ber hinter Java bleibt. Vor allem sollte Oracle aber in seiner Kommunikation klarer werden und transparenter machen, was denn die Pläne und Visionen in Bezug auf Java sind. Dann ließen sich deren Tätigkeiten und Aktionen besser beurteilen und einordnen sowie die Zusammenarbeit mit der Community leichter planen.

Wie sollte sich die Community gegenüber Oracle verhalten?

Manuel Mauky: Die Community sollte auch weiterhin selbstbewusst und kritisch die Entwicklung von Java begleiten und hinterfragen. Wir sollten aber auch wohlwollend und kompromissbereit sein. Ich glaube, dass ein übermäßiges Misstrauen gegenüber allem, was Oracle macht, das ich in Teilen der Community wahrnehme, uns nicht weiterbringt.

Manuel Mauky
manuel.mauky@saxsys.de

Manuel hat in Görlitz Informatik studiert und arbeitet seitdem bei der Saxonia Systems AG als Software-Entwickler und Architekt. Er beschäftigt sich mit allen Aspekten der Anwendungsentwicklung, mit einem Fokus auf das Frontend. Daneben interessiert er sich vor allem für die funktionale Programmierung und ist in zahlreichen Open-Source-Projekten aktiv. Er ist einer der Gründer und Organisatoren der JUG Görlitz und hält regelmäßig Vorträge auf verschiedenen Konferenzen und bei Usergroups.

Crypto 101

Oliver Milke, TRIOLOGY GmbH

Kryptographie ist ein sehr umfangreiches und vielschichtiges Thema, das viel Erfahrung und Wissen erfordert. Entwickler kommen im Alltag immer mal wieder mit Kryptographie in Berührung, insbesondere im DevOps-Umfeld. Um die Artikellänge nicht zu sprengen und beim Wesentlichen zu bleiben, möchte ich mit diesem Artikel praxisrelevante Grundlagen für die tägliche Entwicklungsarbeit schaffen, ohne dabei in die Tiefen von Security abzutauchen. Ich habe mir speziell jene Dinge vorgenommen, über die ich selbst während meiner Einarbeitungszeit in der Abteilung für Security der mobilen Online-Dienste bei VW gestolpert bin.

Kryptographie befasst sich mit Informationssicherheit und ist ein Teilbereich der Kryptologie. Ein weiterer Teilbereich ist die Krypto-Analyse, deren Ziel es unter anderem ist, Schwächen in bestehenden kryptographischen Verfahren aufzudecken oder den gebotenen Schutz zu quantifizieren. Sicherheit („Security", „secure") wiederum umfasst neben Kryptographie noch weitere Aspekte wie Prozesse und den Umgang mit den Methoden der Kryptographie sowie die Bewusstheit solcher Prozesse. Der Form halber ergänze ich hier noch, dass Sicherheit „(Safety", „safe") eher Unversehrtheit beschreibt und demnach zu einem ganz anderen Bereich gehört. Die Unterscheidung

