Zusammenfassung der Kapitel 8, 9, 15, 16 und 22 aus „Practical Cryptography“ von Niels Ferguson und Bruce Schneier

Martin Eismann
(martin.eismann@gmx.fr)

Seminar „Internet Sicherheit“ ,
Technische Universität Berlin

WS 2006/2007 (Version vom 15. Januar 2007)

Zusammenfassung

In der genannten Literatur werden folgende Themen abgehandelt und hier gekürzt wiedergegeben: Aufbau/Initialisierung eines sicheren Kanales (Kapitel 8), Implementierung sicherer Software und Protokolle, Arbeitsspeicherangriffe, Seitenkanalattacken, Testen von Berechnungen auf großen Ganzzahlen (Kapitel 9 und 16), Diffie-Hellman-Protokoll (Kapitel 15), Salzen und Strecken von Paßwörtern, Secure Tokens (Kapitel 22)

1 Einleitung

Das Thema Sicherheit ausgetauschter Information ist alt, bei weitem älter als das Computerzeitalter. Dennoch hat rechnergestützte Kommunikation den Nachrichtenaustausch grundlegend verändert – und mit ihm die Verwundbarkeit desselben: Nachrichten können nicht nur abgefangen und gelöscht, sondern auch mitgelesen und modifiziert werden, ohne daß die Dialogpartner sich darüber im Klaren sind. Gegen diese Maßnahmen muß Kommunikation „gesichert“ werden. Wie sich das bewerkstelligen läßt, ist Thema des Buches Practical Cryptography von Schneier/Ferguson, von dem hier die Kapitel 8, 9, 15, 16 und 22 zusammengefaßt wiedergegeben werden sollen.

Die Grundlage jeder sicheren Kommunikation ist das Vorhandensein eines „gemeinsamen Geheimnisses“, d.h. eines Schlüssels [1] , mithilfe dessen einem Dritten die Teilnahme am Nachrichtenaustausch „verschlossen“ werden sollen. Die Aushandlung dieses Schlüssels während des Verbindungsaufbaus sowie die Benutzung dessen zur Authentifizierung und Chiffrierung der Nutzdaten werden daher jeweils einen der sich anschließenden beiden Themenpunkte abdecken. Punkt vier enthält Betrachtungen zur Implementierung von sicherer

[1] Hier ist vom Sitzungsschlüssel die Rede, der für jede Verbindung erneut generiert wird. „Sicher“ im eigentlichen Sinne wird die Kommunikation aber erst durch einen weiteren Schlüssel gemacht: dem Authentifizierungsschlüssel, der auf anderem Wege ausgetauscht worden ist und sicherstellt, daß die Identität des Partner richtig ist (siehe Punkt 2, Seite 2).

Software und Punkt fünf schließlich das sichere Speichern von Daten, das von den Autoren als eine Spezialform eines sicheren Kanales zu einem Kommunikationspartner in der Zukunft gesehen wird, schließen sich ergänzend an.

2 Die Aushandlung des Sitzungsschlüssels

Wollen zwei Partner vertrauliche Informationen über ein unsicheres Medium austauschen, müssen sie ihre Nachrichten sichern, indem sie sie mittels eines Schlüssels kodieren. Das bedeutet, daß alle Nachrichtenpakete vom Sender nicht nur chiffriert (damit niemand mitliest), sondern auch signiert werden müssen (damit niemand eigene Nachrichten in den Paketstrom einbringt). Wie aber läßt sich ein solcher Schlüssel etablieren, ohne daß ein Mithörender ihn errät?

Bevor in Details gegangen wird, sei hier noch folgendes vorausbemerkt: Erstens findet unsere gesamte Kommunikation zwischen genau zwei Partnern statt. Um die Nachrichtenströme nicht durcheinander zu bringen, muß es im Protokoll eine Rollenverteilung geben, von der Server/Client eine mögliche ist. Zweitens werden wir hier lediglich diskrete Nachrichtenströme behandeln, d.h. die Nutzdaten werden in Segmente aufgespalten und als solche verschlüsselt und versandt. Praktisch alle Systeme arbeiten so. Gehen Nachrichten verloren, muß dem Empfänger klar sein, um welche es sich dabei handelt, ebenso muß erkennbar sein, wenn Nachrichten eingefügt wurden. Zudem muß der Paketstrom beim Empfänger wieder in der richtigen Reihenfolge zusammengesetzt werden.

