daher werden die Verschlüsselungsalgorithmen mit dem Keyword aECDSA kombiniert:

```
$ openssl ciphers  'kEECDH+aECDSA kEECDH kEDH+DSS kEDH'
```

Damit werden zuerst alle Ciphersuites aufgelistet, welche ECDHE-ECDSA einsetzen, danach kommen alle anderen ECDHE-Algorithmen (also ECDHE-RSA). Anschließend folgen EDH-DSS und

EDH-RSA Ciphersuites.

Damit in der Liste auch ECDH Ciphersuites auftauchen, kann folgendes Keyword benutzt werden: 'kECDH'.

Bemerkung: Wenn wir ECDSA mit ECDH bevorzugen wollen, gibt es einen Bug in OpenSSL. Bei ECDH Ciphersuites ist ECDH als Authentifikationsmethode aufgelistet:

ECDH-ECDSA-AES256-SHA384 TLSv1.2 Kx=ECDH/ECDSA Au=ECDH

Daher liefert die Einstellung 'kECDH+aECDSA' kein erfolgreiches Ergebnis.

Empfehlung: Eine Möglichkeit um ECDH-ECDSA zu bevorzugen bietet 'kECDHe' an.

Wenn man alle oben beschriebenen Einstellungen kombiniert, entsteht folgende Liste:

```
$ openssl ciphers -v 'kEECDH+aECDSA kEECDH kEDH+DSS kEDH kECDHe kECDH !NULL !SHA !MD5 !LOW'
```

Mögliche Erweiterung um RSA-Verschlüsselung kann mit dem Keyword 'kRSA' erzielt werden:

```
$ openssl ciphers -v 'kEECDH+aECDSA kEECDH kEDH+DSS kEDH kECDHe kECDH kRSA !NULL !SHA !MD5 !LOW'
```

Wenn man einen OpenSSL s_server mit der beschriebenen Einstellung startet, werden nur die beschriebenen Ciphersuites eingesetzt.

```
/home/developer/openssl-1.0.1g/apps/openssl s_server -tls1_2 -accept 55443 -key ../keys/ec192-private.pem -cert ../keys/ec192-cert.pem -cipher 'kEECDH+aECDSA kEECDH kEDH+DSS kEDH kECDHe kECDH !NULL !SHA !MD5 !LOW'
```

Ausgabe eines Tests mit testssl.sh (die Liste enthält nur ECDH-ECDSA Ciphersuites, da mit OpenSSL nur ein Schlüssel eingestellt werden konnte):

```
xc02c   ECDHE-ECDSA-AES256-GCM-SHA384  ECDH       AESGCM     256
xc024   ECDHE-ECDSA-AES256-SHA384      ECDH       AES        256
xc02e   ECDH-ECDSA-AES256-GCM-SHA384   ECDH/ECDSA AESGCM     256
xc026   ECDH-ECDSA-AES256-SHA384       ECDH/ECDSA AES        256
xc02b   ECDHE-ECDSA-AES128-GCM-SHA256  ECDH       AESGCM     128
xc023   ECDHE-ECDSA-AES128-SHA256      ECDH       AES        128
xc02d   ECDH-ECDSA-AES128-GCM-SHA256   ECDH/ECDSA AESGCM     128
xc025   ECDH-ECDSA-AES128-SHA256       ECDH/ECDSA AES        128
```

Bemerkung: In [TR021022] werden auch Ciphersuites mit Preshared Keys empfohlen: ECDHE-PSK und DHE-PSK. Diese Ciphersuites sind von OpenSSL nicht unterstützt.

Des Weiteren werden keine DH-DSS und keine DH-RSA Ciphersuites aus [TR021022] unterstützt.

7.8.3 Konfiguration der Ciphersuites aus [TR021022] mit Whitelists

Eine explizite Konfiguration von Ciphersuites aus dem vorherigen Abschnitt kann man auch mit einer Whitelist Methode erstellen. Dafür wird der Server wie folgt gestartet:

```
/home/developer/openssl-1.0.1g/apps/openssl s_server -tls1_2
-accept 55443 -key ../keys/ec192-private.pem -cert ../keys/ec192-
cert.pem -cipher 'ECDHE-ECDSA-AES256-GCM-SHA384,ECDHE-ECDSA-
AES256-SHA384,ECDHE-ECDSA-AES128-GCM-SHA256,ECDHE-ECDSA-AES128-
SHA256,ECDHE-RSA-AES256-GCM-SHA384,ECDHE-RSA-AES256-SHA384,ECDHE-
RSA-AES128-GCM-SHA256,ECDHE-RSA-AES128-SHA256,DHE-DSS-AES256-GCM-
SHA384,DHE-DSS-AES256-SHA256,DHE-DSS-AES128-GCM-SHA256,DHE-DSS-
AES128-SHA256,DHE-RSA-AES256-GCM-SHA384,DHE-RSA-AES256-SHA256,DHE-
RSA-AES128-GCM-SHA256,DHE-RSA-AES128-SHA256,ECDH-ECDSA-AES256-GCM-
SHA384,ECDH-ECDSA-AES256-SHA384,ECDH-ECDSA-AES128-GCM-SHA256,ECDH-
ECDSA-AES128-SHA256,ECDH-RSA-AES256-GCM-SHA384,ECDH-RSA-AES256-
SHA384,ECDH-RSA-AES128-GCM-SHA256,ECDH-RSA-AES128-SHA256'
```

8 Auswirkung von Schwachstellen auf Webanwendungen

In diesem Kapitel wird untersucht, wie sich durch spezielle Einstellungen in Apache2- und Nginx-Servern Angriffe auf diese gezielt unterbinden lassen. Weiterhin werden Verbesserungsvorschläge und Schutzmaßnahmen diskutiert, die den vorgestellten Angriffen entgegenwirken.

Die Installation und die Untersuchung der Server wurden zum großen Teil anhand des Buches „Bulletproof SSL and TLS“ von Ivan Ristić durchgeführt [BPSSL2014].

Zusätzlich war ursprünglich für dieses Arbeitspaket auch eine Untersuchung von Denial-of-Service (DoS) Angriffen geplant. Da aber in vorherigen Arbeitspaketen keine DoS-Angriffe mit gravierenden Auswirkungen gefunden wurden, werden diese nicht betrachtet. Unsere Analyse wird sich daher detaillierter der Einstellung von TLS Ciphersuites und den analysierten Angriffen widmen.

8.1 Genutzte Umgebung und Tools

Für die Überprüfung der genutzten Einstellungen wurden folgende Tools eingesetzt:

- Wireshark: für die Verfolgung der TLS-Kommunikation
- Netbeans: Source-Code Analyse
- OpenSSL s_client, testssl.sh [testsslsh], sslyze [sslyze], Java TLS Client: Überprüfung der Server-Konfiguration

8.2 Apache httpd Server

Apache httpd [apachehttpd] ist ein weithin genutzter Web Server, welcher standardmäßig OpenSSL für die Durchführung der TLS-Kommunikation einsetzt.

