Yun Ding

SSL sicher implementieren

Das Protokoll Transport Layer Security (TLS), bekannt auch unter der ursprünglichen Bezeichnung Secure Sockets Layer Protocol (SSL), ist eines der wichtigsten Protokolle für die sichere Kommunikation im Internet – und feiert in diesem Jahr den 20sten Geburtstag. SSL sorgt für die Verschlüsselung von Daten und schützt vor unautorisierter Modifikation. Allerdings wurden auch bei SSL Schwachstellen bekannt: in den Protokollen, SSL-Bibliotheken, Middleware-Komponenten und Anwendungsprogrammen. Der Beitrag gibt einen Überblick, was bei der Nutzung von SSL berücksichtigt werden sollte.

1 SSL in den Schlagzeilen

In den vergangenen Monaten wurden mehrere Schwachstellen von SSL bekannt. Es fing mit dem „Apple go to fail“-Bug im Februar 2014 an, dann folgte der „GnuTLS goto“-Bug im März und schließlich kam im April Heartbleed hinzu, ein OpenSSL-Bug, der als der Super-Gau des Internets bezeichnet wurde.

Im September 2013 wurde darüber spekuliert, ob die NSA eine Backdoor in den NIST-Standard des Pseudozufallsgenerators Dual EC DRBG eingebaut haben könnte. Mit einer solchen Backdoor wäre die NSA in der Lage, Zufallszahlen vorauszusagen. Daraus könnte sie die kryptographischen Schlüssel einer SSL-Kommunikation ableiten, die den Dual EC DRBG verwendet, und damit alle verschlüsselten Nachrichten entschlüsseln.

Und in den Jahren davor waren zahlreiche weitere SSL-Angriffe wie BEAST, CRIME, Lucky 13, RC4 Biases und BREACH bekannt geworden.

In zwei Untersuchungen aus dem Jahr 2012 und 2013[2, 3] fanden Forscher heraus, dass die SSL-Implementierung von 13.500 freien, populären Android-Apps 1.074 (8.0%) und von 697 Apple-Apps 98 (14%) – wahrscheinlich auch heute noch – nicht vor einem Man-in-the-Middle-Angriff (MITM) schützt. Dabei schaltet sich ein Angreifer zwischen die Kommunikationspartnern und kann den Datenverkehr trotz SSL im Klartext mitlesen und ggf. auch modifizieren. Davon waren auch sicherheitskritische Apps beispielsweise für den elektronischen Zahlungsverkehr betroffen.

Eine Anwendung, die SSL nutzt, besteht aus mehreren Ebenen. Die zahlreichen SSL-Schwachstellen lassen sich in die jeweiligen Ebenen einordnen. Dieser Beitrag erläutert die Ursachen für die jeweiligen SSL-Schwachstellen, zeigt ihre Wirkung auf die Sicherheit des Gesamtsystems auf und gibt Empfehlungen für Entwickler, wie sie eine SSL-Anwendung sicher implementieren können.

Dr. Yun Ding

Security Consultant bei der Secorvo Security Consulting GmbH. Ihre Beratungsschwerpunkte sind Kryptografie, Security Engineering und die Sicherheit von Web-, Mobile- und Cloud-Anwendungen.
E-Mail: yun.ding@secorvo.de

2 SSL sicher implementieren

Abbildung 1 zeigt die verschiedenen Ebenen einer auf SSL basierenden Anwendung. Sie lassen sich wie folgt beschreiben:

