Glückliche Dreizehn: Knacken der TLS- und DTLS-Aufzeichnungsprotokolle

Nadhem J. AlFardan und Kenneth G. Paterson
Gruppe für
Informationssicherheit, Royal
Holloway, Universität von London
Egham, Surrey TW20 0EX, Vereinigtes Königreich
Email: {nadhem.alfardan.2009, kenny.paterson}@rhul.ac.uk

***Zusammenfassung - Das* Transport Layer Security (TLS)-Protokoll soll die Vertraulichkeit und Integrität von Daten bei der Übertragung über nicht vertrauenswürdige Netze gewährleisten. TLS hat sich als sicheres Protokoll für das Internet und mobile Anwendungen durchgesetzt. DTLS ist eine Variante von TLS, die zunehmend an Bedeutung gewinnt. In diesem Beitrag werden Angriffe zur Unterscheidung und Wiederherstellung von Klartext gegen TLS und DTLS vorgestellt. Die Angriffe basieren auf einer genauen Zeitanalyse der Entschlüsselungsverarbeitung in den beiden Protokollen. Wir fügen experimentelle Ergebnisse bei, die die Durchführbarkeit der Angriffe in realistischen Netzwerkumgebungen für verschiedene Implementierungen von TLS und DTLS, einschließlich der führenden OpenSSL-Implementierungen, zeigen. Wir stellen Gegenmaßnahmen für die Angriffe vor. Schließlich diskutieren wir die weiteren Auswirkungen unserer Angriffe auf das kryptographische Design von TLS und DTLS.**

***Schlüsselwörter: TLS*, DTLS, CBC-Verschlüsselung, Timing-Angriff, Klartext-Wiederherstellung**

I. Einführung

TLS ist heute das wohl am weitesten verbreitete sichere Kommunikationsprotokoll im Internet. Das Protokoll begann als SSL und wurde von der IETF angenommen und als TLS 1.0 [7] spezi- fiziert. Seitdem hat es sich über TLS 1.1 [8] zur aktuellen Version TLS 1.2 [9] weiterentwickelt. Verschiedene andere RFCs definieren zusätzliche kryptographische Algorithmen und Erweiterungen von TLS. TLS wird heute für die Absicherung einer Vielzahl von Datenübertragungen auf Anwendungsebene verwendet und hat sich zu einem ernsthaften Konkurrenten von IPsec für die allgemeine VPN-Nutzung entwickelt. Es wird von Client- und Serversoftware sowie von kryptografischen Bibliotheken für eingebettete Systeme, mobile Geräte und Webanwendungs-Frameworks weitgehend unterstützt.

Das DTLS-Protokoll ist ein enger Verwandter von TLS, das mit minimalen Änderungen so weiterentwickelt wurde, dass es über UDP anstelle von TCP funktioniert [16]. Dadurch eignet sich DTLS für Anwendungen, bei denen die Kosten des TCP-Verbindungsaufbaus und der TCP-Wiederholungsübertragungen nicht gerechtfertigt sind, z. B. bei Sprach- und Spieleanwendungen. DTLS gibt es in zwei Versionen, DTLS 1.0 [20], das in etwa mit TLS 1.1 und DTLS 1.2 [21], das mit TLS 1.2 übereinstimmt.

Sowohl TLS als auch DTLS sind eigentlich Protokollsuiten und keine Einzelprotokolle. Die Hauptkomponente von (D)TLS, um die es hier geht, ist das Record-Protokoll, das sym- metrische Schlüsselkryptographie (Blockchiffren, Stromchiffren und MAC-Algorithmen) in Kombination mit Sequenznummern verwendet

Die Forschung des zweiten Autors wurde durch ein EPSRC Leadership Fellowship, EP/H005455/1, unterstützt.

Komponenten sind das (D)TLS Handshake-Protokoll, das für die Authentifizierung, den Aufbau des Sitzungsschlüssels und die Aushandlung der Chiffrierung zuständig ist, sowie das TLS Alert-Protokoll, das Fehlermeldungen und Verwaltungsdaten übermittelt. Abgesehen von speziellen authentifizierten Verschlüsselungsalgorithmen (die in TLS- oder DTLS-Implementierungen noch keine breite Unterstützung finden), verwendet das (D)TLS Record Protocol eine MAC-Encode-Encrypt (MEE)-Konstruktion. Dabei werden die zu transportierenden Klartextdaten (zusammen mit bestimmten Header-Bytes) zunächst durch einen MAC-Algorithmus geleitet, um ein MAC-Tag zu erzeugen. Die unterstützten MAC-Algorithmen basieren alle auf HMAC, wobei MD5, SHA-1 und SHA-256 die zulässigen Hash-Algorithmen in TLS 1.2 [9] sind. Dann erfolgt ein Verschlüsselungsschritt. Bei der RC4-Stromchiffre werden lediglich der Klartext und das MAC-Tag verkettet, während bei der CBC-Verschlüsselung (der anderen möglichen Option) der Klartext, das MAC-Tag und ein Verschlüsselungs-Padding in einem bestimmten Format verkettet werden. Im Verschlüsselungsschritt wird der verschlüsselte Klartext mit der gewählten Chiffre

1081-6011/13 $26.00 © 2013 IEEE
DOI 10.1109/SP.2013.42

wird, ist die DES in TLS näher erläutert.

Die weit verbreitete Verwendung von TLS (und die zunehmende Verwendung von DTLS) macht die fortgesetzte Untersuchung der Sicherheit dieser Protokolle von großer Bedeutung. In der Tat wurde die Entwicklung des TLS-Protokolls weitgehend durch kryptografische Angriffe vorangetrieben, die gegen dieses Protokoll entdeckt wurden, darunter die in [25], [5], [17], [3], [4], [10], [18], [1].

Von besonderem Interesse waren in letzter Zeit Angriffe, die auf der Verwendung von verketteten Initialisierungsvektoren (IVs) für den CBC-Modus in SSL und TLS 1.0 beruhen, insbesondere der so genannte BEAST-Angriff [10], der seine Wurzeln in [23], [17], [3], [4] hat. Mit diesem Angriff wurde die vollständige Wiederherstellung des Klartextes gegen TLS erreicht, allerdings nur in Szenarien, in denen ein Angreifer Zugang zu einer ausgewählten Klartextfähigkeit erlangen kann, etwa indem er den Benutzer dazu bringt, zunächst bösartigen Javascript-Code in seinen Browser herunterzuladen. Trotz dieser strengen Anforderungen erregte der BEAST-Angriff im Jahr 2011 große Aufmerksamkeit in der Branche und in den Medien. Zu den möglichen Gegenmaßnahmen gehören die Aufrüstung auf TLS 1.1 oder 1.2, die Aufnahme einer Dummy-Nachricht mit Null-Länge vor der

jede echte TLS-Nachricht, oder der Verzicht auf die CBC-Verschlüsselung zugunsten von RC4 oder eines authentifizierten Verschlüsselungsalgorithmus.

Die andere wichtige Angriffslinie gegen das TLS-Record-Protokoll umfasst [25], [5], [17], [18], [1] und bezieht sich darauf, wie das in MEE-TLS-CBC erforderliche Padding bei der Entschlüsselung gehandhabt wird. Die Probleme hier rühren alle von der Tatsache her, dass das Padding *nach der* Berechnung des MAC hinzugefügt wird und somit unauthentifizierte Daten im verschlüsselten Klartext bildet. Insgesamt gesehen sind die Angriffe in [25], [5], [17], [18], [1] zeigen, dass der Umgang mit Auffüllungen, die während der Entschlüsselung entstehen, ein heikles und komplexes Thema für MEE-TLS-CBC ist.

All diese Angriffe auf den CBC-Modus in TLS könnten vermieden werden, indem RC4 oder ein spezieller authentifizierter Verschlüsselungsmodus verwendet wird, oder indem (D)TLS so umgestaltet wird, dass nur eine Encrypt-then-MAC-Konstruktion verwendet wird. RC4 ist jedoch keine Option für DTLS und wird von NIST nicht für TLS empfohlen [6]; authentifizierte Verschlüsselungsmodi sind nur in TLS 1.2 verfügbar, das noch nicht weithin unterstützt wird.[1] Die Neugestaltung von (D)TLS würde noch radikalere Änderungen erfordern als die Übernahme von TLS 1.2. Es wäre also phantastisch, MEE-TLS-CBC und die damit verbundene Komplexität "wegzuwünschen": Diese Option ist fest in den TLS- und DTLS-RFCs verankert, weit verbreitet und wird dies auch in absehbarer Zukunft bleiben. Andererseits könnten wir hoffen, dass wir nach mehr als einem Jahrzehnt intensiver Studien einen Punkt erreicht haben, an dem wir wissen, wie wir MEE-TLS-CBC sicher implementieren können. In diesem Papier zeigen wir, dass dies nicht der Fall ist.

A. Unsere Ergebnisse

Wir stellen eine Reihe von Angriffen vor, die auf den CBC-Modus in allen TLS- und DTLS-Implementierungen anwendbar sind, die mit TLS 1.1 oder 1.2 oder mit DTLS 1.0 oder 1.2 konform sind. Sie gelten auch für Implementierungen von SSL 3.0 und TLS 1.0, die Gegenmaßnahmen für Padding-Orakel-Angriffe enthalten (Implementierungen, die dies nicht tun, sind natürlich bereits anfällig für bekannte Angriffe).

Die Angriffe gibt es in verschiedenen Varianten: Unterscheidung, teilweise Klartext-Wiederherstellung und vollständige Klartext-Wiederherstellung. Für die Klartext-Wiederherstellungs-Angriffe ist im Gegensatz zu den BEAST-Angriffen keine chosen-plaintext-Fähigkeit erforderlich: Die Angriffe können von einem normalen Man-in-the-Middle (MITM)-Angreifer durchgeführt werden, der nur Chiffretext sieht und selbst zusammengestellte Chiffretexte in das Netz einspeisen kann. Welche Angriffe im Einzelnen möglich sind, hängt von der genauen Größe der MAC-Tags ab, die der vom Handshake-Protokoll ausgehandelte MAC-Algorithmus ausgibt, sowie von der Tatsache, dass genau 13 Byte Kopfdaten in die MAC-Berechnung einfließen (daher unser Titel).

1SSL Pulse (https://www.trustworthyinternet.org/ssl-pulse/) berichtet, dass im Januar 2013 nur 11,4 % von 200.000 befragten Websites TLS 1.2 unterstützen; die meisten gängigen Browser unterstützen TLS 1.2 derzeit nicht.

Die Anwendbarkeit der Angriffe ist auch von der Implementierung abhängig, da verschiedene Implementierungen die RFCs interpretieren. Wir haben mehrere verschiedene Open-Source-Implementierungen von TLS und DTLS untersucht und festgestellt, dass sie alle für unsere neuen Angriffe oder Varianten davon anfällig sind (in einem Fall sogar für alte Angriffe).

Wir haben eine Auswahl der Angriffe in einer experimentellen Umgebung implementiert. Wie bei früheren Angriffen ist das vollständige Brechen von TLS eine Herausforderung, da die Angriffe "gebrochene" TLS-Datensätze erzeugen und somit viele TLS-Sitzungen verbrauchen. Nichtsdestotrotz kann unser Basisangriff den gesamten Klartext für die aktuelle OpenSSL-Implementierung von TLS extrahieren, vorausgesetzt, der Angreifer befindet sich z. B. im selben LAN-Segment wie der anvisierte TLS-Client oder -Server. Dabei werden etwa 2^{23} TLS-Sitzungen benötigt, um einen Klartextblock in einem Angriffsszenario mit mehreren Sitzungen, wie es in [5] betrachtet wurde, zuverlässig wiederherzustellen. Ein solches Szenario ist z. B. möglich, wenn ein Anwendungsprotokoll eine automatische TLS-Wiederverbindung und eine erneute Übertragung des Passworts vornimmt. Angesichts seiner Komplexität scheint dieser grundlegende Angriff nur eine theoretische Bedrohung darzustellen. Varianten davon sind jedoch wesentlich effektiver:

- Die Unterscheidungsangriffe gegen TLS sind für OpenSSL recht praktisch und erfordern nur eine Handvoll Sitzungen, um die Verschlüsselungen ausgewählter Nachrichten zuverlässig zu unterscheiden.
- Das Brechen von DTLS-Implementierungen ist selbst für einen Angreifer aus der Ferne durchaus praktikabel, da wir die Tatsache ausnutzen können, dass DTLS-Fehler nicht fatal sind, um die Angriffe in einer einzigen Sitzung auszuführen, und wir können die Verstärkungstechniken aus [1] wiederverwenden, um die heiklen Zeitsignale zu verstärken, von denen unsere Angriffe abhängen.
- Wir haben auch effizientere Angriffe auf TLS und DTLS, um den Klartext *teilweise* wiederherzustellen. Bei OpenSSL TLS beispielsweise kann ein Angreifer, der ein Byte eines Blocks in einer der beiden letzten Byte-Positionen kennt, jedes der verbleibenden Bytes in diesem Block mit 2^{16} -Sitzungen zuverlässig wiederherstellen.
- Die Komplexität aller unserer Angriffe lässt sich mit Sprachmodelle und sequenzielle statistische Techniken wie in [5], [10]. Ein einfaches Beispiel: Wenn der Klartext base64-kodiert ist, wie es bei der HTTP-Basisauthentifizierung und bei Cookies der Fall ist, dann verringert sich die Anzahl der TLS-Sitzungen, die zur Wiederherstellung eines Blocks erforderlich sind, von etwa 2^{23} auf 2^{19} .
- In der Webumgebung können unsere Techniken kombiniert werden
  mit denen, die im BEAST-Angriff [10] verwendet werden: Client-seitige Malware, die im Browser läuft, kann verwendet werden, um alle benötigten TLS-Sitzungen zu initiieren, wobei ein HTTP-Cookie automatisch vom Browser an einer vorhersehbaren Stelle im Klartextstrom in jeder Sitzung injiziert wird. Die Malware kann auch die Position des Cookies kontrollieren, so dass in jeder Phase des Angriffs nur ein unbekanntes Byte im Zielblock vorhanden ist. Der Angreifer

kombiniert dann die "ein bekanntes Byte"-Variante unseres Angriffs mit der oben beschriebenen base64-Optimierung (unter der Annahme, dass der sensible Teil des Cookies base64-kodiert ist). Nimmt man all diese Verbesserungen zusammen, können HTTP-Cookies nach unserer Schätzung mit 2^{13} Sitzungen pro Cookie-Byte wiederhergestellt werden (wobei alle Sitzungen automatisch generiert werden). Beachten Sie, dass die Malware für diesen Angriff nicht in der Lage sein muss, ausgewählten Klartext in eine bestehende TLS-Sitzung zu injizieren.