8.2.1 Installation

Für die Analyse wurde die aktuelle Apache2-Version gewählt, die ihren Einsatz in der aktuellen Ubuntu 14.04 LTS Version findet. Der Server wurde wie folgt installiert:

```
$ sudo apt-get install apache2
```

Die Version lautet 2.4.7:

```
$ apache2 -version
Server version: Apache/2.4.7 (Ubuntu)
Server built:   Jul 22 2014 14:36:38
```

Standardmäßig ist SSL nicht unter den aktiven Modulen aufgelistet, dies muss man mit dem folgenden Befehl tun:

```
$ sudo a2enmod ssl
```

Die genutzten Schlüssel und Zertifikate werden anschließend in der Datei `/etc/apache2/sites-available/default-ssl.conf`

definiert. Diese Datei muss dann ins Verzeichnis

`/etc/apache2/sites-enabled`

kopiert werden (oder man erstellt einen symbolischen Link auf die Datei im sites-available Verzeichnis). Eine minimale Konfiguration von mehreren Zertifikaten und Schlüsseln sieht dabei wie folgt aus:

```
<IfModule mod_ssl.c>
	<VirtualHost _default_:443>
		#   Enable/Disable SSL for this virtual host.
		SSLEngine on
		#   RSA Cert and Key files
		SSLCertificateFile  /home/developer/TLS-Attacker/resources/server-
cert2048.pem
		SSLCertificateKeyFile /home/developer/TLS-Attacker/resources/privkey2048.pem

		# EC Cert and Key files
		SSLCertificateFile  /home/developer/TLS-Attacker/resources/ec192-cert.pem
		SSLCertificateKeyFile /home/developer/TLS-Attacker/resources/ec192-private.-
pem

		# DSA Cert and Key files
		SSLCertificateFile  /home/developer/TLS-Attacker/resources/dsa-cert.pem
		SSLCertificateKeyFile /home/developer/TLS-Attacker/resources/dsa-private.pem
		…
	</VirtualHosts>
</IfModule>
```

Nach dem Server-Neustart mit

```
$ sudo service apache2 restart
```

kann eine SSL-Verbindung mit einer Test-Webseite erfolgreich erstellt werden.

8.2.2 Einstellungen der Ciphersuites

In der Standard-Einstellung sind unsichere Protokolle und Ciphersuites erlaubt. Eine sichere Konfiguration kann über die Datei:

```
/etc/apache2/mods-enabled/ssl.conf
```

eingerichtet werden. Die vorgegebene Datei enthält folgende (unsichere) Konfiguration für TLS-Protokoll-Version und Ciphersuites:

SSLCipherSuite HIGH:MEDIUM:!aNULL:!MD5

SSLProtocol all

Folgende Einstellungen verändern das Server-Verhalten so, dass nur TLS1.2 und die Ciphersuites aus dem AP3.7 unterstützt werden:

SSLCipherSuite kEECDH+aECDSA:kEECDH:kEDH+DSS:kEDH:kECDHe:kECDH:!NULL:!SHA:!MD5:!LOW

SSLHonorCipherOrder on

SSLProtocol TLSv1.2

Mit der Option **SSLHonorCipherOrder on** wird definiert, dass der *Server* die Ciphersuite-Reihenfolge festlegt, und nicht der Client. Die in der `ssl.conf` Datei definierte Reihenfolge wird aber nicht beachtet, sondern der Server entscheidet über die Reihenfolge selber. Dabei werden einfach Ciphersuites mit längeren Schlüsseln priorisiert. Ein Test mit testssl.sh ergibt folgendes Ergebnis bei dem Test der Ciphersuite-Reihenfolge.

```
ECDHE-RSA-AES256-GCM-SHA384    256
ECDHE-ECDSA-AES256-GCM-SHA384  256
ECDHE-RSA-AES256-SHA384        256
ECDHE-ECDSA-AES256-SHA384      256
DHE-DSS-AES256-GCM-SHA384      256
DHE-RSA-AES256-GCM-SHA384      256
DHE-RSA-AES256-SHA256          256
DHE-DSS-AES256-SHA256          256
ECDH-ECDSA-AES256-GCM-SHA384   256
ECDH-ECDSA-AES256-SHA384       256
ECDHE-RSA-AES128-GCM-SHA256    128
ECDHE-ECDSA-AES128-GCM-SHA256  128
ECDHE-RSA-AES128-SHA256        128
ECDHE-ECDSA-AES128-SHA256      128
DHE-DSS-AES128-GCM-SHA256      128
```

```
DHE-RSA-AES128-GCM-SHA256      128
DHE-RSA-AES128-SHA256          128
DHE-DSS-AES128-SHA256          128
ECDH-ECDSA-AES128-GCM-SHA256   128
ECDH-ECDSA-AES128-SHA256       128
```

Die explizite Konfiguration der Ciphersuite-Reihenfolge wird von Apache nicht unterstützt. Die gleiche Konfiguration kann deswegen auch mit dieser einfacheren Einstellung erzielt werden:

```
SSLCipherSuite 'EECDH:EDH:ECDH:!NULL:!SHA:!MD5:!LOW'
```

8.2.2.1 Fazit

Die in der `ssl.conf` festgelegte Reihenfolge der Ciphersuites wird ignoriert, was auf den ersten Blick unschön erscheint. Der Server-Betreiber kann nämlich nicht Prioritäten bei den ausgewählten Ciphersuites beliebig setzen. Auf der anderen Seite werden damit Fehler beim Konfigurieren der ssl.conf Datei verhindert. Zusätzlich kann die vom Apache httpd festgelegte Reihenfolge der Ciphersuites gemäß o.a. Liste unter Berücksichtigung der Ergebnisse von AP 3.7 als sicher angenommen werden.

8.2.3 Ephemeral Schlüssel

Im AP 3.6 haben wir gezeigt, dass der OpenSSL Server beim Einsatz von Ephemeral Ciphersuites lediglich einen Ephemeral Schlüssel initialisiert, wenn der Server gestartet wird. Dieser Ephemeral Schlüssel wird dann für alle Verbindungen genutzt, bis der Server neu gestartet wird.

Bei Apache2 konnten wir dieses Verhalten nicht beobachten. Der Server hat bei jeder neuen Verbindung einen neuen Ephemeral Schlüssel generiert und benutzt. Dies wurde beispielhaft anhand von zwei nacheinander folgenden TLS-Handshakes gezeigt:

```
Server key: SunPKCS11-NSS EC public key, 256 bits (id 1, session object)

                                    public                  x                   coord:
57589228357079787379036088106510505657208751794320605672088761793013009168399

                                    public                  y                   coord:
44669572253736216554274472083367864588085082173821118075350989423066424117181

  parameters: secp256r1 [NIST P-256, X9.62 prime256v1] (1.2.840.10045.3.1.7)


Server key: SunPKCS11-NSS EC public key, 256 bits (id 1, session object)

                                    public                  x                   coord:
10505876022245392000284542492474772317139323143209344038577139405625828462064 0

                                    public                  y                   coord:
90926244356562719078465936321016196160229843152830629021125353061130972657587

  parameters: secp256r1 [NIST P-256, X9.62 prime256v1] (1.2.840.10045.3.1.7)
```

8.2.3.1 Fazit

Bei jeder neuen Verbindung werden in Apache httpd frische Ephemeral Schlüssel eingesetzt, was positiv zu bewerten ist. Damit wird Perfect Forward Secrecy gewährleistet.

Wie im AP 3.6 untersucht wurde, ist die Generierung von frischen Ephemeral Schlüsseln in der OpenSSL-Implementierung nämlich nicht standardmäßig aktiviert.

8.2.4 Insecure Renegotiation

Insecure Renegotiation (siehe AP 3.1 und [RFC5746]) ist standardmäßig ausgeschaltet. Mit dem sslyze Tool konnte es erfolgreich überprüft werden:

```
$ ./sslyze.py --reneg localhost:443

 SCAN RESULTS FOR LOCALHOST:443 - 127.0.0.1:443

 ---------------------------------------------

  * Session Renegotiation:

      Client-initiated Renegotiations:    OK - Rejected

      Secure Renegotiation:               OK - Supported
```

Bei dem Test versucht das Tool eine Insecure Renegotiation durchzuführen. Wie das Ergebnis zeigt, ist dies nicht möglich.

Es besteht eine Möglichkeit, Insecure Renegotiation explizit in der Datei /etc/apache2/mods-enabled/ssl.conf einzuschalten (was der Administrator vermeiden sollte).

8.2.4.1 Fazit

Der Apache httpd Server ist in der Standardeinstellung gegen Insecure Renegotiation Angriffe geschützt.

8.2.5 CRIME und TLS Kompression

Der CRIME-Angriff ist anwendbar, wenn TLS Kompression auf dem Server explizit aktiviert ist (siehe AP 3.1).