- *Kryptographische Primitive*: SSL bedient sich einer Vielzahl von kryptographischen Primitiven, beispielsweise (Pseudo-) Zufallsgeneratoren, Hash-, Verschlüsselungs- und Authentifikationsalgorithmen. Der Angriff RC4 Biases nutzte eine Schwachstelle des Verschlüsselungsverfahrens RC4 aus, während BEAST und Lucky 13 auf den Schwachstellen im CBC-Mode des AES-Verschlüsselungsalgorithmus aufsetzten.
- *SSL-Protokolle*: SSL-Protokolle setzen sich aus kryptographischen Primitiven zusammen. Die verschiedenen Verfahren zu Authentifikation, Verschlüsselung und Integritätssicherung werden zu fest definierten Ciphersuites zusammengefasst. Die Protokolle schreiben vor, wie Ciphersuites zwischen den Kommunikationspartnern ausgehandelt und kryptographische Schlüsseln ausgetauscht werden. Protokollerweiterungen wie der Kompressionsalgorithmus oder Heartbeat werden definiert. Die Angriffe CRIME und BREACH basierten auf Schwachstellen des Kompressionsalgorithmus.
- *SSL-Bibliotheken*: SSL-Bibliotheken implementieren die SSL-Protokolle. Die Nutzung von Open Source SSL-Bibliotheken, beispielsweise GnuTLS, OpenSSL, Apple Secure Transport und CyaSSL[1], ist weit verbreitet.
- *SSL-Middleware/Wrapper*: Diese nutzen SSL-Bibliotheken und bieten den Anwendungen sicheren Datentransport in einer höheren Protokollebene (wie z. B. HTTPS).
- *Anwendung*: Sichere Kommunikation ist für Anwendungen wie Online-Banking, Online-Shopping, E-Mail und Messaging essentiell. Die Anwendungen nutzen entweder die APIs der SSL-Middleware oder direkt die APIs der SSL-Bibliotheken.
- *User Interface*: Schließlich gibt es noch die Ebene der Interaktion mit dem Benutzer, z. B. in Gestalt von Warnungen, die

1 http://www.yassl.com/yaSSL/Products-cyassl.html

Abbildung 1 | Ebenen einer auf SSL-basierenden Anwendung

Endanwender	This Connection is Untrusted You have asked Firefox to connect securely to , but we can't confirm that your connection is secure. Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.			
Anwendung	Bank	Shopping	Messaging	Browser
Middleware	Apache HttpClient	cURL	PhoneGap	MKNetworkKit
SSL Bibliotheken	GnuTLS	Apple Secure Transport	OpenSSL	JSSE
SSL Protokolle	Secure Protocols	Cipher Suites	Renegotiation	Compression
Krypto-graphische Primitive	Zufallszahl-generator	Hash	Verschlüsselung	Authentifikation

gerne ‚weggeklickt' werden, um weiter surfen oder arbeiten zu können [7].

Die Sicherheit einer auf SSL basierenden Anwendung ist eine Systemeigenschaft. Die Sicherheit des Gesamtsystems wird durch die Sicherheit jeder einzelnen Komponente und den Abhängigkeiten sowie Interaktionen zwischen diesen bestimmt.

2.1 Abhängigkeiten

Ein sicherheitsrelevanter Fehler in einer unteren Ebene wirkt sich auf alle darüber liegenden Ebenen aus. Damit hängt die Sicherheit der höheren Ebenen von der Sicherheit aller darunterliegenden Ebenen ab. Oft können solche Fehler nicht mehr von den höheren Ebenen behoben werden. Ein Beispiel ist der OpenSSL-Bug Heartbleed: Er gefährdet alle Anwendungen, die OpenSSL direkt oder indirekt über die SSL-Middleware nutzen.

2.2 Annahmen

Ein Anwendungsentwickler geht davon aus, dass die zugrunde liegende SSL-Bibliothek sicher implementiert ist. Ein Entwickler für eine SSL-Bibliothek zweifelt normalerweise nicht an der kryptographischen Sicherheit der SSL-Protokolle und der mathematischen Korrektheit der kryptographischen Primitive. Derartige Annahmen sind notwendig, um die Ebenen einer SSL-Implementierung voneinander abzugrenzen. Der Fokus auf eine Ebene reduziert die Komplexität für den jeweiligen Betrachter.

Wie „Apple goto fail" und Heartbleed zeigten, müssen solche Annahmen auf ihre Richtigkeit überprüft werden. Jedoch ist ein Hinweis wie „Nichts ist sicher – sei paranoid und überprüfe alles" nicht praxistauglich. Aufgrund des erforderlichen Expertenwissens und der Komplexität lassen sich SSL-Bibliotheken und –Middleware nicht durch einzelne Personen überprüfen. Daher ist es wichtig zu verstehen, was essentiell für das Funktionieren von SSL ist (siehe Abschnitt 3).