B. *Wie die Angriffe funktionieren*

Unsere neuen Angriffe machen sich die Tatsache zunutze, dass bei der Entschlüsselung von schlecht gepolsterten Daten immer noch eine MAC-Prüfung an *einigen* Daten durchgeführt werden muss, um die bekannten Zeitangriffe zu verhindern. Aber welche Daten sollten für diese Berechnung verwendet werden? In den RFCs zu TLS 1.1 und 1.2 wird empfohlen, die MAC-Prüfung so durchzuführen, als gäbe es ein Pad mit der Länge Null. Wie in diesen RFCs vermerkt:

> *Damit verbleibt ein kleiner Zeitkanal, da die MAC-Leistung bis zu einem gewissen Grad von der Größe des Datenfragments abhängt, der jedoch aufgrund der großen Blockgröße der bestehenden MACs und der geringen Größe des Zeitsignals nicht groß genug sein dürfte, um ausgenutzt zu werden.*

Wir sind davon überzeugt, dass es tatsächlich kleine Zeitunterschiede gibt, die aber im Gegensatz zu dem, was in den RFCs steht, ausgenutzt werden *können*. Kurz gesagt, wenn verschiedene Faktoren wie die Größe der MAC-Tags, die Blockgröße der Blockchiffre und die Anzahl der Header-Bytes zufällig aufeinander abgestimmt sind, gibt es einen Zeitunterschied bei der Verarbeitung von TLS-Datensätzen mit gutem und schlechtem Padding, und dieser Unterschied zeigt sich in der Zeit, in der Fehlermeldungen im Netz erscheinen. Dieser Zeitseiten-Kanal kann dann durch eine sorgfältige statistische Analyse mehrerer Zeitproben dazu gebracht werden, Klartextdaten preiszugeben.

C. *Offenlegung*

Angesichts der großen Anzahl betroffener Implementierungen informierten wir im November 2012 zunächst die Vorsitzenden der IETF TLS Working Group, die IETF Security Area Directors und die Vorsitzenden der IRTF Crypto Forum Research Group (CFRG) über unsere Angriffe. Daraufhin begannen wir mit der Kontaktaufnahme zu einzelnen Anbietern. Patches zur Behebung unserer Angriffe wurden von OpenSSL, NSS, GnuTLS, PolarSSL, CyaSSL, MatrixSSL, Opera, F5, BouncyCastle und Oracle veröffentlicht. Weitere Einzelheiten finden Sie in der Vollversion [2].

D. *Weitere Details zu verwandten Arbeiten*

Padding-Orakel-Angriffe begannen mit Vaudenay [25], der zeigte, dass das Vorhandensein eines *Padding-Orakels*, d. h. eines Orakels, das einem Angreifer mitteilt, ob das Padding korrekt formatiert wurde oder nicht, zum Aufbau einer Entschlüsselungsfunktion genutzt werden kann. Canvel *et al.* [5] zeigten, dass ein solches Orakel für die damals aktuelle Version von OpenSSL wie folgt erhalten werden kann

Ausnutzung eines Zeitunterschieds bei der TLS-Entschlüsselungsverarbeitung. Im Wesentlichen wurde in OpenSSL keine MAC-Prüfung durchgeführt, wenn das Padding falsch formatiert war, während die MAC-Prüfung durchgeführt wurde, wenn das Padding korrekt war. Das wiederum bedeutete, dass im Fall von "ungültigem Padding" schneller eine Fehlermeldung erzeugt wurde als im Fall von "gültigem Padding". Das Padding-Orakel wurde also durch einen zeitlichen Seitenkanal aufgedeckt. Eine Komplikation bei der vollständigen Wiederherstellung des Klartextes besteht darin, dass bei TLS die entsprechenden Fehlermeldungen fatal sind und zur Beendigung der TLS-Sitzung führen. Um dieses Problem zu lösen, betrachteten Canvel *et al.* die Multisession-Einstellung, bei der davon ausgegangen wird, dass derselbe Klartext in vielen Sitzungen an der gleichen Stelle im Chiffretext übertragen wird. Moeller
In [17] wurde darauf hingewiesen, dass der *Verzicht auf die* Prüfung des Auffüllformats keine Option ist, da dies noch einfachere Angriffe ermöglicht. Die korrekte Lösung, wie sie in TLS 1.1 und TLS 1.2 befürwortet wird, besteht darin, das Padding-Format sorgfältig zu prüfen, eine einzige Fehlermeldung für Padding- und MAC-Fehler zu melden und die Verarbeitungszeit des Datensatzes im Wesentlichen gleich zu halten, unabhängig davon, ob das Padding korrekt ist oder nicht. Kürzlich haben wir in [1] gezeigt, dass die OpenSSL-Implementierung von DTLS die bekannten Gegenmaßnahmen für Angriffe nicht übernommen hat. Wir haben auch neuartige Timing-Amplifikationstechniken eingeführt, um vollständige Klartext-Wiederherstellungsangriffe gegen diese DTLS-Implementierung zu entwickeln, obwohl DTLS keine expliziten Fehlermeldungen zur Zeit hat.

Theoretische Unterstützung für die in (D)TLS verwendete MEE-Konstruktion findet sich in [12], [14], [18]. Insbesondere haben Paterson *et al.* [18] die ersten positiven Sicherheitsergebnisse für ein vollständig akkurates Modell MEE-TLS-CBC geliefert, das alle Details des CBC-Verschlüsselungsschritts (der Auffüllen beinhaltet) enthält. Sie haben bewiesen, dass MEE-TLS-CBC Sicherheit durch Length Hiding Authenticated-Encryption bietet, vorausgesetzt, dass seine MAC- und CBC-Blockchiffrierkomponenten natürliche Sicherheitseigenschaften erfüllen, dass die MAC-Tags lang genug sind und dass es so implementiert ist, dass die Entschlüsselung die Ursache von Fehlern nicht aufdeckt. Unsere Angriffe nutzen die Tatsache aus, dass aktuelle Implementierungen von (D)TLS diese letzte Annahme nicht erfüllen. Unsere Angriffe stehen also nicht im Widerspruch zu den Ergebnissen von [18], sondern relativieren deren Anwendbarkeit in der Praxis.

In einer unabhängigen Arbeit haben Pironti *et al.* [19] denselben Zeitkanal in TLS identifiziert, den wir ausnutzen. Sie weisen ihn jedoch als "zu klein, um über das Netzwerk gemessen zu werden" zurück und konzentrieren sich stattdessen darauf, ihn zur Gewinnung von Informationen über die Nachrichtenlänge zu nutzen. Der jüngste CRIME-Angriff nutzt die optionale Verwendung von Komprimierung in TLS in Kombination mit der Möglichkeit, den Klartext auszuwählen, um einen Angriff zur Wiederherstellung des Klartextes durchzuführen.

II. Das (D)TLS-Protokoll

Wir konzentrieren uns auf die kryptografische Funktionsweise der TLS- und DTLS-Record-Protokolle im CBC-Modus

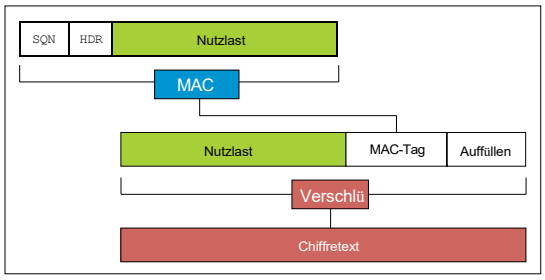

Abbildung 1. D(TLS)-Verschlüsselungsverfahren

Verschlüsselung. Der Kern des Verschlüsselungsprozesses ist in Abbildung 1 dargestellt und wird im Folgenden näher erläutert.

Die durch TLS oder DTLS zu schützenden Daten werden von der Anwendung empfangen und können vor der weiteren Verarbeitung fragmentiert und komprimiert werden. Ein einzelner Datensatz R (betrachtet als Bytefolge mit einer Länge von mindestens Null) wird dann wie folgt verarbeitet. Der Absender führt eine 8-Byte-Sequenznummer `SQN`, die für jeden gesendeten Datensatz[2] inkrementiert wird, und bildet ein 5-Byte-field `HDR`, das aus einem 2-Byte-Versions-field, einem 1-Byte-Typ-field und einem 2-Byte-Längen-field besteht. Dann berechnet es einen MAC über
die Bytes $SQN||HDR||R$; T bezeichne den resultierenden MAC
Tag. Beachten Sie, dass genau 13 Bytes an Daten dem
R hier aufzeichnen, bevor der MAC berechnet wird. Die Größe des MAC-Tags beträgt 16 Byte (HMAC-MD5), 20 Byte (HMAC- SHA-1) oder 32 Byte (HMAC-SHA-256). Wir lassen t diese Größe in Bytes bezeichnen.

Der Datensatz wird dann verschlüsselt, um den Klartext P zu erzeugen, indem $P = R||T$ `||pad` gesetzt wird. Dabei ist `pad` eine Folge von Auffüllungsbytes, die so gewählt wird, dass die Länge von P in Bytes
ein Vielfaches von b ist, wobei b die Blockgröße der gewählten Blockchiffre ist (also $b = 8$ für 3DES und $b = 16$ für AES). In allen Versionen von TLS und DTLS muss das Padding bestehen aus
$p + 1$ Kopien eines Byte-Wertes p, wobei $0 \leq p \leq 255$.
Insbesondere muss immer mindestens ein Byte Füllung vorhanden sein
hinzugefügt. Beispiele für gültige Bytefolgen für `Pad` sind: "0x00", "0x01||0x01" und "0x02||0x02||0x02". Das Auffüllen kann sich über mehrere Blöcke erstrecken, und die Empfänger müssen Folgendes unterstützen
die Beseitigung dieser erweiterten Polsterung.

Im Verschlüsselungsschritt wird der verschlüsselte Datensatz P mit dem CBC-Modus der gewählten Blockchiffre verschlüsselt. TLS 1.1 und
1.2 und beide Versionen von DTLS verlangen einen expliziten IV, der nach dem Zufallsprinzip erzeugt werden sollte. TLS 1.0 und SSL verwenden einen verketteten IV; unsere Angriffe funktionieren bei beiden Optionen. Die Chiffretextblöcke werden also wie folgt berechnet:

$$C_j = E_{K_e}(P_j \oplus C_{j-1})$$

wobei P_i die Blöcke von P , C_0 der IV und K_e der Schlüssel für die Blockchiffre E ist. Bei TLS (und SSL) sind die Daten

[2] Tatsächlich setzt sich dieses 8-Byte-Feld bei DTLS aus einer 16-Bit-Epochennummer und einer 48-Bit-Sequenznummer zusammen. Wir werden die Terminologie missbrauchen und das 8-Byte-field durchgehend als Sequenznummer für TLS und DTLS bezeichnen.

die über die Leitung übertragen wird, hat dann die Form:

$$HDR||C$$

wobei C die Verkettung der Chiffretextblöcke C_i ist (je nach SSL- oder TLS-Version einschließlich oder ausschließlich der IV). Beachten Sie, dass die Sequenznummer nicht als Teil der Nachricht übertragen wird. Bei DTLS sind die über die Leitung übertragenen Daten die gleichen wie bei TLS, mit der Ausnahme, dass die `SQN` als Teil des Datensatz-Headers enthalten ist und die CBC-Modus-IV immer explizit ist.

Vereinfacht gesagt, kehrt der Entschlüsselungsprozess diese Abfolge von Schritten um: Zunächst wird der verschlüsselte Text Block für Block entschlüsselt, um die Klartextblöcke wiederherzustellen:

$$P_j = D_{K_e}(C_j) \oplus C_{j-1},$$

wobei D für den Entschlüsselungsalgorithmus der Blockchiffre steht. Dann wird das Padding entfernt, und schließlich wird der MAC anhand der Header-Informationen (und bei TLS einer Version der Sequenznummer, die beim Empfänger aufbewahrt wird) überprüft. Schließlich wird bei DTLS die Sequenznummer optional auf Wiederholungen geprüft.

In Wirklichkeit ist eine weitaus anspruchsvollere Verarbeitung als diese erforderlich. Der Empfänger sollte prüfen, ob die Größe des Chiffriertextes ein Vielfaches der Blockgröße beträgt und groß genug ist, um mindestens einen Datensatz der Länge Null, ein MAC-Tag der erforderlichen Größe und mindestens ein Byte Padding zu enthalten. Nach der Entschlüsselung sollte der Empfänger prüfen, ob das Format des Paddings einem der möglichen Muster entspricht, wenn er es entfernt, da andernfalls Angriffe möglich sind [17] (SSL erlaubt ein loses Padding-Format, während in TLS 1.0 keine spezifischen Padding-Prüfungen während der Entschlüsselung erzwungen werden, so dass beide potenziell für die Angriffe in [17] anfällig sind). In der Regel wird dazu das letzte Byte des Klartextes untersucht, als Auffüllungslängenbyte `padlen` behandelt und verwendet, um festzulegen, wie viele zusätzliche Bytes der Auffüllung entfernt werden sollen. Hier ist jedoch Vorsicht geboten, da das blinde Entfernen von Bytes zu einer Unterversorgung führen könnte: Es müssen genügend Bytes im Klartext vorhanden sein, um insgesamt `padlen+1` Bytes zu entfernen und genügend Bytes für mindestens einen Datensatz der Länge Null und ein MAC-Tag zu hinterlassen.

Gelingt dies, so kann der MAC neu berechnet und mit dem MAC-Tag im Klartext verglichen werden. Wenn das Padding nicht korrekt formatiert ist, sollten Implementierungen trotzdem eine MAC-Prüfung durchführen, um zu vermeiden, dass ein Timing-Seitenkanal wie in [5] ausgenutzt wird. Da das Format der Auffüllungen in diesem Fall jedoch nicht korrekt ist, ist nicht sofort klar, wo die Auffüllungen enden und wo sich das MAC-Tag befindet: Der Klartext ist nun unparselbar. Die in TLS 1.1 und 1.2 (und damit auch in DTLS 1.0 und 1.2) empfohlene Lösung besteht darin, ein Padding der Länge Null anzunehmen, die letzten t Bytes des Klartextes als MAC-Tag zu interpretieren, den Rest als den Datensatz R zu interpretieren und

MAC-Verifikation auf $SQN||HDR||R$. Dies ist angenommen worden

in OpenSSL und anderswo; GnuTLS hingegen

entfernt `padlen` + 1 Byte vom Ende des Klartextes,

nimmt die nächsten t Bytes als MAC, interpretiert das, was übrig bleibt, als

R und führt dann eine MAC-Verifikation auf $SQN||HDR||R$ durch.