TLS Kompression auf dem httpd Server ist explizit ausgeschaltet. Dies konnte mit dem sslyze Tool nachgeprüft werden:

```
$ ./sslyze.py --compression localhost:443

 SCAN RESULTS FOR LOCALHOST:443 - 127.0.0.1:443

 ---------------------------------------------
```

```
* Deflate Compression:
    OK - Compression disabled
```

Laut [BPSSL2014] wurde TLS Kompression in Apache2 im Jahre 2013 standardmäßig ausgeschaltet, mit Versionen 2.2.6 und 2.4.4.

8.2.5.1 Fazit

Apache httpd ist in der Standardeinstellung gegen den CRIME-Angriff geschützt.

8.3 Nginx

Nginx wird heutzutage als ein beliebter Web-Server und Reverse-Proxy eingesetzt [nginx].

8.3.1 Installation

Für die Tests wurde die neueste Version von Nginx gewählt und kompiliert: nginx-1.7.9. Für die manuelle Kompilierung haben wir uns aus dem Grund entschieden, da Nginx bei der Kompilierung ein OpenSSL-Verzeichnis als Eingabe nimmt und die ganze OpenSSL-Bibliothek mitkompiliert. Dies hat es uns erlaubt, die openssl-1.0.1g Version zu referenzieren und den ganzen Kompilierungsprozess zu verfolgen.

Nginx wurde manuell wie folgt kompiliert:

```
$ ./configure --prefix=/opt/nginx --with-openssl=../openssl-1.0.1g
--with-http_ssl_module -without-http_gzip_module

$ make

$ make install
```

Eine Besonderheit an dieser Kompilierung ist, dass Nginx direkt auf den OpenSSL-Code zugreift und diesen konfiguriert. Wenn spezifische Konfigurations-Optionen für OpenSSL weitergereicht werden sollen, kann dies mit dem Flag `--with-openssl-opt` erreicht werden, z.B.:

```
--with-openssl-opt="enable-ec_nistp_64_gcc_128"
```

Bemerkung: Bevor die Kompilierung komplett funktioniert hat, musste man in einigen der OpenSSL Dateien einen Patch anwenden.

Empfehlung: Der Patch ist unter https://gist.github.com/martensms/10107481 zu finden und fixt lediglich die Generierung der OpenSSL-Dokumentation. Wenn der Patch nicht angewendet wird, kommt es zu Fehlern bei der Bearbeitung von POD-Dateien (Plain Old Documentation). In neueren OpenSSL-Versionen sollte dieses Problem gelöst werden. Der Patch hat keine Sicherheitsimplikationen.

Nach der Installation kann TLS auf dem Server unter /opt/nginx/conf/nginx.conf aktiviert werden,

indem man folgenden Code hinzufügt:

```
server {
    listen       8443 ssl;
    server_name  localhost;
    ssl_certificate      /home/developer/TLS-Attacker/resources/ec256-cert.pem;
    ssl_certificate_key  /home/developer/TLS-Attacker/resources/ec256-private.pem;
    ssl_session_cache    shared:SSL:1m;
    ssl_session_timeout  5m;

    ssl_ciphers  HIGH:!aNULL:!MD5;
    ssl_prefer_server_ciphers  on;

    location / {
        root   html;
        index  index.html index.htm;
    }
}
```

Im Gegensatz zu Apache2 kann in Nginx nur eine Zertifikats-Datei und eine Schlüssel-Datei eingestellt werden. Dieses Problem und ein Patch dafür wurden schon unter http://mailman.nginx.org/pipermail/nginx-devel/2013-October/004376.html beschrieben. Der Patch wurde aber noch nicht in den Code eingepflegt und nach unseren Tests erlaubt die neueste Nginx Version noch nicht mehrere Schlüssel einzubinden. Der Server wird daher bei unseren Tests mit einem EC-Zertifikat initialisiert.[9]

Der Server kann anschließend mit

```
$ /opt/nginx/sbin/nginx
```

gestartet werden. Der Befehl zum Neustart lautet:

```
$ /opt/nginx/sbin/nginx –s reload
```

8.3.2 Einstellungen der Ciphersuites

Die oben vorgestellte TLS-Einstellung basiert auf einer Nginx default Konfiguration. Diese Konfiguration erlaubt z.B. TLS Ciphersuites mit SHA-1 oder 3DES. Wie bei Apache2, können hier TLS Ciphersuites angepasst werden, indem man z.B. ssl_ciphers auf den folgenden Wert setzt:

```
kEECDH+aECDSA:kEECDH:kEDH+DSS:kEDH:kECDHe:kECDH:!NULL:!SHA:!MD5:!
LOW
```

9 Damit ist der Server nicht so konfigurierbar, dass er alle Ciphersuites aus dem [TR021022] unterstützen kann.

Nach dem Server-Neustart bleiben nur folgende TLS Ciphersuites, die von dem Server angeboten werden:

```
ECDHE-ECDSA-AES256-GCM-SHA384  256
ECDHE-ECDSA-AES256-SHA384      256
ECDH-ECDSA-AES256-GCM-SHA384   256
ECDH-ECDSA-AES256-SHA384       256
ECDHE-ECDSA-AES128-GCM-SHA256  128
ECDHE-ECDSA-AES128-SHA256      128
ECDH-ECDSA-AES128-GCM-SHA256   128
ECDH-ECDSA-AES128-SHA256       128
```

Ähnlich wie bei Apache2, wird auch bei Nginx *keine* explizite Konfiguration der Ciphersuite-Reihenfolge unterstützt. Die gleiche Ciphersuite-Konfiguration wie oben kann deswegen auch mit dieser einfachen Einstellung erzielt werden:

```
ssl_ciphers 'EECDH:EDH:ECDH:!NULL:!SHA:!MD5:!LOW'
```

8.3.2.1 Fazit

Die vom Server-Betreiber festgelegte Reihenfolge der Ciphersuites wird (ähnlich wie bei Apache httpd) ignoriert, es können daher keine Prioritäten für die ausgewählten Ciphersuites gesetzt werden. Die vom Nginx festgelegte Reihenfolge der Ciphersuites kann aber gemäß o.a. Liste unter Berücksichtigung der Ergebnisse von AP 3.7 als sicher angenommen werden.

Der Server war leider zur Zeit der Untersuchung nur mit einem Zertifikat konfigurierbar. Erweiterung der Server-Konfiguration durch Einsatz von mehreren Zertifikaten würde die Sicherheit und Kompatibilität erhöhen.

8.3.3 Ephemeral Schlüssel

Im AP 3.6 haben wir gezeigt, dass der OpenSSL Server beim Einsatz von Ephemeral Ciphersuites lediglich einen Ephemeral Schlüssel initialisiert, wenn der Server gestartet wird. Dieser Ephemeral Schlüssel wird dann für alle Verbindungen genutzt, bis der Server neu gestartet wird.