OWASP empfiehlt bei der Auswahl von SSL-Software die Verwendung von FIPS zertifizierten kryptographischen Modulen [5]. Aber es ist Trugschluss zu glauben, dass zertifizierte SSL-Software sicher und frei von Schwachstellen ist. So hat z. B. das OpenSSL FIPS Object Module die FIPS 140-2 Level 1 Validierung bestanden.[2] Dabei wurden jedoch nur die einzelnen kryptographischen Primitive validiert. Eine Zertifizierung des Zusammenwirkens der kryptographischen Primitive liegt außerhalb des Validierungsbereiches des FIPS 140-2.[3] Allerdings bestehen die SSL-Protokolle gerade aus einer Zusammensetzung von kryptographischen Primitiven; dies macht erst die Funktionalität von SSL aus.

2.3 Interaktionen

SSL-Bibliotheken, -Middleware und die darauf basierenden Anwendungen kommunizieren über APIs. Einerseits nimmt eine Anwendung dadurch die Dienste einer SSL-Middleware oder -Bibliothek in Anspruch. Andererseits kann die Anwendung über die APIs das Sicherheitsverhalten der SSL-Middleware bzw. -Bibliothek anpassen. Die Anpassung kann das gesamte Sicherheitsverhalten sowohl erhöhen (Abschnitt 5.3) als auch schwächen (Abschnitt 4.3).

3 SSL-Zertifikatsprüfung

SSL schützt die Vertraulichkeit der ausgetauschten Nachrichten durch Verschlüsselung. Der dabei verwendete kryptographische Schlüssel wird zwischen den authentifizierten Kommunikationspartnern ausgetauscht. Dieser Schlüssel darf nur dem Sender und dem Empfänger bekannt sein. Die Authentizität der Kommunikationspartner basiert dabei auf Public-Key-Infrastrukturen (PKI); dazu beglaubigt ein Zertifikat die Bindung zwischen einem öffentlichen Schlüssel und seinem legitimen Besitzer (z. B. einer Person oder einer Domain im Internet).

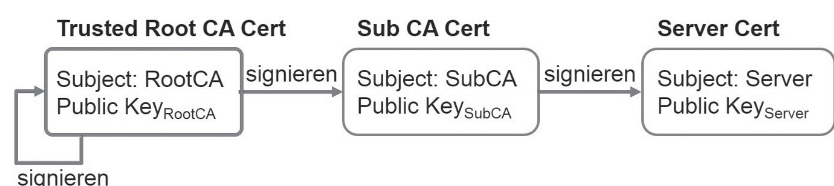
Ein Zertifikat wird mit dem geheimen Schlüssel des Zertifikatsausstellers digital signiert. Diese Signatur wird mit dem öffentlichen Schlüssel des Zertifikatsausstellers überprüft. Um die Authentizität des Ausstellerschlüssels zu prüfen, wird wiederum ein Zertifikat benötigt. Dieses beglaubigt die Echtheit des Ausstellerzertifikats. Auf diese Weise entsteht eine Kette von Zertifikaten (Abbildung 2). Die Echtheit eines Zertifikats wird jeweils mit dem öffentlichen Schlüssel des vorangehenden Zertifikats geprüft. Das allererste Zertifikat ist ein Wurzelzertifikat und stellt den Vertrauensanker der PKI (und damit auch des SSL-Protokolls) dar.

Im Jahr 2011 wurden mehrere Zertifizierungsstellen (z. B. Comodo und DigiNotar) kompromittiert. Dies hat gezeigt, auf welchem wackeligen Fundament u. U. der Vertrauensanker – und damit die gesamte Vertrauenskette – stehen können. Die Verlässlichkeit des Vertrauensankers soll jedoch hier nicht weiter vertieft werden.

2 http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-all.htm#1747

3 http://fips140.blogspot.de/2014/04/openssl-heartbleed-bug-and-fips.html

Abbildung 2 | Zertifikatskette



Die Sicherheitsfunktionen von SSL stehen und fallen mit einer korrekten Zertifikatsprüfung. Dabei muss die gesamte Zertifikatskette überprüft werden:

- *Gültigkeitsprüfung*: Ein Zertifikat ist gültig, wenn 1. der aktuelle Zeitpunkt sich innerhalb des Gültigkeitsbereiches des Zertifikats befindet und 2. das Zertifikat nicht bereits zurückgerufen wurde. Sobald der Verdacht besteht, dass der zugehörige geheime Schlüssel eines Zertifikats (wie im Fall Heartbleed) kompromittiert wurde, muss das Zertifikat zurückgerufen werden. Mit der Kenntnis des geheimen Schlüssels kann ein Angreifer die Verschlüsselungsschlüssel für den zukünftigen Nachrichtenaustausch selber berechnen.
- *Echtheitsprüfung*: Es wird überprüft, ob ein Zertifikat von einer vertrauenswürdigen Zertifizierungsstelle ausgestellt und signiert wurde. Dabei muss die gesamte Zertifikatskette überprüft werden. Wenn keine Echtheitsprüfung stattfindet und praktisch jedes Zertifikat akzeptiert wird, kann ein MITM-Angriff durchgeführt werden. Der Angreifer stellt dabei als Man-in-the-Middle ein (eigenes) Zertifikat zur Verfügung. Das erlaubt ihm, jeweils mit dem Sender und dem Empfänger einen kryptographischen Schlüssel auszutauschen. Der Angreifer kann damit problemlos alle Nachrichten zwischen Sender und Empfänger entschlüsseln, sie nach Belieben verändern und weiterreichen.
- *Self-signed-Zertifikat*: Selbst-signierte Zertifikate stammen entweder von einer öffentlichen Zertifizierungsstelle (Wurzelzertifkat und Vertrauensanker) oder werden von Unternehmen für die unternehmensinterne Kommunikation ausgestellt. Aber auch ein Angreifer kann ein self-signed-Zertifikat ausstellen. Daher müssen alle self-signed-Zertifikate, die kein Wurzelzertifikat sind oder explizit als vertrauenswürdig konfiguriert wurden, bei der Zertifikatsprüfung abgelehnt werden.
- *Hostname-Validierung*: Ein Zertifikat ist einer Person bzw. einer Domain zugeordnet. Bei einer Hostname-Validierung wird geprüft, ob der im Zertifikat eingetragene Domainname mit dem verbundenen Host übereinstimmt. Ohne eine Hostname-Validierung ist es möglich, mit einer anderen Webseite als vorgesehen zu kommunizieren.

4 Was schief gehen kann

Die Ursachen, die in der Vergangenheit zu fehlenden bzw. fehlerhaften Zertifikatsprüfungen geführt haben, waren unterschiedlich. Die Unterschungen [1, 2, 3] analysieren die Ursachen und geben konkrete Bespiele an.

4.1 Codequalität

Bei „Apple goto fail“ und „GnuTLS goto“ haben Programmierfehler dazu geführt, dass die Zertifikatsprüfung im Programmablauf übersprungen wurde.

4.2 Komplexe APIs der SSL Middleware

Das Design der Interaktion zwischen den Komponenten ist mit der Codequalität einer Software eng verbunden. SSL-Protokolle lassen zahlreiche Konfigurationsmöglichkeiten zu. Diese Flexibilität wird von SSL-Bibliotheken und -Middleware in Form von APIs mit zahlreichen Parametern und Optionen an die Anwendungsentwickler weitergereicht. Die Semantik der Parameter und Optionen hat einen hohen Detaillierungsgrad, so dass sie für einen nicht auf Sicherheit spezialisierten Entwickler oft schwer verständlich ist. Ein Beispiel ist das Setzen von Verifikationsparametern mit verschiedenen Modes (z. B. SSL_VERIFY_PEER statt Server- bzw. Client-Authentifikation) und Callback-Funktionen.[4]

Komplexe Beziehungen und Inkonsistenzen zwischen dem Rückgabewert und dem Fehlerstatus in den APIs sorgen zusätzlich für Verwirrung bei Anwendungsentwicklern. Beispielsweise setzt GnuTLS [1] bei einem self-signed-Zertifikat den Fehlerstatus, gibt jedoch den Rückgabewert ‚erfolgreich‘ zurück.

Per default führen OpenSSL, GnuTLS, CyaSSL, JSSE SSLSocketFactory, Apache HttpClient Version 3.* und Python ssl module keine Hostname-Validierung durch. Mit diesem unsicheren Standardverhalten (unsafe defaults) wird die Verantwortung für eine korrekte Zertifikatsprüfung an die Anwendungsentwickler weitergereicht. Jedoch wird das unsichere Verhalten nicht ausreichend dokumentiert, daher wissen Anwendungsentwickler oft nicht, dass sie die Hostname-Validierung selber durchführen müssen.

