Entschlüsselung von SSH-Verbindungen für die Angriffsanalyse in High-Interaction-Honeypots

---

Bachelorarbeit

von

Lion Hellstern

Universität Potsdam
Institut für Informatik
Professur Betriebssysteme und Verteilte Systeme

Aufgabenstellung und Betreuung:
Prof. Dr. Bettina Schnor
Dipl.-Inf. Sven Schindler

Potsdam, den 20. September 2014

Hellstern, Lion
lion@familie-hellstern.de
Entschlüsselung von SSH-Verbindungen für die Angriffsanalyse in High-Interaction-Honeypots
Bachelorarbeit, Institut für Informatik
Universität Potsdam, September 2014

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Potsdam, den 20. September 2014

_____

Lion Hellstern

Zusammenfassung

In dieser Bachelorarbeit wird die OpenSSH-Software modifiziert. Die Modifikation dient dazu SSH-Verbindungen eines Honeypots zu entschlüsseln. Angriffe auf den Honeypot über das SSH-Protokoll können dann analysiert werden. Dafür wird zunächst auf Grundlagenwissen und mögliche Lösungsansätze zum Entschlüsseln eingegangen. Danach wird gezeigt, welche Änderungen an der Software vorgenommen werden. Zum Schluss belegt ein Proof of Concept die prinzipielle Funktionalität der Software.

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit handelt von der Entschlüsselung von SSH-Verbindungen des low-interaction Honeypots Honeydv6, der in der Lage ist IPv6-Netzwerke zu simulieren. Das Honeydv6-Netzwerk hat den Zweck, potentielle Angreifer durch Sicherheitslücken in das System zu lassen, um ihr Vorgehen zu analysieren.
Ist ein Angreifer auf eine simulierte Maschine des Honeypots aufmerksam geworden, wird er bemerken, dass der Port 22 des Rechners offen ist. Dadurch weiß er, dass auf diesem Rechner der SSH-Daemon gestartet ist. Er wird versuchen sich über das SSH-Protokoll mit dem Rechner zu verbinden. Da der Rechner darauf ausgelegt ist angegriffen zu werden, gelingt es ihm. Möglich ist dies, in dem der Angreifer das schwache Passwort errät oder einen Exploit einsetzt.
Nach erfolgreichem Systemzugriff wird er weitere Kommandos abschicken, um die Konfiguration zu ändern oder Schadware zu installieren. Da SSH-Verbindungen allerdings verschlüsselt sind, ist unklar, welche Kommandos abgeschickt werden. Wären sie nicht verschlüsselt, könnte man die SSH-Pakete abfangen und wüsste, was auf der kompromittierten Maschine passiert.
Ziel der Arbeit ist es, den SSH-Daemon auf den Rechnern des Honeydv6-Netzwerkes zu modifizieren. Nach der Veränderung soll der Daemon den Schlüssel des symmetrischen Verschlüsselungsverfahrens an den Honeydv6 schicken. Während der SSH-Sitzung werden die Pakete abgefangen und im Nachhinein mithilfe des Schlüssels entschlüsselt. Der Angriff kann jetzt analysiert werden.

Struktur der Arbeit

Der Schwerpunkt der Arbeit liegt in der Modifikation des SSH-Daemons. Dazu werden Teile des originalen Quelltextes erklärt und es wird erläutert an welchen Stellen Veränderungen notwendig sind.
Im Folgenden wird der rote Faden der Arbeit verdeutlicht. Nach dem in der Einleitung das Problem geschildert wurde, wird in dem nächsten Kapitel beschrieben, ob andere Personen bereits einer ähnlichen Problematik gegenüberstanden, und zu welchen Lösungen sie gekommen sind. Danach wird in Kapitel 3 auf die Grundlagen eingegangen, die notwendig sind, um die Arbeit zu verstehen. Dazu gehören Kenntnisse zu dem SSH-Protokoll, zu Honeypots und dem OpenSSH Server.
Durch dieses Wissen können im Anschluss die Ansätze diskutiert werden, mit denen das Ziel erreicht werden kann. Dabei wird im Detail der gewählte Lösungsweg betrachtet. Über seine Umsetzung geht es in dem Kapitel Umsetzung. Dort wird darauf eingegangen, wie die einzelnen Komponenten des SSH-Daemons zusammen spielen, wie der Schlüssel erzeugt und gespeichert wird und letztlich wie dadurch Pakete entschlüsseln werden können.
Das letzte Kapitel liefert eine Zusammenfassung.

2 Related Work

Dieses Kapitel geht auf zwei Artikel und ein Programm ein, die sich mit der Entschlüsselung von SSH-Verbindungen beschäftigen.
Die Arbeit von Xuqing Tian Song und David Wagner[DXS01], mit dem Titel *Timing Analysis of Keystrokes and Timing Attacks on SSH*, nutzt das Protokolldesign von SSH aus. Mit jedem eingetippten Symbol schickt der SSH-Client ein Paket an den Server. Dadurch lassen sich Informationen über das Passwort, wie zum Beispiel seine Länge, extrahieren. Außerdem stellen die Autoren das Programm *Herbivore* vor, welches weitere Informationen über das Passwort herausfindet, in dem es die Tippangewohnheiten eins Clients während einer SSH-Sitzung analysiert. Durch diese Zusatzinformationen soll das Erraten eines Passworts um den Faktor 50 vereinfacht werden.
Dieser Ansatz SSH-Verbindungen zu analysieren, zeigt auf welch vielfältige Art und Weise das in der Einleitung beschriebene Problem gelöst werden könnte. Da wir in dem Szenario dieser Arbeit vollen Zugriff auf den Honeypot haben, gibt es einfachere Wege, SSH-Verbindungen zu entschlüsseln.
In [Gut11] wird darauf eingegangen, dass Benutzer trotz fehlgeschlagener Authentifizierung des Servers, SSH-Verbindungen akzeptieren. Der Grund ist, dass die Fehlersuche den Benutzer zu stark von seinem eigentlichen Vorhaben abbringen würde. Dies ist ein Aspekt, der einen Man-in-the-middle Angriff[BSI07] begünstigt, welcher durch die zentrale Rolle des Honeypots möglich ist.

Das SSH Man-in-the-middle tool(`mitm-ssh`) wurde von Claes M Nyberg entwickelt. Es ist ein geänderter OpenSSH Server, welcher sich zwischen Client und Zielrechner schaltet. Dazu muss der Client zum `mitm-ssh` umgeleitet werden. Um dies zu erreichen, werden zwei Wege vorgeschlagen. Der Erste ist, schneller auf DNS-Anfragen zu antworten als der Ziel-DNS Server. Der zweite Weg ist, sich durch ARP Reques Poisoning [BSI06] zwischen Client und Zielrechner zu schalten. In Bezug auf den Honeydv6 könnte man den Traffic direkt zum `mitm-ssh` leiten, da der Honeydv6 Man-in-the-middle ist. Problematisch ist, dass es zum `mitm-ssh` keine Dokumentation gibt. Es ist nicht nachvollziehbar, welche Änderungen der Autor an der Software gemacht hat. Da `mitm-ssh` möglicher Weise unerwünschte Funktionalität besitzt, ist es für diese Arbeit nicht verwendbar.

3 Grundlagen

Das Secure Shell (SSH) Protokoll dient zum sicheren Login und Arbeiten auf entfernten Computern. Die Sicherheit wird durch kryptografische Verfahren gewährleistet, die Datenintegrität, Authentizität des Benutzers und Vertraulichkeit zwischen Client und Server garantieren. Dadurch bietet es eine sichere Alternative zu älteren Protokollen, wie `rlogin` oder `telnet`, welche Informationen unverschlüsselt im Netzwerk übertragen.
Die erste Implementierung des Protokolls erschien im Jahr 1995 und wurde von Tatu Ylönen entwickelt. Inzwischen existieren neben vielen anderen Implementierungen zwei Weiterentwicklungen. Zum einen die Open Source Variante OpenSSH und die proprietäre Software SSH Tectia. In dieser Arbeit wird der OpenSSH [OPE99] fokussiert.
Neben dem verschlüsselten und authentifizierten Login auf einen Rechner, bietet SSH weitere Funktionalität. Daten können komprimiert übertragen werden, um auch bei niedriger Bandbreite schnell Senden und Empfangen zu können. Weiterhin ist verschlüsselte Übertragung von X11-Datenverkehr möglich, so dass auch grafische Programme benutzt werden können. Dazu wird das X11 Protokoll mittels SSH getunnelt. Das bedeutet, die Pakete des X11 Protokolls werden an die SSH-Anwendung weitergegeben. Diese schickt sie sicher über das Netzwerk und der Empfänger leitet die X11 Pakete an seine dafür vorgesehene Anwendung weiter. Auch andere Protokolle, wie HTML oder SMTP können mit SSH getunnelt und dadurch gesichert werden.