Wie bei Apache2, konnten wir auch bei Nginx dieses Verhalten *nicht* beobachten. Der Server hat bei jeder neuen Verbindung einen neuen Ephemeral Schlüssel generiert und benutzt. Dies wurde beispielhaft anhand von zwei nacheinander folgenden TLS-Handshakes gezeigt:

```
*** ECDH ServerKeyExchange
Server key: SunPKCS11-NSS EC public key, 256 bits (id 2, session object)
                                        public                  x                  coord:
55186964351098839172629543863671166251688165921439447508393592744256701659814
                                        public                  y                  coord:
11096152486096919105178987910803724533870608033985623212917188563111599332885 8
```

```
  parameters: secp256r1 [NIST P-256, X9.62 prime256v1] (1.2.840.10045.3.1.7)


Server key: SunPKCS11-NSS EC public key, 256 bits (id 2, session object)
                                        public                  x                   coord:
105527897166966899729741464325066809607993363077569126244076435189494754505364
                                        public                  y                   coord:
226761211119053664522030921752702115113384803924813451915283819811935100330002
  parameters: secp256r1 [NIST P-256, X9.62 prime256v1] (1.2.840.10045.3.1.7)
```

Dies kann sichergestellt werden, indem bei dem Server-Start folgende Aufrufe gemacht werden, welche eine Ephemeral-Schlüssel-Generierung bei jeder neuen Verbindung erzwingen:

```
SSL_CTX_set_options(ssl->ctx, SSL_OP_SINGLE_DH_USE);
SSL_CTX_set_options(ssl->ctx, SSL_OP_SINGLE_ECDH_USE);
```

Zeile 250 und 1004 in der Datei `nginx-1.7.9/src/event/ngx_event_openssl.c`. Wenn diese zwei Aufrufe nicht vorhanden wären, würde der Nginx Server die Ephemeral Schlüssel nur bei einem Neustart generieren (das Verhalten wäre also mit dem OpenSSL Test-Server identisch).

8.3.3.1 Fazit

Bei jeder neuen Verbindung werden in Nginx frische Ephemeral Schlüssel eingesetzt, was positiv zu bewerten ist. Damit wird Perfect Forward Secrecy gewährleistet, wie es auch bei Apache httpd der Fall ist.

8.3.4 Insecure Renegotiation

Insecure Renegotiation (siehe AP 3.1 und [RFC5746]) ist standardmäßig ausgeschaltet. Mit dem sslyze Tool konnte es erfolgreich überprüft werden:

```
$ ./sslyze.py --reneg localhost:8443
SCAN RESULTS FOR LOCALHOST:8443 - 127.0.0.1:8443
 ---------------------------------------------
  * Session Renegotiation:
      Client-initiated Renegotiations:    OK - Rejected
      Secure Renegotiation:               OK - Supported
```

Bei dem Test versucht das Tool eine Insecure Renegotiation durchzuführen. Wie aus dem Ergebnis sichtbar ist, ist dies nicht möglich.

Es besteht eine Möglichkeit, Insecure Renegotiation explizit in der Datei `/etc/apache2/mods-`

`enabled/ssl.conf` einzuschalten (was der Administrator vermeiden sollte).

8.3.4.1 Fazit

Der Nginx Server ist in der Standardeinstellung nicht gegen Insecure Renegotiation Angriffe anfällig.

8.3.5 CRIME und TLS Kompression

Der CRIME-Angriff ist anwendbar, wenn TLS Kompression auf dem Server explizit aktiviert ist (siehe AP 3.1).

TLS Kompression ist explizit ausgeschaltet. Dies konnte mit dem sslyze Tool nachgeprüft werden:

```
$ ./sslyze.py --compression localhost:8443
```

Laut [BPSSL2014] wurde TLS compression in Nginx im Jahre 2012 ausgeschaltet, mit den Versionen 1.2.2 und 1.3.2.

8.3.5.1 Fazit

Der Nginx Server ist in der Standardeinstellung nicht gegen den CRIME-Angriff anfällig.

8.4 Zusammenfassung: Sicherer Einsatz von Apache httpd und Nginx

In den vorherigen zwei Abschnitten wurden die Standardeinstellungen und Sicherheitseigenschaften von Apache httpd und Ningx analysiert. Die wichtigsten Merkmale sind in Tabelle 5 aufgelistet.

Einstellung / Eigenschaft	**Apache httpd (2.4.7)**	**Nginx (1.7.9)**
Einschränkung der TLS-Version	ja	ja
Einschränkung der Ciphersuites	ja	ja
Server-seitige Auswahl der Ciphersuites-Reihenfolge	ja	ja
Unterstützung der Ciphersuites aus [TR021022]	ja[10]	ja[11]
Einsatz von mehreren Schlüsselpaaren	ja	nein
Frische Ephemeral Schlüssel	ja	ja
Anfälligkeit gegen den CRIME Angriff	nein	nein
Anfälligkeit gegen den Insecure Renegotiation Angriff	nein	nein

Tabelle 5: Zusammenfassung der Sicherheitsmerkmale von Apache httpd und Nginx

10 In [TR021022] werden auch Ciphersuites mit Preshared Keys empfohlen. Diese werden aber durch OpenSSL nicht unterstützt.

11 In [TR021022] werden auch Ciphersuites mit Preshared Keys empfohlen. Diese werden aber durch OpenSSL nicht unterstützt. Zusätzlich können auf dem Server nicht mehrere Schlüsselpaare eingesetzt werden. Dies bedeutet, dass sich der Server-Betreiber für die Unterstützung von ECDSA *oder* RSA Signaturen entscheiden muss.

Wie in der Tabelle zu sehen ist, erfüllen beide Server die wichtigsten Sicherheitseigenschaften. Der einzige Nachteil bei Nginx ist, dass dieser Server nur mit einem Schlüsselpaar konfigurierbar ist.

Im Folgenden werden wir auf eine empfohlene und praxisnahe Konfiguration von diesen zwei Servern eingehen. Wir werden die wichtigsten Schritte zusammenfassen, die ein Server-Administrator durchführen muss, um eine sichere Konfiguration zu erreichen. Dabei gehen wir davon aus, dass die Server wie in den obigen Kapiteln auf einem Linux-System installiert werden. Darüber hinaus verfügt der Administrator über RSA und EC Zertifikate, welche von vertrauenswürdigen CAs ausgestellt wurden. Wir werden die Schlüssel und Zertifikate wie folgt referenzieren:

- **rsa-key.pem**: (mindestens) 2048 Bit RSA-Schlüssel
- **rsa-cert.pem**: passendes Zertifikat zu dem RSA-Schlüssel
- **ec-key.pem**: EC-Schlüssel (z.B. für eine elliptische Kurve secp256r1)
- **ec-cert.pem**: passendes Zertifikat zu dem EC-Schlüssel

8.4.1 Sichere Konfiguration von Apache httpd

Standardmäßig ist bei Apache httpd SSL nicht unter den aktiven Modulen aufgelistet, dies wird mit dem folgenden Befehl erreicht:

```
$ sudo a2enmod ssl
```

Die genutzten Schlüssel und Zertifikate werden anschließend in der Datei

```
/etc/apache2/sites-available/default-ssl.conf
```

definiert. Diese Datei muss dann ins Verzeichnis

```
/etc/apache2/sites-enabled
```

kopiert werden. Eine Konfiguration mit den vorhandenen Zertifikaten kann wie folgt definiert werden:

```
<IfModule mod_ssl.c>
	<VirtualHost _default_:443>
		#   Enable/Disable SSL for this virtual host.
		SSLEngine on
		#   RSA Cert and Key files
		SSLCertificateFile  [path-to-key]/rsa-cert.pem
		SSLCertificateKeyFile [path-to-key]/rsa-key.pem

		# EC Cert and Key files
		SSLCertificateFile  [path-to-key]/ec-cert.pem
```

```
            SSLCertificateKeyFile [path-to-key]/ec-key.pem
            …
        </VirtualHosts>
</IfModule>
```

Nach einem Server-Neustart mit

$ sudo service apache2 restart

kann eine SSL-Verbindung mit einer Test-Webseite erfolgreich erstellt werden.

Eine sichere Konfiguration der TLS-Version und Ciphersuites kann über die Datei:

`/etc/apache2/mods-enabled/ssl.conf`

eingerichtet werden. Dafür sollten folgende Angaben verändert werden:

SSLHonorCipherOrder on

SSLProtocol all -SSLv2 -SSLv3

SSLCipherSuite 'EECDH:EDH:ECDH:kRSA:!aNULL:!eNULL:!MD5:HIGH:!LOW:!RC4:!3DES:!EXP:!DSS:!SRP:!CAMELLIA:!IDEA:!SEED:!PSK'

Damit wird der Server mit den TLS-Versionen 1.0, 1.1 und 1.2 konfiguriert. Es werden Ciphersuites aus [TR021022] eingesetzt, wobei die Liste um obligatorische RSA-Verschlüsselung erweitert wurde.[12] Über die Reihenfolge der Ciphersuites entscheidet der Server.