Unser Beispiel geht von einem DH (Diffie-Hellman)-Protokoll aus[2]. Gemeint ist damit ein Protokoll, dessen Sicherheit darauf aufbaut, daß ein bestimmtes mathematisches Problem bisher noch nicht gelöst wurde – das Diskreter-Logarithmus-Problem: Wenn nur g, p, $g^x \ mod \ p$ und $g^y \ mod \ p$ bekannt sind, läßt sich dennoch nicht x, y oder gar g^{xy} berechnen. g, g^x und g^y werden unverschlüsselt ausgetauscht, daher kann auch ein Angreifer sie sehen. Ein Beispiel soll das illustrieren:

1. Beide Partner wählen sich eine Primzahl $p = 19$ und eine Basis $g = 3$ aus.	
2. Partner 1 benutzt seine Geheimzahl $x = 8$ und verschickt $g^x \ mod \ p$	$3^8 \ mod \ 19 = 64$
3. Partner 2 benutzt seine Geheimzahl $y = 11$ und verschickt $g^y \ mod \ p$	$3^{11} \ mod \ 19 = 10$
4. Partner 1 berechnet $(g^y \ mod \ p)^x \ mod \ p$	$10^8 \ mod \ 19 = 17$
5. Partner 2 berechnet $(g^x \ mod \ p)^y \ mod \ p$	$6^{11} \ mod \ 19 = 17$

Beide Partner kommen auf 17 (den gemeinsamen Schlüssel), was auf dem Potenzgesetz $(g^a)^b = (g^b)^a$ aufbaut. Auch das zunächst verwirrende *mod* 19 beeinträchtigt diese Gesetzmäßigkeit nicht, ist allerdings Grund dafür, daß ein Angreifer x,y und vor allem g^{xy} nicht sehen kann. In der Realität sind all diese Zahlen natürlich wesentlich größer.

Damit steht der Sitzungsschlüssel fest. Allerdings wird dieser nicht für die Verschlüsselung der Verbindung direkt verwendet, vielmehr leiten wir von ihm vier weitere Schlüssel ab, um die Sicherheit zu diversifizieren (dazu unter Punkt 2).

[2]Die gleiche Funktion übt das Protokoll RSA aus, das hier nicht betrachtet wird

Trotzdem bleibt noch eine Frage offen: Die Authentizität des Kommunikationspartners zu Beginn eines Protokollaufbaus. Um zu vermeiden, daß uns ein Angreifer eine falsche Identität vorspielt [3] , müssen sich beide Partner eindeutig authentifizieren, bevor sie die ersten Nutzdaten austauschen. Nach welcher Methode die Authentifizierung vonstatten geht, wird hier nicht Thema sein, aber es kann vorausgesetzt werden, daß beide Partner sich auch hier an einem „gemeinsames Geheimnis" erkennen, d.h. an einem Schlüssel, der nur für die Authentifizierung verwendet wird und z.B. bei persönlichem Kontakt festgelegt wurde. Die Authentifizierung schließt sich gleich an die Etablierung des Sitzungsschlüssels an. Dabei wird das Authentifizierungsverfahren auf sämtliche bisher in dieser Sitzung ausgetauschte Daten angewandt. Mit dieser pauschalen Bestätigung sollen Sicherheitslöcher vermieden werden, die entstehen, wenn Teile des Protokolls nicht bestätigt werden. Ein Sitzungsschlüssel existiert nur für eine einzige Sitzung und wird danach nicht wieder verwendet. Das soll verhindern, daß Nachrichten aus alten, mitgeschnittenen Verbindungen erneut in den Nachrichtenstrom eingespielt werden und Verwirrung stiften.

Das ist auch einer der Gründe, warum man nicht den Authentifizierungsschlüssel als Sitzungsschlüssel verwendet, sondern einen komplett neuen Schlüssel generiert. Aber selbst ohnedem will man den Authentifizierungsschlüssel nicht als Sitzungsschlüssel verwenden und ihn dadurch während der gesamten Verbindungsdauer den feindlichen Angriffen aussetzen: Seine Erstellung ist einfach schwieriger, etwa wenn sie durch persönlichen Kontakt zustande kam. Die anfängliche Authentifizierung ist zwar für das Protokoll von entscheidender Wichtigkeit, aber auch in kurzer Zeit abwickelbar, und länger sollte der Schlüssel nicht in Gebrauch sein.

Es sollte nun auch klar werden, warum wir nicht einfach x und y, die „Geheimzahlen" der Teilnehmer, als Schlüssel verwenden: Man müßte jedem, mit dem man kommunizieren will, seine Geheimzahl verraten. Stattdessen kombinieren wir sie (ohne sie einander verraten zu müssen!) zu einem Schlüssel, den man nur erraten kann, wenn man zweimal das Diskreter-Logarithmus-Problem löst – ein mit geeigneten Algorithmen in der Theorie nicht mehr unmögliches, aber extrem aufwendiges Unterfangen.

Neben der Tatsache, daß sich beide Partner auf die im Rahmen des DH-Protokolls verwendeten Parameter einigen müssen (z.B. g – siehe oben), so daß beider Sicherheitsansprüche befriedigt werden (im wesentlichen genügend große Primzahlen), gibt es mit diesen Parametern noch ein Problem: Aus ihnen leitet sich direkt der Sitzungsschlüssel ab. Werden sie konstant gewählt, haben wir auch immer den gleichen Sitzungsschlüssel, und ein Angreifer kann einem der Partner die mitgeschnittenen Nachrichten seines Gegenübers aus einer früheren Verbindung erneut vorspielen, anhand der Parameter insbesondere die Anforderung eines Verbindungsaufbaus. Diese Phantom-Verbindungen bergen zwar primär keine Sicherheitsrisiken, verursachen aber gefälschte Traffic-Logs. Dieses Problem wird gelöst, indem man die Parameter nicht nur von den beiderseitigen Mindestanforderungen, sondern auch von einem Zufallswert abhängig macht, der für jede Sitzung erneut generiert wird.