3.1 Interne Funktionsweise

Die interne Architektur von SSH beruht auf drei Protokollen. Sie befinden sich im OSI-Modell oberhalb von TCP. Das *SSH Transport Layer Protocol* nimmt die TCP-Pakete entgegen. Es ver- und entschlüsselt sie, komprimiert sie und gewährleistet Vertraulichkeit und Integrität. Außerdem ist es für die Authentifizierung des Server zuständig.
Das *SSH User Authentication Protocol*, sowie das *SSH Connection Protokoll* liegen oberhalb des SSH Transport Layer Protocols. Sie authentifizieren den Client und multiplexen mehrere logische Channel über eine SSH-Verbindung. Ein Channel kann als Terminalsession, als X11-Verbindung oder zur Portweiterleitung benutzt werden.

Abbildung 3.1 veranschaulicht den initialen Nachrichtenaustausch eines SSH-Verbindungsaufbaus, welcher durch das *SSH Transport Layer Protocol* geregelt wird. Zu Beginn tauschen Client und Server die Softwareversionsnummer aus. Diese Informationen, in Form von *"SSH-protoversion-softwareversion SP comments CR LF*[1] *"*, dienen später als *identification string*.
Als nächstes findet die Einigung über die benötigten Algorithmen statt. Dafür besitzen Client und Server jeweils eine Liste von bevorzugten Algorithmen. Diese werden über die SSH-MSG-

[1] SP: Space CR: Carriage Return LF: Line Feed

KEXINIT Nachrichten dem anderen mitgeteilt. Benutzt werden die Algorithmen, die an oberster Stelle in den Clientlisten sind und auch vom Server unterstützt werden. Zur Auswahl stehen unter anderem *aes256-cbc, arcfour, 3des-cbc, none* zum ent-/verschlüsseln, *hmac-sha1, hmac-sha1-96, hmac-md5, none* zur Generierung eines Message Authentication Codes (MAC) und *none, zlib* zum Komprimieren [TY06].
Danach wird in dem *Key Exchange*-Block der Diffie-Hellman-Schlüsselaustausch durchgeführt.

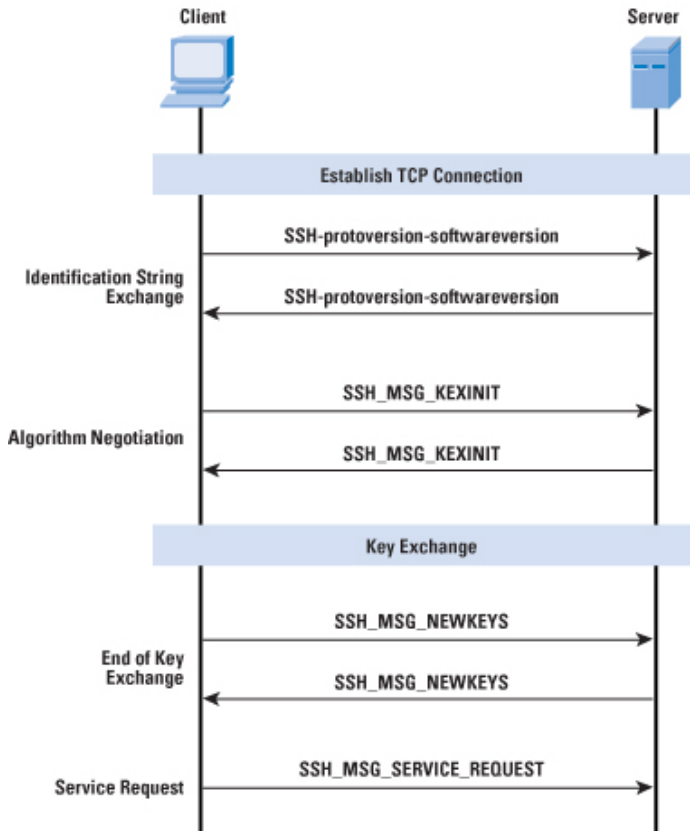

Abbildung 3.1: Trasport Layer Protocol - Nachrichtenaustausch [Sta09]

3.2 Diffie-Hellman-Schlüsselaustausch

Die Kommunikationspartner verschlüsseln den Kanal, über den sie ihre Nachrichten versenden, mithilfe eines gemeinsamen Geheimnisses. Dieses Geheimnis wird durch das Diffie-Hellman Verfahren erzeugt und ausgetauscht. Der Austausch geschieht derart, dass man das Geheimnis nicht erfährt, auch wenn man den Austausch mitliest.
Gegen einen Man-in-the-Middle Angriff ist das Verfahren jedoch nicht geschützt. Damit SSH gegen diese Art von Angriffen geschützt ist, wird bei dem Geheimnisaustausch zusätzlich eine digitale Signatur mitgesendet. Das bedeutet, dass der Server, abgesehen von dem gemeinsamen Geheimnis des Diffie-Hellman-Schlüsselaustauschs, zusätzlich ein eigenes Geheimnis mit einbringt,

das er als einziger kennt. [TY06] Dadurch kann er authentifiziert werden. Folgende Schritte werden zum Schlüsselaustausch im Schritt Key Exchange der Abbildung 3.1 ausgeführt:

1. Client und Server sind nach der SSH-MSG-KEXINIT Nachricht die Primzahl p und die Zahl g, mit $1 \leq g \leq p-1$ bekannt. g ist Erzeuger einer zyklischen Gruppe mit der Ordnung p.

2. Client generiert die Zufallszahl x und berechnet $e = g^x mod\, p$. Der Client sendet e zum Server.

3. Server generiert die Zufallszahl y und berechnet $f = g^y mod\, p$. Der Server empfängt vom Client e. Der Server berechnet $K = e^y mod\, p$ und $H = hash($ *identification string client* $||$[2] *identification string server* $||$ *SSH-MSG-KEXINIT vom Client* $||$ *SSH-MSG-KEXINIT vom Server* $||$ *Server öffentlicher Schlüssel* $||e||f||K)$. Der Server berechnet die Signatur s auf H mit seinem privaten Schlüssel. Der Server sendet: (*Server öffentlicher Schlüssel* $||f||s)$.

4. Client verifiziert den öffentlichen Schlüssel des Servers. Der Client berechnet $K = f^x mod\, p$ und H. Dann vergleicht er H mit der Signatur.

Nun besitzen Client und Server ein gemeinsames Geheimnis. Außerdem ist der Server authentifiziert. H gilt als *session identifier* für diese Verbindung.
Zur Verschlüsselung, sowie zur Berechnung des Message-Authentication-Codes(MAC) wird allerdings nicht K benutzt, sondern ein Hash, der sich aus dem Geheimnis K errechnet. Zur Berechnung des Hashes, muss die gleiche Hashfunktion benutzt werden, die im Diffie-Hellman-Schlüsselaustausch benutzt wurde. Der Schlüssel für die Client-zu-Server Verbindung ist *Hash(K || H || "C" || session identifier)*. Der Schlüssel für die Server-zu-Client Verbindung ist *HASH(K || H || "D" || session identifier)*.
Das Ende des Schlüsselaustauschs wird durch die SSH-MESSAGE-NEWKEYS Nachricht signalisiert. Danach wird eine SSH-MSG-SERVICE-REQUEST Nachricht versandt, um die Clientauthentifizierung zu beginnen oder einen neuen Channel mit dem SSH Connection Protokoll zu öffnen.