Ein Test mit testssl.sh ergibt folgende Ergebnisse bzgl. der unterstützen Ciphersuites:

TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384

TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384

TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384

TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA

TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA

TLS_DHE_RSA_WITH_AES_256_GCM_SHA384

TLS_DHE_RSA_WITH_AES_256_CBC_SHA256

TLS_DHE_RSA_WITH_AES_256_CBC_SHA

TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384

TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384

TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA

TLS_RSA_WITH_AES_256_GCM_SHA384

TLS_RSA_WITH_AES_256_CBC_SHA256

TLS_RSA_WITH_AES_256_CBC_SHA

12 **TLS_RSA_WITH_AES_128_CBC_SHA** ist in TLS 1.2 als eine “mandatory ciphersuite” definiert.

```
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_RSA_WITH_AES_128_CBC_SHA256
TLS_RSA_WITH_AES_128_CBC_SHA
```

8.4.2 Sichere Konfiguration von Nginx

Nach der Installation von Nginx muss TLS unter

`/opt/nginx/conf/nginx.conf`

aktiviert und konfiguriert werden. Eine sichere Konfiguration kann dabei wie folgt aussehen:

```
server {
    ...
    ssl_certificate      [path-to-key]/rsa-cert.pem;
    ssl_certificate_key  [path-to-key]/rsa-key.pem;
    ssl_session_cache    shared:SSL:1m;
    ssl_session_timeout  5m;

    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
      ssl_ciphers   EECDH:EDH:ECDH:kRSA:!aNULL:!eNULL:!MD5:HIGH:!LOW:!RC4:!3DES:!EXP:!
DSS:!SRP:!CAMELLIA:!IDEA:!SEED:!PSK;
    ssl_prefer_server_ciphers  on;
    ...
}
```

Im Gegensatz zu Apache httpd kann in Nginx nur ein Zertifikat konfiguriert werden. In unserem Beispiel haben wir ein RSA-Zertifikat benutzt.

Mit dieser Konfiguration wird der Server TLS-Versionen 1.0, 1.1 und 1.2 unterstützen. Es werden Ciphersuites aus [TR021022] eingesetzt, wobei die Liste um obligatorische RSA-Verschlüsselung erweitert wurde. Es werden aber keine ECDSA Ciphersuites angeboten, da nur ein RSA-Zertifikat zur Verfügung gestellt wurde. Über die Reihenfolge der Ciphersuites entscheidet der Server.

Nach dieser Einstellung bietet der Server folgende Ciphersuites an:

TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384

TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA

TLS_DHE_RSA_WITH_AES_256_GCM_SHA384

TLS_DHE_RSA_WITH_AES_256_CBC_SHA256

TLS_DHE_RSA_WITH_AES_256_CBC_SHA

TLS_RSA_WITH_AES_256_GCM_SHA384

TLS_RSA_WITH_AES_256_CBC_SHA256

TLS_RSA_WITH_AES_256_CBC_SHA

TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256

TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA

TLS_DHE_RSA_WITH_AES_128_GCM_SHA256

TLS_DHE_RSA_WITH_AES_128_CBC_SHA256

TLS_DHE_RSA_WITH_AES_128_CBC_SHA

TLS_RSA_WITH_AES_128_GCM_SHA256

TLS_RSA_WITH_AES_128_CBC_SHA256

TLS_RSA_WITH_AES_128_CBC_SHA

8.5 Jenseits von SSL/TLS: OpenSSH

OpenSSH [openssh] ist ein Programmpaket zur Unterstützung von Secure Shell (SSH). Dabei greift OpenSSH auf kryptographische Funktionen von OpenSSL zu, die über die OpenSSL-API angesprochen werden. Aus diesem Grund wird im Folgenden die Unterstützung von Perfect Forward Secrecy und der CRIME-Angriff untersucht.

Die Untersuchung wurde anhand des aktuellen Source-Codes durchgeführt. Die OpenSSH-Version ist 6.7p1.

8.5.1 Ephemeral Schlüssel

Wie in den vorherigen Kapiteln beschrieben wurde, benutzt ein OpenSSL-Server für jede Verbindung den gleichen DH/ECDH-Schlüssel, wenn dies mit Flags `SSL_OP_SINGLE_DH_USE` und `SSL_OP_SINGLE_ECDH_USE` im SSL-Context nicht explizit verändert wird.

In OpenSSH wird OpenSSL anders als bei Nginx oder Apache2 eingesetzt. OpenSSH benutzt nicht den ganzen OpenSSL-Context, es werden nur bestimmte Funktionen der OpenSSL-Schnittstelle genutzt. So kümmert sich der OpenSSH-Code selbstständig um die Erzeugung von Ephemeral Schlüsseln und um die Berechnung des Shared Secret. Dies ist in der Funktion

`kexecdh_server(Kex *kex)`

aus der Datei `kexecdhs.c` zu sehen:

```
void
kexecdh_server(Kex *kex)
{
	EC_POINT *client_public;
	EC_KEY *server_key;
	const EC_GROUP *group;
	BIGNUM *shared_secret;
	Key *server_host_private, *server_host_public;
	u_char *server_host_key_blob = NULL, *signature = NULL;
	u_char *kbuf, *hash;
	u_int klen, slen, sbloblen, hashlen;

	if ((server_key = EC_KEY_new_by_curve_name(kex->ec_nid)) == NULL)
		fatal("%s: EC_KEY_new_by_curve_name failed", __func__);
	if (EC_KEY_generate_key(server_key) != 1)
		fatal("%s: EC_KEY_generate_key failed", __func__);
	group = EC_KEY_get0_group(server_key);

	if (kex->load_host_public_key == NULL ||
	    kex->load_host_private_key == NULL)
		fatal("Cannot load hostkey");
	server_host_public = kex->load_host_public_key(kex->hostkey_type);
	if (server_host_public == NULL)
		fatal("Unsupported hostkey type %d", kex->hostkey_type);
	server_host_private = kex->load_host_private_key(kex->hostkey_type);

	debug("expecting SSH2_MSG_KEX_ECDH_INIT");
	packet_read_expect(SSH2_MSG_KEX_ECDH_INIT);
	if ((client_public = EC_POINT_new(group)) == NULL)
		fatal("%s: EC_POINT_new failed", __func__);
```

```
packet_get_ecpoint(group, client_public);
packet_check_eom();

if (key_ec_validate_public(group, client_public) != 0)
       fatal("%s: invalid client public key", __func__);

/* Calculate shared_secret */
klen = (EC_GROUP_get_degree(group) + 7) / 8;
kbuf = xmalloc(klen);
if (ECDH_compute_key(kbuf, klen, client_public,
    server_key, NULL) != (int)klen)
       fatal("%s: ECDH_compute_key failed", __func__);
```

Wie in dem Code zu sehen ist, generiert der Server in jedem Vorgang seinen Ephemeral Schlüssel mit der Funktion `EC_KEY_generate_key` neu. Dieser Schlüssel wird anschließend für die Berechnung des Shared Secrets in der Funktion `ECDH_compute_key` benutzt.

Ein ähnliches Vorgehen kann beim Ephemeral Diffie-Hellman Schlüsselaustausch gefunden werden. Dies zeigt sich in der Datei `kexedhs.c`.

8.5.1.1 Fazit

Der Ephemeral Schlüssel wird korrekterweise bei jeder Verbindung neu generiert.

8.5.2 CRIME und Kompression in OpenSSH

Der CRIME-Angriff auf TLS-Kompression funktioniert nur in bestimmten Szenarien, in welchen der Angreifer sein Opfer zwingen kann, bekannte Klartexte mit unbekannten Daten zu verketten, und diese später verschlüsselt über die Leitung zu verschicken. Anhand der Länge der verschlüsselten Daten kann der Angreifer Rückschlüsse über die versendeten Daten ziehen.