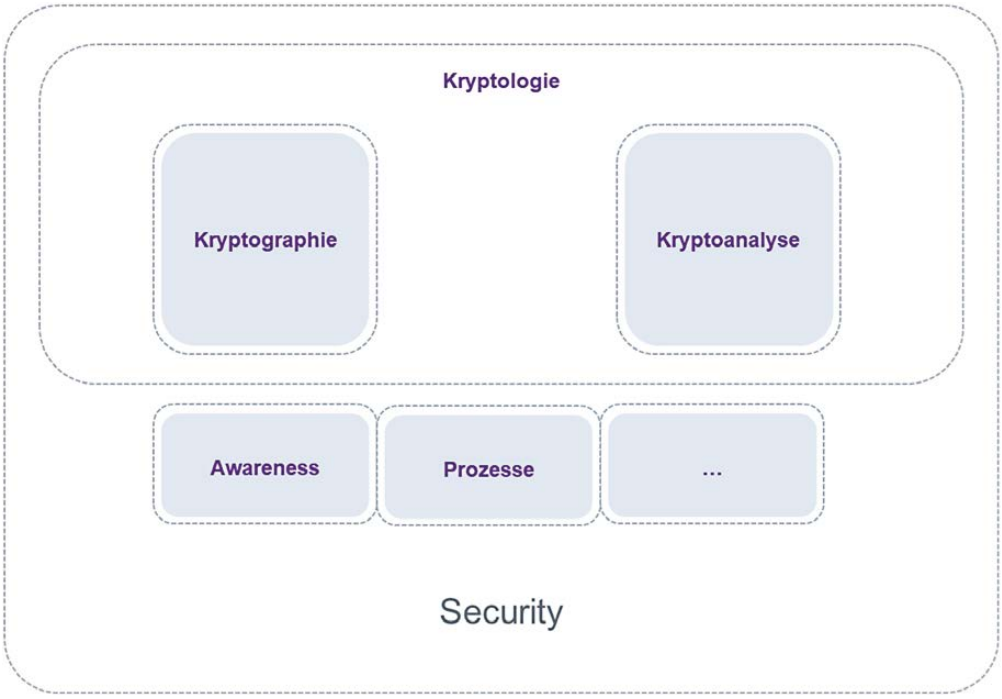

Abbildung 1: Einordnung der Kryptographie

ist im Deutschen leider nicht so deutlich *(siehe Abbildung 1)*.

Kryptographie wird eingesetzt, um drei Ziele zu erreichen:

- *Vertraulichkeit*
  Die Gewissheit, dass Nachrichten nicht mitgelesen werden
- *Integrität*
  Die Gewissheit, dass Nachrichten nicht verändert sind
- *Authentizität*
  Die Gewissheit, dass Nachrichten von einem mir bekannten Absender stammen

Kryptographische Primitive

Die im vorhergehenden Abschnitt dargestellten Ziele werden durch verschiedene Operationen, die sogenannten kryptographischen Primitive, ermöglicht. Man kann sie sich als die Grundrechenarten der Kryptographie vorstellen:

- *Kryptographischer Hash*
  Einwegfunktionen, die neben weiteren Eigenschaften vor allem kollisionsresistent sind.
- *Symmetrische Verschlüsselung*
  Verschlüsselungsverfahren, bei denen derselbe Schlüssel sowohl für die Verschlüsselung als auch für die Entschlüsselung verwendet wird. Diese Verfahren sind typischerweise sehr schnell.
- *Asymmetrische Verschlüsselung*
  Verschlüsselungsverfahren, bei denen es zwei voneinander abhängige Schlüssel gibt, wobei der jeweils andere Schlüssel die Umkehr einer Operation erlaubt, auch Public-Key-Verfahren genannt. Diese Verfahren sind typischerweise langsamer als symmetrische Verfahren.
- *Digitale Signatur*
  Zur Sicherstellung von Authentizität und Integrität einer Nachricht. Vereinfacht dargestellt handelt es sich um die asymmetrische Verschlüsselung des Hashs einer Nachricht.
- *Kryptographisch sichere Zufallszahlengenerierung*
  Man kann sich leicht vorstellen, dass echter Zufall durch eine deterministische Maschine nicht leistbar ist. Dennoch benutzen kryptographische Verfahren Noncen (Zahlen, die nur ein einziges Mal eingesetzt werden dürfen) oder Initialisierungsvektoren (Startblock bei Block-Chiffren). Sie werden typischerweise zufällig generiert und haben somit einen Einfluss auf die Sicherheit anderer Operationen.
- *Weitere*
  Es gibt noch weitere Primitive, auf die ich hier nicht eingehen will. Einen Einstiegspunkt für Interessierte bietet der gleichnamige Wikipedia-Artikel *(siehe „https://en.wikipedia.org/wiki/Cryptographic_primitive")*.

Die Kombination solcher Primitive nennt man „Krypto-System". Die Art und Weise der Kombination hängt vom Anwendungsfall ab, zum Beispiel Transport-Layer-Security (TLS) oder Secure Shell (SSH).

Sicherheit von Kryptographie

Die Sicherheit von kryptographischen Verfahren basiert zu großen Teilen auf Einwegfunktionen. Wie der Name bereits suggeriert, handelt es sich um Funktionen, zu denen (noch) keine effiziente Umkehrfunktion vorhanden ist. Beispiele für Einwegfunktionen sind neben den kryptographischen Hash-Funktionen auch die Multiplikation von Primzahlen (Grundlage für RSA) sowie die diskrete Exponentialfunktion in endlichen Körpern oder die Multiplikation von Punkten auf elliptischen Kurven (Grundlage des ElGamal-Verfahrens und des Diffie-Hellman-Verfahrens).