Bei TLS sollte jeder bei der Entschlüsselung auftretende Fehler

Dies bedeutet, dass eine verschlüsselte Fehlermeldung an den Absender gesendet und die Sitzung beendet wird, wobei alle Schlüssel und anderes kryptografisches Material entsorgt werden. Bei DTLS können solche Fehler als nicht schwerwiegend eingestuft werden, und die Sitzung würde mit der Verarbeitung des nächsten Chiffretextes fortgesetzt.

Es sollte nun klar sein, dass die Implementierung der grundlegenden Entschlüsselungsvorgänge von TLS und DTLS eine gewisse Sorgfalt bei der Umsetzung erfordert, da es viel Spielraum für Kodierungsfehler und unzureichendes Parsing gibt. Außerdem sollte dies alles so implementiert werden, dass die Verarbeitungszeit nichts über den Klartext (einschließlich der Füllbytes) verrät. Wie wir noch sehen werden, hat sich dies als Herausforderung für die Implementierer erwiesen: Keine der von uns untersuchten Implementierungen ist völlig korrekt, und die Empfehlung von TLS 1.1 und 1.2, das MAC-Tag so zu extrahieren und zu überprüfen, als ob das Padding null Länge hätte, lässt einen ausnutzbaren zeitlichen Seitenkanal.

A. Einzelheiten zu HMAC

Wie bereits erwähnt, verwenden TLS und DTLS ausschließlich den HMAC-Algorithmus [13], wobei in TLS 1.2 HMAC-MD5, HMAC-SHA-1 und HMAC-SHA-256 unterstützt werden.[3] Zur Berechnung des MAC-Tags T für eine Nachricht M mit dem Schlüssel K_a wendet HMAC den spezifizierten Hash-Algorithmus H zweimal iterativ an:

$$T = H((K_a \oplus \texttt{opad})||H((K_a \oplus \texttt{ipad})||M)).$$

Hier sind `opad` und `ipad` spezifische 64-Byte-Werte, und der Schlüssel K_a wird mit Nullen aufgefüllt, um ihn auf 64 Byte zu bringen, bevor die XOR-Operationen durchgeführt werden. Bei allen in TLS verwendeten Hash-Funktionen H *wird* bei der Anwendung von H selbst ein Verschlüsselungsschritt namens *Merkle-Damga˚rd-Verstärkung* verwendet. Dabei wird ein 8 Byte langes Feld, gefolgt von einer Auffüllung in einem bestimmten Byte-Format, an die zu hashende Nachricht M angehängt. Das Padding ist mindestens 1 Byte lang und richtet die Daten an einer 64-Byte-Grenze aus. Die entsprechenden Hash-Funktionen haben ebenfalls eine iterative Struktur, bei der die Nachrichten in Abschnitten von 64 Byte (512 Bit) unter Verwendung einer Komprimierungsfunktion verarbeitet werden, wobei die Ausgabe jedes Komprimierungsschritts mit dem nächsten Schritt verkettet wird. Die Komprimierungsfunktion beinhaltet wiederum eine komplexe Rundenstruktur, wobei in jeder Runde viele grundlegende arithmetische Operationen an den Daten durchgeführt werden.

In Kombination bedeuten diese Merkmale, dass HMAC-Implementierungen für MD5, SHA-1 und SHA-256 ein ausgeprägtes Zeitprofile aufweisen. Nachrichten M mit einer Länge von bis zu 55 Byte können in einem einzigen 64-Byte-Block kodiert werden, was bedeutet, dass die erste, innere Hash-Operation in HMAC in 2 Auswertungen der Kompressionsfunktion durchgeführt wird, wobei 2 weitere erforderlich sind

3TLS-Chiffren, die HMAC mit SHA-384 verwenden, sind in RFC 5289 spezifiziert.
(ECC cipher suites for SHA256/SHA384) und RFC 5487 (Pre-Shared Keys SHA384/AES), aber wir betrachten diesen Algorithmus hier nicht weiter.

für die äußere Hash-Operation, also insgesamt 4 Kompressionsfunktionsauswertungen. Nachrichten M mit 56 bis 64 + 55 = 119 Bytes können in zwei 64-Byte-Blöcken kodiert werden, was bedeutet, dass für die innere Hash-Operation 3 Kompressionsfunktionsauswertungen und für die äußere Operation 2 weitere, also insgesamt 5, erforderlich sind. Im Allgemeinen ist für jede zusätzlichen 64 Bytes eine zusätzliche Auswertung der Kompressionsfunktion erforderlich.
der Nachrichtendaten, wobei die genaue Anzahl durch die Formel $I^{\pounds^{-55}}$ / + 4 bestimmt wird, wobei .e die Nachrichtenlänge in Bytes ist. Eine einzelne Auswertung einer Kompressionsfunktion dauert
in der Regel etwa 500 bis 1000 Hardware-Zyklen (je nach Hash-Funktion und Einzelheiten der Implementierung), was bei modernen Prozessoren eine Zeit im sub-µs-Bereich ergibt.

Es sei daran erinnert, dass bei TLS der MAC auf dem Klartext *nach* Entfernung der Füllung berechnet wird. Daher könnte man erwarten, dass die Gesamtlaufzeit für die Entschlüsselungsverarbeitung einige Informationen über die Größe des entpackten Klartextes preisgibt, vielleicht bis zu einer Auflösung von 64 Byte angesichts der obigen Diskussion. Unser unterscheidender Angriff nutzt dies aus, aber wir werden zeigen, dass noch viel mehr möglich ist.

III. Ein Angriff mit Unterscheidungskraft

In diesem Abschnitt beschreiben wir einen einfachen Unterscheidungsangriff gegen die MEE-TLS-CBC-Konstruktion, wie sie in TLS verwendet wird. Dies ist eine Vorstufe zu unseren Klartext-Wiederherstellungsangriffen, aber wir stellen fest, dass selbst ein Unterscheidungsangriff gegen ein so wichtiges Protokoll normalerweise als eine bedeutende Schwäche angesehen wird.

Bei einem Unterscheidungsangriff wählt der Angreifer ein Nachrichtenpaar aus (M_0 , M_1). Eine davon wird verschlüsselt, z. B. M_d , und der daraus resultierende Chiffretext wird dem Angreifer übergeben. Die Aufgabe des Angreifers ist es, den Wert des Bits d zu bestimmen. Um zu verhindern, dass der Angreifer auf triviale Weise gewinnt, haben wir
erfordern, dass M_0 und M_1 die gleiche Länge haben.

Wir konzentrieren uns auf den Fall, dass b = 16 ist, d. h. die Blockchiffre ist AES. Eine Variante des Angriffs funktioniert für b = 8. Angenommen, der MAC-Algorithmus ist *HMAC-H*, wobei H entweder MD5, SHA-1 oder SHA-256 ist. M_0 bestehe aus 32 willkürlichen Bytes, gefolgt von 256 Kopien von 0xFF. M_1 bestehe aus 287 willkürlichen Bytes, gefolgt von 0x00. Man beachte, dass beide Nachrichten 288 Bytes lang sind und somit genau 18 Klartextblöcke fit. Unser Angreifer
sendet das Paar (M_0 , M_1) zur Verschlüsselung und erhält einen MEE-TLS-CBC-Chiffretext $HDR||C$. C besteht nun aus einer CBC-Verschlüsselung einer verschlüsselten Version von M_d , wobei
der Kodierungsschritt fügt ein MAC-Tag T und einige Auffüllungen hinzu
`pad`. Da das Ende von M_d mit einer Blockgrenze übereinstimmt, werden die zusätzlichen Bytes T `||pad` in separaten Blöcken von M_d verschlüsselt. Der Angreifer bildet nun einen neuen Chiffretext $HDR||C^t$, in dem C^t die gleiche 16-Byte-IV wie C behält (wenn explizite IVs verwendet werden), schneidet aber den nicht-IV-Teil von C auf 288 Bytes ab. Dies hat zur Folge, dass die Blöcke von C die T `||pad` enthalten.

Nun reicht der Angreifer $HDR||C^t$ zur Entschlüsselung ein. Wenn der Datensatz, der C zugrunde liegt, M_0 war, dann ist der Klartext P^t

entsprechend C^t scheint mit dem gültigen 256-Byte-Padding-Muster 0xFF … zu enden. 0xFF. In diesem Fall werden alle diese Bytes entfernt, und die verbleibenden 32 Bytes des Klartextes werden als Kurznachricht und MAC-Tag interpretiert. Für
Wenn H beispielsweise SHA-1 ist, dann haben wir eine 12-Byte-Nachricht und ein 20-Byte-MAC-Tag. Die MAC-Verifizierung schlägt (mit überwältigender Wahrscheinlichkeit) fehl, und es wird eine Fehlermeldung zurückgegeben
für den Angreifer. Wenn der zugrundeliegende Datensatz M_1 war, dann ist P^t
scheint mit dem gültigen 1-Byte-Auffüllmuster 0x00 zu enden.
In diesem Fall wird ein einziges Byte entfernt, und die verbleibenden 287 Bytes Klartext werden als lange Nachricht und MAC-Tag interpretiert. Auch hier schlägt die MAC-Verifizierung fehl und der Angreifer erhält eine Fehlermeldung.

Wenn $d = 0$ ist, C also M_0 verschlüsselt, wird eine kurze Nachricht, die aus 13 Byte Header und höchstens 16 Byte Nachricht besteht (wenn der Hash-Algorithmus MD5 ist), durch den MAC-Algorithmus geleitet. Zur Berechnung des MAC
erfordert 4 Auswertungen der Kompressionsfunktion von H.
Wenn $d = 1$ ist, verschlüsselt $C\,M_1$, und eine lange Nachricht, die aus 13 Byte Header und mindestens 255 Byte Nachricht besteht, wird durch den MAC-Algorithmus geleitet. Die Berechnung des MAC erfordert dann mindestens 8 Evaluierungen der Kompressionsfunktion von H, mindestens 4 mehr als im Fall $d = 0$. Daher erwarten wir, dass die Zeit, die für die Erstellung der Fehlermeldung bei einem Entschlüsselungsfehler benötigt wird, bei $d = 1$ etwas größer ist als bei $d = 0$, in der Größenordnung von einigen μs bei einem modernen Prozessor. Dieser Zeitunterschied ermöglicht dann theoretisch einen Angriff auf die in TLS verwendete MEE-TLS-CBC-Konstruktion.

A. Praktische Überlegungen

Bei der Beschreibung des Angriffs haben wir die Zeit, die für das Entfernen des Paddings benötigt wird, ignoriert. Diese ist für die beiden zu verarbeitenden Nachrichten unterschiedlich, und der Unterschied ist entgegengesetzt zu dem bei der MAC-Prüfung, da die Entfernung des Paddings für M_0 länger dauert als für M_1 . Ebenso haben wir alle anderen Zeitunterschiede ignoriert, die bei anderen Verarbeitungsschritten auftreten könnten. Wie wir in Abschnitt V sehen werden, erweisen sich diese Unterschiede in der Praxis als kleiner als der MAC-Zeitunterschied.

Der Angriff nutzt die Anforderung der (D)TLS-RFCs aus, dass Implementierungen in der Lage sein müssen, Datensätze mit variabler Länge zu entschlüsseln, erfordert aber nicht, dass die Implementierungen tatsächlich Datensätze mit solchen Auffüllungen senden. Eine Angriffsvariante ist möglich, wenn nur Auffüllungen mit minimaler Länge unterstützt werden, allerdings mit einem kleineren Zeitsignal.

Bei TLS werden die Fehlermeldungen über das Netz gesendet, so dass sie vom Angreifer leicht entdeckt werden können. Diese Nachrichten unterliegen jedoch dem Netzwerk-Jitter, und dieser kann groß genug sein, um die Zeitdifferenz, die sich aus den 4 zusätzlichen Kompressionsfunktionsbewertungen ergibt, zu überdecken. Andererseits kann das Zeitsignal recht groß sein, wenn die kryptografische Verarbeitung in einer eingeschränkten Umgebung durchgeführt wird, z. B. auf einem 8- oder 16-Bit-Prozessor oder sogar auf

ein Smartphone. Darüber hinaus kann der Jitter erheblich reduziert werden, wenn der Angreifer als separater Prozess auf dem Rechner läuft, der die TLS-Entschlüsselung durchführt. Dies könnte in virtualisierten Umgebungen möglich sein, z. B. in einem Cloud-Szenario, wie es in [22] untersucht wurde. Der Angriff zerstört auch die TLS-Sitzung, da solche Fehler bei TLS fatal sind. Der Angriff kann über L Sitzungen hinweg wiederholt werden, wobei M_d in jeder Sitzung verschlüsselt wird und eine statistische Verarbeitung zur Extraktion des Zeitsignals verwendet wird.

Bei DTLS gibt es keine Fehlermeldungen, aber die Techniken von [1] können zur Lösung dieses Problems eingesetzt werden. Dort senden die Autoren ein Paket mit einem Chiffriertext C, dicht gefolgt von einer DTLS-Nachricht, wobei letztere immer eine Antwortnachricht provoziert. Ein Zeitunterschied, der durch die Entschlüsselung von C entsteht, zeigt sich dann als Unterschied in der Ankunftszeit der Antwortnachrichten. Die Signalverstärkungstechniken aus [1] können ebenfalls verwendet werden, um die Zeitdifferenz zu erhöhen - hier besteht die Idee darin, mehrere Pakete, die alle C enthalten, kurz hintereinander zu senden, um eine kumulative Zeitdifferenz zu erzeugen (da C jedes Mal auf dieselbe Weise verarbeitet wird).

Bei dem beschriebenen Angriff haben wir 288-Byte-Nachrichten verwendet. Dadurch wurde sichergestellt, dass nach dem Entfernen der Auffüllungen genügend Bytes übrig blieben, um Platz für eine Nachricht (möglicherweise mit der Länge Null) und ein MAC-Tag zu schaffen. Dies gewährleistet
dass C^t alle eventuell durchgeführten Plausibilitätsprüfungen besteht
während der Entschlüsselung. Diese Sicherheitsprüfungen können jedoch
in Varianten unseres Basisangriffs ausnutzbar. Wenn beispielsweise eine Implementierung feststellt, dass nach dem Depadding nicht mehr genügend Bytes für einen MAC übrig sind, kann sie die MAC-Verifizierung ganz auslassen, was zu einer größeren Zeitdifferenz führt.

