Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Härtung von WireGuard durch Hardware-Sicherheitsmodule gegen Quantenangriffe

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science

eingereicht von: Daniel Busch
geboren am:
geboren in:

Gutachter/innen: Prof. Dr. Jens-Peter Redlich
Prof. Dr. Florian Tschorsch

eingereicht am: .....

Inhaltsverzeichnis

1 Einführung

Während das Aufkommen von Quantencomputern Forschung und Industrie gleichermaßen dazu bewegt, neue quantensichere Alternativen zur aktuellen asymmetrischen Kryptoprimitiven zu erwägen, mangelt es den bisher entwickelten Techniken zu Schlüsselkapselung an Sicherheit oder Praktikabilität. Dies wirkt sich auch auf das aufkommende Virtual Private Network-Protokoll (VPN) WireGuard aus, das zum Schutz privater Verbindungen über einen öffentlichen, unsicheren Kanal wie das Internet verwendet wird. Um dies zu mitigieren, wollen wir die Option, einen symmetrischen Preshared Key zu verwenden, die in WireGuard eingebaut ist, verwenden und aufrüsten.

1.1 Motivation

VPN-Tunnel sind ein wichtiges Werkzeug zur Sicherung von Geheimhaltung und Authentizität bei der Kommunikation über unsichere Netzwerke. Für diese Schutzziele werden in gängigen Lösungen derzeit meistens asymmetrische Kryptoprimitiven verwendet, die gegenüber den derzeit in der Entwicklung befindlichen Quantencomputern nicht sicher sind. Die einsetzende Erkenntnis, dass Quantencomputer langfristig wahrscheinlich alle klassischen asymmetrischen Kryptoprimitiven hinfällig machen werden, kommt für die Entwicklung zukünftiger Systeme noch keinesfalls zu spät, da die bisher entwickelten Quantencomputer noch nicht die nötige Rechenleistung besitzen. Um jedoch endgültig durch quantensichere Primitiven Abhilfe zu schaffen, die von Quantencomputern nicht auf solche Weise gebrochen werden können, sind zunächst aufwendige Standardisierungs- und Überprüfungsprozesse nötig, denen dann eine praxissichere Implementierung und schlussendlich breite Verwendung bei den Endnutzern folgen sollte. Hierbei kann man höchstens die ersten beiden Schritte als abgeschlossen ansehen [13], und die Reife von Softwaresystemen benötigt, besonders im Bereich der sicheren Kommunikation, meistens Jahre bis Jahrzehnte. In der Zwischenzeit kann ein einfacher, wenn auch etwas uneleganter Ansatz als Übergangslösung hilfreich sein, um gegen Store-now-Decrypt-Later-Angriffe zu schützen: Hierbei zeichnet ein Angreifer bereits jetzt verschlüsselte Kommunikation auf, die in Zukunft durch einen Quantencomputer gebrochen werden kann.

Wir wollen uns insbesondere mit dem aufstrebenden VPN-Protokoll WireGuard befassen, um die Möglichkeit einer solchen Lösung darin zu demonstrieren. Sie sollte sowohl mit derzeitigen WireGuard-Endpunkten kompatibel sein als auch die Sicherheit gegenüber Attacken mithilfe von Quantencomputern erhöhen: Nicht indem Postquantenkryptographie verwendet wird, sondern

indem der im Protokoll verwendete symmetrische Preshared Key besser geschützt und dessen Verwendung gefördert wird.

1.2 Ziele

Im Folgenden soll WireGuard mithilfe eines Hardwaresicherheitsmoduls so erweitert werden, dass der in der Schlüsselvereinbarung verwendete Preshared Key gegen Akteure gesichert wird, die die Ressourcen für einen breitflächigen Angriff mittels eines eventuell entwickelten Quantencomputers besitzen. Die Entschlüsselung bereits erfolgter Kommunikation soll auch nicht mehr möglich sein, wenn einer der beiden Tunnelendpunkte später kompromittiert wird. In diesem Falle kann der Preshared Key momentan einfach aus der Konfiguration ausgelesen werden, was sich mithilfe eines Hardwaresicherheitsmoduls verhindern lässt, indem man dieses so konfiguriert, dass der Preshared Key zwar zur Schlüsselableitung verwendet, aber nicht ausgelesen werden kann. Das Hardwaresicherheitsmodul ist dann natürlich so konstruiert, dass es auch physischen Versuchen, den Schlüssel daraus zu extrahieren widersteht, und kann so die Vorwärtssicherheit der getunnelten Pakete schützen.

1.3 Struktur

Die weitere Arbeit ist wie folgt aufgebaut: In Abschnitt 2 wird zunächst ein Überblick über die relevanten Technologien gegeben, beginnend mit dem Noise-Protokollframework, auf dem WireGuard aufbaut, dann WireGuard selbst, und zuletzt die PKCS#11-Schnittstelle für Hardwaresicherheitsmodule. Die relevanten kryptographischen Primitiven werden mit Blick auf ihre Verwendung in WireGuard diskutiert. Abschnitt 3 beschreibt unser Angreifermodell und die Herangehensweise bei der Umsetzung sowie die dabei zugrunde liegenden Designentscheidungen. In Abschnitt 4 wird dann der eigentliche Prozess der Modifikation der Komponenten beschrieben und die wichtigsten Codeänderungen sowie die getroffenen Designentscheidungen dargelegt. In Abschnitt 5 folgen die Ergebnisse in Bezug auf Funktionalität und Leistung, und es wird auf die Kompromisse eingegangen, die sich daraus ergeben. Der letzte Abschnitt fasst das Erreichte schließlich zusammen und gibt Empfehlungen für zukünftige Entwicklung zu diesem Thema sowie einen Ausblick auf Bereiche, in denen verwandte Arbeit möglich wäre.

2 Hintergrund

Bevor wir mit der eigentlichen Implementierung beginnen, erinnern wir zunächst an die relevanten Komponenten, für die wir am Ende das gewünschte Zusammenspiel erzielen wollen.

2.1 Noise-Protokollframework

Noise ist ein Framework zum Bau von kryptographischen Netzwerkprotokollen, die auf einer gemeinsamen Basis von Kryptoprimitiven beruhen [9]. Durch seine im Vergleich zu gängigen Verfahren wie TLS geringe Komplexität will Noise das Potenzial für Schwachstellen möglichst gering halten. Eine konkrete Instanz von Noise besteht aus zwei Teilen: Einerseits ist ein Bündel von Kryptoprimitiven für die Anwendungsfälle Hashing, Diffie-Hellman-Schlüsselaustausch und authentisierte, symmetrische Verschlüsselung, also ein AEAD-Verfahren nötig, die im Protokoll verwendet werden sollen. Andererseits muss festgelegt werden, welche Sicherheitsgarantien im Handshake benötigt werden. Solche Sicherheitsgarantien erfordern eine korrespondierende Reihe an durchzuführenden Diffie-Hellman-Schlüsselberechnungen, die diese Garantien bieten sollen. Der Diffie-Hellman-Schlüsselaustausch ist jedoch Quantencomputern gegenüber schwach, da diese das diskrete Logarithmusproblem effizient lösen und damit die privaten Schlüssel des Verfahrens finden können. Als zusätzliche Option gibt es in Noise die Möglichkeit zur Einbindung eines zuvor vereinbarten symmetrischen Schlüssels, dem sogenannten „Preshared Key“. Diese Option erhöht nach dem derzeitigen Wissensstand die Sicherheit gegenüber Angreifern im Besitz eines Quantencomputers, da ein solcher zwar mithilfe der Grover-Suche die Sicherheitsmarge des Preshared Keys effektiv halbieren kann [6], was aber immer noch keine praktische Durchführbarkeit einer Schlüsselsuche bei ausreichender Sicherheitsmarge bedeutet. Das Protokoll von WireGuard ist im Kern eine Noise-Protokollinstanz, und wir wollen versuchen, diese unter Verwendung des Preshared Keys weiter zu sichern.