Das Prinzip von Kerckhoff besagt, dass die Sicherheit von kryptographischen Verfahren nicht von der Geheimhaltung des Verfahrens abhängen darf. Vielmehr muss es von der Geheimhaltung des Schlüssels abhängen. Dadurch entsteht die Möglichkeit zur Widerlegung der systematischen Sicherheit. Unsichere Verfahren können infolgedessen vermieden werden. Das Gegenteil bezeichnet man landläufig als „Security By Obscurity".

Schwachstellen in Krypto-Systemen entstehen entweder direkt in der Spezifikation (BEAST, siehe *„https://de.wikipedia.org/wiki/Transport_Layer_Security#BEAST")* eines Verfahrens oder in einer konkreten Implementierung eines Verfahrens (Heart Bleed, siehe *„https://de.wikipedia.org/wiki/Heartbleed").* Implementierungen können außerdem Seitenkanal-Angriffe *(siehe „http://www.cryptofails.com/post/70097430253/crypto-noobs-2-side-channel-attacks")* ermöglichen, indem zum Beispiel aus der benötigten Verarbeitungszeit einer Operation Schlussfolgerungen hergeleitet werden können.

Passwörter sicher speichern

Die von einem Benutzer gewählten Passwörter in Klartext zu speichern, ist ein „No-Go" – das wissen die meisten Entwickler. Aber wie genau speichert man denn Passwörter? Oft habe ich schon den Begriff „Verschlüsselung" in diesem Zusammenhang gehört, wobei das typischerweise eines von zwei Dingen bedeutet. Entweder verkompliziert man die Passwort-Speicherung, weil man jetzt auch noch „das Passwort" für eine Verschlüsselung speichern muss oder Verschlüsselung wird fälschlicherweise verwendet und meint eigentlich „Hashing".

Aufgrund der Tatsache, dass Hashes deterministische Einwegfunktionen sind, sind sie hervorragend geeignet, um Passwörter abzusichern. Allerdings kann man daher einfach Hash-Tabellen bilden, um einmal berechnete Hashes im Zusammenhang mit ihrem Klartext-Ursprung zu speichern. Auf diese Weise entsteht eine Zeit-effiziente Umkehrfunktion. Für den Fall, dass man zum Beispiel eine ganze Datenbank mit gehashten Passwörtern vorliegen hat (etwa durch den Einbruch in ein System), entsteht ein zusätzliches Problem. Rät man Passwörter und hasht sie (teuer), kann man alle vorhandenen Passwörter dagegen prüfen (günstig) und somit in einem Schritt alle Passwörter angreifen.

Abhilfe schafft ein sogenannter „Salt". Salts sind zufällig und pro Passwort individuell zu generieren, zusätzlich mit dem Passwort zu speichern und sie reichern das Passwort vor dem Hashen an. Dadurch unterscheiden sich Passwörter mindestens um ihren Salt, was zwei wesentliche Effekte hat. Zum einen sieht man zwei Datensätzen nicht mehr sofort an, ob es sich um das gleiche Passwort handelt. Zum anderen ist es dadurch nicht mehr möglich, alle Passwörter einer Datenbank gleichzeitig zu überprüfen.

Hash-Funktionen werden jedoch auch zu anderen Zwecken auf deutlich größere Datenmengen angewendet, weshalb eines der verfolgten Entwicklungsziele für manche Hash-Funktionen die Performance ist. Das ist unglaublich praktisch, um den Hash von einem Linux-Image zu berechnen und Passwörter mittels Brute Force stumpf auszuprobieren. Deswegen ist eine wichtige Anforderung an Hash-Funktionen für Passwörter die Langsamkeit. Eine einfache Lösung ist die wiederholte Anwendung einer Hash-Funktion. Die bessere Alternative ist eine Hash-Funktion, bei deren Entwicklung darauf geachtet wurde, dass sie konfigurierbar langsam ist. Wenn eine Hash-Funktion gleichzeitig noch sehr viel Arbeitsspeicher benötigt, sind die Parallelisierung oder der Einsatz von spezieller Hardware unwirtschaftlich.