Man beachte, dass der Angriff auch dann wie beschrieben funktionieren würde, wenn die in [11] für TLS spezifizierten trunkierten MACs verwendet würden, da die vollständige HMAC-H-Berechnung weiterhin durchgeführt wird, aber nur bestimmte Bytes des berechneten Tags mit Bytes des Klartextes verglichen werden.

Über die erfolgreiche Durchführung dieses Angriffs berichten wir in Abschnitt V.

IV. Angriffe zur Klartext-Wiederherstellung

A. Allgemeiner Ansatz

Wie wir im vorigen Abschnitt gesehen haben, hängt die Verarbeitungszeit für einen (D)TLS-Datensatz (und damit auch die Erscheinungszeit von Fehlermeldungen) von der Menge der Auffüllungen ab, die der Empfänger als im verschlüsselten Klartext enthalten interpretiert. Indem ein Angreifer jedoch einen Ziel-Chiffretextblock am Ende des verschlüsselten Datensatzes platziert, kann er dafür sorgen, dass der diesem Block entsprechende Klartextblock als Füllung interpretiert wird, und somit die Verarbeitungszeit von den Klartextbytes abhängig machen. Es scheint jedoch, dass große Mengen gültiger Auffüllungen erforderlich sind, um einen signifikanten Zeitunterschied zu erzeugen, und dies ist bei einem Klartext-Wiederherstellungsangriff schwer zu erreichen. Wir zeigen, dass dieses Hindernis für die Wiederherstellung des Klartextes unter bestimmten Umständen überwunden werden kann.

C^* sei ein beliebiger Chiffretextblock, dessen entsprechenden Klartext P^* der Angreifer wiederherstellen möchte. C^t sei der Chiffretextblock, der C^* vorausgeht. C^t könne der IV oder der letzte Block des vorausgehenden Chiffretextes sein, wenn C^* der
first Block eines Chiffretextes. Wir haben:

$$P^* = D_K(C^*) \oplus C^t.$$

Für einen beliebigen Block B von Klartext oder Chiffretext schreiben wir
$B = [B_0\, B_1 \ldots B_{b-1}]$, wobei B_i die Bytes von B bezeichnet.
Insbesondere gilt: $P^* = [P^*_0\ P^*_1 \ldots P^*_{b-1}]$.

Wie üblich gehen wir davon aus, dass der Angreifer in der Lage ist, die (D)TLS-geschützte Kommunikation abzuhören und beliebige Nachrichten in das Netz einzuschleusen. Für TLS oder DTLS mit deaktivierter Sequenznummernprüfung brauchen wir keine Möglichkeit, zu verhindern, dass Nachrichten ihr Ziel erreichen. Ebenso wenig benötigen wir die Möglichkeit, einen Klartext auszuwählen.

B. Vollständige Klartext-Wiederherstellung

Der Einfachheit halber nehmen wir im Folgenden an, dass die IVs im CBC-Modus explizit sind (wie in TLS 1.1, 1.2 und DTLS 1.0, 1.2). Außerdem nehmen wir an, dass $b = 16$ ist (unsere Blockchiffre ist also AES). Es ist einfach, Varianten unserer Angriffe für implizite IVs und für $b = 8$ zu konstruieren. Wir beginnen damit, nur TLS zu betrachten, wobei Details für DTLS folgen werden. Wir gehen auch davon aus, dass die TLS-Implementierung den Ratschlag in den RFCs TLS 1.1 und 1.2 befolgt, den MAC so zu prüfen, als ob es ein Pad der Länge Null gäbe, wenn das Padding falsch formatiert ist. Wir werden die Sicherheit anderer Implementierungsoptionen in Abschnitt VI untersuchen. Am wichtigsten ist, dass wir aus Gründen, die noch deutlich werden, vorerst von $t = 20$ ausgehen (so dass der MAC-Algorithmus HMAC- SHA-1 ist). Wir betrachten in Kürze $t = 16$ und $t = 32$ (HMAC-MD5 und HMAC-SHA-256).

Angreifer Block von 16 Bytes, und man betrachte die Entschlüsselung

eines $^{\text{Catt}}(\Delta)$ der Form
Chiffretextes

$$^{\text{Catt}}(\Delta) = HDR||C_0||C_1||C_2||C^t \oplus \Delta||C^*$$

wobei es 4 nicht-IV-Chiffretextblöcke gibt, der vorletzte Block $C^t \oplus \Delta$ ist eine XOR-maskierte Version von C^t und der letzte Block ist C^*. Der entsprechende 64-Byte-Klartext ist $P = P_1||P_2||P_3||P_4$, wobei

$$\begin{aligned} P_4 &= D_K(C^*) \oplus (C^t \oplus \Delta) \\ &= P^* \oplus \Delta. \end{aligned}$$

Beachten Sie, dass P_4 eng mit dem unbekannten Ziel-Klartextblock P^* verbunden ist. Wir betrachten 3

2) P_4 endet mit einem gültigen Auffüllmuster der Länge von mindestens 2 Byte: In diesem Fall werden mindestens 2 Byte des Auffüllmusters entfernt, und die nächsten 20 Byte werden als MAC-Tag T interpretiert. Damit verbleibt ein Datensatz R mit einer Länge von höchstens 42 Byte, was bedeutet, dass die MAC-Verification an einer Nachricht mit einer Länge von höchstens 55 Byte durchgeführt wird.
3) P_4 endet mit einem beliebigen anderen Byte-Muster: In diesem Fall entspricht das Byte-Muster nicht einem gültigen Padding. Gemäß der Vorschrift in TLS 1.1 und 1.2
RFCs wird der Klartext so behandelt, als ob er keine Bytes enthält

Die letzten 20 Bytes werden als MAC-Tag T interpretiert, und die verbleibenden 44 Bytes Klartext werden als Datensatz R verwendet. Die MAC-Verifizierung wird dann an einer 57-Byte-Nachricht durchgeführt.

In allen Fällen scheitert die MAC-Verifizierung (mit überwältigender Wahrscheinlichkeit) und es wird eine Fehlermeldung ausgegeben. Beachten Sie, dass die MAC-Verifizierung in Übereinstimmung mit der Diskussion in Abschnitt II-A in den Fällen 1 und 3 5 Evaluierungen der Kompressionsfunktion für SHA-1 erfordert, während in Fall 2 nur 4 Evaluierungen notwendig sind. Daher können wir hoffen, Fall 2 von den Fällen 1 und 3 zu unterscheiden, indem wir das Auftreten der Fehlermeldung im Netz zeitlich festlegen. Hier ist der Zeitunterschied der, der für eine einzige SHA-
1 Kompressionsfunktionsauswertung (im Vergleich zu 4 solchen Auswertungen in unserem unterscheidenden Angriff). Beachten Sie, dass die Größe des Headers (13 Byte) in Verbindung mit der Größe des MAC-Tags (20 Byte) entscheidend für die Erzeugung dieses besonderen Zeitverhaltens ist.

In Fall 2 wird angenommen, dass der Klartext keine besonderen
Struktur ist das wahrscheinlichste Auffüllungsmuster das der Länge 2, nämlich **0x01||0x01**, wobei alle längeren Auffüllungsmuster etwa 256 Mal unwahrscheinlicher sind. Wenn also die
Der Angreifer wählt eine Maske Δ so, dass er die
Fall 2, nachdem er $^{\text{Catt}}(\Delta)$ zur Entschlüsselung vorgelegt hat, kann er ableiten, dass P_4 mit **0x01||0x01** endet, und anhand der Gleichung
verschiedene Fälle, die
zwischen ihnen alle Möglichkeiten abdecken, was bei der Entschlüsselung von $^{\text{Catt}}(\Delta)$ passieren kann:

1) P_4 endet mit einem 0x00-Byte: In diesem Fall wird ein einziges Byte des Paddings entfernt, die nächsten 20 Bytes werden als MAC-Tag T interpretiert, und die verbleibenden 64 - 21 = 43 Bytes Klartext werden als Datensatz R genommen. MAC
Die Verifizierung wird dann an einer 13 + 43 = 56-Byte-Nachricht $SQN||HDR||R$ durchgeführt.

$P_4 = P^* \oplus \Delta$, können nun die letzten 2 Bytes von P^* wiederhergestellt werden. (In der Tat, durch Wiederholung des Angriffs mit einer Maske Δ^t, die ist

modified von Δ im drittletzten Byte, kann der Angreifer den Fall eines Auffüllmusters der Länge 2 leicht von allen längeren Mustern trennen).

Es bleibt die Frage: Wie löst der Angreifer Fall 2 aus, so dass er die letzten 2 Bytes von P^* extrahieren kann? Wir erinnern uns, dass der Angreifer die Freiheit hat, Δ zu wählen.

Folge von Chiffretexten $^{\text{Catt}}(\Delta)$ mit Werten von Δ, die über alle möglichen Werte in den letzten 2 Bytes Δ_{14} , Δ_{15} variieren, dann wird der Angreifer (im schlimmsten Fall) nach 2^{16} Versuchen mit Sicherheit einen Wert für Δ wählen, so dass $^{\text{Catt}}(\Delta)$ den Fall 2 auslöst.

Sobald die letzten 2 Bytes von P^* extrahiert worden sind, wird die

kann der Angreifer die verbleibenden Bytes von P^* auf effizientere Weise wiederherstellen, indem er von rechts nach links arbeitet. Diese Phase ist im Wesentlichen identisch mit dem ursprünglichen Padding-Orakel-Angriff von Vaudenay [25].

Um zum Beispiel das drittletzte Byte zu extrahieren, kann der Angreifer sein neues Wissen über die letzten beiden Bytes von P^* nutzen, um nun Δ_{14} , Δ_{15} so zu setzen, dass P_4 mit 0x02||0x02 endet. Dann kann er

erzeugt Kandidaten $^{Catt}(\Delta)$ wie zuvor, ändert aber Δ_{13} nur. Nach höchstens 2^8 Versuchen wird er einen Chiffretext produzieren

was wiederum in Fall 2 fällt, der zeigt, dass er es geschafft hat

einen Wert 0x02 im drittletzten Byte von $P_4 = P^* \oplus \Delta$ zu setzen. Daraus kann er $_{13}$ P^* wiederherstellen. Die Wiederherstellung jedes nachfolgenden Bytes in P^* erfordert höchstens 2^8 Versuche, was insgesamt $14 \cdot 2^8$ Versuche ergibt, um die Extraktion von P^* abzuschließen.

1) Praktische Überlegungen: In der Praxis gibt es bei TLS zwei schwerwiegende Komplikationen. Erstens wird die TLS-Sitzung zerstört, sobald der Angreifer seinen allerersten Angriffs-Chiffretext einreicht. Zweitens ist der Zeitunterschied zwischen den beiden Fällen sehr gering, so dass er wahrscheinlich durch Netzwerk-Jitter und andere Quellen des Zeitunterschieds verdeckt wird.

Das erste Problem kann bei TLS durch einen Multisession-Angriff überwunden werden, bei dem angenommen wird, dass derselbe Klartext über viele Sitzungen hinweg an derselben Stelle wiederholt wird (wie z. B. in [5]). Wir haben in einem solchen Fall Masken Δ verwendet

so dass keine weitere Anpassung des Angriffs erforderlich ist, um dieser Einstellung Rechnung zu tragen - natürlich ändern sich die Blöcke C^t und C^* für jede Sitzung.

Das zweite Problem kann in der gleichen Einstellung mit mehreren Sitzungen gelöst werden, indem der Angriff viele Male für jeden Δ-Wert wiederholt wird und dann eine statistische Verarbeitung der aufgezeichneten Zeiten durchgeführt wird, um abzuschätzen, welcher Wert von Δ am ehesten dem Fall 2 entspricht. In der Praxis hat sich gezeigt, dass ein einfacher Perzentil-Test (und sogar eine Mittelwertbildung) gut funktioniert - siehe Abschnitt V für weitere Einzelheiten. Unter der Annahme, dass für jeden Δ-Wert L Versuche verwendet werden, verbraucht der beschriebene Angriff etwa

$L \cdot 2^{16}$ Sitzungen, wobei ein Chiffretext $^{Catt}(\Delta)$ versucht wird in

jede Sitzung.

2) Effizientere Varianten: Die Angriffskomplexität kann erheblich reduziert werden, wenn man davon ausgeht, dass die Sprache, aus der die Klartexte gezogen werden, mit einer Markov-Kette von finiter Länge modelliert werden kann. Dies ist eine angemessene Annahme für natürliche Sprachen sowie für Protokollnachrichten der Anwendungsschicht wie HTML, XML usw. Dieses Modell kann verwendet werden, um die Auswahl von Klartext-Bytes in der Reihenfolge abnehmender Wahrscheinlichkeit zu steuern und daraus die Δ-Bytes zu bestimmen, die benötigt werden, um zu prüfen, ob eine Vermutung für die Klartext-Bytes zu einer gültigen Auffüllung führt oder nicht. Ähnliche Techniken wurden in [5] verwendet,

[10] in Kombination mit sequenziellen statistischen Verfahren, um die Komplexität der Wiederherstellung von

kennt den Wert des Bytes P^* . Dann setzt er den Startwert Wert von Δ so, dass $\Delta_{14} = P^*_{14} \oplus$ 0x01, so dass bei

$^{Catt}(\Delta)$ entschlüsselt wird, wird das vorletzte ^{14} bereits Byte von P_4

Klartexten mit geringer Entropie zu verringern. Beachten Sie, dass dieser Ansatz nicht gut funktioniert, wenn die optionale Komprimierung von TLS verwendet wird. Eine andere Möglichkeit ist, dass die Klartextbytes aus einem reduzierten Raum von Möglichkeiten gezogen werden. Bei der HTTP-Basisauthentifizierung sind Benutzername und Passwort beispielsweise Base64-kodiert, was bedeutet, dass jedes Byte des Klartextes nur 64 mögliche Werte hat. Ähnliche Beschränkungen gelten oft für die sensiblen Teile von HTTP-Cookies.