3.3 OpenSSH

Mit der Veröffentlichung von OpenBSD 2.6 wollte man eine saubere, schnelle und freie Implementierung von SSH mitliefern. Dazu wurde sich für eine von Björn Grönvall verbesserte Version des *free ssh 1.2.12* von Tatu Ylönen entschieden. Am Ende des Jahres 1999 wurde OpenSSH 1.2.2 veröffentlicht. Zum größten Teil entwickelt von Tatu Ylönen und durchdringend weiterentwickelt von Björn Grönvall, Theo de Raadt, Niels Provos, Markus Friedl, Bob Beck, Aaron Campbell und Dug Song. Sie entfernten jeglichen Code, der unter einer einschränkenden Lizenz stand, entfernten Bugs und fügten viele Features hinzu. [OSS99a]
Im Jahr 2008 sind ca. 86% aller SSH-Server im Internet OpenSSH-Implementationen [OPE99]. Ein wichtiger Aspekt, da im Rahmen dieser Bachelorarbeit eine SSH Implementierung modifiziert werden soll, um die Kommandos von Angreifern des Honeypots zu entschlüsseln. Ist diese

[2] Konkatenation

Implementierung nicht kompatibel mit der von den Angreifern, kann keine Verbindung aufgebaut werden.
In Tabelle 3.1 sieht man die Ausgabe des Programms cloc[3], angewandt auf den Sourcecode von

Tabelle 3.1: count lines of code: openssh-6.6p1

Language	files	blank	comment	code
C	224	11072	14129	73313
Bourne Shell	86	4133	3049	23295
C/C++ Header	110	2180	3958	7612
m4	2	317	63	4700
Bourne Again Shell	6	226	332	2173
make	6	126	16	690
awk	1	7	23	340
XML	1	13	16	61
Perl	1	10	8	54
C Shell	1	0	0	1
SUM:	438	18084	21594	112239

OpenSSH 6.6p1. Zu erkennen ist, dass die meisten Zeilen in C und Bourne Shell geschrieben wurden und das Projekt mit insgesamt 112239 Zeilen Code auf 438 Dateien aufgeteilt, komplex ist. Der C-Code hat 19% Kommentare. Dies hilft sehr bei dem Verstehen des Codes, da es keine öffentliche Dokumentation gibt.
Die Komplexität des Projekts lässt sich darauf begründen, dass SSH wesentlich mehr Funktionalität bietet, als eine Verbindung zu einem entfernten Terminal aufzubauen. Neben dem ssh und dem sshd Programmen existieren weitere Programme. Die Programme `ssh-agent`, `ssh-add`, `ssh-keygen`, `ssh-keyscan` und `ssh-keysign` zum Verwalten von Schlüsseln für symmetrische und asymmetrische Verschlüsselungsverfahren. Mit ihrer Hilfe können Schlüssel erzeugt und zum Signieren benutzt werden. Das Programm sftp, welches wie das FTP-Protokoll arbeitet aber über SSH kommuniziert. Sowie das Programm scp zum sicheren Kopieren.
Die Komplexität des Projekts wird zusätzlich dadurch erhöht, dass es von OpenBSD auf andere unixoide Systeme, wie Linux oder MAC OS portiert wurde. Mit Cygwin lasst sich OpenSSH auch auf einem Windowsbetriebssystem installieren[OSS99b]. Da die portierbare Variante des Codes aus der Sicht seiner Entwickler nicht mehr den Anforderungen eines sicheren und grundsoliden Programms erfüllte[OSS99a], gibt es eine grundlegende und eine portierbare Version von OpenSSH. Diejenigen, mit einem “p“ für portierbar in der Versionsnummer laufen auf Linux-, MAC- und Windowssystemen.
Die kryptografischen Funktionen wurden nicht selber implementiert, sondern von den OpenSSL Bibliotheken eingebunden.

[3]Count lines of code ist ein Programm zur Quellcodeanalyse

3.4 Honeydv6

Der Honeydv6 ist eine Erweiterung des “Honeyd“ Honeypots [HON08]. Er wurde in der Arbeitsgruppe von Prof. Dr. Bettina Schnor an der Universität Potsdam entwickelt und von Dipl.-Inf. Sven Schindler implementiert. Mit der Hilfe des Honeypots werden derzeit verschiedene Experimente durchgeführt.
Der Honeyd ist ein low-interaction Honeypot. Das bedeutet, er simuliert kein vollständiges Betriebssystem, sondern nur gewisse Aspekte. Der Honeyd implementiert den Netzwerkstack und ist somit in der Lage, auf Netzwerkpaketen von unterschiedlichen Protokollen zu antworten. Was im internen passiert, wenn ein Paket eintrifft, kann durch Scripte definiert werden. Dabei ist der Honeyd in der Lage, ein gesamtes Netzwerk zu simulieren. [Sch12] Er eignet sich gut, Angriffe zu analysieren, die ein ganzes Netzwerk betreffen, wie zum Beispiele Netzwerk-scanns, Würmer oder Viren.
Im Gegensatz dazu eignen sich high-interaction Honeypots dazu, gezielte Angriffe einzelner Systeme zu analysieren. Sie sind einzelne Computersysteme mit physikalischer oder virtueller Hardware. Schafft es ein Angreifer sie zu kompromittieren, hat er vollen Zugriff auf das System. Über dem Betriebssystem wacht in den meisten Fällen eine Software, die das Handeln des Angreifers protokolliert.

Der Honeyd kann Pakete an andere Systeme weiterleiten, anstatt sie direkt zu beantworten. Somit könnte ein Angreifer zu einem high-interaction Honeypot weitergeleitet werden. Der Honeyd ist in diesem Szenario der Vermittler zwischen Angreifer und high-interaction Honeypot. Dadurch ist es möglich, sowohl das gesamte Netzwerk, sowie spezielle Angriffe zu analysieren.
In IPv6-Netzwerken ergibt sich dabei jedoch ein Problem. Um als Angreifer einen einzelnen Rechner zu attackieren, muss er dessen Adresse kennen. Da es in IPv6-Netzwerken allerdings viele Adressen gibt[4], ist es unwahrscheinlich, dass einzelne gefunden werden.
Dieses Problem löst der Honeydv6, indem er dynamisch Hosts erzeugt. Wird ein TCP-SYN in das von dem Honeydv6 verwalteten Netzwerk gesendet, wird unter Berücksichtigung eines Zufallsfaktors, ein Host erzeugt und auf die Zieladresse des TCP-SYN-Pakets konfiguriert. Der Angreifer bekommt davon nichts mit und denkt er habe einen Rechner gefunden. Der Zufallsfaktor bestimmt, ob ein Host erzeugt wird oder nicht, da es auffällig wäre, wenn jede Adresse, die getestet wird, antwortet. Außerdem werden die Hosts nicht erst erzeugt, sondern vorgehalten und nur umkonfiguriert. Der Honeydv6 besitzt gewissermaßen ein Cache von Hosts, um die Performance zu steigern.
Der Honeydv6 ist dadurch nicht nur Vermittler, sondern auch Verwalter. Die simulierten Hosts werden mit der Software QEMU [QEM14] emuliert.

[4] Das Netzwerk des Honeydv6 ist 2^{48} Knoten groß

4 Lösungsansätze

In diesem Kapitel werden Ansätze vorgestellt, mit denen es möglich ist, die SSH-Verbindung eines Angreifers lesbar zu machen. Die sechs möglichen Lösungsstrategien sind in der Abbildung 4.1 veranschaulicht und werden im folgenden erklärt.