Für Protokolle existiert noch ein Problem, das für andere Software-Bereiche längst gelöst ist: Wo an anderen Stellen der Komplexität mit Modularisierung begegnet wird, sind Protokol-

[3] Es handelt sich um die sogenannte „Man-In-The-Middle-Attack". Ein Angreifer baut zu beiden Partnern eine Verbindung auf, in der er zwar alle Nachrichten zwischen ihnen weiterleitet, aber auch mitlesen kann: Nachrichten von A müssen ja zuerst dekodiert werden, um sie mit dem neuen Schlüssel kodiert an B weiterzusenden.

le monolithisch aufgebaut, ohne daß Techniken zur Modularisierung bekannt wären. Eine höhere Komplexität jedoch kann die gerade bei Protokollen kritischen Sicherheitsbemühungen untergraben.

3 Authentifizierung und Chiffrierung unter Verwendung des Sitzungsschlüssels

Nachdem der Sitzungsschlüssel mit dem Partner ausgehandelt wurde und sich letzterer im Zuge dessen auch als authentisch erwiesen hat, muß der Sitzungsschlüssel zur Anwendung kommen. Einzelheiten dazu werden im Folgenden beschrieben.

Wir wenden uns zunächst der Initialisierung des Protokolls zu. Es handelt sich dabei um eine Routine, die ein Zustands-Objekt der Sitzung zurückliefert. Darin enthalten sind nicht nur die aktuellen Werte des Nachrichtenzählers von sowohl empfangenen als auch versandten Nachrichten. Vor allem enthält dieses Objekt vier Schlüssel, die von dem Sitzungsschlüssel abgeleitet wurden. Dabei handelt es sich um zwei Authentifizierungsschlüssel und zwei Chiffrierungsschlüssel (in beiden Fällen je einen für Versand und Empfang). Hier wird nun deutlich, warum wir ganz zu Beginn festgelegt haben, daß zwischen den Kommunikationspartnern klare Rollen definiert werden müssen: Beide Partner benutzen für die Ableitung den gleichen Sitzungsschlüssel. Sie werden auch für Versand und Empfang die gleichen Schlüssel errechnen. Allerdings muß der Versand-Authentifizierungsschlüssel des Senders beim Empfänger der Empfangs-Authentifizierungsschlüssel sein. Das heißt einer der Partner muß noch während der Initialisierung einen Swap von Empfangs- und Versandschlüsseln vornehmen. Die Rolle entscheidet darüber, welchen der beiden das betrifft (z.B. den Client). Gleiches gilt natürlich für die Chiffrierung.

Jetzt stellt sich die Frage, ob eine Nachricht zuerst verschlüsselt und danach authentifiziert wird oder umgekehrt. (Es gilt als verstanden, daß jede einzelne Nachricht, die über die sichere Verbindung versandt wird, nicht nur verschlüsselt, sondern ebenfalls authentifiziert werden muß, um eine Manipulation zu verhindern. Diese Authentifizierung hat dabei nichts mit der anfänglichen Authentifizierung zu tun, die beim Aushandeln des Protokolls vorgenommen wird, und erst recht nichts mit dem dabei verwendeten permanenten Schlüssel!)

Die Autoren beschäftigen sich zu Beginn mit der Verschlüsseln-zuerst-Methode, bei der die schon verschlüsselte Nutzlast eines Paketes (Ciphertext) noch einmal authentifiziert wird (Abb. 3). Rechtfertigen ließe sich diese Methode aus einer theoretischen Überlegung her-

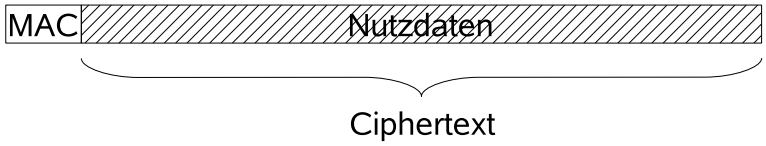

Abbildung 1: Nutzlast wird zuerst verschlüsselt und an den entstandenen Ciphertext der MAC (Message Authenticating Code) angehängt

aus und mit dem Argument, bei falsch authentifizierten Paketen die Entschlüsselung gleich

einzusparen (weniger Prozessorlast am Empfänger). Die Autoren jedoch sprechen sich dagegen aus – nicht zuletzt wegen der marginalen Bedeutung beider Erwägungen und der Bereitschaft, Effizienz für Sicherheit zu opfern.