2.2 WireGuard VPN

WireGuard ist ein aufstrebendes VPN-Protokoll mit Fokus auf Einfachheit und Modernität. Besonders zuträglich ist der Performance von WireGuard neben dem kompakten Protokoll eine Implementierung im Linux-Kernel, was den sonst üblichen Overhead beim Routing von VPN-Verkehr im Userland umgeht. Da das Kernelmodul von WireGuard weiterhin aus nur etwa 6000 Zeilen Code besteht, hebt Autor Jason Donenfeld dafür auch hohe Implementie-

rungssicherheit durch gute Auditierbarkeit hervor [4]. Der Operationsmodus von WireGuard orientiert sich an Noise und kann grob in zwei Zustände unterteilt werden, den Handshake und das tatsächliche Tunneln von Paketverkehr. Bevor Pakete durch einen VPN-Tunnel geleitet werden können, muss dieser zunächst aufgebaut werden, wofür der Noise-Handshake zuständig ist. Im Handshake wird das Schlüsselmaterial für den aufzubauenden Tunnel ausgetauscht, woraus letztlich die Schlüssel für die Tunnelpakete abgeleitet werden. WireGuard erneuert im Sinne der Vorwärtssicherheit regelmäßig die Verbindung durch wiederholte Handshakes, was normalerweise etwa alle zwei Minuten geschieht. Somit werden nach Verbindungsaufbau alle zwei Minuten neue Schlüssel ausgetauscht und Transportschlüssel abgeleitet, was für eine einzelne Verbindung kein nennenswerter Aufwand ist. Auf einem VPN-Server, der für das Tunneln von hunderten oder tausenden Verbindungen zuständig ist, wird jedoch für dieses Auffrischen der Verbindung eine nicht unerhebliche Menge an Rechenzeit benötigt. Dies ist bei der Modifikation einer Implementierung zu beachten, wenn man die Performance nicht außen vor lassen möchte.

2.2.1 Kryptoprimitiven

Das WireGuard-Protokoll verwendet für seine verschiedenen Schutzziele jeweils angemessene Kryptoprimitiven. Zur Schlüsselvereinbarung dient zunächst der Elliptic-Curve-Diffie-Hellman-Algorithmus Curve25519 [2]. Dieser ist seit nun mehr als einem Jahrzehnt ein de-facto-Standard für schnellen asymmetrischen Schlüsselaustausch und bietet durch seine Einfachheit wenig Angriffsfläche in seinen Implementierungen. WireGuard verwendet den Curve25519-Kryptoprimitive für zwei Zwecke: Einerseits für den eben genannten Schlüsselaustausch, sowohl mit den statischen Schlüsselpaaren der beiden Kommunikationspartner als auch deren zur Laufzeit generierten ephemeren Schlüsselpaaren, und andererseits zur Identifikation und Routing anhand des statischen öffentlichen Schlüssels von anderen Endpunkten. Wohin ein Paket geroutet wird, hängt also direkt damit zusammen, mit welchen statischen Schlüssel die Verbindung aufgebaut ist.

Das mit Curve25519 ausgetauschte Schlüsselmaterial wird natürlich nicht roh verwendet, um WireGuard-Pakete zu verschlüsseln. Dazu werden die ausgetauschten Geheimnisse gemeinsam mit anderen öffentlichen Informationen unter Verwendung einer Schlüsselableitungsfunktion zu einem möglichst gleichmäßig pseudozufälligen und sicheren Transportschlüssel abgeleitet. Diese Schlüsselableitungsfunktion ist HKDF, die HMAC-based Key Derivation Function [7]. Sie verwendet die Diffusionseigenschaften des darunterliegenden

HMAC dazu, die aus dem Schlüsselmaterial eingehende Entropie gleichmäßig in Blöcke von der vom HMAC vorgegebenen Blockgröße zu verteilen. HKDF besteht aus zwei Funktionen, die in WireGuard gemeinsam hintereinander verwendet werden: Die erste Funktion extrahiert mit dem verwendeten HMAC eine fixe Zahl an Bits aus dem Eingabematerial, und die zweite Funktion kann die Entropie aus diesen Bits in mehrere Blöcke von der Ausgabelänge des HMACs durch seine wiederholte Anwendung strecken. Je nach Verwendungszweck kann hier die benötigte Anzahl an Bits aus dem Eingabematerial extrahiert werden, wobei sich das Sicherheitsniveau jedoch insgesamt nicht über das der Eingabe erhöhen kann, da keine neue Entropie gesammelt wird. Die wiederholte Anwendung der HKDF mit dem im Verlauf des Handshakes gesammelten Schlüsselmaterial ergibt schließlich den Schlüssel für den Datentransport.

Nicht nur im HMAC von HKDF, sondern auch zum Integritätsschutz von gesendeten und empfangenen Daten wird von WireGuard die Hashfunktion BLAKE2s aus der BLAKE2-Familie verwendet. BLAKE2 ist der Nachfolger des Finalisten BLAKE aus der NIST Hash Function Competition, und daher wie BLAKE auch auf Performance ausgelegt [1]. WireGuard verwendet die Variante von BLAKE2s mit Schlüssel teilweise auch als MAC für das Cookie-System, aber sonst wird die Variante ohne Schlüssel entweder in der Schlüsselableitung oder als fortlaufender Hash über alle im Handshake bisher gesendeten Daten zum Integritätsschutz verwendet.

Zuletzt kommt zur eigentlichen Transportverschlüsselung sowie zur Authentisierung von versendeten Daten der Algorithmus ChaCha20-Poly1305 zum Einsatz. Als Authenticated-Encryption-with-Additional-Data-Primitive garantiert ChaCha20-Poly1305 nicht nur die Geheimhaltung der verschlüsselten Daten, sondern kann zusätzlich auch deren Authentizität und die eines weiteren Datenstücks, das im Klartext übertragen wird, sichern [8]. Neben der Transportverschlüsselung dient ChaCha20-Poly1305 auch zur Authentisierung der im Handshake errechneten Hashwerte beim Integritätsschutz sowie der authentischen und geheimen Übertragung von Zeitstempeln, die die Frische des Handshakes sichern und damit Replay-Angriffe verhindern sollen.

2.2.2 Handshake

Da der Handshake für uns von besonderem Interesse ist, wird dieser hier in seinen Einzelheiten beschrieben. Der Handshake wurde zur Vermeidung von Latenz so entworfen, dass er in einem Round-Trip, also einem Paket von Initiator zu Empfänger und einem darauf folgenden Paket von Empfänger zu Initiator durchführbar ist. Falls der Empfänger unter Last steht, während er

eine Handshake-Nachricht erhält, gibt es für ihn zur Vermeidung von DoS-Angriffen die Option, eine direkte Beantwortung abzulehnen und stattdessen eine weniger berechnungsintensive Cookie-Antwort zu senden. Da in dieser Cookie-Nachricht jedoch keine Geheimnisse ausgetauscht werden, sondern sie nur den Handshake verschiebt, ist sie für unsere Zwecke von geringer Relevanz und wir betrachten nur die beiden Handshake-Nachrichten.