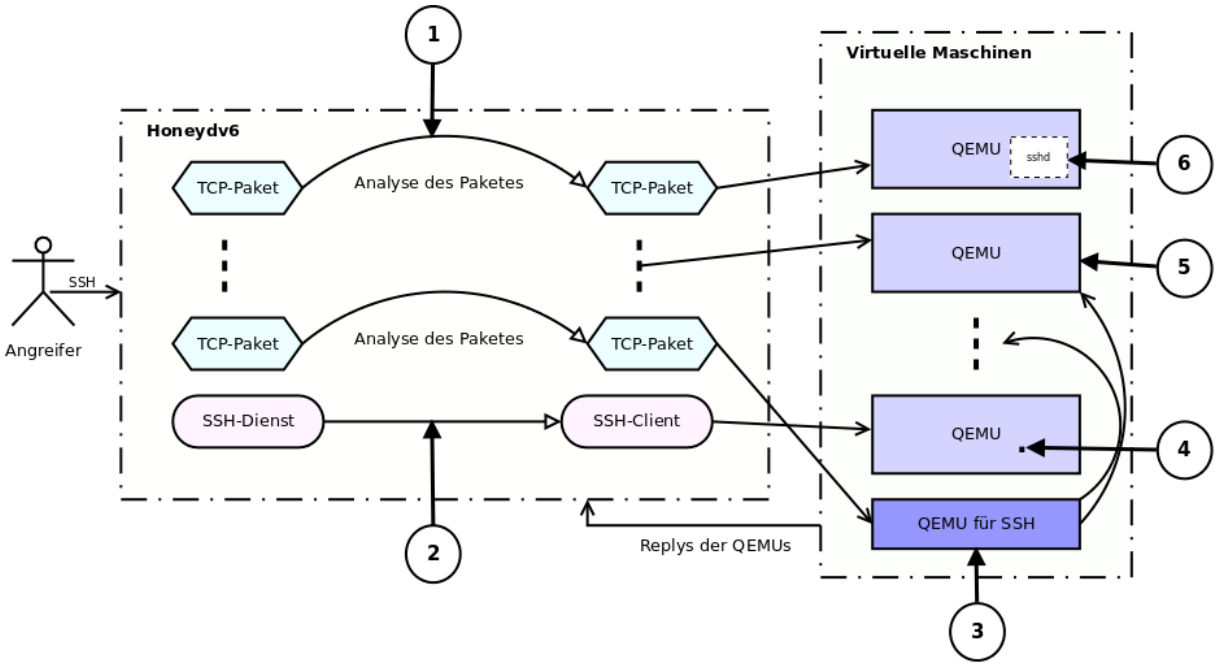

Abbildung 4.1: Lösungsstrategien im Honeydv6

Die Abbildung zeigt einen Angreifer, der Daten des SSH-Protokolls an den Honeydv6 sendet. Das Ziel ist es, diese Daten zu entschlüsseln. Der Honeydv6 hat Einblick in alle eintreffenden Pakete. Je nach dem von welchem Protokoll das Paket ist, wird es unterschiedlich behandelt. Trifft ein TCP-SYN Paket ein, führt der Honeydv6 ein 3-Way-Handshake durch. Gelingt das, wird ein QEMU umkonfiguriert, so dass er die IP-Adresse hat, an welche das TCP-SYN Paket geschickt werden sollte. Nun ist der Honeydv6 Vermittler zwischen dem QEMU und dem Sender. Jetzt wird die verschlüsselte SSH-Verbindung aufgebaut, welche analysiert werden soll.
Dadurch, dass der Honeypot Vermittler zwischen Angreifer und Host ist, bietet es sich an, einen Man-in-the-middle Angriff durchzuführen. Diese Idee wird von den Punkten Eins bis Drei aufgegriffen. Andererseits besitzen wir die Kontrolle über die virtuellen Maschinen, weshalb man versuchen könnte, die SSH-Sitzung dort zu protokollieren und zu analysieren. Dieser Ansatz wird in den Punkten Vier bis Sechs betrachtet.

1. Ein Man-in-the-middle Angriff auf die SSH-Pakete. Bei dem Verbindungsaufbau einer SSH-Sitzung ermittelt der Honeydv6 das Geheimnis des Diffie-Hellman Schlüsselaustausches des Angreifers. Mit seinem eigenen Geheimnis generiert er einen gemeinsamen Schlüssel für die Verbindung zwischen ihm und dem Angreifer. Gleichzeitig startet er einen SSH-Verbindungsaufbau mit dem QEMU, der Zieladresse. Nun besitzt der Honeydv6 zwei Schlüssel und ist der Man-in-the-Middel, wodurch die Entschlüsselung unwirksam ist.
   Vorteil: Unauffällig[1]
   Nachteile: Änderungen im Honeydv6, Schwierig zu Implementieren

2. Ein Man-in-the-Middle Angriff mit der Hilfe des SSH-Dienstes. Es werden ein SSH-Dienst und ein SSH-Client auf dem Honeydv6 gestartet. Der Angreifer kommuniziert mit dem SSH-Dienst. Die eingehenden Kommandos werden an den SSH-Client geschickt, welcher sie über eine SSH-Verbindung zu einem QEMU weiterleitet.
   Vorteil: Unauffällig
   Nachteil: Änderung des Honeydv6

3. Ein Man-in-the-Middle Angriff mit Hilfe des SSH-Dienstes auf einem QEMU. Die eintreffenden SSH-Pakete werden an einen speziellen QEMU gesendet. Dieser hat einen SSH-Dienst und einen SSH-Client gestartet, um SSH-Pakete zu lesen und die Kommandos an andere QEMUs weiter zu senden.
   Vorteile: Unauffällig, fast keine Veränderung des Honeydv6
   Nachteile: die Verwaltung mehrere SSH-Sitzungen ist aufwendig

4. Den SSH-Dienst der QEMUs modifizieren, so dass alle Kommandos in ein Log geschrieben werden. Der Angreifer führt eine normale SSH-Sitzung mit einem QEMU durch. Alle Kommandos, die bei dem QEMU ankommen werden von dem abgeänderten SSH-Dienst protokolliert.
   Vorteile: Einfach zu implementieren, keine Änderung des Honeydv6
   Nachteil: Auffällig, da das Log gefunden werden kann

5. Die gesamte virtuelle Maschine protokollieren. Da die QEMUs virtuell sind, kann ihr Speicher von außerhalb analysiert werden. Die Kommandos des Angreifers liegen im Speicher des QEMUs. Wird der Speicher mit seinen Veränderungen mitgeschrieben, werden auch die Kommandos mit deren Auswirkungen mitgeschrieben.
   Vorteil: Unauffällig
   Nachteil: Es ist aufwendig den Speicher zu protokollieren und die Kommandos zu extrahieren

6. Den SSH-Dienst modifizieren, so dass der Schlüssel nach dem Diffie-Hellman-Verfahren an den Honeydv6 geschickt wird. Der Angreifer startet eine SSH-Verbindung mit einem QEMU. Dieser schickt nach dem Schlüsselaustausch und bevor der Angreifer Kontrolle über die Maschine hat, den Schlüssel heraus. Alle SSH-Pakete werden mitgeschnitten und mit dem Schlüssel im Nachhinein lesbar gemacht.
   Vorteile: Unauffällig, keine Änderung des Honeydv6
   Nachteile: Entschlüsselung Aufwendig

[1] Man-in-the-Middle kann im SSH-Protokoll durch Zertifikate bemerkt werden. Diese Möglichkeit ist dem Angreifer aber nicht gegeben.

Umgesetzter Lösungsansatz

Die beschriebenen Lösungsstrategien haben den Trade-off zwischen der Unauffälligkeit gegenüber dem Angreifer und dem Aufwand der Umsetzung. Dies hängt damit zusammen, dass es desto Auffälliger wird, je stärker die Änderungen den Angreifer betreffen. Wird zum Beispiel eine ganze virtuelle Maschine protokolliert, bemerkt ein Angreifer das nicht. Protokolliert man die Kommandos allerdings direkt in dem SSH-Dienst oder der SHELL, besteht die Gefahr, dass der Angreifer den Log findet.
Den besten Trade-off haben die Punkte Drei und Sechs. Punkt Sechs überzeugt allerdings, da zunächst nicht mehr getan werden muss, als den SSH-Dienst etwas abzuändern. Mit dem herausgeschickten Schlüssel ist dann alles vorhanden, was benötigt wird, um die SSH-Verbindung analysieren zu können.