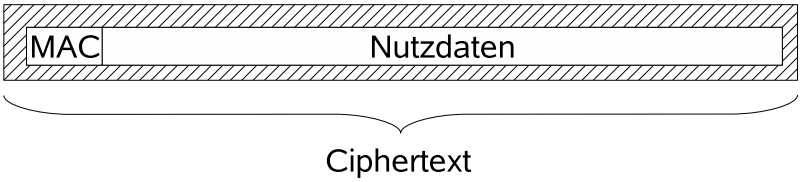

Abbildung 2: Nutzlast wird zuerst authentifiziert und dann mitsamt dem MAC verschlüsselt

Stattdessen soll zuerst authentifiziert und dann verschlüsselt werden (Abb. 3). Die Frage ist nämlich zum einen, welche Funktion zuletzt auf den Datensatz angewandt und damit dem unmittelbaren Angriff ausgesetzt wird: Die Verschlüsselung oder die Authentifizierung. Da im Allgemeinen die Authentifizierung sicherheitskritischer ist als die Verschlüsselung, versucht man, erstere zu schützen, indem man auch sie in den zu verschlüsselnden Datensatz einbezieht, d.h. erst nach der Authentifizierung verschlüsselt.

Dazu kommt der Fall, in dem ein korrektes Paket beim Empfänger zwar die Authentifizierungs-Probe besteht, aber der Ciphertext mit einem falschen Schlüssel dekodiert wird. Deswegen sollte der MAC (Message Authenticating Code) ein Teil des Ciphertextes sein, sodaß die Authentifizierungs-Probe erst dann gemacht werden kann, wenn der Ciphertext entschlüsselt wurde.

Die Nachrichtennummerierung soll sicherstellen, daß Nachrichten in der richtigen Reihenfolge beim Empfänger zusammengesetzt werden, außerdem sollen fehlende Pakete bemerkt werden. Die Nummern müssen streng monoton wachsen; beide Partner haben ihren eigenen, aber gleich großen Nummernraum. Es wird dabei in der Regel eine Zahl nicht länger als 48 Bit verwendet, die Zählung beginnt mit Eins, nicht mit Null, da Null schon mit einer anderen Bedeutung belegt ist: „Noch keine Nachricht versandt“, bzw. „Nummernraum erschöpft“. Tritt letzterer Fall ein, muß ein neuer Schlüssel ausgehandelt werden, auf keinen Fall sollte die Zählung von vorn beginnen!

Was passiert nun, wenn bei der Übertragung Nachrichtenpakete ihre Reihenfolge vertauschen? Viele Implementierungen können damit nicht umgehen, daher sollte das Problem eigentlich von der darunterliegenden Transportschicht gelöst werden. Einzig im Fall von IPsec ist den Autoren bekannt, daß es mit vertauschten Nachrichten innerhalb eines festgelegten Intervalles auf der Skala der Nachrichtennummern umgehen kann.

Als Verschlüsselungsmethode verwenden wir AES im CTR-Modus (Counter-Modus). Letzteres bietet sich an, da wir mit den Nachrichtennummern bereits einmalige Werte haben, die CTR benötigt. Das Verfahren ist ein Themengebiet für sich und wird hier nicht weiter behandelt (siehe Buch 4.5.2-4.5.5 (AES) und 5.5 (CTR)).

Beim Empfang einer Nachricht muß beachtet werden, daß die Nutzdaten des Paketes nur dann zurückgeliefert werden, wenn die Authentifizierung erfolgreich war. Im anderen Fall

muß ein Fehler zurückgeliefert werden und es darf dem System keine weitere Information über die Nutzdaten bekannt werden, auch im Speicher müssen Keystream und Nutzdaten zerstört werden.

4 Über Implementierung

In der Praxis werden die meisten Verschlüsselungsprotokolle nicht auf der kryptographischen Ebene geknackt, sondern auf der Implementierungsebene – aus Gründen der Einfachheit: Man findet viel leichter ein „Loch“ in der Implementierung, als in der Chiffrierung. Nicht nur deshalb muß eine Betrachtung zur Sicherheit auch immer die Implementierung mit abdecken.

Zunächst sei einmal klargestellt, daß keines der gängigen Betriebssysteme mit dem primären Ziel Sicherheit konzipiert wurde. Das allein kann manchmal unsere Bemühungen bereits vereiteln. Trotzdem sollte die Kryptographie schon heute auf möglichst hohem Niveau sein, da sich in Zukunft die Sicherheit der Systeme verbessern könnte und Protokolle erfahrungsgemäß ein langes Leben haben – auch wenn schon längst bessere verfügbar sind.

Der Kern des Implementierungsproblems ist, daß man einfach noch nicht in der Lage ist, ein „korrektes“ Programm zu schreiben – i.e. eines, dessen Verhalten genau seiner Spezifikation entspricht. Leider ist jedoch oft noch nicht einmal die Spezifikation klar umrissen. Somit gibt es also auch keine Kriterien für Tests. Und zweitens tut sich selbst beim Testen eine weitere Unzulänglichkeit auf: Man kann mit Tests nur die Existenz von Fehlern nachweisen, niemals aber deren Abwesenheit. Die Lösung besteht derzeit noch aus dem Verbessern der Testprozesse. Das kryptographische Problem ist dabei nur, daß die Test&Fix-Methode konzeptionell oft dann auch das einzige ist, was Programmierer bei der Entwicklung verwenden.