Bevor der Handshake beginnen kann, müssen unsere beiden Kommunikationspartner Alice und Bob zunächst ihre IP-Addressen gegenseitig kennen, jeweils statische Diffie-Hellman-Schlüsselpaare generiert, und die entstandenen öffentlichen Schlüssel ausgetauscht haben. Alice kann nun den Handshake beginnen, indem sie zusätzlich zum statischen Diffie-Hellman-Schlüsselpaar S_A^{prv}, S_A^{pub} ein weiteres, ephemeres Diffie-Hellman-Schlüsselpaar E_A^{prv}, E_A^{pub} generiert. Sie überträgt nun zur Identifikation der aktuell aufzubauenden Verbindung zunächst eine zufällig gewählte, 4 Byte große Session ID, gefolgt von E_A^{pub}. Nun ist weiter

$$\begin{aligned} C_1 &:= \mathrm{KDF}_1(\text{const.}, E_A^{pub}) \\ (C_2, k_1) &:= \mathrm{KDF}_2(C_1, \mathrm{DH}(E_A^{prv}, S_B^{pub}))) \end{aligned}$$

zu berechnen, um den Schlüssel k_1 zu erhalten, mit dem dann S_A^{pub} verschlüsselt übertragen wird. Hierbei bezeichnet $\mathrm{DH}(\mathrm{prv}, \mathrm{pub})$ das mit einem privaten und einem öffentlichen Schlüssel im Diffie-Hellman-Verfahren abgeleitete Geheimnis und $\mathrm{KDF}_n(\mathrm{Salt}, \mathrm{Input})$ die HKDF-Schlüsselableitung mit BLAKE2s-HMAC, es werden dabei in der Expansionsphase n Blöcke in der Ausgabegröße des HMACs abgeleitet. Zuletzt leitet Alice den Schlüssel k_2 durch

$$(C_3, k_2) := \mathrm{KDF}_2(C_2, \mathrm{DH}(S_A^{prv}, S_B^{pub}))$$

ab und überträgt dann einen frischen Timestamp, der mit k_2 verschlüsselt ist. Es wird nach jeder Berechnung auch ein fortlaufender Hash über alle bisherigen Ergebnisse mitgeführt, der mit den verschlüsselten Daten jeweils nach dem letzten Update im AEAD-Modus authentisiert mitgeliefert wird. Danach werden noch zwei MACs über öffentliche Daten übertragen, die für den Cookie-Mechanismus benötigt werden, was das erste Paket des Handshakes beendet. Beim Empfang dieses Paketes wird Bob nun dieselben Berechnungen durchführen, um seinen internen Zustand mit dem von Alice zu synchronisieren, wobei die Diffie-Hellman-Berechnungen genau reziprok geschehen sollten, um dieselben Geheimnisse zu erhalten.

Nun antwortet Bob auf Alices Handshake-Paket, indem er zunächst ebenfalls ein ephemeres Diffie-Hellman-Schlüsselpaar E_B^{prv}, E_B^{pub} generiert und dann

Alices Session-ID gefolgt von seiner eigenen zufälligen 4-Byte-Session-ID und danach ebenjenem E_B^{pub} überträgt. Nun folgt die andere Hälfte der Schlüsselableitung. Hierzu berechnet Bob:

$$\begin{aligned} C_4 &:= \mathrm{KDF}_1(C_3, E_B^{pub}) \\ C_5 &:= \mathrm{KDF}_1(C_4, \mathrm{DH}(E_B^{prv}, E_A^{pub})) \\ C_6 &:= \mathrm{KDF}_1(C_5, \mathrm{DH}(E_B^{prv}, S_A^{pub})) \\ (C_7, h, k_3) &:= \mathrm{KDF}_3(C_6, Q), \end{aligned}$$

wobei Q der Preshared Key von Alice und Bob ist. Mit k_3 wird nun noch der Hash über alle vorherigen Berechnungen im AEAD-Modus authentisiert und übertragen, zusammen mit den beiden MACs wie oben, was den Handshake abschließt. Wiederum führt Alice beim Empfang der zweiten Nachricht von Bob die dazu symmetrischen Berechnungen aus, um genau das gleiche Schlüsselmaterial wie Bob zu erhalten, mit welchem nach einer weiteren Schlüsselableitung nun getunnelte Pakete übertragen werden können. Falls an irgendeinem Punkt des Handshakes eine der Berechnungen nicht das erwartete Resultat liefert, wird die Verbindung einfach verworfen.

Der symmetrische Preshared Key fließt wie oben zu sehen in die zweite Handshake-Nachricht ein. Dies ist im gesamten WireGuard-Protokoll auch der einzige Ort, an dem dieser verwendet wird, denn er dient – wie anderes initial generiertes Schlüsselmaterial auch – zur Vereinbarung eines symmetrischen Schlüssels, der für die Transportverschlüsselung gebraucht wird.

2.3 PKCS#11

PKCS#11 ist ein Standardinterface zur Einbindung von Hardwaresicherheitsmodulen, in welchen Schlüssel sicher generiert, aufbewahrt, und verwendet werden können [5]. Die Verwendung eines Hardwaresicherheitsmoduls bietet den Vorteil, dass kryptographische Operationen auf einen externen Coprozessor ausgelagert werden können, den sicherheitskritische Daten nicht verlassen müssen. Hierdurch könnte ein möglicher Angreifer zwar die Kontrolle über ein System erlangen und mit dieser darauf laufende Kommunikation beobachten oder manipulieren, aber nicht das dazu benötigte Schlüsselmaterial stehlen. Damit ist ein Sicherheitsverstoß zwar problematisch, während er stattfindet, aber der Einsatz eines Hardwaresicherheitsmoduls reduziert die langfristigen Konsequenzen, da nicht alle zukünftigen Kommunikationen kompromittiert sind und auch keine neuen Geheimnisse vereinbart werden müssen. Die Verwendung eines Hardwaresicherheitsmoduls durch PKCS#11 bietet also erhöhte Vorwärtssicherheit und kann bei Sicherheitsverstößen den Verwaltungsaufwand eines Services wie WireGuard reduzieren.

Unter anderem sind in PKCS#11 Version 3.0 auch Routinen zur HKDF-Schlüsselableitung und die BLAKE2-Hashfunktion spezifiziert [14]. Da Implementierungen von PKCS#11 v3.0 aber aufgrund des jungen Alters dieser Version der Spezifikation noch nicht sehr verbreitet sind, musste hierfür mit v2.0 vorlieb genommen werden. In dieser älteren Spezifikation sind jedoch weder BLAKE2s noch HKDF verfügbar. Trotzdem bietet PKCS#11 ein Grundgerüst, um die nötige Funktionalität als Proof-of-Concept darin größtenteils nahtlos zu integrieren [5]. Hierfür ist die Verwendung der SoftHSMv2-Implementierung von PKCS#11 hilfreich: SoftHSMv2 ist kein Hardwaresicherheitsmodul im eigentlichen Sinne, sondern stellt lediglich ein PKCS#11-Interface mit Software-Backend bereit. Dies missachtet zwar den Hardwareaspekt eines Hardwaresicherheitsmoduls, aber umgeht damit auch die Kosten für die Beschaffung und den Aufwand für den Einsatz von richtiger Hardware und ermöglicht darüber hinaus recht einfach Modifikationen.

PKCS#11 ist als Sammlung von C-Funktionen gegeben, die als Abstraktion über die spezifische Hardwaresicherheitsmodul-Implementierung auf dieser die notwendigen Kryptoprimitiven ausführen. Je nach Art des Primitivs werden dafür vorgesehene Funktionen verwendet, beispielsweise die Funktion `C_Encrypt` für symmetrische Verschlüsselung, `C_Verify` für die Verifikation von Signaturen oder `C_DeriveKey` für das Ableiten von Schlüsseln.