SSH definiert so wie TLS auch eine Möglichkeit, Daten vor dem Verschlüsseln zu komprimieren. Dieses Problem wurde schon von mehreren Sicherheitsexperten bemängelt, es wurde bisher aber keine konkrete Möglichkeit gefunden, wie man den CRIME-Angriff direkt auf SSH anwendet. Der Angreifer kann nämlich sein Opfer nicht dazu zwingen, bestimmte Daten wiederholt über den SSH-Kanal zu schicken.

Trotzdem ist es nicht empfehlenswert Kompression in SSH einzuschalten. In OpenSSH ist SSH-Kompression standardmäßig ausgeschaltet und kann erst mit dem Kompression-Paramter (`Compression yes`) in `der ssh_config` Datei explizit eingeschaltet werden.

8.5.2.1 Fazit

Die Datenkompression in OpenSSH wird standardmäßig deaktiviert, was als positiv zu bewerten ist.

9 Schutzmaßnahmen

Dieses Kapitel fasst Schutzmaßnahmen gegen Schwächen zusammen, die in den Analysen der vorigen Kapitel und Arbeitspakete festgestellt wurden. Die Schwächen werden hierfür thematisch zusammengefasst, sofern dies sinnvoll und möglich ist. Es wird auf die detaillierte Analyse in den entsprechen Arbeitspaketen verwiesen. Der Übersichtlichkeit halber beschränkt sich dieses Kapitel auf sicherheitskritische Schwächen.

9.1 Schutzmaßnahmen für den Zufallszahlengenerator

9.1.1 Mehrfache Verwendung von Entropie aus der Entropiedatei

Die Verwendung einer Datei, um Entropie zwischen verschiedenen Programmaufrufen zu speichern stellt eine sinnvolle Methode dar, um direkt nach dem Programmstart hochwertige Zufallszahlen liefern zu können. Allerdings sorgen verschiedene Eigenschaften von OpenSSL dafür, dass die Verwendung einer solchen Datei dazu führen kann, dass der RNG mehrfach mit den gleichen Daten geseedet wird:

- die Datei wird nach dem Lesen nicht gelöscht (s. AP2, Bemerkung 4),
- es wird ein Standard-Dateiname gewählt (s. AP2, Bemerkung 5).

Um diese Datei sicher zu verwenden, muss sichergestellt werden, dass

- die Entropie-Datei nach dem Lesen von Daten gelöscht wird (s. AP2, Empfehlung 4),
- jedes Programm eine eigene Entropie-Datei verwendet (s. AP2, Empfehlung 5),

Darüber hinaus ist es unter Umständen möglich, dass andere Benutzer Zugriff auf die Entropiedatei erhalten, da

- die Zugriffsrechte auf die Datei evtl. nicht korrekt gesetzt werden (s. AP2, Bemerkung 12)

Daher muss sichergestellt werden, dass

- die Zugriffsrechte auf die Datei korrekt gesetzt werden (s. AP2, Empfehlung 12).

9.1.2 Nutzung/Seeding des RNG mit zu wenig Entropie

Ein weiteres Problem ist dass Seeding des RNG aus Quellen, die nicht ausreichend Entropie liefern.

- Es wird /dev/urandom als Entropiequelle genutzt (s. AP2, Bemerkung 9).
- /dev/urandom wird sogar als erste Entropiequelle genutzt, d.h. im Normalfall wird keine Entropie aus /dev/random genutzt (s. AP2, Bemerkung 19).

- Der „entropy gathering daemon“ wird standardmäßig aktiviert. Über diesen sind jedoch keine detaillierten Analysen verfügbar (s. AP2, Bemerkung 21 und 22).

Um das Seeding mit /dev/random zu garantieren, sollte

- das Seeding mit /dev/urandom deaktiviert werden (s. AP2, Empfehlung 9 und 19),
- der „entropy gathering daemon“ deaktiviert werden (s. AP2, Empfehlung 21 und 22).

Die Funktion RAND_pseudo_bytes() liefert evtl. Zufallsdaten mit geringer Entropie:

- Selbst wenn der RNG nicht geseedet wurde, liefert RAND_pseudo_bytes() Zufallsdaten (s. AP2, Bemerkung 10 und 24).

Daher sollte auf die Nutzung von RAND_pseudo_bytes() verzichtet werden oder vor dem Aufruf von RAND_pseudo_bytes() überprüft werden, ob der RNG ausreichend geseedet wurde (s. AP2, Empfehlungen 10 und 24).

9.1.3 Threading- und Fork-Safety

Die Verwendung von OpenSSL in nebenläufigen Programmen ist umständlich. So gibt es sowohl bei durch Threading als auch durch Forking parallelisierten Programmen Schwächen.

- Der RNG ist in der Default-Konfiguration nicht Thread-Safe (s. AP2, Bemerkung 14).
- Der RNG-Zustand wird in einem Buffer fester Größe gespeichert, auf den Threads gleichzeitig zugreifen, die kollidieren könnten (s. AP2, Bemerkung 16).

Daher sollte

- entweder auf den Zugriff auf den RNG aus verschiedenen Threads verzichtet, oder
- die notwendigen Locking-Funktionen gesetzt (s. AP2, Empfehlung 14) und die Locks während des Zugriffs auf den RNG gehalten (s. AP2, Empfehlung 16) werden.

Wenn ein weiterer Prozess durch Aufruf des fork()-Systemaufrufs erzeugt wird, dann wird auch der aktuelle RNG-Zustand kopiert. Damit hat der RNG-Zustand des geforkten Prozess den gleichen Zustand wie der des Mutterprozess.

- Der Zustand des RNGs des Mutterprozesses kann im Kindprozess relativ einfach rekonstruiert werden (s. AP2, Bemerkung 15).

Deshalb muss der RNG nach einem fork()-Aufruf neu geseedet werden (s. AP2, Empfehlung 15).

9.1.4 Erzeugung schwacher DSA-Schlüssel

Die Schlüsselerzeugung von DSA-Schlüsseln wird, sofern kein expliziter Seed angegeben wird, ein Seed mittels RAND_pseudo_bytes() erzeugt (s. [AP2], Bemerkung 25). Diese Funktion garantiert

allerdings nicht ausreichend Entropie (s. [AP2], Abs. 1.3.1.1). Für die Schlüsselerzeugung muss immer RAND_bytes() korrekt genutzt werden um ausreichend Entropie zu garantieren (s. AP2, Empfehlung 25).

9.2 Schutzmaßnahmen für die Schlüsselerzeugung

Die für die Schlüsselerzeugung genutzte Funktion für die Primzahlerzeugung in OpenSSL garantiert mit der Anzahl der Iterationen des Miller-Rabin-Primzahltest lediglich eine Wahrscheinlichkeit von ungefähr $1-2^{-80}$ dafür, dass die Zahl tatsächlich prim ist (s. AP3.3, Bemerkung 3). Für langfristige Schlüssel oder Schlüssel, bei denen die Erzeugungszeit nicht relevant ist, sollte die Anzahl von Iterationen erhöht werden (s. AP3.3, Empfehlung 3).

9.3 Schutzmaßnahmen für die Entschlüsselung mit RSA-PKCS#1

Im AP 3.6 (Abschnitt Nicht Konstante PKCS#1 Bearbeitung) wurde festgestellt, dass die PKCS#1 Verifikation nicht zeitkonstant durchgeführt wird und dass Unterschiede von 200 Ticks sichtbar sind. Obwohl dies zu keinen praktischen Timing-basierten Bleichenbacher-Angriffen führen kann (der Unterschied ist zu klein), können andere Sicherheitsprobleme auftreten. Wie Zhang et al. gezeigt haben [ZJRR], kann dieses Verhalten zu Flush-Reload Side-Channels führen, z.B. in PaaS-basierten Umgebungen.