In einem verwandten Angriffsszenario kann der Angreifer, wenn er bereits eines der letzten beiden Bytes von P^* kennt, das andere Byte mit viel geringerer Komplexität als in unserer bisherigen Analyse wiederherstellen

nahelegen würde. Dies ist dann ein Klartext-Wiederherstellungsangriff mit teilweise bekanntem Klartext. Nehmen wir zum Beispiel an, der Angreifer

gleich 0x01 ist. Dann durchläuft er die 2^8 möglichen Werte für Δ_{15} und findet schließlich einen, bei dem die letzten beiden Bytes von P_4 gleich 0x01||0x01 sind, wodurch Fall 2 ausgelöst wird. Er kann dann
den Rest von P^* mit der gleichen Komplexität wiederherzustellen
wie zuvor. Insgesamt kann dieser Angriff, bei dem 15 Byte von
Klartext, bei dem 1 von 2 der letzten Bytes des Zielblocks bekannt sind, benötigt nur $15L \cdot 2^8$ Sitzungen, wobei L die Anzahl der Versuche ist, die für jeden Δ-Wert an jeder Byteposition verwendet werden.
Durch die Kombination der beiden Varianten kann dies noch weiter reduziert werden. Für base64-kodierten Klartext werden beispielsweise nur $15L \cdot 2^6$ Sitzungen benötigt, um einen Block zu entschlüsseln.

3) Die Kombination von Lucky 13 mit BEAST: Eine wesentliche Einschränkung unserer bisher beschriebenen Angriffe besteht darin, dass sie viele TLS-Sitzungen beanspruchen. Diese Einschränkung kann überwunden werden, indem unsere Angriffe mit Techniken aus dem BEAST-Angriff [10] kombiniert werden, um TLS-geschützte HTTP-Cookies anzugreifen.

Speziell im Zusammenhang mit einem Webbrowser, der über TLS mit einem Webserver kommuniziert, kann der Benutzer dazu gebracht werden, Malware von einer betrügerischen Website in seinen Browser herunterzuladen. Diese Malware, die vielleicht in Javascript implementiert ist, kann dann alle TLS-Sitzungen initiieren, die für unseren Angriff erforderlich sind, wobei der Browser automatisch das gezielte HTTP-Cookie an die erste HTTP-Anfrage des Browsers anhängt. Durch Anpassung der Länge dieser ersten HTTP-Anfrage kann die Malware außerdem sicherstellen, dass in jedem Ziel-Ciphertext-Block nur ein unbekanntes Byte HTTP-Cookie-Klartext enthalten ist. Dies ermöglicht es unserem Angreifer, die oben beschriebene Angriffsvariante auszuführen. Angenommen, der Zielteil von
der Cookie ist base64-kodiert, der Angriff verbraucht $L \cdot 2^6$ Sitzungen pro Byte des HTTP-Cookies. Wie wir in den folgenden Abschnitten erörtern werden
Wie in Abschnitt V näher erläutert, haben wir festgestellt, dass die Einstellung $L = 2^7$ in unserer Versuchsanordnung eine zuverlässige Wiederherstellung des Klartextes ermöglicht, d. h. ein Angriff, der HTTP-Cookies mit etwa 2^{13} Sitzungen pro unbekanntem Cookie-Byte wiederherstellt.

C. Klartext-Wiederherstellung für andere MAC-Algorithmen

Ein entscheidendes Merkmal unseres obigen Angriffs ist die Beziehung zwischen der Größe des in der MAC-Berechnung enthaltenen Headers (fixed bei $h = 13$ Byte), der MAC-Tag-Größe t und der Blockgröße b. Wenn TLS beispielsweise so konzipiert ist, dass $h = 12$ ist, dann zeigt eine ähnliche Fallanalyse wie zuvor, dass unser Chiffriertext $^{\text{Catt}}(\Delta)$ die Eigenschaft hätte, eine schnellere MAC-Verification zu haben, wenn P_4 auch mit dem einzelnen Byte 0x00 endet (dem gültigen Auffüllmuster der Länge 1). Dies würde einen verbesserten 2^8 Angriff gegen TLS mit CBC-Modus und HMAC-SHA-1 ermöglichen. In gewissem Sinne *ist* 13 *ein* Glücksfall, aber 12 wäre noch besser gewesen!

In ähnlicher Weise haben wir (weniger effiziente) Varianten unserer Angriffe für HMAC-MD5 und HMAC-SHA-256, bei denen die Tag-Größen t 16 bzw. 32 Byte betragen. Da t in diesem Fall ein Vielfaches von b ist, ist die Analyse weitgehend identisch für

beide Fälle, und wir betrachten nur HMAC-MD5 im Detail. Diesmal ist $^{Catt}(\Delta)$ so, dass wir in Fall 2 fallen (gültiges Auffüllen mit einer Nachricht von höchstens 55 Byte Größe, was
schnelle MAC-Verification) nur dann, wenn $P_4 = P^* \oplus \Delta$ endet mit
ein gültiges Padding der Länge 6 oder mehr. Ohne zusätzliche Informationen über P^* bräuchte der Angreifer (im schlimmsten Fall) 2^{48} Versuche, um das richtige Δ zu konstruieren, um diesen Fall auszulösen;
zu erkennen, dass er dies getan hat, wäre angesichts der großen Anzahl von möglichen Δ-Werten schwieriger. Dies ist kein attraktiver Angriff, insbesondere im Hinblick auf die oben erwähnten praktischen Erwägungen für TLS. Andererseits gibt es attraktive Angriffe mit teilweise bekanntem Klartext für HMAC-MD5 und HMAC-SHA-256. Zum Beispiel, wenn jede 5
von den letzten 6 Bytes von P^* bekannt sind, können wir die restlichen 11 Bytes mit $11L - 2^8$ Sitzungen wiederherstellen. Der Angriff kann
Der Klartext kann auch effizienter gemacht werden, wenn der Klartext eine niedrige Entropie hat, indem man Kandidaten für die letzten 6 Bytes von P^* in der Reihenfolge abnehmender Wahrscheinlichkeit ausprobiert und dann die restlichen Bytes von P^* wiederherstellt, sobald der richtige 6-Byte-Kandidat gefunden wurde. Dies wäre z. B. eine gute Option für die Wiederherstellung von Passwörtern.

Eine ähnliche Analyse kann für abgeschnittene MAC Algorithmen, wie in [11] beschrieben. Wenn z. B. bei einem 80-Bit (10-Byte)-MAC-Tag 11 der letzten 12 Bytes von P^* bekannt sind, können wir die verbleibenden 5 Bytes mit $5L - 2^8$ Sitzungen wiederherstellen.

Schließlich stellen wir fest, dass die Angriffe "Lucky 13 + BEAST" unabhängig von der Größe des MAC-Tags gleich gut funktionieren.

D. Anwendung der Angriffe auf DTLS

Bislang haben wir uns auf TLS konzentriert. Die für DTLS erforderlichen Änderungen sind die gleichen wie bei unserem Unterscheidungsangriff in Abschnitt III: Wir können die Techniken aus [1] verwenden, um die Zeitunterschiede zu verstärken und die TLS-Fehlermeldungen zu emulieren. Die Möglichkeit der Verstärkung reduziert die Komplexität des Angriffs drastisch: Im Wesentlichen können wir jeden Δ-Wert mit nur wenigen Paketzügen genau testen, anstatt L Versuche zu benötigen. Es gibt einen weiteren entscheidenden Unterschied, den wir hervorheben möchten: Wie bereits erwähnt, behandelt DTLS Fehler, die während der Entschlüsselung auftreten, nicht als fatal. Das bedeutet, dass der gesamte Angriff gegen DTLS in einer *einzigen* Sitzung durchgeführt werden kann, d. h. ohne dass derselbe Klartext an derselben Stelle im Klartext über mehrere Sitzungen hinweg wiederholt werden muss und ohne dass das Handshake-Protokoll abgewartet werden muss, um für jede Sitzung erneut durchgeführt werden.

Diese Unterschiede machen unseren Angriff für DTLS durchaus praktikabel. Dies gilt insbesondere, wenn die optionale Überprüfung der Sequenznummern bei DTLS deaktiviert ist. Selbst wenn dies nicht der Fall ist, sind die Angriffe in der Praxis durchaus durchführbar, vorausgesetzt, es sind genügend DTLS-Nachrichten verfügbar oder das durch DTLS geschützte Protokoll der oberen Schicht erzeugt Antworten auf gesendete Nachrichten in konsistenter Weise. Diese Probleme werden in [1] und im nächsten Abschnitt ausführlicher diskutiert, wo wir über die erfolgreiche Implementierung unserer Angriffe für die OpenSSL-Implementierung von TLS und DTLS berichten.

V. EXPERIMENTELLE ERGEBNISSE FÜR OPENSSL

A. Experimenteller Aufbau

Wir haben Version 1.0.1 von OpenSSL auf dem Client und dem Server ausgeführt. In unserem Laboraufbau sind ein Client, der Angreifer und der angegriffene Server alle mit demselben VLAN auf einem 100-Mbps-Ethernet-Switch verbunden. Der angegriffene Server lief auf einem Rechner mit einem Single-Core-Prozessor mit 1,87 GHz und 1 GByte RAM, während der Angreifer auf einem Rechner mit einem Dual-Core-Prozessor mit 3,4 GHz und 2 GByte RAM lief.

Um den (D)TLS-Client zu simulieren, haben wir `s_client verwendet`, ein generisches Tool, das als Teil des OpenSSL-Distributionspakets verfügbar ist. Wir modifizierten den Quellcode von `s_client`, um unsere Testanforderungen zu erfüllen. Außerdem haben wir ein einfaches Python-Skript entwickelt, das `s_client` bei Bedarf aufruft. Unser Angriffscode wurde in C geschrieben und ist in der Lage, beliebige Pakete abzufangen, zu manipulieren und in das Netzwerk zu injizieren.

Im Falle von TLS fängt der Angreifer das "gezielte" Paket ab, manipuliert es und sendet dann die manipulierte Version an den Zielserver, wodurch die TLS-Sitzung abgebrochen wird. Dieses manipulierte Paket zwingt den Client und den Zielserver dazu, die TCP-Synchronisation zu verlieren, was zu einer Verzögerung beim Abbau der TCP-Verbindung führt. Um den Abbau der TCP-Verbindung zu beschleunigen, sendet der Angreifer beim Erkennen der verschlüsselten TLS-Warnmeldung gefälschte RST-Pakete an den Client und das Zielsystem und zwingt beide Systeme, die der beendeten TLS-Sitzung zugrunde liegende TCP-Struktur unabhängig voneinander zu zerstören.

Alle Zeitangaben in diesem Dokument basieren auf Hardware-Zyklen, die für die Prozessorgeschwindigkeit spezifik sind. So entsprechen beispielsweise 187 Hardware-Zyklen auf unserem Zielserver mit einer Geschwindigkeit von 1,87 GHz einer absoluten Zeit von 100 ns. Zum Zählen der Hardware-Zyklen haben wir eine bestehende C-Bibliothek verwendet, die unter der GNU GPL v3[4] lizenziert ist.

B. Statistische Analyse

Die von uns in jedem Experiment erfassten Netzwerkzeitpunkte stammen aus einer schiefen Verteilung mit langen Schwänzen und vielen Ausreißern. Wir haben jedoch festgestellt, dass die Verwendung grundlegender statistischer Verfahren (Mediane und allgemeiner Perzentile) für die Analyse unserer Daten ausreichend ist.

C. Unterscheidungsangriff für OpenSSL TLS

Abbildung 2 zeigt die experimentelle Verteilung der Zeitwerte für den in Abschnitt III beschriebenen TLS-Unterscheidungsangriff. Die Abbildung zeigt, dass es mit genügend Stichproben möglich sein sollte, Verschlüsselungen der Nachricht M_0 (bestehend aus 32 beliebigen Bytes gefolgt von 256 Kopien von **0**xFF) von Verschlüsselungen der Nachricht M_1 (bestehend aus 287 beliebigen Bytes gefolgt von **0x00**) zu unterscheiden.

[4]code.google.com/p/fau-timer

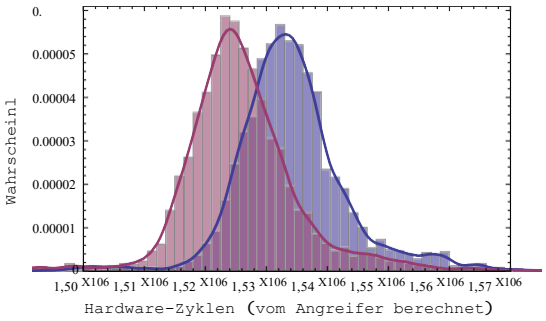

Abbildung 2. Verteilung der Zeitwerte (Ausreißer entfernt) für den Unterscheidungsangriff auf OpenSSL TLS, mit schnellerer Verarbeitungszeit im Fall von M_0 (in rot) im Vergleich zu M_1 (in blau).

L	Erfolgswahrscheinlichkeit
1	0.756
2	0.769
4	0.858
8	0.914
16	0.951
32	0.983
64	0.992
128	1

Tabelle I
OPENSSL TLS UNTERSCHEIDET DIE ERFOLGSWAHRSCHEINLICHKEITEN VON ANGRIFFEN.

Wir haben einen einfachen Schwellentest verwendet, um einen konkreten Angriff zu entwickeln: Wir berechnen einen Schwellenwert T auf der Grundlage des Profilings, sammeln L Zeitproben, filtern Ausreißer, berechnen den Median der verbleibenden Zeitproben und geben dann 1 aus, wenn der Medianwert größer als T ist, und 0, wenn er kleiner ist. Tabelle I zeigt die Erfolgswahrscheinlichkeiten für diesen konkreten Unterscheidungsangriff; es ist ersichtlich, dass der Angriff auch dann zuverlässig ist, wenn nur eine mäßige Anzahl von Proben zur Verfügung steht. Der Angriff hat bereits einen signifikanten Vorteil gegenüber dem Raten, wenn $L = 1$ ist, d.h. wenn nur eine Probe zur Verfügung steht.