3 Prinzip der Umsetzung

Der Fall, gegen den wir WireGuard sichern wollen, ist der eines Angreifers, der bereits jetzt WireGuard-Verkehr mitschreibt, um ihn später mithilfe eines Quantencomputers zu entschlüsseln. Es handelt sich also um einen Store-now-Decrypt-later-Angriff, bei dem zusätzlich der Preshared Key kompromittiert werden muss. Der Quantencomputer würde hierbei dazu dienen, mithilfe von Shors Algorithmus das diskrete Logarithmusproblem zu lösen [12], auf dessen Schwierigkeit die Sicherheit des Diffie-Hellman-Schlüsselaustauschs beruht. Ein Angreifer könnte nach der Aufzeichnung von WireGuard-Traffic mithilfe dieses Verfahrens und einer Methode zum Bestimmen des Preshared Keys den Schlüssel für die belauschte Verbindung im Nachhinein bestimmen und den gesamten Datenverkehr von WireGuard entschlüsseln, indem er die privaten Diffie-Hellman-Schlüssel bestimmt und dann den restlichen Schlüsselaustausch nachvollzieht.

Da die Ressourcen eines Widersachers mit den Ressourcen für die ersten beiden Notwendigkeiten des Angriffs bereits recht umfangreich sein müssen, liegt das Aufdecken des geheimen Schlüssels höchstwahrscheinlich auch in

seinem Fertigkeitsbereich. Dies kann beispielsweise durch netzwerkseitiges Eindringen oder durch die physische Inbesitznahme des Servers geschehen. Beides sind Möglichkeiten, die ein Angreifer, der sich jetzt schon auf das Brechen der Forward-Secrecy von WireGuard vorbereitet, in Betracht ziehen kann, und die ihm mit entsprechender Vorbereitung in Zukunft offen stehen könnten.

Für den beschriebenen Angriff ist es unerheblich, ob die WireGuard-Implementierung im Kernel oder im Userland angesiedelt ist. Letzteres soll im Folgenden zur Vereinfachung der Implementierung verwendet werden, weil unser Angreifermodell sowieso die vollständige Kompromittierung des Endpunktes auf Softwareebene beinhaltet. Daher verwenden wir in der Umsetzung die Userspace-WireGuard-Implementierung Boringtun [3], weil dabei die aufwändige Praxis der Kernelentwicklung wegfällt.

3.1 Designziele

Um die beschriebene Angriffsmöglichkeit zu mitigieren, wollen wir die bisher beschriebenen Komponenten zusammenführen, indem wir die Schlüsselableitung in WireGuard so abändern, dass sie stattdessen durch das PKCS#11-Interface in ein Hardwaresicherheitsmodul ausgelagert werden kann. Dies sorgt dafür, dass auch ein Angreifer mit Zugriff auf den WireGuard-Endpunkt gar nicht oder nur mit extremem Aufwand an den Schlüssel im Hardwaresicherheitsmodul herankommt. In Ermangelung eines entsprechenden, programmierbaren Moduls, das wir entsprechend modifizieren könnten, um HKDF mit BLAKE2 zu unterstützen, verwenden wir SoftHSMv2, das die PKCS#11-Schnittstelle softwareseitig implementiert [10]. Dadurch geht zwar die tatsächliche Hardwaresicherheit zunächst verloren, für eine ordentliche Implementierung in Hardware kann die Arbeit an SoftHSMv2 dennoch als Blaupause dienen. SoftHSMv2 ermöglicht es uns weiterhin relativ problemlos, die unter dem PKCS#11-Interface liegende Logik zu modifizieren. Da es sich um Software handelt, können wir so die benötigten Kryptoprimitiven für den Handshake in Software implementieren.

Zunächst sollte der PKCS#11-Schnittstelle von SoftHSMv2 ein Mechanismus zur Schlüsselableitung mit HKDF hinzugefügt werden. Dieser kann durch `C_DeriveKey` aufgerufen werden, was wir in Boringtun bei der Bearbeitung des zweiten Handshake-Pakets mit rust-cryptoki, einem Rust-Wrapper für PKCS#11 umsetzen wollen [11]. Wir interpretieren dabei den Wert für den Preshared aus dem Konfigurationsinterface von WireGuard nicht als Schlüssel, sondern als Label für ein Objekt auf dem Hardwaresicherheitsmodul, um mithilfe dieses Labels den Preshared Key zu finden und anhand von diesem

den finalen Transportschlüssel abzuleiten. Um diese Operationen auf einem PKCS#11-Token durchführen zu können, was auf Softwarebene essenziell einem solchen Modul entspricht, muss man sich jedoch dem Token gegenüber mit einer PIN authentisieren, was idealerweise genau einmal beim Einrichten der Verbindung geschehen sollte. Wir werden hier dem Angreifermodell bereits gerecht, indem wir die PIN mit dem Label konfigurieren, aber es wäre für Implementierungen mit Hardware eleganter, einen geschützten Authentifikationsweg wie in PKCS#11 spezifiziert zu konfigurieren. Da SoftHSMv2 einen solchen nicht unterstützt, müssen wir hier darauf verzichten.

Der bisherige Operationsfluss von WireGuard soll nach außen hin auf Netzwerkebene erhalten bleiben. Das bedeutet, dass wir alle Nachrichten des Protokolls funktional unverändert lassen müssen. So wird ermöglicht, die Änderungen transparent auf einer Seite des Tunnels einzusetzen, ohne die Kompatibilität mit einer anderen WireGuard-Implementierung am anderen Ende des Tunnels zu verlieren. Indem wir nur die Implementierung der Schlüsselableitung ändern, nicht jedoch ihre eigentliche Funktion, erreichen wir genau dies. Während wir die Kompatibilität auf Netzwerkebene erhalten wollen, wird eine Modifikation des Konfigurationsformates unumgänglich sein, damit ein gewöhnlicher Preshared Key von einem, der in einem Hardwaresicherheitsmodul aufbewahrt wird unterschieden und entsprechend behandelt werden kann.

Dieses Vorgehen erhöht nicht die eigentliche Quantensicherheit von WireGuard als Protokoll, sondern fördert eine Verwendung des optionalen, aber im Kern schon quantensicheren Preshared Keys. Dies bietet für das beschriebene Angreifermodell ein weiteres Stück Vorwärtssicherheit, die unter Verwendung von derzeitigen asymmetrischen Primitiven nicht garantiert ist. Der Preshared Key ist das Einzige, was einen Angreifer mit Quantencomputer von einem Store-now-Decrypt-later-Angriff trennt. Wir erhöhen die Vorwärtssicherheit von WireGuard durch den Schutz des Schlüssels mit einer solchen Integration indirekt, da sie auf der Geheimhaltung des symmetrischen Schlüssels basiert.

4 Implementierung

Wie bereits erörtert wurde, sind Modifikationen von verschiedenen existierenden Komponenten an mehreren Stellen nötig, um das gewünschte Zusammenspiel zu erreichen. Im Folgenden wollen wir diese mit ihren beabsichtigten Effekten beschreiben.

4.1 SoftHSMv2

Zunächst musste in SoftHSMv2 `C_DeriveKey` so modifiziert werden, dass ein entsprechender Vendor-spezifischer Mechanismus für HKDF verfügbar ist. Dieser kann mit der Mechanismus-Identifikation

```
CKM_VENDOR_DEFINED | 0x484B4446
```

angesprochen werden. Der Mechanismus selbst bietet dasselbe Interface wie der `CKM_HKDF_DATA`-Mechanismus der PKCS#11 v3.0-Spezifikation, da in der Entwicklung dieser Spezifikation die nötigen Details bereits ausgearbeitet wurden und es hier keinen Grund gibt, vom bewährten Muster abzuweichen.