In der OpenSSL 1.0.1j Version wurden einige dieser Probleme behoben und die PKCS#1 Validierung zeitkonstant neu implementiert. Damit sollten Timing-basierte Side-Channels beseitigt werden.

9.4 Generierung von Ephemeral Schlüsseln in DHE und ECDHE Ciphersuites

Im AP 3.6 (Abschnitte TLS-DH: Kleine Untergruppen und Elliptische Kurven) wurde analysiert, dass standardmäßig auf dem Server jeweils ein Ephemeral Schlüssel für ECDHE und DHE Ciphersuites erzeugt wird. Diese Schlüssel werden nicht erneuert, sie werden eingesetzt bis der Server gestoppt wird.

Dieses Verhalten kann man explizit im SSL-Context anpassen, mit den Befehlen:

SSL_CTX_set_options(ctx, SSL_OP_SINGLE_DH_USE);

SSL_CTX_set_options(ctx, SSL_OP_SINGLE_ECDH_USE);

Wir empfehlen jedoch, dass diese Optionen standardmäßig aktiviert werden, so dass pro Verbindung immer ein neuer Ephemeral Schlüssel generiert wird.

9.5 Unterstützung von SSLv3

Standardmäßig unterstützt die untersuchte OpenSSL-Version SSLv3 oder unsichere Ciphersuites.

Deswegen empfehlen wir den OpenSSL-Code mit den im AP 3.2 empfohlenen Flags zu kompilieren, oder einen kompilierten OpenSSL-Server mit sicheren Ciphersuites und Protokoll-Versionen zu konfigurieren, wie im AP 3.7 beschrieben wurde.

Bibliographie

[ANSIX9.26] American Bankers Association: „Sign-On Authentication for Wholesel Financial Tranaction“, American National Standard for Financial Institutions, 1990

[AP2] W. Meyer zu Bergsten, R. Korthaus: „Quellcode-basierte Untersuchung von kryptographisch relevanten Aspekten der OpenSSL-Bibliothek“, Arbeitspaket 2: Random Number Generator, Version 1.0, September 2014

[apachehttpd] Apache httpd. http://httpd.apache.org/

[ASK05] Onur Acıiçmez, Werner Schindler, Çetin K. Koç: „Improving Brumley and Boneh Timing Attack on Unprotected SSL Implementations“. ACM CCS 2005. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=DCD55885759798FF35E87947FEEC7E58?doi=10.1.1.63.3510&rep=rep1&type=pdf

[BB2003] D. Boneh, D. Brumley: „Remote timing attacks are practical“, Proceedings of the 12th Usenix Security Symposium, 2003

[BBPV2012] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren: Practical realisation and elimination of an ECC-related software bug attack. CTRSA, 2012

[BDFPS14] Bhargavan, Delignat-Lavaud, Fournet, Pironti, Strub: „Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS“, Security and Privacy 2014, http://prosecco.gforge.inria.fr/personal/karthik/pubs/triple-handshakes-and-cookie-cutters-sp14.pdf

[BEAST] Thai Duong, Juliano Rizzo: Here Come The ⊕ Ninjas. http://packetstormsecurity.com/files/download/105499/Beast-SSL.rar

[BDPLR] K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, M. Ray: „Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension“, http://tools.ietf.org/html/draft-bhargavan-tls-session-hash-00

[Blei] Daniel Bleichenbacher: „Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1“, Crypto,1998

[BnetzA] Bundesnetzagentur für Elektrizität, Gas, Telekommunikation, Post und Eisenbahnen: „Bekanntmachung zur elektronischen Signatur nach dem Signaturgesetz und der Signaturverordnung (Übersicht über geeignete Algorithmen)“, vom 13.1.2014

[BPSSL2014] Ivan Ristić: „Bulletproof SSL and TLS“, Feisty Duck, 16. August 2014

[BR93] Mihir Bellare and Phillip Rogaway: „Entity authentication and key distribution“, in Douglas R. Stinson, editor, Advances in Cryptology – CRYPTO’93

[BREACH] Yoel Gluck, Neal Harris, Angelo Prado, BREACH. SSL, „Gone in 30 Seconds“, http://breachattack.com/, http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf

[BT2011] B. B. Brumley,N. Tuveri: „Remote Timing Attacks are Still Practical“, Cryptology ePrint Archive: Report 2011/232, Mai 2011

[CAB] CA/Browser Forum: „Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates“, v.1.2.3

[COMBA] P. G. Comba: „Exponentiation cryptosystems on the IBM PC“, IBM Systems Journal, Vol. 29, No. 4, December 1990

[CK01] R. Canetti and H. Krawczyk: „Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels“, Advances in Cryptology - EUROCRYPT ’01

[CP2005] C. Percival: „CACHE MISSING FOR FUN AND PROFIT“, Proceedings of BSDCan 2005

[CRIME-BS13] Tal Be'ery, Amichai Shulman. „A perfect CRIME? Only TIME will tell“, Blackhat 2013, https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf

[CRIMEimp] Adam Langley: „CRIME“, https://www.imperialviolet.org/2012/09/21/crime.html

[CRIME-koto] Krzysztof Kotowicz, „If it's CRIME, then I'm guilty“, http://blog.kotowicz.net/2012/09/if-its-crime-than-im-guilty.html

[EarlyCCSadv] OpenSSL Security Advisory, „Early CCS“, https://www.openssl.org/news/secadv_20140605.txt, 5. Juni 2014

[EarlyCCSimp] Adam Langley, „Early ChangeCipherSpec Attack“, https://www.imperialviolet.org/2014/06/05/earlyccs.html

[EarlyCCSLep] Masashi Kikuchi: „How I discovered CCS Injection Vulnerability (CVE-2014-0224)“, http://ccsinjection.lepidum.co.jp/blog/2014-06-05/CCS-Injection-en/index.html

[FAUT] S. Schinzel, I. Schmitt: „FAU-Timer“, https://code.google.com/p/fau-timer/, 2013

[FIPS186-1] U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology: „DIGITAL SIGNATURE STANDARD (DSS)“, 15. Dezember 1998

[FLAGS] https://www.openssl.org/docs/crypto/X509_VERIFY_PARAM_set_flags.html

[GCM] U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology: „Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC“, November 2007

[GKS13] Florian Giesen, Florian Kohlar, and Douglas Stebila: „ On the security of TLS renegotiation“,. CCS '13, 2013

[HAC] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone: „Handbook of Applied Cryptography“, Fifth Printing, 2001, ISBN 0-8493-8523-7

[Heartbleed] „The Heartbleed Bug“, http://heartbleed.com/

[HeartbleedAdv] OpenSSL Security Advisory:“Heartbleed,“ https://www.openssl.org/news/secadv_20140407.txt,7. April 2014

[HeartbleedCloud] Nick Sullivan: „Heartache and Heartbleed: The insider's perspective on the aftermath of Heartbleed“, 2014, http://events.ccc.de/congress/2014/Fahrplan/events/6212.html

[HeartbeatRFC] R. Seggelmann, M. Tuexen, M. Williams: „Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension“, 2012

[HOST] https://www.openssl.org/docs/crypto/X509_check_host.html

[IANAECC] Internet Assigned Numbers Authority: „Transport Layer Security (TLS) Parameters: EC Named Curve Registry“, http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-8, vom 8.12.2014

[JKSS12] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk: „On the security of TLS-DHE in the standard model. In Reihaneh Safavi-Naini and Ran Canetti, editors“, Advances in Cryptology – CRYPTO 2012

[KKO63] A. Karatsuba, YU. Ofman , „Multiplication of multidigit numbers on automata“, Soviet Physics – Doklady, 7 (1963), 595–596.