4.3 Sicherheit steht „im Weg“

Wenn eine SSL-Bibliothek bzw. -Middleware per Standard sicher konfiguriert ist, gibt sie bei einer fehlgeschlagenen Zertifikatsprüfung erwartungsgemäß eine Fehlermeldung zurück. Um eine solche Fehlermeldung während der Entwicklungs- bzw. Testphase temporär aus dem Weg zu räumen, deaktivieren Anwendungsentwickler häufig die Zertifikatsprüfung oder ändern das Standardsicherheitsverhalten der darunter liegenden SSL-Middleware – und versäumen es, dies vor dem Release der Software wieder zu korrigieren.

Moderne Softwarearchitekturen verfügen über die Eigenschaften der Änderbarkeit, Erweiterbarkeit und Testbarkeit. Diese Eigenschaften können genutzt werden, um das Standard-Sicher-

4 https://www.openssl.org/docs/ssl/SSL_CTX_set_verify.html

Abbildung 3 | Angepasster Trust Manager

SSLTest.java | DisableValidationTrustManager.java

```
import javax.net.ssl.*;
import java.security.cert.*;

public class DisableValidationTrustManager implements X509TrustManager {
    public DisableValidationTrustManager() {
    }

    public void checkServerTrusted(java.security.cert.X509Certificate[] p1, String p2) {
          System.out.println("I don't validate any certs!");
          return;
    }
```

SSLTest.java | DisableValidationTrustManager.java

```
TrustManager tm[] = {new DisableValidationTrustManager()};
SSLContext context;
try {
  context = SSLContext.getInstance("TLS");
  context.init(null, tm, null);
} catch (NoSuchAlgorithmException e) {
  e.printStackTrace();
} catch (KeyManagementException e) {
      e.printStackTrace();
}
```

heitsverhalten der SSL-Middleware anzupassen. Der Apache HttpClient bietet beispielsweise für die Hostname-Validierung verschiedene Verifizierer: AllowAllHostnameVerifier, StrictHostnameVerifier und den als Default konfiguierten BrowserCompatHostnameVerifier. AllowAllHostnameVerifier kann konfiguriert werden, um die Hostname-Validierung zu deaktivieren. Abbildung 3 zeigt einen angepassten Trust Manager in Java, der keine Zertifikatsprüfung durchführt.
Wenn die als temporär gedachte Anpassung im Anwendungscode fest kodiert ist, wird es oft vergessen, sie für die Produktion wieder zu entfernen. Dies ist eine häufige Ursache für fehlende oder fehlerhafte Zertifikatsprüfung. Deswegen sollten Testartefakte vom Produktionscode entkoppelt werden (siehe Abschnitt 5.2).

5 Wie macht man es richtig?

5.1 SSL-Middleware sicher einsetzen

In Abschnitt 4.2 haben wir auf die Gefahren von komplexen, z. T. inkonsistenten APIs sowie von unsicherem Standardverhalten der SSL-Middleware und –Bibliotheken hingewiesen. Bei der Verwendung von SSL-Middleware muss die API-Dokumentation (ggf. sogar der betroffenen Abschnitt des Quellcodes) genau verstanden werden. Außerdem kann sich das Standardsicherheitsverhalten der SSL-Bibliotheken und –Middleware von Version zu Version ändern.

Statt sich auf das Standardverhalten zu verlassen, sollten daher die Sicherheitsparameter explizit gesetzt werden. Dabei müssen insbesondere die folgenden Eigenschaften in der SSL-Middleware und –Bibliothek überprüft werden:

- Wird die Zertifikatsprüfung für die gesamte Zertifikatskette durchgeführt?
- Werden self-signed-Zertifikate akzeptiert?
- Wird eine Hostname-Validierung durchgeführt?

5.2 Testartefakte von Produktion entkoppeln

Self-signed-Testzertifikate und angepasste, reduzierte Zertifikatsprüfungen während der Entwicklungs- und Testphase sind üblich und legitim. Statt die unsichere Variante fest in den Anwendungscode zu kodieren, sollte sie vom Produktionscode entkoppelt werden.

So können gängige Software-Entwurfsmuster (design patterns) verwendet werden, die die Software änderbar und testbar machen. Beispielsweise wird bei der Verwendung des Entwurfsmusters „Abstract Factory" die Erzeugung bzw. Anbindung von konkreten Trust-Managern oder Hostname-Verifizierern nicht festgelegt. Angepasste Trust-Manager und Hostname-Verifizierer für den Testzweck werden damit vom Produktionscode isoliert und können über eine gesonderte Testkonfiguration eingebunden werden. Separate Schlüsselspeicher mit Testzertifikaten sollten ebenfalls verwendet werden.