Im Detail gibt es bei diesem Lösungsweg einige Aspekte, die beachtet werden müssen. Es ist notwendig, dass der Schlüssel zum Honeydv6 übertragen wird, bevor der Angreifer Zugriff auf die Maschine hat. Ansonsten könnte der Angreifer mit dem netstat-Kommando[2] herausfinden, dass die Maschine eine Verbindung zu dem Honeydv6 hat. Deshalb ist es wichtig zu identifizieren, an welcher Stelle im Quellcode des SSH-Dienstes der Schlüsselaustausch geschieht. Wie in Kapitel 3.1 gezeigt, ist der Schlüsselaustausch das Erste, was bei dem Verbindungsaufbau stattfindet. Befindet sich die laufende Software an dieser Stelle, ist sichergestellt, dass der Schlüssel erzeugt wurde, der Angreifer allerdings noch keinen Zugriff auf die Maschine hat.
Zur Übertragung des Schlüssels an den Honeypot wird ein Socket erzeugt. Über diesem wird der Schlüssel als Bytefolge mit seiner Länge verschickt. Der Datentyp des Schlüssels ist *u_char*, damit er auf unterschiedlichen Systemen gleich interpretiert wird.
Unabhängig vom SSH-Dienst der virtuellen Maschine, muss der Honeydv6 den gesamten SSH-Datenverkehr zwischenspeichern und den unterschiedlichen Sitzungen der QEMUs zuordnen.
Will man eine Sitzung analysieren, müssen die Pakete untersucht werden. Mit jedem Zeichen, dass der Angreifer auf seiner Tastatur eintippt, wird ein Paket übermittelt. Mit den Paketen besitzt man lediglich ein Log über die gedrückten Tasten des Angreifers. Darum müssen die einzelnen Zeichen bzw. Pakete für die Analyse zu Kommandos zusammen gesetzt werden.
Außerdem muss entschlüsselt werden. Das ist nicht in einem Schritt getan. Ein SSH-Paket setzt sich aus unterschiedlichen Teilen zusammen. Abbildung 5.2 veranschaulicht den Aufbau eines SSH-Pakets. Es besitzt einen verschlüsselten Part und einen Part in dem der Message-Authentication-Code ist. Der verschlüsselte Part setzt sich wiederum aus der Paketlänge, der Paddinglänge, den Daten und dem Padding zusammen. Dabei können die Daten komprimiert sein.
Der verfolgte Ansatz zum Entschlüsseln der Pakete ist ein weiterer modifizierter OpenSSH-Server. Er kann mit dem Aufbau von SSH-Paketen umgehen und benötigt lediglich die richtigen Parameter. Anstatt eine Verbindung zu einem Client aufzubauen, führt sie einen lokalen Schlüsselaustausch mit fest in den Quellcode einprogrammierten Daten durch. Dann wird die Funktion zur Abarbeitung eines Pakets mit den Paketdaten als Parameter aufgerufen. Die hierzu notwendigen Informationen sind: der geheime Schlüssel, der Hashwert, die Session-ID der Sitzung, die Paketdaten und die Sequenznummer des Pakets. In welchen Variablen die Informationen gespeichert werden müssen, wird im Kapitel 5 Umsetzung behandelt.

[2] Ein Unixtool zum anzeigen von Netzwerkinformationen

5 Umsetzung

Für die Umsetzung werden zwei Modifikationen der OpenSSH-Software benötigt. Zum einen die Variante, die auf den virtuellen Maschinen läuft. Sie wird den Schlüsselaustausch mit dem Angreifer durchführen. Danach sendet sie alle notwendigen Information an den Honeydv6 und arbeitet dann als normaler SSH-Dienst. Zum anderen die Variante welche zum Entschlüsseln benutzt wird. Sie wird unabhängig von dem Angriff gestartet, erhält alle erforderlichen Informationen und entschlüsselt die Pakete.

5.1 Schlüsselversand mit Hilfe von OpenSSH

Der OpenSSH-Server ist ein durch die Datei *sshd.c* definierter Daemon. In dieser Datei befindet sich eine MAIN()-Funktion, welche bei dem Start des OpenSSH-Servers ausgeführt wird. In der MAIN()-Funktion wird nach der Behandlung möglicher Optionen, die dem OpenSSH-Server mitgegeben werden können, ein Socket geöffnet und auf dem Port 22 auf Verbindungsanfragen gewartet. Trifft eine Anfrage ein, wird die SSH-Sitzung initialisiert. Da das Hauptmerkmal von SSH ist sicher zu sein, wird wie in der Abbildung 3.1 aus dem Kapitel 3.1 Interne Funktionsweise der Schlüsselaustausch durchgeführt, um danach eine verschlüsselte Kommunikation führen zu können.
Während des Schlüsselaustausches werden private, öffentliche und ein gemeinsamer Schlüssel erzeugt. Dazu können verschiedene kryptografische Verfahren benutzt werden. Um die Schlüssel korrekt zu erzeugen und auszutauschen, müssen Datenstrukturen verwendet werden, welche auf jedem System gleich interpretiert werden.
Damit die Art und Weise verständlich wird, wie der SSH-Daemon dies Umsetzt, ist es sinnvoll, seine Funktionsaufrufe zu verfolgen. Gelangt man zu dem Punkt, an dem der gemeinsame Schlüssel erzeugt wird, weiß man, wie dieser abgespeichert wird und welche Informationen zu seiner Erzeugung benötigt wurden.

Die Abbildung 5.1 visualisiert die Funktionsaufrufe. Dabei werden Funktionen aus den Dateien *sshd.c, kex.c, dispatch.c, kexc25519.c* und *kexc25519s.c* verwendet. *kex.c* steht für key exchange. Durch diese Datei wird der Schlüsselaustausch koordiniert.
In *dispatch.c* werden dynamisch Funktionen aufgerufen. Das bedeutet, es gibt ein Array von Zeigern auf Funktionen, mit der Definition *dispatch_fn *dispatch[DISPATCH_MAX]*. Je nach dem welcher Typ von Paket bei dem SSH Daemon eintrifft, kann die für den Typ zuständige Funktion im Array ausgewählt und aufgerufen werden. So wird für ein `new_keys`-Paket die Funktion zum Aushandeln neuer Schlüssel, statt der Funktion zum entschlüsseln von SSH-Paketen aufgerufen.
In den Dateien *kexc25519s.c* und *kexc25519.c* werden Funktionen der SSL-Bibliothek aufgerufen, um kryptografische Verfahren durchzuführen. In diesen Dateien wird der Schlüssel erzeugt.