Drittens wird von den Autoren die laxe Einstellung gerügt, die in der IT-Industry immer noch vorherrscht. Anbieter lassen sich mitsamt dem EULA einen Haftungsauschluß attestieren und sind alle Verantwortung los. In der Autoindustrie etwa sei dies völlig anders. Auch ein Vergleich mit der Flugzeugindustrie soll uns inspirieren: Der bekanntlich extrem hohe Sicherheitsstandard führt zwar zu ebenso hohen Zusatzkosten, aber dafür ist eine technisch so prekäre Sache wie das Fliegen heute eine alltägliche Angelegenheit. Angesichts der Kosten für „Flugzeugabstürze“ in der IT wäre auch dort eine aufwendigere Entwicklung ihren Preis wert.

Wir grenzen den Begriff „korrekte Software“ ab von „sicherer Software“: Erstere hat eine erwiesene Entsprechung mit seiner gewünschten Funktionalität. Sichere Software dagegen garantiert die Abwesenheit einer Funktionalität: Funktion X ist nicht ausführbar, egal wie sehr man (i.d.R. der Angreifer) sich bemüht. Es gilt aber der (im Buch deutlich hervorgehobene) Satz: „Standard-Implementierungstechniken sind völlig ungeeignet um sicheren Code zu erstellen.“ Was können wir also tun? Zumindest können wir Sicherheitsprobleme bei der Implementierung beschreiben:

4.1 Schwachstellen in der Speicherverwaltung

Der falsche Umgang mit dem Arbeitsspeicher kann große Probleme schaffen. Information, die nicht mehr benötigt wird, sollte immer zerstört werden, besonders das Zustands-Objekt

der Sitzung, in der die Schlüssel gespeichert sind. Wird die Information nach Beendigung des Programmes im Speicher zurückgelassen, ist sie für entsprechende Programme leicht zugänglich, zumal einige Betriebssysteme den Speicher nicht erst löschen, ehe sie ihn neu vergeben.

In C kann man sich um die Zerstörung der Daten selbst kümmern – es sei denn, ein übereifriger Compiler hat die `memset`-Funktion wegoptimiert. Auch C++ bietet die Möglichkeit, Speicherbereiche zu löschen, aber wie bei C muß sie vom Programmierer auch genutzt werden.
Java ist in diesem Bereich völlig unhandlich. Mit der Zerstörung nicht mehr referenzierter Objekte ist ein nicht-berechenbarer und kaum steuerbarer Automatismus beauftragt – der Garbage Collector. Es gibt zwar eine Möglichkeit, ihn mit dem Löschen der nicht mehr verwendeten Objekte zu beauftragen, allerdings ohne Garantie dafür, daß es auch wirklich passiert.
Eigentlich brauchen wir eine Sprache, die automatisch sicherstellt, daß jegliche nicht mehr verwendete Information sofort gelöscht wird.

Ein anderes Problem ist der virtuelle Speicher. Wird mehr Arbeitsspeicher benötigt, als vorhanden ist , wird ein Teil auf die Festplatte ausgelagert. Viele Betriebssysteme löschen diese Daten nicht einmal, wenn der Computer heruntergefahren wird, geschweige denn, wenn er abstürzt. Es gibt auch keinen Mechanismus, um das zu unterbinden. (Allerdings gibt es Betriebssysteme, denen man die Auslagerung bestimmter Speicherbereiche verbietet kann, selten auch solche, die ausgelagerte Daten verschlüsseln.) Den Umgang mit dem virtuellen Speicher können wir auf der Implementierungsebene kaum verbessern, hier ist eine Maßnahme von seiten der Betriebssystemhersteller nötig.

Aber selbst wenn wir regulär Daten aus dem Speicher löschen, ist es möglich, daß die Änderung in der Code-Kopie des Cache-Speichers hängenbleibt und nicht in den RAM geschrieben wird. Wie man letzteres erzwingt, ist den Autoren für kein Betriebssystem bekannt. Zumindest aber kann in fast keinem Fall eine andere Instanz als das Betriebssystem selbst auf den Cache zugreifen.

Ein weiteres Problem bereitet eine physische Schwachstelle der Arbeitsspeicher-Chips: Der Speicher hält seine Daten länger als nur für die Zeit, die er unter Strom steht, besonders wenn häufig an der gleichen Stelle gleiche Daten abgelegt werden. Die Autoren sprechen zwar von SRAM und DRAM – heute längst überholt – aber dieser Effekt muß bei modernen Chipsätzen erst noch ausgeschlossen werden. Eine eher sperrige Methode, um für kleinere Datenmengen dieses Problem zu unterbinden, ist die Ver-x-oderung der Daten mit einem Zufallsstring, der in regelmäßigen Abständen (100ms) geändert wird, wobei auch die Daten wieder neu ver-x-odert werden. Bei größeren Datenmengen sollte man diese lieber als ganzes über längere Zeit verschlüsselt speichern und nur den Schlüssel mit der beschriebenen Methode im RAM halten.

Weitere Gefahren für im Speicher liegende Geheimnisse sind zur Laufzeit vorhanden, etwa wenn zwei Programme auf den gleichen Speicherbereich Zugriff haben, weil sie die gleiche Programmbibliothek benutze. Auch kann man in manchen Windows-Versionen einen Debugger an einen bereits laufenden Prozeß hängen, der natürlich auch den Speicher lesen kann. In UNIX kann man unter bestimmten Umständen von einem Programm einen Core-Dump erzwingen, d.h. eine Datei mit dem verwendeten Speicher anlegen. Und ein Superuser kann selbstverständlich den gesamten Speicher lesen.