[KR02] Vlastimil Klíma, Tomáš Rosa: „Attack on Private Signature Keys of the OpenPGP format, PGP programs and other applications compatible with OpenPGP“, 2002, https://eprint.iacr.org/2002/076.pdf

[LMSS2014] M. Lochter, J. Merkle, J.-M. Schmidt, T. Schütze: „Requirements for Standard Elliptic Curve“, Oktober 2014, http://eprint.iacr.org/2014/832

[Lucky13] N.J. AlFardan and K.G. Paterson: „Lucky Thirteen: Breaking the TLS and DTLS Record Protocols“, IEEE Security and Privacy, 2013, http://www.isg.rhul.ac.uk/tls/Lucky13.html

[Lucky13imp] Adam Langley: „Lucky Thirteen attack on TLS CBC“, 2013, https://www.imperialviolet.org/2013/02/04/luckythirteen.html

[MALSHA1] A. Albertini, J.-P. Aumasson, M. Eichlseder, F. Mendel, M. Schläffer: „Malicious Hashing: Eve's Variant of SHA-1“, Cryptology ePrint Archive: Report 2014/694, 03. September 2014, http://eprint.iacr.org/2014/694
FAQ: http://malicioussha1.github.io/#faq

[mitmAdv] OpenSSL Security Advisory [07-Jan-2009]: „Incorrect checks for malformed signatures“, https://www.openssl.org/news/secadv_20090107.txt

[MS14] Daniel A. Mayer, Joel Sandin: „Time Trial - Racing Towards Practical Remote Timing Attacks“, Blackhat 2014

[MSWS] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, and Jörg Schwenk, Sebastian Schinzel, Erik Tews: „Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks“, USENIX Security ,2014

[NCsupport] Tom Leek: „OpenSSL ECC named curve support“, 2013, http://security.stackexchange.com/questions/43992/openssl-ecc-named-curve-support

[NETSEC] John Viega, Matt Messier, Pravir Chandra: „Network Security with OpenSSL“, Frist Edition, 2002, ISBN 0-596-00270-X

[nginx] Nginx, http://nginx.org/

[openssh] OpenSSH. http://www.openssh.com/

[OpenSSLDH] OpenSSL Dokumentation zu DH. https://www.openssl.org/docs/ssl/SSL_CTX_set_tmp_dh_callback.html

[P1OVECT1] PKCS#1 Test Vectors, Original: ftp://ftp.rsa.com/pkcs-1v2/p1ovect1.txt (offline), available at ftp://ftp.sunet.se/pub/security/pca/docs/PKCS/ftp.rsa.com/pkcs-1v2/p1ovect1.txt

[PCK1996] Paul C. Kocher: „Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems“, Advances in Cryptology - CRYPTO ’96

[PKCS1] B. Kaldinski, J. Staddon: „PKCS #1: RSA Cryptography Specifications“, Version 2.0, October 1998, available at https://www.ietf.org/rfc/rfc2437.txt

[PKI] Carlisle Adams, Steve Lloyd: „Understanding PKI“, Second Edition, 2003, ISBN 0-672-32391-5

[POODLE] Bodo Möller, Thai Duong, Krzysztof Kotowicz: „This POODLE bites: Exploiting the SSLv3.0 Fallback“, 2014, https://www.openssl.org/~bodo/ssl-poodle.pdf

[POODLEadv] OpenSSL Security Advisory, „POODLE und SSLv3.0 Fallback“, https://www.openssl.org/news/secadv_20141015.txt, 15. Oktober 2014

[RenegAdv] OpenSSL Security Advisory: „TLS Renegotiation Attack“, https://www.openssl.org/news/secadv_20091111.txt, 11. November 2009

[RFC2785] R. Zuccherato: Methods for Avoiding the "Small-Subgroup" Attacks on the Diffie-Hellman Key Agreement Method for S/MIME, https://www.ietf.org/rfc/rfc2785.txt

[RFC2785] Internet Engineering Task Force: RFC2785: „Methods for Avoiding the "Small-Subgroup" Attacks on the Diffie-Hellman Key Agreement Method for S/MIME“, http://tools.ietf.org/html/rfc2785, März 2000

[RFC3280] Internet Engineering Task Force: RFC3280: „Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile“, http://tools.ietf.org/html/rfc3280, April 2001

[RFC3447] Internet Engineering Task Force: RFC3447: „Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1“, http://tools.ietf.org/html/rfc3447, Februar 2003

[RFC3779] Internet Engineering Task Force: RFC3779: „X.509 Extensions for IP Addresses and AS Identifiers“, http://tools.ietf.org/html/rfc3779, Juni 2004

[RFC4419] Internet Engineering Task Force: RFC4419: „Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol“, http://tools.ietf.org/html/rfc4419, März 2006

[RFC5246] Internet Engineering Task Force: RFC5246: „The Transport Layer Security (TLS) Protocol Version 1.2“, https://tools.ietf.org/html/rfc5246, August 2008

[RFC5639] Internet Engineering Task Force: RFC5639: „Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation“, https://tools.ietf.org/html/rfc5639, März 2010

[RFC5580] Internet Engineering Task Force: RFC5580: „Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile“, http://tools.ietf.org/html/rfc5280, März 2008

[RFC5746] Internet Engineering Task Force: RFC5746: „Transport Layer Security (TLS) Renegotiation Indication Extension“, http://tools.ietf.org/html/rfc5746, Februar 2010

[RFC6125] Internet Engineering Task Force: RFC6125: „Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)“, http://tools.ietf.org/html/rfc6125, März 2011

[RSAblAdv] OpenSSL Security Advisory [17 March 2003], RSA Blinding, https://www.openssl.org/news/secadv_20030317.txt

[RSAblinding] RSA Blinding patch and a recent snapshot, OpenSSL mailing list, http://comments.gmane.org/gmane.comp.encryption.openssl.devel/4919

[RS2000] Jean-Francois Raymond, Anton Stiglic: „Security Issues in the Diffie-Hellman Key Agreement Protocol“, Dezember 2000

[RTOpenSSL] Issue Tracking System for OpenSSL, 2014, http://rt.openssl.org/Ticket/Display.html?id=3180&user=guest&pass=guest

[RTOpenSSL] Issue Tracking System for OpenSSL, 2014, http://rt.openssl.org/Ticket/Display.html?id=3180&user=guest&pass=guest

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage: Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds, CCS '09

[SAFEECC] D.J. Bernstein: „SafeCurves: choosing safe curves for elliptic-curve cryptography“, http://safecurves.cr.yp.to/, vom 19. Januar 2014

[SSL3] A. Frier, P. Karlton, P. Kocher: "The SSL 3.0 Protocol", Netscape Communications Corp., Nov 18, 1996.

[sslyze] SSLyze, https://github.com/iSECPartners/sslyze

[testsslsh] testssl.sh: Testing SSL/TLS, https://testssl.sh/

[TR021021] BSI Technische Richtline BSI TR-02102-1: „Kryptografische Verfahren: Empfehlungen und Schlüssellängen“, Version 2014-01 vom 10.2.2014

[TR021022] BSI Technische Richtline BSI TR-02102-2: „Verwendung von Transport Layer Security (TLS)“, Version 2014-01

[TRUST] https://www.openssl.org/docs/apps/x509.html#TRUST_SETTINGS

[VFY] https://www.openssl.org/docs/ssl/SSL_CTX_set_verify.html

[VFY-CERT] https://www.openssl.org/docs/crypto/X509_verify_cert.html

[x509test] https://github.com/yymax/x509test

[ZDNET] http://www.zdnet.com/68-percent-of-top-free-android-apps-vulnerable-to-cyberattack-researchers-claim-7000032875/

[ZJRR] Y. Zhang, A. Juels, M.K. Reiter, T. Ristenpart: „Cross-Tenant Side-Channel Attacks in PaaS Clouds“, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security

[ZJRR12] Yinqian Zhang, Ari Juels, Mike Reiter, and Thomas Ristenpart: Cross-VM Side Channels and Their Use to Extract Private Keys, CCS '12