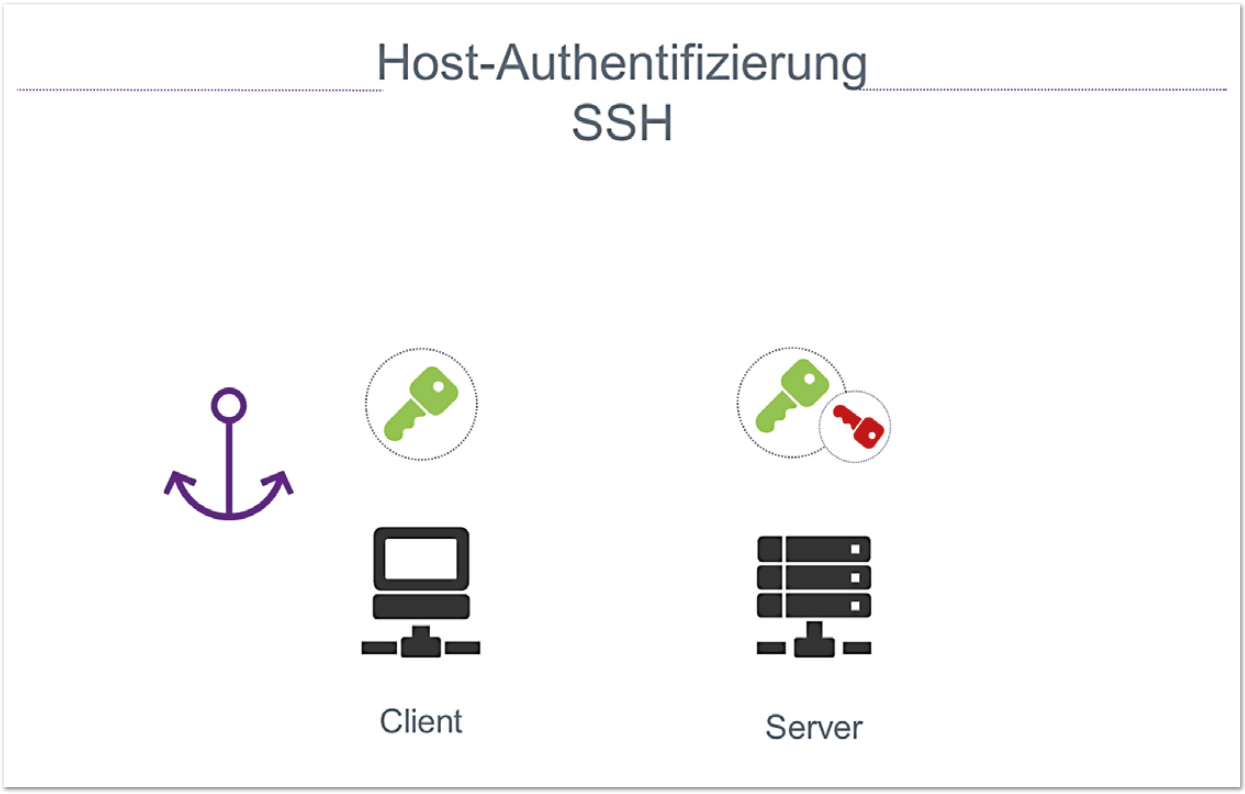

Abbildung 2: Vertrauensanker bei der Host-Authentifizierung bei SSH

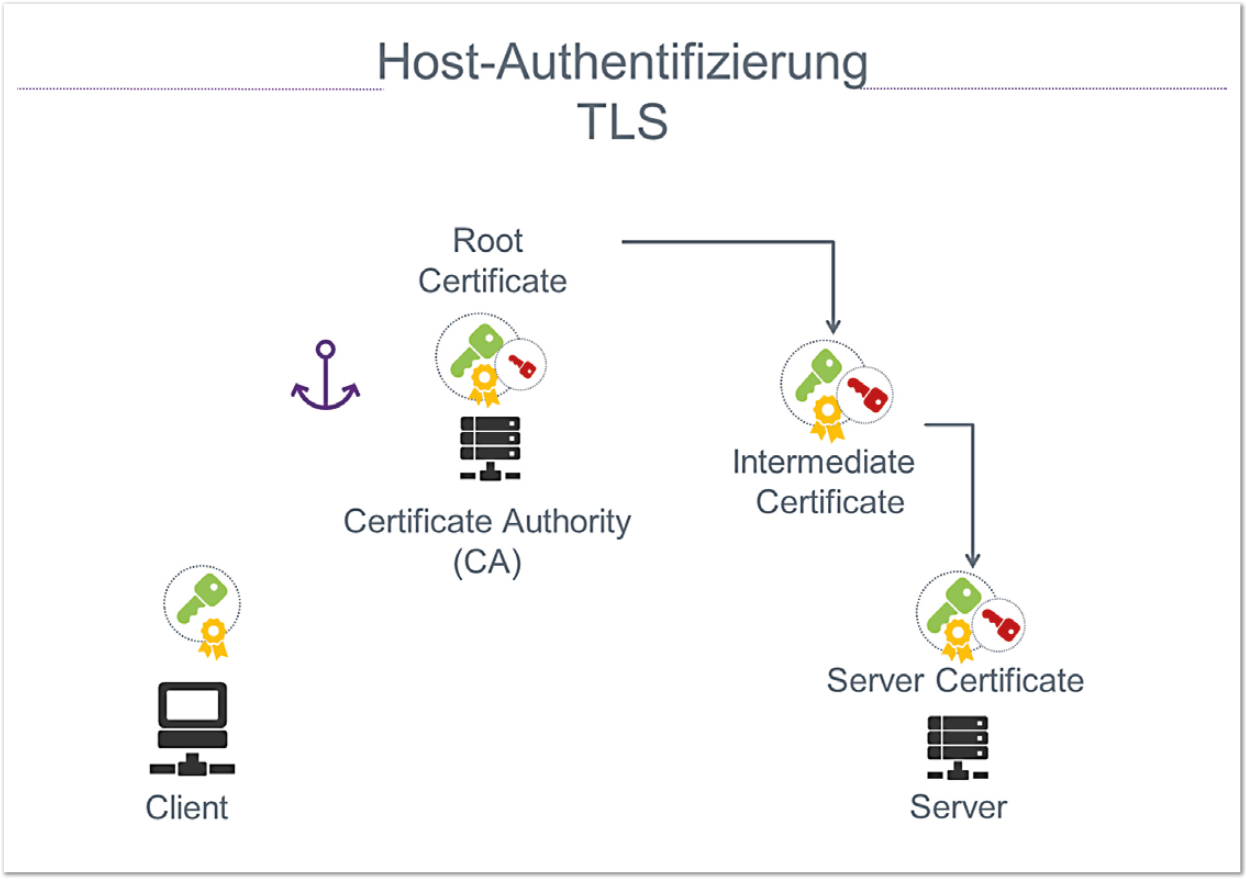

Abbildung 3: Vertrauensanker bei der Host-Authentifizierung bei TLS

Eine für Passwörter geeignete Hash-Funktion ist „bCrypt". Sie erzeugt aus einem Salt einen Kostenfaktor und aus dem Passwort einen String, der sowohl den Salt als auch den Hash enthält. „12" ist ein guter Kostenfaktor für den Moment. Im Security Stack Exchange *(siehe „https://security.stackexchange.com/questions/211/how-to-securely-hash-passwords/31846#31846")* findet ihr eine sehr ausführliche Antwort, die weitere gute Hash-Funktionen vorstellt und eine ausführliche Darstellung bietet.

Host-Authentifizierung bei SSH

Die meisten Entwickler haben sich schon mal per SSH auf einem Server eingeloggt. Bei der ersten Verbindung zu einem SSH-Server wird man typischerweise gefragt, ob man dem Server vertrauen möchte – na klar möchte man das; oder vielleicht doch lieber nicht?

Bei der Verbindungsinitiierung stellt sich der Server vor, indem er seinen öffentlichen Schlüssel bereitstellt. Anhand dessen kann der Client entscheiden, ob er den Verbindungsaufbau fortführen will, weil er den Schlüssel kennt – im Grunde wie die Gästeliste für eine VIP-Party. Im weiteren Verlauf des Handshake beweist der Server, dass er im Besitz des zugehörigen privaten Schlüssels ist *(siehe Abbildung 2)*.