Beim Aufrufen von `C_DeriveKey` mit dem neuen Mechanismus wird nun analog zu anderen Klassen von Kryptoprimitiven im Backend OpenSSL verwendet, um die HKDF-Schlüsselableitung mit den gewünschten Parametern durchzuführen. Bevor dies geschieht, werden zunächst die Eingabeparameter auf ihre Zulässigkeit überprüft. Dann wird der benötigte Mechanismus selbst auf Gültigkeit überprüft und weitere mechanismusspezifische Parameter werden validiert, unter anderem die zu verwendende Hashfunktion. Falls keine Probleme gefunden wurden, wird der abzuleitende Schlüssel geladen und an das Backend mit den Parametern der auszuführenden Operation abgegeben. Dieses führt den Kryptoprimitive aus und gibt das Ergebnis zurück. Zuletzt wird die Gültigkeit der Ausgabe überprüft, es wird ein neues Objekt, das letztere hält, erstellt und mit sicheren Parametern initialisiert, und falls nötig auf die Tokendatei gespeichert.

Bei dem hierbei verwendeten Operationsfluss können wir uns in Teilen an den bereits bestehenden Mechanismen in SoftHSMv2 orientieren. Beispielsweise gibt bei der Speicherung des erstellten Objektes nur eine Stelle, an der wir von der bereits existierenden Implementierung abweichen müssen, da statt eines Schlüsselobjekts lediglich ein Datenobjekt erstellt wird. Die Umsetzung des Kryptoprimitivs wurde in die neu erstellten Strukturen für Schlüsselableitungs-Kryptoprimitiven eingesetzt, die andere Schlüsselableitungsfunktionen auch fassen können, wobei wir den Aufbau hierbei dem Rest von SoftHSMv2 anpassen. Der einzige hierbei bisher implementierte Primitive ist HKDF, aber weitere könnten auf eine ähnliche Art umgesetzt werden. Es wurde ausschließlich mit der OpenSSL-Seite von SoftHSMv2 gearbeitet, da OpenSSL eine API bietet, die auf einer relativ hohen Ebene operiert und somit nötigen Boilerplate-Code reduziert. Der eingefügte Mechanismus für HKDF ist nicht mit einer Botan-Implementierung versehen, weil für die Demonstration eine der beiden Bibliotheken bereits genügt und OpenSSL standardmäßig von SoftHSMv2 bevorzugt wird.

4.2 rust-cryptoki

Um den neu erstellten Mechanismus nun in Rust bedienen zu können, musste auch das rust-cryptoki-Crate modifiziert werden. Dieses soll dann in Boringtun dazu dienen, die PKCS#11-API in sicherem Rust zu wrappen. Da rust-cryptoki bisher keine Unterstützung für Vendor-spezifische Mechanismen besaß, wurde eine generische Anbindung für solche implementiert. Hierzu wurde die Repräsentation von Mechanismen im Crate so geändert, dass sie Vendor-spezifische Mechanismen mit einer beliebigen, zur Laufzeit bestimmten Mechanismus-Identifikation unterstützt, und weiterhin eine beliebige Parameterstruktur für Mechanismen mit Parametern als Bytearray in das PKCS#11-Backend übergibt. Diese Parameterstruktur wird hinter den Kulissen auf im Rust-Sinne unsafe Art in dieses Bytearray umgewandelt, weshalb darin vorkommende Referenzen manuell auf Speichersicherheit geprüft werden müssen. Unter Verwendung von rust-cryptoki kann man nun in Boringtun den in SoftHSM hinzugefügten Mechanismus nach Deklaration der Parameterstruktur, die wie oben erklärt analog zu Version 3 von PKCS#11 aufgebaut ist, zur Schlüsselableitung verwenden.

4.3 Boringtun

Im Zentrum der Arbeit steht die Modifikation von Boringtun, der WireGuard-Userspace-Implementierung [3]. Nach der Modifikation von rust-cryptoki können wir in Boringtun beim Handshake nun die PKCS#11-Implementierung von HKDF aufrufen, um den Preshared Key bei der Schlüsselableitung zu verwenden. Dies passiert an zwei Stellen, nämlich beim Senden und beim Empfangen der zweiten Handshake-Nachricht. Der Ablauf ist jedoch zweimal der gleiche: Es wird bei der Konfiguration des Tunnels eine Read-Only-Session zum konfigurierten Token geöffnet, um in dieser beim Handshake den Preshared Key anhand des zuvor konfigurierten Labels zu finden. Mit dem Preshared Key wird mit dem neu implementierten HKDF-Mechanismus in SoftHSMv2 durch rust-cryptoki hindurch der finale Chaining Key abgeleitet, und der Handshake wird wie vorgesehen fortgesetzt. Die Session wird beim Löschen des Tunnels wieder geschlossen. Es werden keine abgeleiteten Schlüssel auf dem Token gespeichert, die Daten werden nur für die Dauer der Session vorgehalten und bei deren Abbau verworfen. Dieses Verfahren ist aus einer Ressourcenmanagement-Perspektive sinnvoll, damit die abgeleiteten Schlüssel nicht nach und nach über den Verlauf vieler Sessions den Speicher des Hardwaresicherheitsmoduls füllen. Dies könnten wir noch weiter verbessern, indem wir alle neu erstellten Objekte sofort nach der Verwendung löschen, damit ein solches Füllen auch im Verlauf einer einzelnen Session nicht geschehen

kann, aber bisher ist dies noch nicht implementiert.

Um zu wissen, wo der Schlüssel zu finden ist, musste weiterhin das Konfigurationsinterface von Boringtun bearbeitet werden, damit das Schlüsselobjekt identifiziert werden kann. Die Konfigurationsschnittstelle von Boringtun erkennt nun an einem Schlüssel, der mit $ beginnt, ein Label für PKCS#11, und parst dieses entsprechend. Das geparste Label wird intern wie der Schlüssel als Byte-Array gehandhabt, aber ist mit einem anderen Tag versehen. Beim Handshake wird dieses Tag erkannt und das Label entsprechend als solches interpretiert. Anstatt einen Base64-Kodierten 32-Byte-Schlüssel durch

```
PresharedKey = K3G3KZ4EsQ5wPLaHidauPxWivdMbr9Y/f34jUgI2/K4=
```

in einer Konfigurationsdatei anzugeben, wird nun

```
PresharedKey = $thisisanobjectlabel
```

spezifiziert, was dafür sorgt, dass Boringtun nun nach dem Objekt mit genau diesem Label sucht, wobei der gegebene, mit $ beginnende String mit Nullbytes gepaddet und auf 32 Byte abgeschnitten wird.

Nicht nur Boringtun musste jedoch für die Handhabung von PKCS#11-Objektlabeln modifiziert werden: Auch wg, das Konfigurationstool für Wireguard-Tunnel aus dem WireGuard-tools-Paket, welches standardmäßig für das Einrichten von Tunneln verwendet wird, benötigte Anpassungen. In diesem wird beim Laden von Konfigurationsdateien und beim Parsen von Schlüsseln von der Kommandozeile normalerweise ein Base64- oder Hex-String erwartet. Damit dieses Tool auch zum Aufsetzen der modifizierten Boringtun-Variante genutzt werden kann, musste hier auch der Test auf ein Label mit $ eingeführt werden, um das Label dann als Schlüssel an Boringtun weiterzugeben. Andernfalls würde wg beim Lesen des Schlüssels mit einem Parsingfehler abbrechen. Prinzipiell könnte man auf die Modifikation von wg verzichten, indem man den Tunnel stattdessen durch den von Boringtun erstellten Konfigurationssocket manuell einrichtet, also handelt es sich hierbei letztlich um eine Vereinfachung bei der Handhabung.