4.2 Programmierhinweise

Einige Hinweise zur Entwicklung sicheren Codes sollen auch hier gegeben werden, obwohl sie eigentlich in ein Programmierhandbuch gehören.

- Einfachheit: Komplexität ist der größte Feind der Sicherheit. Alle unnötigen Optionen sollten weggelassen werden, besonders wenn sie die Sicherheit abstufen können.
- Modularisierung: Komplexität kann man auch durch Aufspalten der Software in einzelne Segmente begegnen, die – wo nötig – über eine schlanke Schnittstelle miteinander verbunden sind.
- Zusicherungen: Überall, wo die Konsistenz eines Programmes wirkungsvoll überprüft werden kann, sollte das getan werden – und entsprechende Mitteilungen über das Ergebnis gemacht werden.
- Pufferüberläufe sind schon seit Jahrzehnten bekannt, und etwa so lange gibt es auch schon Lösungen dafür. C und C++ sind sehr anfällig dafür. Die Hälfte aller Sicherheitsprobleme kommt durch Pufferüberläufe.
- Testläufe: Code sollte vom Entwickler selbst und von einer weiterer Person getestet werden. Zufällig generierte Testfälle sind hilfreich, ebenso wie kleine Tests beim Starten eines Programmes.

4.3 Seitenkanalangriffe

Seitenkanäle sind z.B. die verbrauchte Rechenzeit, der Stromverbrauch, elektromagnetische Abstrahlung, Speicherbedarf, Reaktionen auf Falscheingaben u.ä. Eine Analyse des Stromverbrauches ist besonders bei Smart Cards sehr wirkungsvoll.

Beispielhaft soll hier noch auf die Rechenzeitanalyse (Timing attack) eingegangen werden. Der Angreifer generiert zwei disjunkte Mengen von Eingabedaten. Ist ein bestimmtes Bit im Schlüssel „Eins“, werden Daten aus Menge A etwas schneller berechnet als Daten aus der Menge B. Ist das Bit „Null“, gibt es keinen Unterschied. Die Eingabemengen sind so gewählt, daß jeweils ein bestimmtes Bit des Schlüssels angesteuert werden kann. Hat man dieses entschlüsselt, fixiert man das nächste. Bei mehreren Millionen Anfragen ergibt sich ein statistisches Mittel des Zeitunterschiedes, sodaß auch ein Verrauschen der Rechenzeit keine Verbesserung bringt: Durch ausreichend viele Anfragen wird das Rauschen, das nichts anderes ist als die Verzögerung um einen zufällig gewählten Wert, wieder geglättet. Die Verzögerung sollte daher nicht zufällig gewählt werden, sondern genau so, daß jede Berechnung die gleiche Zeitspanne benötigt. Jetzt sind Rechenzeitanalysen zwar ausgeschlossen, aber es läßt sich ein Unterschied in der elektromagnetischen Abstrahlung des Rechners feststellen, je nachdem, ob er gerade rechnet oder „nur“ verzögert.

Obgleich man Seitenkanalangriffe kaum vermeiden, sondern allenfalls erschweren kann, sollte der Versuch bei der Implementierung nicht ausgelassen werden. In der Praxis spielen Seitenkanalangriffe kaum eine Rolle.

4.4 Aspekte der Protokollimplementierung – große Ganzzahlen

Im folgenden soll es nicht mehr wie bisher vordergründig um die Entwicklung von Anwendungssoftware allgemein gehen, sondern um die Implementierung von Verbindungsprotokollen.

Da wir bei der Verschlüsselung oft mit sehr großen Zahlen rechnen müssen, sind Fehler im Berechnen großer Ganzzahlen sicherheitskritisch. Wir können diese Berechnungen nicht auf die programmiersprachliche Ebene heben und so kontrollieren, da sie dadurch zu langsam würden. Besser ist optimierter Code für die jeweilige Hardware, der oft in etablierten Bibliotheken bereitgestellt wird. Unsere Aufgabe wird es sein, diese zu testen. Das ist nötig, da z.B. Routinen zur Division großer Ganzzahlen Codesegmente enthalten, die nur alle 2^{32} Divisionen ausgeführt wird. Auf 64-Bit-CPUs wird es gänzlich unmöglich, Fehler in solchen Segmenten durch zufälliges Testen zu finden. Trotzdem sind theoretische Fälle bekannt, wo Schlüssel aufgrund solcher Fehler preisgegeben werden.

Eine Möglichkeit, eine Rechenoperation modulo eine kleine Primzahl zu testen, ist das Wooping. Man macht sich zunutze, daß zum Beispiel $10 * 11 \ mod \ 7$ dasselbe ergibt (nämlich 5) wie $[(10 \ mod \ 7) * (11 \ mod \ 7)] \ mod \ 7$. Das gleiche läßt sich für die Addition sagen $(10 + 11)$. Man hält also zu allen großen Ganzzahlen n den Woop-Wert $n \ mod \ 7$ und führt die Berechnung einmal auf den großen Ganzzahlen durch und einmal auf den Woop-Werten. Der Woop-Wert beider Ergebnisse muß übereinstimmen, dann hat die Bibliothek richtig gerechnet. Wooping funktioniert nicht nur auf den konventionellen Rechenoperationen selbst, sondern auch auf deren Modulo-Varianten.