Idealerweise erfolgt der Austausch des öffentlichen Schlüssels über einen anderen Kanal als bei der ersten Verbindung, denn dieser öffentliche Schlüssel ist der Vertrauensanker für die Authentifizierung des Servers gegenüber dem Client. Eine Möglichkeit ist ein USB-Stick. Ansonsten sollte man sich vom Administrator beispielsweise per Telefon den Fingerprint (Hash) des öffentlichen Schlüssels bestätigen lassen, um sicherzugehen, sich nicht mit einem Fake-Server zu verbinden. Das ist beim heimischen Raspberry Pi sicherlich wenig kritisch als bei der Verbindung zu einem Remote-Backup-Server, zu dem das eigene Backup übertragen wird.

Host-Authentifizierung bei TLS

Die Anzahl der SSH-Server, zu denen man sich regelmäßig verbindet, ist überschaubar. Etwas anders gelagert ist die Situation bei Webseiten. Jeden einzelnen öffentlichen Schlüssel einer Webseite vorab zu speichern, ist einfach unpraktikabel. Dennoch besteht der Bedarf zu entscheiden, ob eine Webseite diejenige ist, für die sie sich ausgibt, insbesondere dann, wenn Kreditkarten-Daten oder andere sensible Informationen ausgetauscht werden. An dieser Stelle wird das Konzept von Schlüsselpaaren einfach mehrstufig erweitert und das Ergebnis ist eine Zertifikatskette.

An die Stelle eines öffentlichen Schlüssels für jeden Server tritt ein öffentlicher Schlüssel einer Zertifizierungsstelle („Certificate Authority", CA). Dieser öffentliche Schlüssel ist mit dem zugehörigen privaten Schlüssel signiert und wird „Root-Zertifikat" genannt. Mit genau diesem privaten Schlüssel sind weitere öffentliche Schlüssel signiert, die „Intermediate Certificates". Es gibt typischerweise mehrere Stufen von Intermediate Certificates. Im Wesentlichen wird dadurch das Ziel verfolgt, den zentralen privaten Schlüssel so selten wie möglich aus dem Tresor zu holen. Ein Schlüsselpaar ist entweder zur Ausstellung weiterer Intermediate Certificates oder zur Verwendung für eine Webseite gedacht *(siehe Abbildung 3)*.

Durch diese Verkettung muss der Client nur dem Root-Zertifikat vertrauen und kann dennoch sichergehen, dass der Webserver der richtige ist. Zusammengefasst bedeutet das, dass ein Zertifikat ein öffentlicher Schlüssel eines asymmetrischen Verfahrens ist, der seinerseits mit einem asymmetrischen Verfahren signiert wurde. Mithilfe eines Certificate Signing Request (CSR) wird um diese Signierung eines öffentlichen Schlüssels gebeten und gleichzeitig der Beweis geführt, dass der Antragsteller im Besitz des zugehörigen privaten Schlüssels ist. Der CSR enthält daneben noch weitere Informationen, beispielsweise die URL der Webseite, für die das Zertifikat genutzt werden soll, und darüber, wer es beantragt.

Cipher Suites am Beispiel von TLS

Anleitungen zu Konfigurationen von Web-Servern enthalten regelmäßig auf den ersten Blick unleserliche Zeichenfolgen, wie zum Beispiel „ECDHE_RSA_WITH_AES_256_CBC_SHA384". Es ist allgemein bekannt, dass TLS-Verbindungen verschlüsselt sind. Das konkrete Verschlüsselungsverfahren wird jedoch durch die Cipher Suite festgelegt. Im gewählten Beispiel erfolgt die Verschlüsselung mit AES (CBC-Modus, 256 Bit Schlüssel). Zusätzlich wird ein HMAC auf Basis von SHA-384 gebildet, der die Integrität und Authentizität der vertraulichen Inhalte sichergestellt.