D. Klartext-Wiederherstellungsangriffe für OpenSSL TLS

1) Teilweise Wiederherstellung des Klartextes: Abschnitt IV beschreibt einen Angriff, bei dem das 15 Byte P^*_{14} wiederhergestellt werden kann, wenn P^*_{14} bekannt ist. Dazu wird Δ_{14} so eingestellt, dass $P^*_{14} \oplus \Delta_{14}$ gleich 0x01 ist, und dann werden alle möglichen Werte von Δ_{15} ausprobiert, um herauszufinden, welcher davon $P^*_{15} \oplus \Delta_{15}$ ebenfalls gleich 0x01 macht. Abbildung 3 zeigt die mittlere serverseitige Entschlüsselungszeit in Abhängigkeit von Δ_{15} für die besonderen Werte von $P^*_{14} = 0x01$ (also $\Delta_{14} = 0x00$) und $P^*_{15} = 0xFF$. Eine deutliche Verringerung der Verarbeitungszeit ist für den erwarteten Wert von Δ_{15} deutlich unterscheidbar.

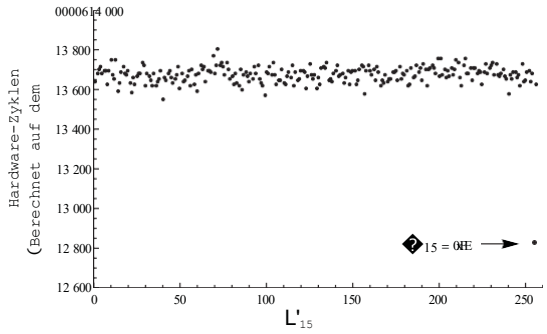

Abbildung 3. Mittlere OpenSSL-TLS-Server-Zeiten (in Hardware-Zyklen) bei $P^*_{14} = 0x01$ und $P^*_{15} = 0xFF$. Wie erwartet, führt $\Delta_{15} = 0xFE$ zu einer schnelleren Bearbeitungszeit.

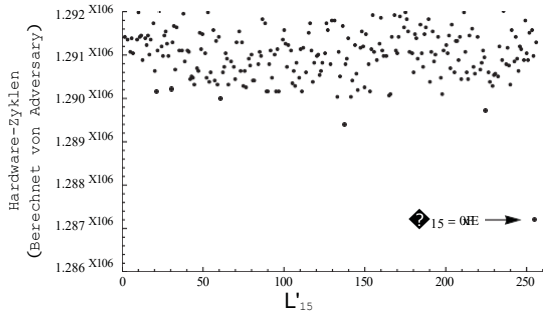

Abbildung 4. OpenSSL TLS mittlere Netzwerkzeit in Bezug auf die Hardware Zyklen, wenn $P^*_{14} = 0x01$ und $P^*_{15} = 0xFF$. Wie erwartet $\Delta_{15} = 0xFE$ führt zu einer schnelleren Bearbeitungszeit.

Abbildung 5 zeigt die Erfolgswahrscheinlichkeiten für den Angriff. Jeder Datenpunkt in der Kurve basiert auf mindestens 64 Versuchen. Jede Kurve in der Grafik steht für eine unterschiedliche Anzahl von Gesamtsitzungen, die bei dem Angriff verbraucht wurden (entsprechend den unterschiedlichen Werten für L, der Anzahl der Versuche für jeden Δ-Wert). Die *x-Achse* stellt das Perzentil dar, das in unserem statistischen Test verwendet wird: Wenn der Perzentilwert p ist, dann nehmen wir als korrekten Wert für Δ_{15} denjenigen, für den der p-te Perzentilwert der Zeitverteilung (gemessen über L Stichproben) *minimiert* ist. Es ist offensichtlich, dass eine Reihe von Perzentilen gut funktionieren, einschließlich des Medians. Wie erwartet, steigt die Erfolgswahrscheinlichkeit des Angriffs mit zunehmendem L. Wir erreichen bereits eine Erfolgswahrscheinlichkeit von 1, wenn $L = 2^8$ ist, wobei die Gesamtzahl der benötigten Sitzungen 2^{16} beträgt. Ebenso haben wir eine festzustellen, nämlich $\Delta_{15} = 0xFE$.

Bemerkenswert ist auch die Stabilität der Verarbeitungszeit für andere Byte-Werte. Diese serverseitigen Zeiten deuten darauf hin, dass ein Angriff, der auf der Zeitmessung von

Fehlermeldungen im Netz basiert, eine gewisse Aussicht auf Erfolg hat. Abbildung 4 zeigt die entsprechende Verteilung der mittleren Netzwerkzeitpunkte in unserem Versuchsaufbau. Die Daten sind eindeutig rauschhafter, aber die "Delle" bei $\Delta_{15} = 0xFE$ ist

Erfolgswahrscheinlichkeit von 0,93 bei $L = 2^7$, wobei die Gesamtzahl der Sitzungen 2^{15} beträgt.

In Anbetracht dieser Ergebnisse gehen wir davon aus, dass der Angriff leicht auf die Wiederherstellung von 15 unbekannten Bytes aus einem Block ausgedehnt werden kann, wenn man eines der letzten beiden Bytes erhält. Wir haben diese Variante nicht implementiert.

2) Vollständige Wiederherstellung des Klartextes: Der nächste Schritt wäre die

den Angriff zur vollständigen Wiederherstellung des Klartextes aus Abschnitt IV durchführen. In diesem Fall würde der Angreifer insgesamt $L \cdot 2^{16}$ Versuche benötigen

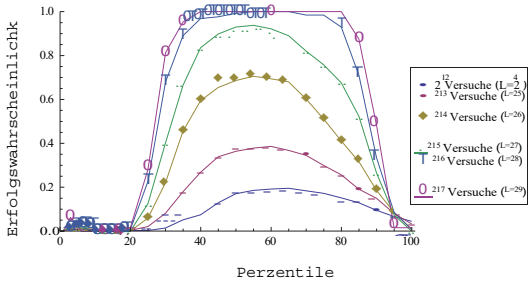

Abbildung 5. OpenSSL TLS teilweise Klartext-Wiedererstellung: prozentualer Anteil Erfolgswahrscheinlichkeiten für die Wiedererlangung von P_{15}^* unter der Annahme, dass P_{14} bekannt ist.

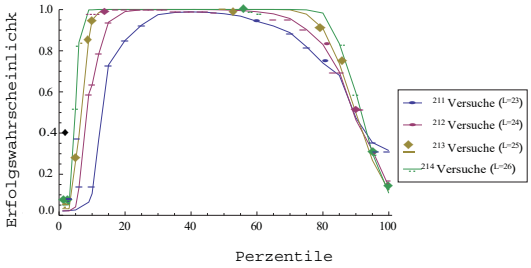

Abbildung 6: OpenSSL DTLS Wiederherstellung von partiellem Klartext: prozentualer Anteil Erfolgswahrscheinlichkeiten für die Wiedererlangung von P_{15}^* mit P_{14}^* bekannt, $n = 10$.

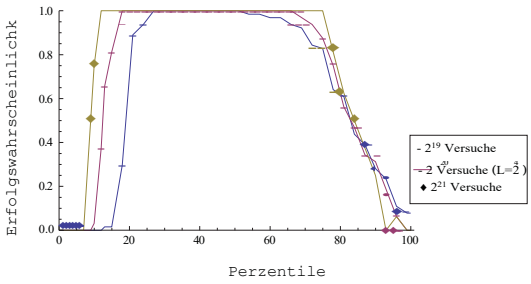

Abbildung 7. OpenSSL DTLS 2-Byte-Wiederherstellung: Prozentualer Erfolg Wahrscheinlichkeiten für die Wiederherstellung von P_{14}^* und P_{15}^*, $n = 10$.

um herauszufinden, welcher Maskenwert den Fall 2 auslöst. Im Falle von TLS nimmt dies aufgrund der zugrundeliegenden TCP- und TLS-Verbindungsauf- und -abbauzeiten eine beträchtliche Zeit in Anspruch. Bei $L = 2^7$ schätzen wir zum Beispiel, dass die 2^{23} -Sitzungen in unserem Setup etwa 64 Stunden dauern würden. Sobald jedoch die letzten beiden Bytes eines Blocks erfolgreich wiederhergestellt wurden, können die restlichen Bytes in diesem Block in viel kürzerer Zeit wiederhergestellt werden. Wir haben den Angriff zur vollständigen Wiederherstellung des Klartextes für TLS nicht implementiert. Unsere Ergebnisse unten
für DTLS deuten stark darauf hin, dass der vollständige Angriff für TLS mit $L = 2^7$ funktionieren würde, wenn auch langsam.

E. Klartext-Wiederherstellungsangriffe für OpenSSL DTLS

Wie in Abschnitt IV-D erläutert, können wir die Timing- und Amplifikationstechniken aus [1] in Kombination mit den zuvor beschriebenen Angriffen verwenden, um DTLS anzugreifen. Nun sendet der Angreifer eine Anzahl (n) gefälschter Pakete, gefolgt von einer DTLS-Heartbeat-Anfrage und wartet auf die entsprechende Heartbeat-Antwort. Dieser Vorgang wird L-mal für jeden Maskenwert wiederholt. Der Angreifer wählt n und L, *um einen* Kompromiss zwischen der Erfolgswahrscheinlichkeit des Angriffs und der Gesamtzahl der eingeschleusten Pakete zu finden. Wir haben experimentell festgestellt, dass $n = 10$ eine gute Wahl ist, um stabile Zeitwerte zu erreichen. Andererseits entspricht $n = 1$ dem, was bei TLS zu erwarten ist, ohne dass der Overhead des TCP- und TLS-Verbindungsaufbaus in Kauf genommen werden muss (man beachte, dass das Rauschen bei DTLS im Allgemeinen etwas höher ist, da wir von einer Fehlermeldung der Anwendungsschicht und nicht von einer nativen TLS-Fehlermeldung abhängen). Höhere Werte von n könnten verwendet werden, wenn der Angreifer vom Server entfernt ist.

Abbildung 6 zeigt die prozentualen Erfolgswahrscheinlichkeiten

Wiederherstellung beider Bytes mit einer Erfolgswahrscheinlichkeit von 0,93 für 2^{19} Versuche ($L = 2^3$). Die Qualität dieser Ergebnisse ist ein Beweis dafür, dass der Angriff leicht zu einem vollständigen Klartext-Wiederherstellungsangriff erweitert werden sollte. Abbildung 8 zeigt unsere Ergebnisse für $n = 1$, was, wie wir uns erinnern, als experimentelles Modell für TLS dient. Wir sehen, dass die 2-Byte-Wiederherstellung bei 2^{23} Versuchen ($L = 2^7$) zuverlässig ist; wir erreichen bereits mit 2^{22} Versuchen eine Erfolgsquote von mehr als 80 %.

F. Anspruchsvollere Netzwerkumgebungen

Wir haben keine Experimente durchgeführt, bei denen sich der Angreifer nicht im selben LAN wie der Server befindet. Angesichts der geringen zeitlichen Unterschiede würden wir erwarten, dass die Angriffe fehlschlagen, wenn der Angreifer entfernt ist, d. h. mehr als ein paar Hops vom Server entfernt ist, oder dass eine sehr große Anzahl von Sitzungen erforderlich wäre, um zuverlässige Ergebnisse zu erhalten. Dennoch gibt es realistische Szenarien, in denen die

zur Wiederherstellung von P^*_{13} unter der Annahme, dass P^*_{14} bekannt ist, für $n =$ 10. Es ist zu erkennen, dass der Angriff sehr effektiv ist und zuverlässig P^*_{11} Wiedererlangung des unbekannten Klartextbytes mit nur 2 Versuchen ($L = 2^3$). Auch für Versuche ($L = 1$), die Erfolgswahrscheinlichkeit 2^8 beträgt 0,266.

Wir haben auch einen 2-Byte-Wiederherstellungsangriff gegen OpenSSL DTLS durchgeführt; dieser Angriff ist praktisch der erste Schritt des in Abschnitt IV beschriebenen Angriffs zur vollständigen Klartext-Wiederherstellung. Abbildung 7 zeigt die Erfolgswahrscheinlichkeiten für die Wiederherstellung von P^*_{14} und P^*_{15} wenn $n = 10$. Auch hier ist der Angriff sehr effektiv,

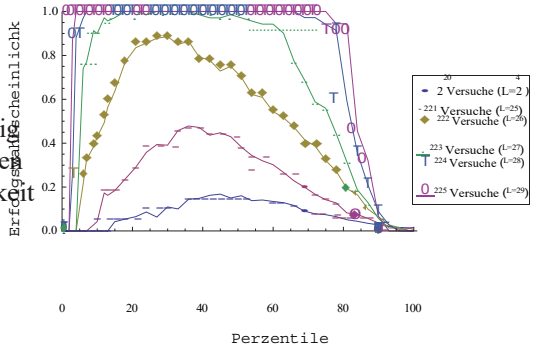

Abbildung 8. OpenSSL DTLS 2-Byte-Wiederherstellung: Prozentualer Erfolg Wahrscheinlichkeiten für die Wiederherstellung von P^*_{14} und P^*_{15}, $n = 10$.

Die Anforderung der Nähe kann z. B. erfüllt werden, wenn ein feindlicher Netzbetreiber seine Kunden angreift, oder in Cloud-Computing-Umgebungen. Bei DTLS können die Zeitsignale praktisch um einen beliebigen Betrag verstärkt werden, so dass wir davon ausgehen, dass die Angriffe aus der Ferne durchgeführt werden können.

VI. Andere Implementierungen von TLS

1) GnuTLS: Die GnuTLS[5] -Implementierung von MEE-TLS-CBC geht mit schlechten Auffüllungen anders um als in den RFCs empfohlen: Anstatt von Auffüllungen der Länge Null auszugehen, wird das letzte Byte des Klartextes verwendet, um zu bestimmen, wie viele Klartextbytes entfernt werden müssen (unabhängig davon, ob diese Bytes korrekt formatierte Auffüllungen sind oder nicht). Da dieser Ansatz eine natürliche Alternative zu den Empfehlungen der RFCs für den Umgang mit fehlerhaftem Padding darstellt, wird er hier analysiert; die vollständigen Details der Analyse finden Sie in der Vollversion [2].