Normalerweise sind die Ganzzahlen natürlich mindestens tausend Bit lang; die entsprechenden Operationen in Bibliotheken kommen vor allem bei der Schlüssel-Berechnung vor. Auch verwendet man nicht eine Zahl wie Sieben für die Berechnung des Woop-Wertes, sondern diese Primzahlen sind zwischen 64 und 128 Bit lang. Und geheim – denn das Wooping findet während der normalen Arbeitszeit der Bibliothek statt, sodaß eine sicherheitskritische Anwendung sofort abbrechen kann, sollte der Wooping-Test fehlschlagen. Da wir aber annehmen müssen, daß ein Angreifer den Fehler in unserer Bibliothek eher gefunden hat als wir, ist zu befürchten, daß er genau so einen (falschen!) Schlüssel verwendet, der den Fehler ausnutzt und dazu noch beim Wooping-Test das richtige Ergebnis liefert, um uns seine Nachrichten unterzuschieben.

Für eine sichere Verbindung nach dem DH-Algorithmus ist eine fehlerhafte Bibliothek nicht sicherheitskritisch. Der Test der Berechung wird zwar nicht auf der Seite der Berechnung selbst durchgeführt, aber im Fehlerfall bricht das Protokoll einfach ab. Anders ist das beim RSA-Protokoll, das verbreiteter ist als Diffie-Hellman und im wesentlichen die gleiche Funktion erfüllt. Hier kann eine fehlerhafte Berechnung zur Preisgabe der Nutzdaten oder sogar des Schlüssels führen. Da wir oft mit großen Ganzzahlen rechnen, interessiert uns eine Methode, mit der wir die Effizienz der Modulo-Multiplikation steigern können. Die klassische Berechnung für $A * B \ mod \ N$ ist eine Division, bei der man wiederholt ein passendes Vielfaches von N vom Ergebnis $A * B$ abzieht. Die grundlegende Idee von Montgomery dagegen ist, die Division durch N mit einer Division durch 2^k zu ersetzen, da sich diese mit einer einfachen Stellenverschiebung in der Binärdarstellung der Zahl realisieren läßt. Das Ergebnis dessen ist immer eine Zahl zwischen 0 und $2N - 1$, die sich nunmehr leicht auf $modulo \ N$ reduzieren läßt.

Werden kryptographische Protokolle implementiert, die über einen sicheren Kanal laufen, sollten zwar alle Sicherheitsmerkmale des Kanals genutzt werden, um das Protokoll zu entlasten, aber aufgrund der Regeln der Modularisierung sollte die Abhängigkeit des Protokolls von der darunterliegenden Schicht möglichst klein sein. Das erfordert z.B. auch die evtl. redundante Implementierung des Replay-Schutzes (Schutz vor der Wiedereinspielung alter Nachrichten durch einen Angreifer).

Trifft eine Nachricht ein, muß zunächst Protokollname und Version geprüft werden. Als nächstes wird eine Zuordnung zur Instanz des Protokolls benötigt, denn möglicherweise existiert zwischen beiden Partnern nicht nur eine, sondern mehrere Verbindungen. Als letztes wird die Nachrichtennummer geprüft, um die Nachricht innerhalb ihres Protokolls einordnen zu können. Läßt die Nachricht, bzw. ihre Nummer die Interferenz eines Dritten vermuten, muß das Protokoll abgebrochen werden.

Ein richtiges Timeout-Verhalten ist ebenfalls erforderlich. Werden Protokolle nicht nach einer ausreichenden Zeitspanne ohne Rückmeldung auf die versendeten Nachrichten beendet und ihr Zustandsobjekt aus dem Speicher gelöscht (!), ist man einem eventuellen Angriff von millionenfachen Verbindungsaufbau-Anforderungen ausgesetzt: Der Speicher kann durch die erzeugten Zustandsobjekte all dieser Verbindungen leicht überlaufen, und das System verweigert die Kooperation.

5 Das sichere Speichern von Daten

Wir haben bisher als selbstverständlich vorausgesetzt, daß für eine sichere Verbindung ein permanenter Schlüssel existiert. Aber wo wird dieser Schlüssel abgelegt, ohne daß unbefugter Zugriff möglich ist? PCs sind aufgrund der gelegentlichen Fremdnutzung zu unsicher. Besser sind schon PDAs. Eine andere Idee wäre, alle Schlüssel mit einem Paßwort zu chiffrieren und das Paßwort im menschlichen Hirn abzulegen. Die chiffrierten Schlüssel könnten dann überall gespeichert werden. Das Problem ist die Wahl des Paßwortes: Je kürzer das Paßwort, desto empfindlicher ist es gegen Angriffe. Ist es zwar lang, aber ein Wort aus dem natürlichen Wortschatz einer Sprache, läßt es sich durch Ausprobieren herausfinden. Es muß also lang sein und aus möglichst zufällig gewählten Zeichen bestehen – eine unpraktische Lösung. Hier bieten sich Paßphrasen an: sie lassen sich leichter merken und sind im Idealfall einzigartig.