Woher kommt jedoch der verwendete Schlüssel, der dem Server und dem Client vorliegen muss? Er wird mithilfe des Diffie-Hellman-Verfahrens auf Basis der diskreten Exponentialfunktion auf elliptischen Kurven bestimmt. Das mathematische Verfahren ermöglicht es, dass sich zwei Parteien über einen unsicheren Übertragungsweg dennoch auf einen wirklich geheimen Schlüssel einigen können. Weiterhin überprüft der Browser mit dem RSA-Verfahren, ob der Server wirklich derjenige ist, für den er sich mit seinem Zertifikat ausgibt. An dieser Stelle muss der Server unter Beweis stellen, dass er im Besitz des passenden privaten Schlüssels ist. Schlüsselgenerierung und Zertifikatsprüfung finden als Teil des TLS-Handshake statt und sind die Grundlage für die anschließend symmetrisch verschlüsselte Kommunikation.

Ganz allgemein ausgedrückt, legt eine Cipher Suite fest, welche kryptographischen Primitive in einem Protokoll zum Einsatz kommen. Welche Elemente nötig sind, hängt dann wiederum vom Protokoll ab, deswegen unterscheiden sich zum Beispiel die Cipher Suites zwischen TLS 1.2 und TLS 1.3.

Weiterführendes Material und Tools

Unabhängig von der Frage, ob Passwörter durch die Anwendung hinreichend sicher gespeichert sind, besteht meine Empfehlung, keine schwachen Passwörter zu erlauben und starke Passwörter zu ermöglichen (unter anderem nicht auf zwölf Zeichen zu begrenzen). Für eine einfache Realisierung von Passwort-Policies in Java eignet sich Passay *(siehe „http://www.passay.org")*. Wenn Kryptographie Teil des Anwendungsfalls ist, ist man mit Bouncy Castle *(siehe „http://www.bouncycastle.org/java.html")* gut bedient. Darin ist auch eine Implementierung von bCrypt enthalten.

Einen niederschwelligen Einstieg in die sichere Konfiguration von gängigen Webservern bietet der Mozilla-Config-Generator *(siehe „https://mozilla.github.io/server-side-tls/ssl-config-generator")*. Die Konfiguration kann anschließend mit dem Qualys-SSL-Lab-Server-Test *(siehe „https://www.ssllabs.com/ssltest")* bewertet werden. Diese Überprüfung lässt sich übrigens auch dank des offiziellen API *(siehe „https://www.ssllabs.com/projects/ssllabs-apis")* prima mit dem gewünschten Monitoring-Tool automatisieren.

Bei der Entscheidung, welches Verfahren eingesetzt werden kann und wie viele Bit ein Schlüssel haben sollte, hilft keylength.com *(siehe „https://www.keylength.com")*. Hier sind unter anderem die Empfehlungen von NIST und BSI zusammengetragen.

Im Blog von Bruce Schneier *(siehe „https://www.schneier.com")* findet ihr viel zum Thema „Security" aus der Perspektive eines sehr erfahrenen Kryptographen.

Fazit

Die grundlegenden Konzepte der hier vorgestellten kryptographischen Aspekte lassen sich gut überblicken. Ich habe bewusst viele Details ausgelassen, um einen Überblick zu ermöglichen. Eine detaillierte Betrachtung wird schnell sehr umfangreich. Die Folien zum Vortrag für diesen Artikel sind auf meinem Speaker Deck *(siehe „https://speakerdeck.com/omilke/crypto-101")* verfügbar.

Für den Umgang mit Kryptographie möchte ich euch noch zwei Direktiven ans Herz legen: Erstens, selbstgemacht ist schlecht, und zweitens, neu ist schlecht. Dabei geht es vor allem darum, dass die eingesetzten Verfahren gründlich geprüft sind, was typischerweise weder bei neuen noch bei selbstgemachten Verfahren der Fall ist. Die zur Verfügung gestellten Links helfen, mit Aufwand sichere Webserver bereitzustellen.

Oliver Milke
dev@oliver-milke.de

Oliver Milke (siehe „http://oliver-milke.de") arbeitet als Software Craftsman bei der TRIOLOGY GmbH und entwickelt dort am Cloudogu EcoSystem. Seit mehr als zehn Jahren ist Software-Entwicklung sein tägliches Brot und mehr als nur ein Beruf. Mit Security und Kryptographie ist er intensiv im Bereich „Mobile Online-Dienste" bei Volkswagen in Berührung gekommen. Er ist Mit-Organisator der JUG-Ostfalen und verbringt seine Freizeit am liebsten mit Sport.