Erstens weisen wir darauf hin, dass die Verarbeitung im GnuTLS-Stil genauso anfällig für Unterscheidungsangriffe ist wie die RFC-konforme Verarbeitung. Tatsächlich funktioniert der in Abschnitt III beschriebene Angriff genauso wie zuvor[6] . Als Nächstes stellen wir einen Angriff vor, der das ganz rechte Byte des Klartextes in einem beliebigen Zielblock wiederherstellt für
Verarbeitung von Auffüllungen im GnuTLS-Stil. C^* bezeichne den Ziel-Chiffretextblock, C^t den vorherigen Chiffretext
Block und Δ bezeichnen einen Maskenblock von 16 Bytes. Wir betrachten die Entschlüsselung eines Chiffretextes $^{Catt}(\Delta)$ in der Form:

$$^{Catt}(\Delta) = HDR \;||\, C_0 \,||C_1 \,||C_2 \,|| \ldots ||C_{18} \,||C^t \oplus \Delta||C^*$$

in der es 20 Nicht-IV-Chiffretextblöcke gibt, der letzte Block ist eine XOR-maskierte Version von C^t und der letzte Block ist C^* , der Ziel-Chiffretextblock. Der entsprechende 320-Byte-Klartext ist $P = P_1 \,||P_2 \,|| \ldots \,||P_{19} \,||P_{20}$, wobei

$$\begin{aligned} P_{20} &= D_K(C^*) \oplus (C^t \oplus \Delta) \\ &= P^* \oplus \Delta. \end{aligned}$$

Jetzt brauchen wir nur noch zwei verschiedene Fälle zu betrachten, die zusammen alle Möglichkeiten abdecken:

1) P_{20} endet mit einem 0x00-Byte: In diesem Fall wird ein einzelnes Byte des Paddings entfernt, die nächsten 20 Bytes werden als MAC-Tag T interpretiert, und die verbleibenden 320 - 21 = 299 Bytes Klartext werden als Datensatz R genommen. MAC
   Die Verifizierung wird dann an einer 13 + 299 = 312-Byte-Nachricht $SQN||HDR||R$ durchgeführt.
2) P_{20} endet mit einem beliebigen anderen Byte-Wert: In diesem Fall werden mindestens zwei Bytes "Padding" entfernt, die nächsten 20 Bytes werden als MAC-Tag T interpretiert, und die verbleibenden Bytes des Klartextes werden als Datensatz verwendet R. Da die Ausgangslänge der Nachricht mit 320 Byte lang genug ist, um die Entfernung von 256 Byte Padding und 20 Byte MAC zu ermöglichen und trotzdem noch

5www.gnu.org/software/gnutls/

6 Da der Angriff nur Klartexte betrifft, die korrekt gepolstert sind, funktioniert er bei *jedem* korrekten Entschlüsselungsalgorithmus.

ein Datensatz, der nicht Null ist, wird keine Prüfung der Länge fehlschlagen. Die MAC-Überprüfung wird dann für eine Nachricht durchgeführt

$SQN||HDR||R$, die *höchstens* 311 Bytes enthält.

In beiden Fällen scheitert die MAC-Verifizierung (mit überwältigender Wahrscheinlichkeit) und es wird eine Fehlermeldung ausgegeben. Beachten Sie, dass die MAC-Verifizierung gemäß der Diskussion in Abschnitt II-A in Fall 1 9 Auswertungen der Kompressionsfunktion für SHA-1 umfasst, während in Fall 2 höchstens 8 Auswertungen erforderlich sind. Daher können wir hoffen, die beiden Fälle durch sorgfältiges Timing zu unterscheiden, wie zuvor.

Der Angriff zur Wiederherstellung des Klartextes aus einem einzigen Byte ist einfach: Der Angreifer injiziert eine Folge von Chiffretexten $^{C_{att}}(\Delta)$ mit Werten von Δ, die über alle möglichen Werte variieren
im letzten Byte Δ_{15} , dann wird der Angreifer (im schlimmsten Fall) nach 2^8 Versuchen sicher einen Wert für Δ wählen, so dass $^{C_{att}}(\Delta)$ den Fall 1 auslöst. Wenn dies erkannt wird, weiß er, dass P_{20} mit einem 0x00-Byte endet, und kann auf den Wert des
das letzte Byte von P^* über die blockweise Gleichung $P_{20} = P^* \oplus \Delta$.
Für weitere Verbesserungen dieses Angriffs und der Analyse für
HMAC-MD5 und HMAC-SHA-256, siehe [2]

Wir stellen fest, dass wir keine Angriffe für die GnuTLS-ähnliche Verarbeitung gefunden haben, die mehr als das letzte Byte des Zielblocks extrahieren können. Dies ist angesichts der Tatsache, dass die Entschlüsselungszeit für die GnuTLS-ähnliche Verarbeitung nur vom letzten Byte des Klartextes abhängt, nicht überraschend.

Wir haben mit Version 3.0.21 von GnuTLS gearbeitet, um die oben genannten Angriffe zu implementieren. Wir stellten fest, dass die Netzwerk-Timings für Fehlermeldungen viel lauter waren als bei OpenSSL. Außerdem haben wir einige Kodierungsfehler in GnuTLS identifiziert, die unseren Angriff behinderten. Dennoch waren wir in der Lage, bis zu 4 Bits des letzten Bytes eines jeden Blocks zuverlässig wiederherzustellen. Einzelheiten finden Sie in der Vollversion [2]. Obwohl wir weniger Klartext extrahieren konnten als unser OpenSSL-Angriff, waren für GnuTLS weit weniger TLS-Sitzungen erforderlich. Dies zeigt, dass das Ignorieren der Empfehlungen der RFCs schwerwiegende Folgen für die Sicherheit haben kann.

2) NSS: Network Security Services (NSS)[7] ist ein quelloffener Satz von Bibliotheken, die unter anderem TLS implementieren. Sie ist weit verbreitet, unter anderem in Mozilla-Client-Produkten und Google Chrome. Der NSS-Code[8] verfolgt denselben Ansatz wie GnuTLS, was den Code anfällig für Angriffe macht, die ein einziges Byte Klartext pro Block auslesen.

3) PolarSSL: Wir haben auch die TLS-Implementierung von PolarSSL[9] untersucht. Der Code[10] verhält sich ähnlich wie OpenSSL, indem er eine Variable `padlen` auf 0 setzt, wenn die Auffüllungsprüfung fehlschlägt, und dann den MAC anhand eines Datensatzes ohne padlen-Bytes verifiziert. Dies würde es zu einem

7http://www.mozilla.org/projects/security/pki/nss

8 Wir haben mit Version 3.13.6 gearbeitet, die unter https://ftp.mozilla.org/pub/ mozilla.org/security/nss/releases/NSS 3 13 6 RTM/src/ verfügbar ist.

9polarssl.org/

10 Wir haben mit Version 1.1.4 gearbeitet, die unter polarssl.org/trac/browser/ trunk/library/ssl tls.c verfügbar ist.

anfällig für die in Abschnitt IV beschriebenen Angriffe. In seiner Standardkonfiguration sendet PolarSSL jedoch keine TLS-Warnmeldungen, wenn Entschlüsselungsfehler auftreten. Dies schützt PolarSSL vor unserem Angriff, bedeutet aber auch, dass es in diesem Punkt *nicht* RFC-konform ist.

4) yaSSL: Die yaSSL[11] eingebettete SSL-Bibliothek, CyaSSL, ist auf eingebettete und Echtzeit-Betriebssystemumgebungen ausgerichtet. Sie scheint nur wenige bekannte Schwachstellen zu haben, wobei seit 2005 nur 5 in der CVE-Datenbank[12] gemeldet wurden. Der CyaSSL-Code[13] führt keine ordnungsgemäßen Auffüllungsprüfungen durch, sondern untersucht lediglich das letzte Byte des Klartextes und verwendet dieses, um zu bestimmen, wie viele Bytes entfernt werden müssen. Dieser Ansatz macht den Code anfällig für den alten Angriff aus [17], der ein Byte Klartext pro Block wiederherstellt. Dies war die einzige Implementierung, die wir gefunden haben, die noch diese grundlegende Säge enthält.

5) Java: Wir haben die Java-Implementierungen von TLS von BouncyCastle[14] und OpenJDK[15] untersucht.

Der BouncyCastle-Code führt eine sorgfältige Prüfung der Padding-Länge durch (wie im letzten Byte des Klartextes angegeben), behandelt aber das Padding als Länge 1, wenn sich das Padding-Format bei der Prüfung als falsch herausstellt (eine Variable `paddingsize` wird auf 0 gesetzt, aber dann wird die Klartextgröße um einen Betrag `paddingsize+minLength` reduziert, wobei `minLength` auf 1 größer als die MAC-Tag-Größe gesetzt ist). Dies weicht geringfügig von der Empfehlung der RFCs ab, das Padding als Länge Null zu behandeln, ermöglicht aber dennoch die Anwendung unserer Angriffe in den Abschnitten III und IV (für Fall 3 des Hauptangriffs zur Wiederherstellung des Klartextes in Abschnitt IV wird die MAC-Verification schließlich an einer 56-Byte-Nachricht durchgeführt, was aber immer noch 5 Evaluierungen der Kompressionsfunktion für SHA-1 erfordert).

Der OpenJDK-Code scheint der Empfehlung der RFCs zu folgen, indem er das Padding als Null-Länge behandelt, wenn sich das Padding-Format bei der Überprüfung als falsch erweist. Dies liegt daran, dass dieser Fall durch eine Ausnahmebehandlung abgefangen wird, während der die Variable, die die Klartextlänge angibt, nicht geändert wird. Dies macht sie potenziell anfällig für unsere Angriffe in den Abschnitten III und IV.

VII. Gegenmaßnahmen

A. Zufällige Zeitverzögerungen hinzufügen

Eine natürliche Reaktion auf zeitbasierte Angriffe besteht darin, zufällige Zeitverzögerungen in den Entschlüsselungsprozess einzubauen, um die statistische Analyse zu vereiteln. Tatsächlich ist diese Gegenmaßnahme überraschend ineffektiv, wie wir in der Vollversion [2] zeigen.

[11] yassl.com/yaSSL/Home.html

[12] www.cvedetails.com/vulnerability-list/vendor_id-3485/Yassl.html

[13] Wir haben mit Version 2.3.0 gearbeitet, die unter yassl.com/yaSSL/Source/ output/src/internal.c.html verfügbar ist.

[14] www.bouncycastle.org/viewcvs/viewcvs.cgi/java/crypto/src/org/bouncycastle/crypto/tls/TlsBlockCipher.java?view=markup

[15] hg.openjdk.java.net/jdk7/l10n/jdk/file/3598d6eb087c/src/share/classes/sun/security/ssl/SSLSocketImpl.java und hg.openjdk.java.net/jdk7/2d/jdk/file/85fe3cd9d6f9/src/share/classes/sun/security/ssl/CipherBox.java

B. RC4 verwenden

Die einfachste Gegenmaßnahme für TLS ist der Wechsel zur Verwendung der RC4-Stromchiffre anstelle der CBC-Verschlüsselung. Dies ist jedoch keine Option für DTLS. Wenn eine Stromchiffre in TLS verwendet wird, ist kein Padding erforderlich. Folglich wird keiner der Angriffe in diesem Dokument funktionieren. RC4 wird in den TLS-Implementierungen weitgehend unterstützt, die gleiche Gegenmaßnahme ist gegen den BEAST-Angriff wirksam und wurde als Reaktion auf BEAST weitgehend übernommen. Die Verwendung einer Stromchiffre in einer MEE-Konstruktion wird von der Theorie gut unterstützt [12]. Die ersten Bytes des vom RC4-Generator ausgegebenen Schlüsselstroms weisen jedoch gewisse kleine Verzerrungen auf, und TLS verwirft diese vor Beginn der Verschlüsselung nicht. Aus diesem Grund raten wir von der Verwendung von RC4 ab.

C. Authentifizierte Verschlüsselung verwenden

Eine weitere Möglichkeit besteht darin, von MEE-TLS-CBC zu einem speziellen authentifizierten Verschlüsselungsalgorithmus wie AES-GCM oder AES-CCM zu wechseln, die in den RFCs 5288 [24] bzw. 6655 [15] für die Verwendung in TLS standardisiert wurden. Theoretisch sollten damit alle Angriffe, die auf Schwächen in der MEE-Konstruktion beruhen, unterbunden werden. Allerdings können wir Implementierungsfehler nicht ausschließen, und uns ist keine detaillierte Analyse der Implementierungen dieser Algorithmen in (D)TLS auf mögliche Seitenkanäle bekannt. Ein weiteres Problem ist, dass die authentifizierte Verschlüsselung erst in TLS 1.2 hinzugefügt wurde und diese Version von TLS noch nicht von vielen Implementierungen unterstützt wird.

D. Sorgfältige Implementierung der MEE-TLS-CBC-Entschlüsselung

Unsere letzte Option besteht darin, die MEE-TLS-CBC-Entschlüsselung sorgfältiger zu implementieren.