5.3 Anpassung für mehr Sicherheit

Die Eigenschaften einer SSL-Middleware bzw. –Bibliothek, wie die Änderbarkeit, Erweiterbarkeit und Testbarkeit, können auch so genutzt werden, dass die Sicherheit erhöht wird. Mit „SSL Pinning" beispielsweise wird im Vorfeld eine Bindung zwischen einem Host/Service und dessen Zertifikat (bzw. Public Key) hergestellt. Die SSL-Middleware bzw. –Bibliotheken sind dann so eingestellt, dass nur das vorkonfigurierte Zertifikat akzeptiert wird. OWASP stellt hierzu Codebeispiele für verschiedene Plattformen zur Verfügung.[5]

5.4 SSL sicher konfigurieren

Die verschiedenen Optionen, die die SSL-Protokolle zulassen, können sowohl im Programm als auch über Konfigurationseinstellungen gewählt werden. Die Zertifikatsprüfung ist nur eine von vielen Konfigurationsmöglichkeiten. Weitere Konfigurationen beziehen sich auf die SSL-Protokolle, Ciphersuites, Renegotiation, Kompression usw.

Für die SSL-Middleware bzw. –Bibliotheken sind folgende Konfigurationseinstellungen zu empfehlen:

- Nur sichere SSL-Protokolle wie TLS v1.2, TLS v1.1, TLS 1.0 verwenden
- Nur sichere Ciphersuites verwenden, die sowohl Authentifikation als auch Verschlüsselung (mit einer Schlüssellänge ≥ 128 Bit) unterstützen. Insbesondere ist die Ciphersuite für Forward Secrecy[6] zu empfehlen. Unsichere Ciphersuites wie anonymes DH, Nullcipher, RC4 und 3DES sind zu vermeiden
- RSA- bzw. DSA-Schlüssel müssen mindestens 1024 Bit lang sein
- Deaktivieren der clientseitigen Renegotiation
- Deaktivieren der TLS-Kompression

Auf Anwendungsebene sind die folgenden Konfigurationseinstellungen zu empfehlen:

- Keine Vermischung von TLS- und Nicht-TLS-Inhalten auf einer Webseite
- Markieren aller Cookies als „Secure"
- Verwenden von HTTP Strict Transport Security
- Kein Caching von sensitiven Inhalten

Weitere Best Practices zur sicheren SSL-Konfiguration finden sich in [4] und [5].

5.5 SSL testen

Bevor eine SSL-Anwendung eingesetzt wird, muss sie daraufhin getestet werden, wie sie auf ungültige, self-signed oder fremde (d. h. einer anderen Domäne zugeordnete) Zertifikate reagiert. Werkzeuge wie sslsniff[7] und mitmproxy[8] können dabei zur Prüfung eingesetzt werden.

Während der Entwicklungs- und Testphase werden Testschlüssel und Testzertifikate benötigt. Neben Java Keytool und der OpenSSL-Bibliothek, die sehr mächtig, aber zugleich komplex ist, können Open Source Tools wie Xca[9] und gnoMint[10] verwendet werden. Xca basiert auf OpenSSL, während gnoMint auf GnuTLS basiert. Beide bieten leichtbedienbare graphische Benutzerschnittstellen an.

5 https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning#Examples_of_Pinning
6 Zu Perfect Forward Secrecy (PFS) siehe Fox, Gateway, DuD 11/2013, S. 729.
7 http://thoughtcrime.org/software/sslsniff/
8 http://mitmproxy.org/
9 http://sourceforge.net/projects/xca/
10 http://gnomint.sourceforge.net/

Auf der OWASP-Webseite [6] sind zahlreiche Werkzeuge zu finden, die unsichere SSL-Konfigurationen aufspüren.