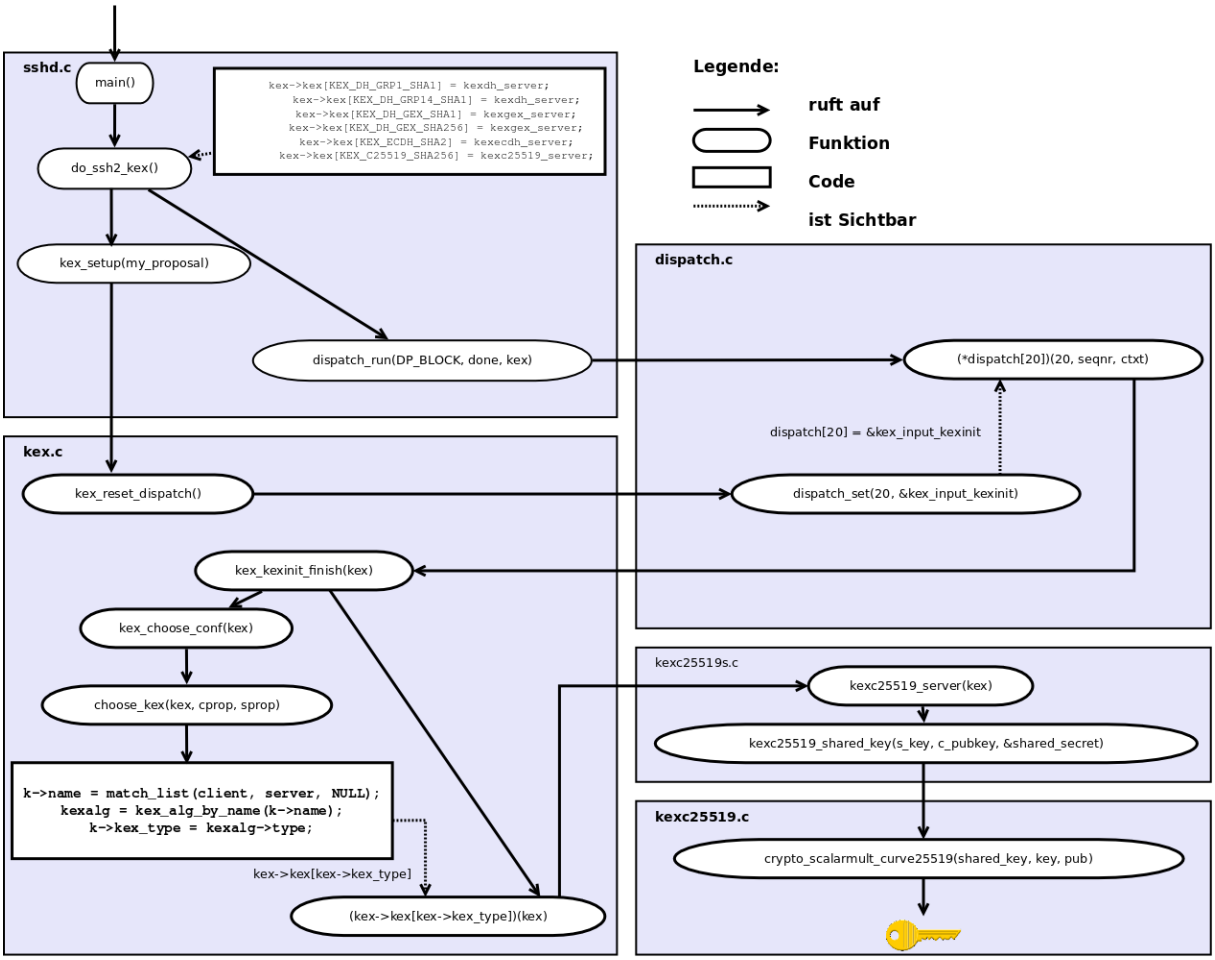

Abbildung 5.1: Funktionsaufrufe vom SSH-Daemon bis zum gemeinsamen Schlüssel

Zunächst wird in MAIN() die Funktion DO_SSH2_KEX() aufgerufen. Sie startet den Schlüsselaustausch und hat Zugriff auf die *kex*-Struktur. In dem *kex-strukt* werden Informationen zu kryptografischen Verfahren, Versionsnummern der SSH-Software, Sitzungs IDs und andere für den Schlüsselaustausch relevante Daten gespeichert. In der Funktion DO_SSH2_KEX() wird zuerst KEX_SETUP(MY_PROPOSAL) ausgeführt. Sie initialisiert das *kex-strukt* mit den serverseitigen Einstellungen und setzt die zwanzigste Stelle des *dispatch*-Arrays auf die KEX_INPUT_KEXINIT() Funktion. Die zweite Funktion von DO_SSH2_KEX() ruft danach mittels DISPATCH_RUN() die Funktion, die an entsprechender Stelle (20) im Dispatch-Array definiert wurde auf.
Anschließend werden mit dem Client die Verfahren ausgehandelt. Im Bezug auf das Kapitel 3 wird durch die Funktion KEX_INPUT_KEXINIT() die SSH-MSG-KEXINIT Nachricht und damit die angebotenen Verfahren des Clients empfangen. Die Funktion CHOOSE_KEX(KEX,CPROP,SPROP) vergleicht die Liste des Clients mit der eigenen Liste. Abhängig vom Namen des kryptografischen Verfahrens, *kexalg*, wird der Typ bestimmt. Der Typ ist eine Zahl von Eins bis Sieben.
Je nach dem welches Verfahren ausgehandelt wurde, wird am Ende der KEX_KEXINIT_FINISH()-Funktion die zugehörige Funktion aus dem Array ausgewählt. Es wird von dem Strukt *kex* auf ein Arrays mit dem Namen *kex* zugegriffen. An der Stelle *kex*->kex_type steht ein Funktionspointer. Dieser wird mit dem Parameter *kex* aufgerufen. Initialisiert wurde dieses Array zu Beginn in der DO_SSH2_KEX()-Funktion.
In dem Beispiel der Abbildung 5.1 wurde das Verfahren Curve25519 ausgehandelt. Mit diesem kann ein gemeinsamer geheimer Schlüssel ausgetauscht werden. In der Datei *kexc25519s.c* wird eine Funktion aus *kexc25519.c* aufgerufen, welche mithilfe der SSL-Bibliotheken das Verfahren anwendet und den gesuchten Schlüssel generiert.
Nachdem KEXC25519_SHARED_KEY(S_KEY, C_PUBKEY, &SHARED_SECRED) aufgerufen wurde, befindet sich in dem Buffer *shared_secred* der Schlüssel. Dieser kann jetzt an den Honeypot geschickt werden.

Schlüsselversand

Zum Versenden wird ein Socket erstellt. Auf einem vorher definierten Port werden der Schlüssel und der Hash an den Honeypot versandt. Dieser wartet bereits auf eingehende Verbindungen und gibt die empfangenen Daten als Hexstring aus oder speichert sie ab.
Da sich diese Arbeit hauptsächlich mit der Modifikation der OpenSSH-Software befasst, werden die Daten in dem Kapitel 5.4 an einen simulierten Honeypot geschickt. Dabei erfüllt der simulierte Honeypot einzig die Aufgabe, auf Pakete des OpenSSHs zu warten, um sie auszugeben. Der Quellcode befindet sich im Anhang A.7.

5.2 Entschlüsselung mit Hilfe von OpenSSH

Die Entschlüsselung von SSH-Kommandos erfordert Kenntnisse über die Zusammensetzung der Pakete. Die Abbildung 5.2 zeigt den Aufbau von SSH-Paketen. In [TY06] wird die Paketlänge als *uint32* definiert. Soll ein Paket entschlüsselt werden, müssen zunächst die ersten 32 Bit entschlüsselt werden. Dadurch erhält man die Packetlänge und weiß, wie viel Byte entschlüsselt werden müssen. Außerdem muss das Padding entfernt werden und die richtige Menge an Bytes dekomprimiert werden. OpenSSH führt diese Schritte bei jedem empfangenen Paket durch. Aus diesem Grund ist es sinnvoll einen modifizierten OpenSSH-Server zum Entschlüsseln zu verwenden.
Im Folgenden dieses Kapitels wird auf die Architektur von OpenSSH eingegangen. Dadurch erhält der Leser einen Überblick darüber, welche Vorgänge ablaufen, wenn ein Paket eintrifft. Danach wird erläutert, in welchen Ebenen die Architektur modifiziert werden kann, um die Paketdaten und den Schlüssel dem OpenSSH bekannt zu machen. Am Schluss wird die Implementierung der Modifikation vorgestellt.