Die wichtigste Anforderung ist die Gewährleistung einer einheitlichen Verarbeitungszeit für alle MEE-TLS-CBC-Chiffretexte einer bestimmten Größe. Das heißt, die Gesamtverarbeitungszeit sollte nur von der Größe des Chiffriertextes abhängen und nicht von den Merkmalen des zugrundeliegenden Klartextes (einschließlich Padding). Das Grundprinzip, das dabei befolgt werden muss, ist recht einfach: Da die größten Zeitunterschiede aus der MAC-Verarbeitung resultieren, sollten Implementierungen sicherstellen, dass derselbe Umfang an MAC-Verarbeitung durchgeführt wird, unabhängig davon, welche Länge der zugrunde liegende Klartext aufweist. Dieses einfache Prinzip wird jedoch durch die Notwendigkeit kompliziert, auch den zugrunde liegenden Klartext einer sorgfältigen Prüfung zu unterziehen und dabei die Einführung weiterer zeitlicher Seitenkanäle zu vermeiden und dafür zu sorgen, dass ein angemessener Umfang an MAC-Verarbeitung durchgeführt wird, selbst wenn diese Prüfungen fehlschlagen. Eine weitere Komplikation entsteht dadurch, dass die Anzahl der zu prüfenden Bytes in der Auffüllungsprüfung vom letzten Byte des letzten Klartextblocks abhängt, so dass die Laufzeit der Auffüllungsprüfung selbst bei gleichmäßiger MAC-Verarbeitung immer noch eine kleine Menge an Informationen über den Klartext verraten kann. Mit diesen Bemerkungen im Hinterkopf fahren wir nun fort, eine detaillierte Beschreibung zu geben, wie man eine zeitkonstante Verarbeitung von MEE-TLS-CBC-Chiffretexten erreichen kann, indem man geeignete Verfahren für den Klartext verwendet.

fähige Überprüfung der Korrektheit. Im Folgenden bezeichnen wir mit `plen` die Länge (in Byte) des Klartextes P, den wir unmittelbar nach der CBC-Entschlüsselung des Geheimtextes erhalten, mit `padlen` das letzte Byte dieses Klartextes, interpretiert als ganze Zahl zwischen 0 und 255, und mit t die Länge der MAC-Tags (in Byte). Außerdem bezeichnen `HDR`, `SQN` den (D)TLS-Datensatzkopf und den erwarteten Wert der Sequenznummer für diesen Datensatz. Unser empfohlenes Verfahren lautet dann wie folgt:

1) Zunächst wird der Chiffriertext auf seine Unbedenklichkeit geprüft: Die Länge in Bytes muss ein Vielfaches der Blockgröße b sein und mindestens $max\{b, t + 1\}$ (für verkettete IVs) oder $b + max\{b, t + 1\}$ (für explizite IVs). Wenn diese Bedingungen nicht erfüllt sind, dann einen `fatalen Fehler` zurückgeben.
2) Entschlüsseln Sie den verschlüsselten Text, um den Klartext P zu erhalten; jetzt `plen` wird ein Vielfaches von b und mindestens $max\{b, t+1\}$ sein.
3) Wenn t + `padlen` +1 > `plen`, dann ist der Klartext nicht lang genug, um die Füllung (wie im letzten Byte des Klartextes angegeben) plus ein MAC-Tag zu enthalten. In diesem Fall, Führen Sie eine Schleife aus, als ob es 256 Bytes Füllmaterial gäbe, mit einer Dummy-Prüfung in jeder Iteration. Dann lassen Sie P^t die firsten `Plen` - tbytes von P bezeichnen, berechnen Sie einen MAC auf $SQN||HDR||P^t$ und führen Sie einen Konstantzeitvergleich von den berechneten MAC mit den letzten t Bytes von P . Rückgabe `Fataler Fehler`.
4) Andernfalls (wenn t + `padlen` + 1 ≤ `plen`) werden die letzten `padlen` +1 Bytes von P überprüft, um sicherzustellen, dass sie alle gleich (dem letzten Byte von P), um sicherzustellen, dass die Schleife alle Bytes prüft (und nicht anhält, sobald die erste Nichtübereinstimmung entdeckt wird). Wenn dies fehlschlägt, führen Sie eine Schleife, als gäbe es 256 - `padlen` - 1 Bytes an Auffüllen, mit einer Dummy-Prüfung in jeder Iteration, und führen Sie dann eine MAC-Prüfung wie im vorherigen Schritt durch. Rückgabe `Fataler Fehler`.
5) Andernfalls (das Padding ist jetzt korrekt formatiert) führen Sie eine Schleife aus, als ob 256 - `padlen` - 1 Bytes Padding vorhanden wären, wobei Sie in jeder Iteration eine Dummy-Prüfung durchführen. Dann bezeichne P^t die ersten plen-padlen-1-t-Bytes von P , und T *die* nächsten t Bytes von P (die verbleibenden - der von P ist ein gültiges Padding). Führen Sie die MAC-Berechnung für $SQN||HDR||P^t$ durch, um ein MAC-Tag T^t zu erhalten. Dann setzt man $L_1 = 13+plen-t$, $L_2 = 13+plen-padlen-1-t$, und führt eine zusätzliche $I^{L\,1-55}/ - I^{L\,2-55}/$ MAC Auswertungen von Kompressionsfunktionen (auf Dummy-Daten). Abschließend wird ein zeitunabhängiger Vergleich von T und T^t durchgeführt. Wenn diese gleich sind, wird P^t zurückgegeben. Andernfalls wird ein `fataler Fehler` zurückgegeben.

Bei der Implementierung des obigen Verfahrens wäre es verlockend, scheinbar unnötige Berechnungen wegzulassen, die beispielsweise durchgeführt werden, wenn t +`padlen` +1 > `plen`. Diese werden jedoch benötigt, um andere Timing-Seitenkanäle zu verhindern, wie sie in [1] für die GnuTLS-Implementierung von DTLS berichtet werden. Beachten Sie auch, dass die im letzten Schritt durchgeführten Dummy-Berechnungen eine Kompressionsfunktion sind

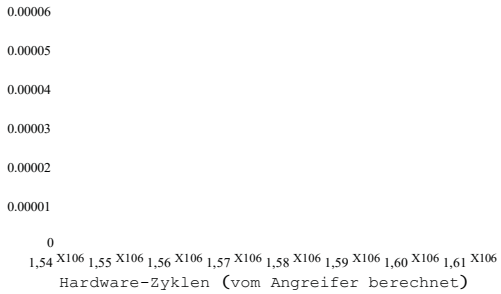

Abbildung 9. Verteilung der Zeitwerte (Ausreißer entfernt) für den Unterscheidungsangriff auf OpenSSL TLS, unter Verwendung unseres Entschlüsselungsverfahrens.

Bewertungen und nicht vollständige MAC-Berechnungen. Dadurch ergibt sich eine MAC-Berechnungszeit, die unabhängig davon, wie viel Füllmaterial entfernt wird, gleich ist (und der in früheren Schritten durchgeführten Zeit entspricht).

Wir haben das obige Verfahren implementiert, indem wir OpenSSL Version 1.0.1 modifiziert haben, die gleiche Version, die wir für unsere Angriffe verwenden. Dann haben wir unseren Unterscheidungsangriff aus Abschnitt III gegen den modifizierten Code ausgeführt. Jedes Paket des Angriffs besteht die Auffüllungsprüfung, schlägt aber bei der MAC-Verifizierung fehl, was den Server veranlasst, die TLS-Sitzung zu schließen und eine verschlüsselte Warnmeldung zu senden. Abbildung 9 zeigt die Verteilung der Zeitwerte (in Hardwarezyklen) nach der Implementierung unseres Verfahrens. Diese Abbildung sollte mit Abbildung 2 verglichen werden: allein die visuelle Inspektion zeigt, dass der Zeitunterschied erheblich geringer ist. Tatsächlich hat sich der Abstand zwischen den Medianen der beiden Verteilungen von etwa 8500 auf etwa 1100 Hardwarezyklen verringert (von etwa 2,5μs auf 0,32μs). Dieser geringe Abstand bedeutet wiederum, dass 128 Sitzungen erforderlich sind, um eine Erfolgswahrscheinlichkeit von 0,68 zu erreichen, während vor unseren Modifizierungen nur eine Sitzung ausreichte, um eine Erfolgswahrscheinlichkeit von 0,756 zu erreichen. Für den Klartext-Wiederherstellungsangriff hat der Angreifer Zugang zu Zeitdifferenzen, die etwa ein Viertel davon betragen, d. h. etwa 80ns auf unserer Hardware. Beachten Sie auch, dass die beiden Verteilungen im Vergleich zu Abbildung 2 umgekehrt sind, d. h. die Verarbeitung von 0xFF-Paketen dauert jetzt im Durchschnitt länger als die von 0x00-Paketen. Wir glauben, dass dies durch den Overhead verursacht wird, der durch einen SHA1_Update-Funktionsaufruf entsteht, der bei 0xFF-Paketen, aber nicht bei 0x00-Paketen auftritt.

Um eine weitere Verringerung der Zeitdifferenz zu erreichen, wäre ein ausgeklügelterer "Konstantzeit"-Programmieransatz erforderlich. Der OpenSSL-Patch in den Versionen 1.0.1d, 1.0.0k und 0.9.8y, der sich mit den Angriffen in diesem Dokument befasst, ist ein Beispiel dafür wie dies erreicht werden kann. Die Komplexität des OpenSSL-Patches ist beachtlich, da etwa 500 Zeilen neuer C-Code erforderlich sind. Für weitere Diskussionen und Erklärungen siehe www.imperialviolet.org/2013/02/04/luckythirteen.html.

VIII. Diskussion

Wir haben eine Reihe von Angriffen gegen (D)TLS-Implementierungen demonstriert. Wir wiederholen, dass die Angriffe nur auf den Chiffretext abzielen und daher von einem Standard-MITM-Angreifer durchgeführt werden können, *ohne die* Möglichkeit, den Klartext zu wählen. Die möglichen Angriffe hängen entscheidend von den Details der Low-Level-Implementierung ab sowie von Faktoren wie dem Verhältnis zwischen der MAC-Tag-Größe t und der Blockgröße b. Alle von uns untersuchten Implementierungen waren für einen oder mehrere Angriffe anfällig. Es ist eine interessante offene Frage, ob ähnliche Timing-Angriffe gegen die TLS-Verschlüsselung mit einer Variante des CRIME-Angriffs entwickelt werden könnten.

Für TLS benötigen wir einen Multi-Session-Angriff, in manchen Fällen mit vielen Sitzungen. Dies schränkt die Praktikabilität der Angriffe ein, aber es ist zu beachten, dass sie mit Standardtechniken wie Sprachmodellen und sequentieller Schätzung weiter verbessert werden können. Sie können auch in einem BEAST-ähnlichen Angriff verbessert werden, um eine effiziente Wiederherstellung von HTTP-Cookies zu ermöglichen. Die Zeitunterschiede, die wir erkennen müssen, liegen nahe an oder unter den Jitterwerten, die man typischerweise in realen Netzwerken findet. Insbesondere muss sich unser Angreifer relativ nahe (in Bezug auf die Anzahl der Netzknoten) an dem angegriffenen Rechner befinden. Dennoch sollten die Angriffe als realistische Bedrohung für TLS angesehen werden, und wir haben eine Reihe von geeigneten Gegenmaßnahmen beschrieben. Die Angriffe sind für DTLS viel schwerwiegender, da dieses Protokoll fehleranfällig ist und die Timing-Amplifikationstechniken aus [1] zur Verfügung stehen. Eine sehr sorgfältige Implementierung des MEE-TLS-CBC-Entschlüsselungsalgorithmus ist erforderlich, um diese Verstärkungstechniken zu vereiteln. In Anbetracht dessen empfehlen wir dringend die Verwendung eines geeigneten authentifizierten Verschlüsselungsalgorithmus anstelle des CBC-Modus für DTLS.

Danksagung

Wir danken Xuelei Fan, David McGrew, Adam Langley, Brad Wetmore und den anonymen Gutachtern für nützliches Feedback. Wir danken auch Eric Rescorla für den Hinweis, dass unsere Angriffe mit BEAST-ähnlichen Techniken im Webumfeld verbessert werden können.

Referenzen

[1] N. AlFardan und K. G. Paterson. Plaintext-recovery attacks against Datagram TLS. In *NDSS*, 2012.

[2] N. AlFardan und K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS record protocols. Vollständige Version dieses Papiers, verfügbar unter www.isg.rhul.ac.uk/tls, 2013.

[3] G. V. Bard. Die Anfälligkeit von SSL für Angriffe im gewählten Klartext. *IACR Cryptology ePrint Archive*, 2004:111, 2004.

[4] G. V. Bard. Ein schwieriger, aber durchführbarer blockweise-adaptiver Selected-Plaintext-Angriff auf SSL. In *SECRYPT*, Seiten 99-109, 2006.

[5] B. Canvel, A. P. Hiltgen, S. Vaudenay, und M. Vuagnoux. Passwort-Abfangen in einem SSL/TLS-Kanal. In D. Boneh, Editor, *CRYPTO*, Band 2729 von *LNCS*, Seiten 583-599. Springer, 2003. ISBN 3-540-40674-3.

[6] C. M. Chernick, C. Edington III, M. J. Fanto, und R. Rosenthal. Richtlinien für die Auswahl und Verwendung von Trans- port Layer Security (TLS)-Implementierungen. In *NIST Spe- cial Publication 800-52, Juni 2005, National Institute of Standards and Technology. Verfügbar unter http://csrc.nist.gov/ publications/ nistpubs/ 800-52/ SP-800-52.pdf* , 2005.

[7] T. Dierks und C. Allen. Das TLS-Protokoll Version 1.0. RFC 2246, Internet Engineering Task Force, 1999.

[8] T. Dierks und E. Rescorla. Das Transport Layer Security (TLS) Protokoll Version 1.1. RFC 4346, Internet Engineering Task Force, 2006.

[9] T. Dierks und E. Rescorla. Das Transport Layer Security (TLS) Protokoll Version 1.2. RFC 5246, Internet Engineering Task Force, 2008.

[10] T. Duong und J. Rizzo. Hier kommen die ⊕ Ninjas. Unveröffentlicht
Manuskript, 2011.

[11] D. Eastlake 3. Transport Layer Security (TLS)-Erweiterungen: Extension Definitions. RFC 6066, 2011.

[12] H. Krawczyk. Die Reihenfolge von Verschlüsselung und Authentifizierung zum Schutz der Kommunikation (oder: Wie sicher ist SSL?). In *CRYPTO*, Seiten 310-331, 2001.

[13] H. Krawczyk, M. Bellare, und R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), 1997.

[14] U. Maurer und B. Tackmann. On the soundness of authenticate-then-encrypt: formalizing the malleability of symmetric encryption. In *ACM CCS*, Seiten 505-515, 2010.

[15] D. McGrew und D. Bailey. AES-CCM Cipher Suites for Transport Layer Security (TLS). RFC 6655 (Proposed Standard), 2012.

[16] N. Modadugu und E. Rescorla. Der Entwurf und die Implementierung von Datagramm-TLS. In *NDSS*, 2004.

[17] B. Moeller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures, 2004. http://www.openssl.org/~bodo/tls-cbc.txt.

[18] K. G. Paterson, T. Ristenpart, und T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT*, Seiten 372-389, 2011.

[19] A. Pironti, P.-Y. Strub, und K. Bhargavan. Identifizierung von Website-Benutzern durch TLS-Trafficking-Analyse: New attacks and effective countermeasures. Technical Report 8067, INRIA, September 2012.

[20] E. Rescorla und N. Modadugu. Datagram Transport Layer Security. RFC 4347, Internet Engineering Task Force, 2006.

[21] E. Rescorla und N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, Internet Engineering Task Force, 2012.

[22] T. Ristenpart, E. Tromer, H. Shacham, und S. Savage. Hey, you, get off of my cloud: exploring information leakage in third party compute clouds. In E. Al-Shaer, S. Jha, und
A. D. Keromytis, Editors, *ACM Conference on Computer and Communications Security*, Seiten 199-212. ACM, 2009.

[23] P. Rogaway. Probleme mit der vorgeschlagenen IP-Kryptographie. Unveröffentlichtes Manuskript, 1995. http://www.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt.

[24] J. Salowey, A. Choudhury, und D. McGrew. AES Galois Counter Mode (GCM) Cipher Suites für TLS. RFC 5288 (Vorgeschlagener Standard), 2008.

[25] S. Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In *EUROCRYPT*, pages 534-546, 2002.