Aber am besten ist es dennoch, ein „kurzes“ Paßwort intern in einen starken Schlüssel zu verwandeln: Mit der Salzen-und-Strecken-Methode. Diese Methode hasht den Eingabestring, wovon das Paßwort ein Teil ist, rekursiv, liefert am Ende einen Schlüssel von 256 Bit Länge zurück und garantiert folgendes: Ein Angreifer muß erstens für jedes auszuprobierende Paßwort 2^{20} oder mehr Rechenschritte ausführen (wegen „Streckung“ durch Rekursion). Ein Nutzer, der das Paßwort kennt, bemerkt von dieser Rechnung zeitlich fast nichts. Zweitens gibt es zu einem bestimmten Paßwort keinen eindeutigen Schlüssel, da beim ersten Rechenschritt auch ein mitgespeicherter Zufallswert (das „Salz“) abfällt, der durch die restlichen $2^{20} - 1$ Berechnungen mitgeschleppt werden muß. Dadurch können keine Paßwort-Schlüssel-Datenbanken angelegt werden, die eine Brute-Force-Methode zeitlich abkürzen würden.

Eine der besten und praktischsten Methoden, um Schlüssel zu speichern, sind Secure Tokens. Sie bestehen aus einem permanenten Speicher, auf dem der Schlüssel abgelegt ist. Gegen eine Brute-Force-Attacke gibt es eine Zugangssperre nach n Fehlversuchen. Problematisch wird diese Methode allerdings bei Benutzern, die den Schlüssel in ihrem (angeschalteten) Rechner stecken lassen und sich zur Mittagspause aufmachen.

Und was, wenn wir dem PC nicht trauen können, über den wir das Paßwort für unseren Secure Token eingeben? Wir müßten eigentlich den sämtlichen sicherheitskritischen Teil der Anwendung auf den Token auslagern, der dann schnell zu einem kleinen Computer mit Eingabemöglichkeit und Display heranwächst. Eine Banküberweisung etwa wird dann vom PC an den Token gesandt, mit dem sie der Benutzer signiert und dem PC zurücksendet.

Fingerabdruckscanner sind leicht zu umgehen, wie ein japanisches Experiment gezeigt hat. Daher sind sie als Zusatz-Merkmal lediglich in Anwendungen mit niedrigen Sicherheitsanforderungen brauchbar, etwa um zu verhindern, daß Kollegen sich mal eben ihre Paßwörter ausborgen. Single-sign-on-Systeme, die mit einem Master-Schlüssel alle anderen Schlüssel speichern, sind ebenfalls noch keine Lösung: Es gibt einfach keine Standardisierung, wie sich alle Anwendungen automatisch ihre Schlüssel von dem Single-sign-on-System holen können.

Um Datenverlust zu vermeiden, sollte man die Schlüssel nicht nur einmal hinterlegen. Vielmehr empfiehlt es sich, einen einfach zu benutzenden Token im täglichen Gebrauch zu verwenden, und einen sicheren zu besitzen (etwa in einem Banktresor), um nicht Zugang zu Daten zu verlieren, nur weil man z.B. dreimal ein Paßwort falsch eingegeben hat.

Selten verwendet man auch Schlüssel, die aus mehreren Teilen bestehen, die je einer anderen Person anvertraut werden. Zugang kann dann nur gewährt werden, wenn eine Mindestzahl von Personen sich zusammentut, bzw. gleichzeitig präsent ist. Für den Rest der Zeit müßte gegenseitiges Mißtrauen garantiert werden können, was aber z.B. in den meisten Firmenvorständen nicht der Fall ist. Diese Lösung ist aus all diesen Gründen unpraktisch.

Wie aus dem Arbeitsspeicher nach einer sicheren Verbindung müssen auch von permanenten Medien Geheimnisse wieder gelöscht werden, sobald sie nicht mehr gebraucht werden, um Mißbrauch zu vermeiden. Magnetische Medien sind schwer zu löschen. Das Beste ist, 50-100 mal zufällige Daten in genau (!) den betreffenden Bereich zu schreiben und möglichst noch ein wenig davor und danach (bei Festplatten). Falls sich der Benutzer davon überzeugen läßt, ist die beste Lösung natürlich die vollkommene physische Zerstörung des Datenträgers.

6 Schluß

Anhand des letzten Satzes wird schon das vielleicht wichtigste Hindernis für Sicherheit bei der Telekommunikation deutlich: Das richtige Bewußtsein dafür existiert nur bei einer Minderheit der Nutzer. Gleiches gilt für die Hersteller besonders von Software. Wie wir gesehen haben, ist nämlich rein technisch Sicherheit schon zu einem durchaus befriedigendem Ausmaß realisierbar – auch wenn ihre Praktikabilität User und Hersteller gleichermaßen oft nicht zu der nötigen Kooperation zu begeistern vermag.

Literatur

[1] Niels Ferguson, Bruce Schneier: Practical Cryptography; Kapitel 8, 9, 15, 16 und 22, 2003.