5 Ergebnisse

Für die so gewonnene Implementierung gibt es einige Faktoren, die sich auf die Praktikabilität des Konzeptes auswirken und daher für eine tatsächliche Umsetzung beachtet werden sollten. Wir gehen im Folgenden auf Performance und Verwendbarkeit ein und arbeiten einige Trade-offs heraus.

5.1 Overhead

Um den Performanceunterschied im Vergleich zur unmodifizierten WireGuard-Implementierung von Boringtun sichtbar zu machen und diesen relativ zur Performance vom verwendeten Hardwaresicherheitsmodul einzuordnen, verwenden wir mehrere Benchmarks. Wir testen den Handshake jeweils mit und ohne Schlüsselableitung im Hardwaresicherheitsmodul auf die folgenden zwei Arten: Einerseits messen wir die Zeit für 100 einzelne Handshakes, während die Verbindung lediglich aufrechterhalten wird, und andererseits Handshakes auf einer Verbindung, auf der 10000 Pakete übertragen werden. Hierbei wird jeweils die Latenz der übertragenen Pakete gemessen, die Dauer des Handshakes selbst, zusammen mit der Zeit für die Verarbeitung der Pakete in Boringtun während des Handshakes, und zuletzt die Zeit, die für die Schlüsselableitung in SoftHSMv2 benötigt wird. Die ersten beiden Metriken geben den Vergleich der Performance für die tatsächliche Benutzung, und die beiden letzten zeigen, um wie viel sich ein einzelner Handshake verteuert. Dies lässt unter Hinzunahme von Performancedaten für tatsächliche HSMs Rückschlüsse auf den Overhead bei einer praktischen Verwendung zu. Wir geben zu jeder Metrik jeweils den Durchschnitt der gemessenen Werte sowie die unkorrigierte Standardabweichung der Stichprobe an. Wenn Boringtun so konfiguriert ist, dass der Schlüssel wie üblich aus der Konfigurationsdatei gelesen und PKCS#11 nicht verwendet wird, kann die Zeit für die Schlüsselableitung in SoftHSMv2 konstruktionsgemäß nicht erfasst werden. Daher ist diese Metrik im besagten Fall nicht angegeben.

Metrik	mit PKCS#11		normal	
	$\varnothing$	σ	$\varnothing$	σ
Ping-Latenz	0,94	0,28	1,00	0,33
Handshake-RTT	13,31	3,48	11,12	2,33
Handshake-Berechnungen	11,41	3,17	9,37	2,19
Schlüsselableitung	0,73	0,22	-	-

Tabelle 1: Handshake-Test mit 10000 übertragenen Paketen, Angaben jeweils in ms

Metrik	mit PKCS#11		normal	
	∅	σ	∅	σ
Ping-Latenz	1,35	1,46	1,29	1,27
Handshake-RTT	13,22	2,36	11,19	2,52
Handshake-Berechnungen	11,27	2,27	9,41	2,32
Schlüsselableitung	0,70	0,20	-	-

Tabelle 2: Handshake-Test mit 100 einzelnen Handshakes, Angaben jeweils in ms

Die Tests wurden mithilfe von zwei virtuellen Maschinen durchgeführt, die in ein gemeinsames virtuelles Netzwerk geschaltet sind. Beide virtuellen Maschinen verwenden dieselbe Debian-Installation in Version 6.0.8-1. Der Hostrechner der beiden Maschinen wird von einem i7-5820K mit 16 GB Arbeitsspeicher gesteuert, von denen jede virtuelle Maschine je 4 GB erhält. Eine der virtuellen Maschinen ist mit der gewöhnlichen Kernelimplementierung von WireGuard konfiguriert, es wird die Liunx-Version 6.0.0-4 verwendet (srcversion 2C06E87677EE83A1EE5A688). Die andere virtuelle Maschine verwendet die modifizierte Variante von Boringtun, auf dieser wurden die Daten erhoben. Während des Tests liefen auf den virtuellen Maschinen neben den getesteten Anwendungen jeweils nur ein Terminalemulator. Die beiden Maschinen können ohne den WireGuard-Tunnel mit einer Latenz von 0,431ms (gemittelt über 1000 Pakete) kommunizieren. Damit der Benchmark in einer vertretbaren Zeit ausgeführt werden konnte, haben wir für den Test die Zeitspanne, nach der ein Rekeying von Boringtun begonnen wird, von 120 auf 5 Sekunden reduziert.

Aus den Ergebnissen lässt sich ablesen, dass der Handshake sich durch die Verwendung von PKCS#11 sich um etwa eine Millisekunde verlängert, wobei der Großteil dieser Millisekunde im HSM selbst zugebracht wird. Etwa 0.3ms werden dabei jedoch beim Aufrufen von PKCS#11 durch rust-cryptoki in Anspruch genommen. Gemessen an der Gesamtdauer des Handshakes, die etwa 10ms beträgt, verlängert die Integration von PKCS#11 diesen also zwar merklich, aber nicht wesentlich. Da der Handshake für gewöhnlich nur alle zwei Minuten durchgeführt wird, und Pakete mit dem alten Transportschlüssel währenddessen weiterhin akzeptiert werden, ist die zusätzliche Last für einen einzelnen VPN-Tunnel also relativ gering. Am größten könnte der Einfluss der zusätzlichen Millisekunde bei seltenen, aber latenzkritischen Verbindungen durch den Tunnel sein: Hierbei muss, wenn der Tunnel mit geringer Frequenz benutzt wird, bei jedem Verbindungsaufbau durch den Tunnel ein neuer Handshake durchgeführt werden, wodurch die zusätzliche

Zeit für die Schlüsselableitung am ehesten bemerkbar wird. Abhängig vom tatsächlichen Hardwaresicherheitsmodul in der Praxis muss bei der Interpretation dieser Ergebnisse zusätzlich beachtet werden, dass die circa 0.7ms für die Schlüsselableitung in SoftHSMv2 für tatsächliche Hardware und deren Treiber nicht repräsentativ sein müssen. Eine sichere Konfiguration wird dann voraussichtlich mehr Zeit im Handshake in Anspruch nehmen, was vor dem Einsatz am besten durch Benchmarks des zu verwendenden Hardwaresicherheitsmoduls zu bestimmen ist. Von Seiten der Boringtun-Modifikationen sind mindestens 0.3ms Overhead garantiert, was jedoch gegenüber der Schlüsselableitung in Hardware wahrscheinlich verschwinden wird und gegenüber dem gesamten Handshake bereits verschwindet.

5.2 Anwendung

Die gewonnene Integration könnte in verschiedenen Situationen eingesetzt werden, in denen VPNs für gewöhnlich auch Verwendung finden. Je nach Szenario ergeben sich unterschiedliche Möglichkeiten zur Umsetzung eines solchen Vorhabens: Die kleinste Anwendungsmöglichkeit wäre ein von einem Privatanwender eingerichteter WireGuard-Tunnel zwischen einem Heimnetzwerk und einem mobilen Tunnelende oder mehreren vorkonfigurierten Endpunkten. Für einen derartigen Einsatz ist die Beschaffung eines kommerziellen Hardwaresicherheitsmoduls höchstwahrscheinlich überflüssig, da sich solche Anwender vermutlich auch mit einem Trusted Platform Module, wie es mittlerweile in der Mehrzahl der Laptops und mittlerweile auch Chipsets von PC-Motherboards verbaut ist, behelfen könnten, sofern die entsprechenden Standards dort zukünftig implementiert werden. Unser Angreifermodell ist in diesem Szenario jedoch kaum zutreffend und die Sensibilität so gering, dass eine Umsetzung hier höchstens noch besonders sensiblen Anwendern zugutekäme.