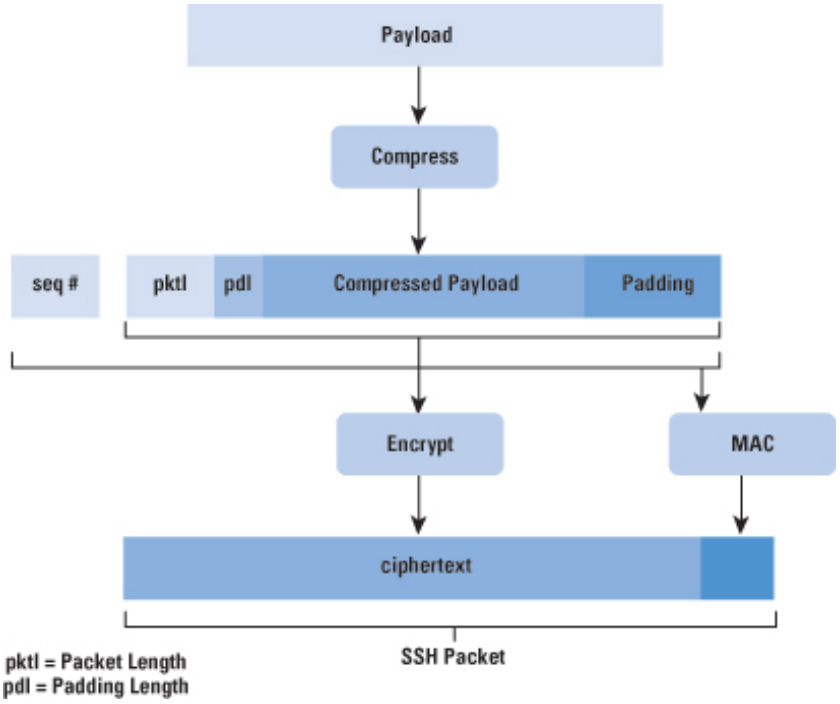

Abbildung 5.2: Paketaufbau in SSH [Sta09]

Architektur des OpenSSH-Servers

Für diese Arbeit sind zwei Teilausschnitte der gesamten Architektur wichtig. Zum einen der Abschnitt, der die Pakete behandelt. In diesem Abschnitt kann der Payload eines SSH-Pakets eingefügt werden. Zum anderen der Abschnitt, der für die Erstellung und Verwaltung der Schlüssel zuständig ist. Dieser Part wurde schon zum Teil im vorherigen Abschnitt 5.1 betrachtet.
Die Abbildung 5.3 veranschaulicht diese beiden Ausschnitte der Architektur. Die linke Seite Beschäftigt sich mit der Einbringung der Keys und die rechte Seite mit der Einbringung des Payloads. Beide Seiten greifen auf die Funktionen der *cipher.c* Datei zu, welche ihrerseits auf die Funktionen der Envelope (EVP) Cipher zugreift. Die EVP Cipher bietet Schnittstellen für symmetrische Verschlüsselungsfunktionen an.

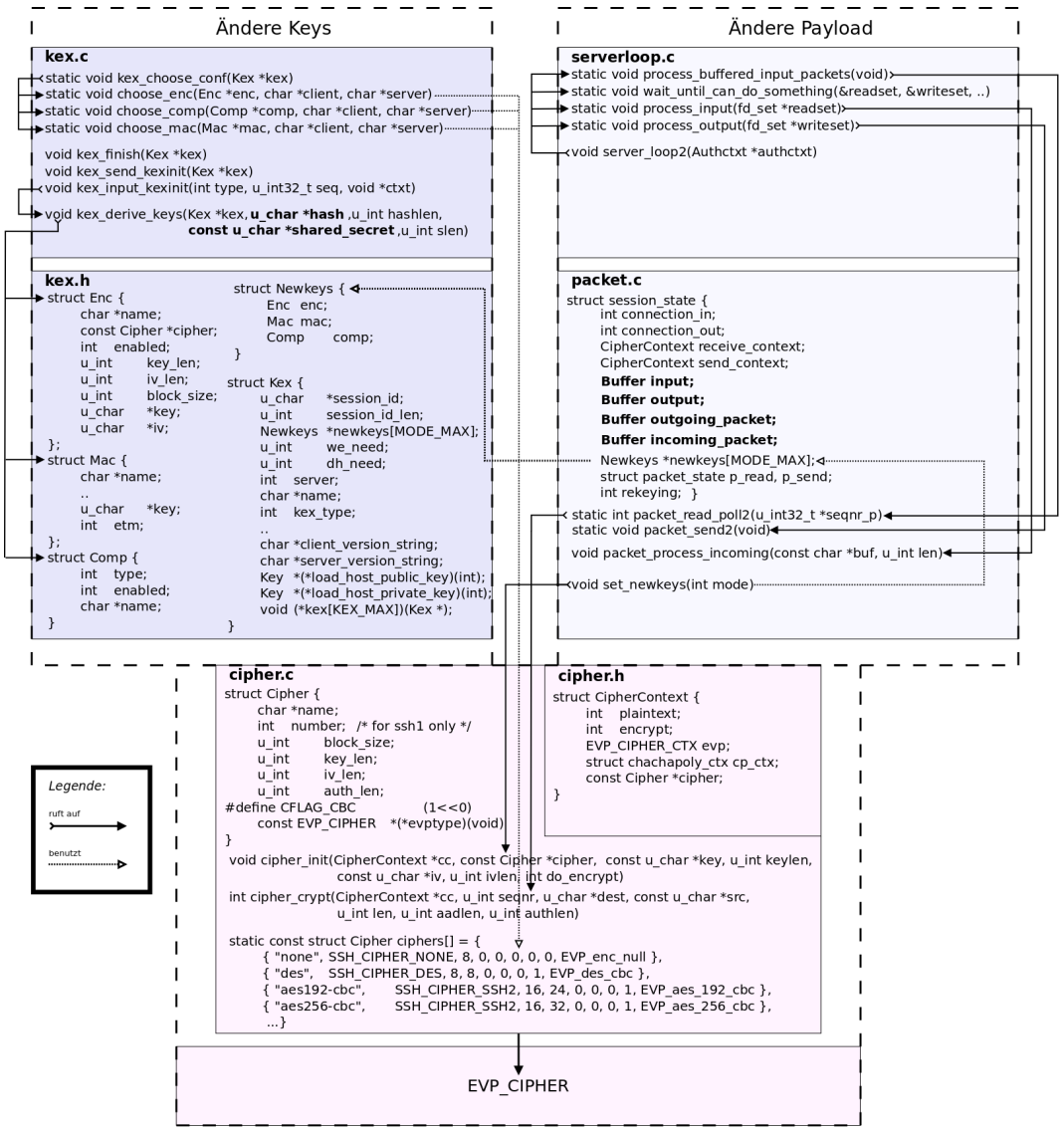

Abbildung 5.3: Architekturausschnitt des OpenSSH, der für die Entschlüsselung zuständig ist

In Abschnitt 5.1 wurde der Weg vom Start des Daemons bis zur Generierung des Schlüssels gezeigt. Nun wird betrachtet, wie der Schlüssel gespeichert und verändert wird, damit die Entschlüsselung funktioniert.
Als letzten Schritt wurde die KEXC25519_SHARED_KEY(S_KEY, C_PUBKEY, &SHARED_SECRED)-Funktion aufgerufen. Der Schlüssel liegt im Buffer *shared_secret*. Danach wird KEX_C25519_HASH() aufgerufen. Dies führt dazu, dass in der Variablen *hash* ein Hashwert steht, welcher als Sitzungs ID benutzt wird.
Der Hash und der Schlüssel müssen, wie in Kapitel 3.2 erläutert, mit dem *session identifiert*, also der Sitzungs ID und einem Buchstaben durch eine Hashfunktion zu einer neuen Bitfolge transformiert werden. Dies wird mit den Buchstaben von A bis F vollzogen. Dadurch ergeben sich sechs Bitfolgen, welche geheim sind und sich gegenseitig unterscheiden. Sie werden als Schlüssel, als initialer *Initialization vector* (IV) und als Schlüssel zur MAC-Berechnung für die Verbindungen Client-zu-Server und Server-zu-Client verwendet.
Diese sechs Hashes werden durch die Funktion KEX_DERIVE_KEYS(..) berechnet. Nach der Berechnung stehen die Hashes in der Struktur NEWKEYS *CURRENT_KEYS[MODE_MAX]. Wird im Laufe des Schlüsselaustausches die SSH-MSG-NEWKEYS Nachricht empfangen, wird die SET_NEWKEYS(..)-Funktion aufgerufen. Durch den Aufruf werden die Werte der *current_keys* Struktur in den Strukturen *Enc*, *Mac* und *Comp* übernommen und durch die Funktion CIPHER_INIT() wird der Cipher der *Enc*-Struktur initialisiert. *Enc*, *Mac* und *Comp* stehen wiederum in der Newkeys Struktur der Datei *kex.h*. Somit sind sie in den Modulen sichtbar, die *kex.h* einbinden.