6 Fazit

Weitere SSL-Schwachstellen werden auch in Zukunft entdeckt werden. Die Zertifizierung einer SSL-Middleware schafft zwar Vertrauen, ist aber keine Garantie für ihre Fehlerfreiheit. Wie bei jeder Zertifizierung ist es hilfreich, nachzulesen, was eigentlich zertifiziert wurde. Folgende Checkliste fasst für Entwickler kurz zusammen, wie sie eine SSL-Anwendung sicher bzw. sicherer implementieren können:

- *Zertifikatsprüfung korrekt durchführen*
  - Die Zertifikatsprüfung darf für die Produktion keinesfalls deaktiviert werden.
  - Die gesamte Zertifikatskette überprüfen.
  - Self-signed-Zertifikate (mit der Ausnahme eines Wurzelzertifikats) ablehnen.
  - Hostnamen-Validierung durchführen.
- *SSL-Middleware sicher einsetzen*
  - Sicherheitsparameter für SSL-Middleware immer explizit setzen.
  - Auf komplexe APIs der SSL-Middleware achtgeben und die API-Dokumentation verstehen.
  - Überprüfen, ob die Zertifikatsprüfung der Middleware korrekt durchgeführt wurde.
- *Unsichere Testartefakte von Produktionscode entkoppeln*
- *SSL-Anwendungen und –Konfigurationen ausreichend testen:* zahlreiche Werkzeuge bieten Unterstützungen.

SSL sicher zu implementieren ist eine komplexe Aufgabe. Sie kann aus der Sicht des Systems Security Engineerings betrachtet werden. Die Sicherheit eines auf SSL basierenden Systems ist das Produkt der Interaktionen zwischen den beteiligten Systemkomponenten: Das Gesamtsystem ist nur so sicher wie das schwächste Glied in dieser Kette. Die Interaktionen (d. h. die APIs, über die die Komponenten aufeinander zugreifen) müssen so entworfen werden, dass die Semantik auf einer geeigneten Abstraktionsebene konsistent ist.

Bei einem System macht jede Komponente Annahmen über die Komponenten, mit denen sie interagiert. Allein die Annahme, dass eine andere Komponente angemessene Sicherheitsqualitäten besitzt, gewährleistet keine Sicherheit und ist somit nicht haltbar. Die Sicherheit anderer Komponenten muss überprüft werden.

So werden Annahmen missbraucht, um Verantwortung zu verschieben. Beispielsweise verlagert eine SSL-Middleware bei unvollständigen Zertifikatsprüfung die Verantwortung an die Anwendungsebene. Ein Anwendungsentwickler wiederum verschiebt über Warnmeldungen die Verantwortung an die Anwender. Die Hoffnung, dass ein Anwender dann richtig entscheidet, erfüllt sich leider meist nicht.

Die Implementierung von SSL ist interdisziplinär. Security-Experten, Entwickler, Anwender und Angreifer sind involviert. Dabei besteht zwischen diesen Gemeinden oft eine große Kluft. Das fehlende Verständnis, wie SSL funktioniert und insbesondere welche Bedeutung die korrekte Zertifikatsprüfung dabei spielt, ist eine wesentliche Ursache für die Schwachstellen in SSL-basierten Anwendungen.

SSL zu implementieren ist nicht nur eine technische Herausforderung. Neben dem Secure Coding und den aktuellen Diskussionen um die Sicherheit der Open Source Software spielen Usability, Psychologie und auch Politik (insbesondere bei den Zertifizierungsstellen) eine entscheidende Rolle.

Referenzen

[1] M. Georgiev, S. Iyengar, S. Jana et al., „The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software", 2012, http://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf

[2] S. Fahl, M. Harbach, L. Baumgaertner and B. Freisleben, „Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security", 2012, http://www2.dcsec.uni-hannover.de/files/android/p50-fahl.pdf

[3] S. Fahl, M. Harbach, H. Perl et al., „Rethinking SSL Development in an Applied World", 2013, http://android-ssl.org/files/p49.pdf

[4] Qualys SSL Labs: „SSL/TLS Deployment Best Practices" https://www.ssllabs.com/downloads/SSL_TLS_Deployment_Best_Practices_1.3.pdf

[5] OWASP: „OWASP Transport Layer Protection Cheat Sheet" https://owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

[6] OWASP: „Testing for Weak SSL/TLS Ciphers, Insufficient Transport Layer Protection (OWASP-EN-002)" https://www.owasp.org/index.php/Testing_for_Weak_SSL/TSL_Ciphers,_Insufficient_Transport_Layer_Protection_%28OWASP-EN-002%29

[7] J. Sunshine, S. Egelman, H. Almuhimedi et al., „Crying Wolf: An Empirical Study of SSL Warning Effectiveness". In: Proceedings of the 18th USENIX Security Symposium, 2009.