Die nächste Einsatzmöglichkeit könnte in einem klassischen VPN-Service bestehen. Hierbei bietet ein Provider ein Ende eines Tunnels an Kunden an, die dann mit ihrem Endgerät die andere Seite einnehmen, beispielsweise zum Zweck der Anonymisierung im Internet. Für dieses Szenario wäre es auf der Serverseite bereits denkbar, ein stärkeres Hardwaresicherheitsmodul einzusetzen. In der Konfiguration des Tunnels würde dann aber das Problem des Schlüsselaustauschs auf industrieller Skala auftreten, sodass hier ein Ausbau des Mechanismus zur Schlüsselgenerierung auf eine mit der benötigten Schlüsselverteilung kompatible Art sinnvoll wäre. Ideen hierzu finden sich im letzten Abschnitt, aber hierfür wären voraussichtlich zusätzlicher Entwicklungs- und Konfigurationsaufwand nötig.

Zuletzt wäre noch ein Szenario im industriellen Einsatz in Form eines Ende-zu-Ende-Tunnels oder eines VPN-Servers zur Einwahl in das Firmennetz denkbar. In beiden Fällen würde hier die Administration von Seiten der Firma erfolgen, sodass die Schlüsselverteilung sich möglicherweise signifikant vereinfacht: Bei bereitgestellten Endgeräten oder nur zwei Endpunkten kann das Schlüsselmaterial in die Geräte unter der notwendigen Geheimhaltung vor dem Einsatz problemlos eingefügt werden, was wesentlich einfacher als im obigen VPN-Service-Szenario ist. Bei dieser Verwendungsmöglichkeit ist es auch am wahrscheinlichsten so, dass die transportierten Daten in ihrer Sensibilität dem Angreifermodell gerecht werden.

5.3 Diskussion

Der erhöhten Sicherheit, die wir durch das Hardwaresicherheitsmodul erhalten, steht zusätzlich zu dem geringen Performanceunterschied der erhöhte Aufwand zur Konfiguration und Administration sowie natürlich den Kosten des Moduls selbst gegenüber. Da die gewonnene Marge an Sicherheit durch die Seltenheit und Schwierigkeit des relevanten Angriffsszenarios relativ gering ist, bedeuten auch die netzwerkseitige Transparenz und im Vergleich zu gewöhnlichem WireGuard nur minimal veränderte Konfiguration hier nicht viel. Zusätzlich muss der Schlüssel im Hardwaresicherheitsmodul auf irgendeine Weise mit dem anderen Endpunkt übereinstimmen, was abhängig vom endgültigen Einsatzszenario vergleichsweise schwierig oder einfach sein kann. In einigen kommerziellen Hardwaresicherheitsmodulen sind Mechanismen zur gemeinsamen Schlüsselgenerierung vorhanden, die solche Operationen unterstützen, aber falls einer der beiden Kommunikationspartner kein Hardwaresicherheitsmodul verwendet, müsste der Schlüssel anders geteilt werden. Dazu könnte er beispielsweise vom anderen Kommunikationspartner generiert und dann in das Modul importiert werden. Hierbei stellt sich zusätzlich die Frage nach der Sicherheit bei der Übertragung des Schlüssels, und der Konfigurationsaufwand erhöht sich durch die Notwendigkeit zur sicheren Übertragung und zum sicheren Importieren.

Insgesamt lässt sich erkennen, dass die vorgeschlagene Lösung nur für Szenarien mit überdurchschnittlich hohen Sicherheitsanforderungen interessant ist, wenn man die Kosten für deren Installation berücksichtigt. Nachdem ein Tunnel auf die hier beschriebene Art eingerichtet ist, ist der Aufwand zur Instandhaltung recht gering, sodass sich für konstante Verbindungen, die gegen starke Angreifer gesichert werden müssen, der größte Nutzen abzeichnet. Auch noch sinnvoll kann eine Verwendung beim Server im VPN-Dienst-Szenario sein, wenn der Schlüsselaustausch einfach genug ausführbar wird, beispiels-

weise durch ein gemeinsames Schlüsselableitungsverfahren. Die Verbindung bei wechselnden Clients neu zu konfigurieren, ist dann genau so schwer wie für normale Wireguard-Verbindungen, solange der Preshared Key weiterhin verwendet und nicht neu vereinbart wird. Die Kompromittierung eines Servers zu verhindern ist wesentlich wichtiger als die einzelner Clients, da der Server alle Schlüssel verwalten muss, daher kann man hier von einem gesteigerten Nutzen sprechen. Auf der anderen Seite des Spektrums beim Sicherheit-Kosten-Trade-off stehen Konfigurationen von ständig wechselnden Kommunikationspartnern mit geringen Sicherheitsanforderungen, speziell an Vorwärtssicherheit, die lediglich auf Anonymisierung Wert legen. Für ein solches Szenario ist diese Arbeit nicht besonders interessant.

6 Fazit

Die präsentierte Arbeit legt den Grundstein für einen erweiterten Schutz von WireGuard-Instanzen gegen Angreifer am oberen Ende des Ressourcenspektrums durch die Verwendung eines Hardwaresicherheitsmoduls und demonstriert die grundsätzliche Möglichkeit eines solchen Vorhabens. Obwohl es sich zunächst nur um ein Proof-of-Concept-Projekt handelt, können Teile der Implementierung Hinweise für eine zukünftige praxisreife Integration bieten.

Auch wenn die praktische Verwendung einer solchen Verstärkung von WireGuard für viele Verwendungsfälle wünschenswert wäre, ist der zu betreibende Aufwand auf dem Weg dahin hoch und hängt maßgeblich von der Herstellung oder Programmierung von Hardwaresicherheitsmodulen mit den benötigten Kryptoprimitiven ab. Durch die auf Protokollebene transparente Implementierung wird die Praktikabilität einer solchen Umsetzung jedoch erhöht und kann für besonders sensible Zwecke tatsächlich sinnvoll erscheinen. Diese Sensibilität kann genau gegenüber den Angreifern gegeben sein, die die Möglichkeiten zur Durchführung einer solchen Attacke besitzen könnten. Für durchschnittliche Heimanwender ist der diskutierte Trade-offs aber höchstwahrscheinlich nicht lohnenswert.

6.1 Empfehlungen

Da wie oben erwähnt für eine tatsächlich auf der erwünschten Ebene sichere Implementierung ein entsprechendes Hardwaresicherheitsmodul mit Unterstützung für HKDF und BLAKE2s als Vendor-spezifischen Mechanismus benötigt wird, ist eine rasche Umsetzung von PKCS#11, Version 3 in Hardware wünschenswert. Da BLAKE2b in PKCS#11 v3 sowieso in mehreren

Varianten spezifiziert ist, sollte der zusätzliche Aufwand für die Bereitstellung von BLAKE2s sehr gering sein, da beide sich im Wesentlichen nur in der Breite einiger Parameter unterscheiden.

Ein weiteres Feature, das für eine sichere Konfiguration hilfreich wäre, ist die Verwendung eines Protected Authentication Path gemäß PKCS#11. Ein solcher ermöglicht es dem Benutzer, den Zugang zum Token nicht durch ein in Software übergebenes Passwort zu regeln, sondern stattdessen direkt mit dem Token in Verbindung zu treten, um darin die Authentisierung für den Zugriff auf die gespeicherten Schlüssel zu bewirken. Ein solches Verfahren würde die Sicherheit zwar nur in einem Szenario erhöhen, in dem der Angreifer zur Laufzeit Kontrolle über den Tunnel erlangen möchte, was im erklärten Angreifermodell nicht inbegriffen ist. Für unser Szenario wäre es aber durchaus wünschenswert: Eine direkte Kompromittierung von außen könnte durch eine solche Konfiguration mit geschütztem Authentifikationspfad erschwert oder sogar verhindert werden.

Da die Sicherheit eines symmetrischen Schlüssels immer von beiden Kommunikationspartnern abhängt, käme es einer möglichst sicheren Umsetzung auch zugute, wenn beide Endpunkte sie verwenden würden. In diesem Punkt ist die Transparenz der präsentierten Lösung für die endgültige Sicherheit also ein Nachteil, da der es einem „faulen“ Endpunkt offen steht, auf die Sicherung durch ein Hardwaresicherheitsmodul zu verzichten. Für die Kompatibilität ist genau diese Transparenz jedoch ein Gewinn, was die Verwendung erleichtern sollte.

6.2 Ausblick

Anhand dieser Arbeit lassen sich verschiedene Erkenntnisse für zukünftige Implementierungen ableiten: Zunächst bietet WireGuard aufgrund des simplen Protokolls und der kompakten Implementierung tatsächlich eine gute Basis für zukünftige Arbeit. Diese beiden Eigenschaften kommen jedoch genau aus der Einfachheit von WireGuard, die durch tiefgreifende Modifikationen gefährdet wird.

Für den praktischen Einsatz einer wie hier präsentierten Lösung gibt es noch einiges zu tun: Die Parallelisierung der Tunnel in Boringtun könnte darunter leiden, wie rust-cryptoki und PKCS#11 mit Nebenläufigkeit umgehen. In Anwendungen mit mehreren Tunneln an einem Endpunkt kann schlechte Parallelisierung die Performance stark senken, was die Verwendbarkeit einschränken würde.

Das Konfigurationsinterface ist auch noch verbesserungswürdig, weil hier

bisher keine Methode existiert, einen PKCS#11-Tokenslot auszuwählen, was uns effektiv auf ein einzelnes Token einschränkt. Zusammen mit dem Fehlen einer praktikablen Lösung für die Authentisierung gegenüber dem oder den Token sollte dieser Punkt am besten in einer Art angegangen werden, die die gesamte Konfiguration vereinfacht. Hier könnte man möglicherweise auch ein anderes Medium als die Token-Labels verwenden, um die richtigen Schlüssel zu finden, falls die Situation eine elegantere Lösung zulässt.

Noch viel wichtiger aber ist, dass sich mit der Verwendung eines Hardwaresicherheitsmoduls auch Möglichkeiten eröffnen, um die asymmetrischen Schlüssel von WireGuard-Endpunkten zu schützen: Hier sind die nötigen Primitiven für die benötigten Diffie-Hellman-Operationen auf elliptischen Kurven bereits in den meisten Hardwaresicherheitsmodulen implementiert, sodass zusammen mit BLAKE2s und HKDF dann alle nötigen Berechnungen in ein solches Modul ausgelagert werden könnten. Dies würde auch Verbindungen ohne Preshared Key vor Angriffen schützen, allerdings wiederum viele der Kosten, die bei der hier vorgestellten Anwendung anfallen, mit sich bringen. Der Konfigurationsaufwand wäre jedoch etwas geringer, da man die asymmetrischen, privaten Schlüssel nicht mit dem anderen Endpunkt teilen zu braucht.

Für eine praktische Umsetzung kann man also zusammenfassend feststellen, dass die Möglichkeit einer so diskutierten Absicherung von Boringtun besteht, aber noch einige anwendungsspezifische Probleme beim Schlüsselaustausch zu lösen sind. Der Implementierungsaufwand ist insgesamt nicht zu hoch, auch wenn der vorgestellte Proof-of-Concept-Code einer praxisreifen Lösung noch nicht gleich kommt. Es muss hierbei der Trade-off zwischen Aufwand und gewonnener Sicherheit beachtet werden, der für kleinere Anwender nicht unbedingt lohnenswert erscheint, sondern auf industriellen Einsatzszenarien am besten ist.

Literatur

[1] Jean-Philippe Aumasson u. a. *BLAKE2: simpler, smaller, fast as MD5*. Cryptology ePrint Archive, Paper 2013/322. https://eprint.iacr.org/2013/322. 2013. URL: https://eprint.iacr.org/2013/322.

[2] Daniel J. Bernstein. „Curve25519: New Diffie-Hellman Speed Records“. In: *Public Key Cryptography - PKC 2006*. Hrsg. von Moti Yung u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, S. 207–228. ISBN: 978-3-540-33852-9.

[3] Cloudflare und Mitwirkende. *Boringtun, Commit 370a9ed2906c1c83a634306675318e43aa10ae68*. GitHub Repository. URL: https://github.com/cloudflare/boringtun.

[4] Jason A. Donenfeld. „WireGuard: Next Generation Kernel Network Tunnel“. In: *Network and Distributed System Security Symposium*. 2017.

[5] Susan Gleeson u. a., Hrsg. *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40 Plus Errata 01*. OASIS Standard Incorporating Approved Errata 01. 13. Mai 2016. URL: http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/errata01/os/pkcs11-base-v2.40-errata01-os-complete.html.

[6] Lov K. Grover. „A fast quantum mechanical algorithm for database search“. In: *Symposium on the Theory of Computing*. 1996.

[7] Hugo Krawczyk. *Cryptographic Extraction and Key Derivation: The HKDF Scheme*. Cryptology ePrint Archive, Paper 2010/264. https://eprint.iacr.org/2010/264. 2010. URL: https://eprint.iacr.org/2010/264.

[8] Yoav Nir und Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. Juni 2018. DOI: 10.17487/RFC8439. URL: https://www.rfc-editor.org/info/rfc8439.

[9] Trevor Perrin. „The Noise protocol framework“. In: *noiseprotocol.org* (2018). URL: https://noiseprotocol.org/noise.pdf.

[10] OpenDNSSEC Project und Mitwirkende. *SoftHSMv2 (Debian-Paketarchive), Commit 7f99bedae002f0dd04ceeb8d86d59fc4a68a69a0*. GitHub Repository. URL: https://github.com/opendnssec/SoftHSMv2.

[11] PARSEC Project und Mitwirkende. *rust-cryptoki, Commit 89055f2a30e30d07a99e5904e9231d743c75d8e5*. GitHub Repository. URL: https://github.com/cloudflare/boringtun.

[12] Peter W. Shor. „Algorithms for quantum computation: discrete logarithms and factoring“. In: *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), S. 124–134.

[13] Maryna V. Yesina, Ye. V. Ostrianska und Ivan Gorbenko. „Status report on the third round of the NIST post-quantum cryptography standardization process“. In: *Radiotekhnika* (2022).

[14] Chris Zimman und Dieter Bong, Hrsg. *PKCS #11 Cryptographic Token Interface Base Specification Version 3.0*. OASIS Standard. 15. Juni 2020. URL: `https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/os/pkcs11-base-v3.0-os.html`.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 23. Februar 2023 .....