Die rechte Seite der Abbildung 5.3 veranschaulicht die Paketbehandlung.
Nach dem Schlüsselaustausch beginnt die SSH-Sitzung. Während einer Sitzung wird jedes eingegebene Symbol des Clients einzeln an den Server übertragen. Wird in dem simulierten Terminal Output erzeugt, wird es an den Client gesandt.
Durch eine *select()*-Funktion[1] wird auf Pakete des Clients gewartet. Eintreffende Pakete werden von dem entsprechenden File Descriptor gelesen und in den *incoming_packet* Buffer gespeichert. Dieser Vorgang wird durch die Funktionen der Datei *serverloop.c* organisiert.
In der Funktion WAIT_UNTIL_CAN_DO_SOMETHING(..) wird auf ein Paket gewartet und in der Funktion PROCESS_INPUT(..) wird es in den Buffer kopiert. Entschlüsselt und dekomprimiert wird der Buffer durch PROCESS_BUFFERED_INPUT_PACKETS(..). Anschließend wird das Ergebnis in dem *input* Buffer gespeichert.
Die PROCESS_BUFFERED_INPUT_PACKETS(..)-Funktion leitet den Aufruf an die *packet.c* Ebene weiter. Dort wird PACKET_READ_POLL2(..) aufgerufen. Diese Funktion ist dafür zuständig die Paketlänge heraus zu finden, den MAC zu überprüfen und das Padding ab zu trennen. Weiter Entschlüsselt und dekomprimiert sie. Das wird durch Aufrufe der *cipher.c* Ebene erreicht.
In *cipher.c* wird die CIPHER_CRYPT(..)-Funktion aufgerufen. Sie bekommt den *CipherContext*, in dem die Newkeys Struktur steht, eine Quelle, ein Ziel und eine Sequenznummer. Mit diesen Daten werden Funktionen des EVP_Cipher aufgerufen.

[1] Die *select()*-Funktion beobachtet ein oder mehrere File Descriptoren und kehrt zurück, wenn mindestens einer von ihnen bereit für Eingabe/Ausgabe ist.

Einstiegspunkte für die Modifikation

Dieser Abschnitt soll die Frage beantworten, welche Ebenen der Architektur modifiziert werden sollen. Für die Schlüsseländerung gibt es die Möglichkeiten vor oder nach der KEX_DERIVE_KEYS(..)-Funktion zu modifizieren. Der Schlüssel muss geändert werden, weil der modifizierte OpenSSH mit dem Schlüssel des ursprünglichen Angreifers entschlüsseln soll.
Der Payload kann in der Serverloopebene, der Paketebene oder der Cipherebene eingebracht werden. Er muss geändert werden damit die Pakte des Angreifers, statt neu eintreffende Pakete entschlüsselt werden.
Wird der Schlüssel unmittelbar nach der Erzeugung durch die SSL-Bibliothek zum Honeypot geschickt, muss danach noch die KEX_DERIVE_KEYS(..)-Funktion aufgerufen werden. Der Vorteil an dieser Variante ist, dass der Schlüssel eher losgeschickt wird. Damit der Schlüssel korrekt verwendet wird, muss die Konfiguration des SSH die selbe sein, wie auf der virtuellen Maschine. Dazu wird die KEX_CHOOSE_CONF(..)-Funktion aufgerufen. Diese Funktion stellt die entsprechenden kryptografischen Verfahren ein.
Der Einstiegspunkt für die Payloadeinbringung ist entscheidender. Je nach dem in welcher Ebene die zu entschlüsselnden Daten eingereicht wird, ergeben sich Vor- und Nachteile.

- *In der Serverloopebene*
  Nach der Einstellung der Konfiguration und der Abänderung des Schlüssels wird der SSH-Server normal betrieben. Ein anderes Programm schickt die Pakete an den Server. Der Server entschlüsselt die Pakete und gibt das Ergebnis aus.
  Vorteil: Keine Betrachtung der anderen Ebenen Notwendig.
  Nachteile: Es muss ein Programm geschrieben werden, welches SSH-Pakete verschicken kann. Zur Entschlüsselung müssen beide Programme gestartet werden.

- *In der Paketebene*
  Nach der Einstellung der Konfiguration und der Abänderung des Schlüssels wird der Payload in den *input* Buffer geschrieben. Danach wird die Funktion PACKET_READ_POLL2(..) aufgerufen.
  Vorteil: Keine Änderung des Payloads notwendig.
  Nachteil: Für jedes Paket muss die Funktion aufgerufen werden.

- *In der Cipherebene*
  Die Funktion CIPHER_CRYPT(..) erhält als Parameter alle notwendigen Informationen und wird unabhängig vom SSH-Server verwendet.
  Vorteil: Kein OpenSSH-Server Start notwendig.
  Nachteil: Der Payload muss geändert werden.

Umgesetzt wurde die Variante der Paketebene. In zukünftigen Arbeiten ist es sinnvoll, nur die CIPHER_CRYPT(..)-Funktion zu verwenden. Der Aufwand den Payload an zu passen, ist jedoch nicht gering, weshalb sich in dieser Arbeit für eine höhere Ebene entschieden wurde.
Das Paket in der Serverloopebene zu entschlüsseln ist weniger aufwendig zu implementieren, dafür aber ungünstiger in der Praxis bei dem Entschlüsseln von Kommandos.

5.3 Modifikation

Modifikationen für den OpenSSH zur Entschlüsselung wurden in den Dateien *sshd.c*, *kex.c*, *packet.c* und *cipher.c*, sowie den zugehörigen Headerdateien durchgeführt. Modifikationen für den OpenSSH zum Schlüsselversand wurden in der Datei *kexc25519s.c* realisiert.
Die gezeigten Listings beinhalten keinen vollständigen Code, da sie zur besseren Übersicht gekürzt wurden. Der vollständige Code befindet sich im Anhang A.
Die Umsetzung zeigt nur, dass die prinzipielle Durchführung des Entschlüsselns von Angreiferkommandos durch die modifizierten OpenSSH Varianten möglich ist. Damit die Programme in der Praxis angewandt werden können, muss der Schlüsselversand an den Honeypot organisiert werden und das Entschlüsseln mehrere Pakete vereinfacht werden. In Kapitel 5.4 wird ein proof of concept demonstriert.
Das Listing 5.1 zeigt den Code der Datei *kexc25519s.c* vor der DERIVE_KEY-Funktion. Zu sehen ist, dass das *shared_secred* von dem Typen Buffer in den Typen u_char umgewandelt wird. Danach wird ein Socket erstellt und der Hash, sowie der Schlüssel an die definierte Adresse gesendet.

```
1         u_int key_len = buffer_len(&shared_secret);
2         u_char *key = xmalloc(key_len);
3         u_char *h = hash;
4         u_int h_len = hashlen;
5
6         /*Ruft die Funktion memcpy(key, shared_secret->buf +
              shared_secret->offset, key_len) in Buffer.c auf*/
7         lion_buffer_get(&shared_secret, key, key_len);
8
9         int sockfd, port;
10        struct sockaddr_in serv_addr;
11        struct hostent *server;
12        port = 101475;
13        sockfd = socket(AF_INET, SOCK_STREAM, 0);
14        if (sockfd < 0) exit(0);
15        server = gethostbyname("127.0.0.1");
16        serv_addr.sin_family = AF_INET;
17        serv_addr.sin_port = htons(port);
18
19        if (connect(sockfd,(struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0) exit(0);
20        if (send(sockfd, h, h_len, 0) != h_len) exit(0);
21        if (send(sockfd, key, key_len, 0) != key_len) exit(0);
```

Listing 5.1: Codeausschnitt des Schlüsselversandes

Die Entschlüsselung wird in der *sshd.c*-Datei gestartet. Dort befindet sich die SSH_DECRYPTER()-Funktion. Sie ist in Listing 5.2 zu sehen. In den ersten 15 Zeilen werden Variablen definiert. Alle notwendigen Informationen werden hier dem Programm übergeben. Der Hexadezimalstring der Variablen *rawpacket* wurde von einer zuvor geschehenen SSH-Verbindung mithilfe des Programmes Wireshark mitgeschnitten. Die paket_seqnr 11 ist das erste Paket, das nach dem Schlüsselaustausch und dem Login des Clients übertragen wurde, da zuvor Pakte zur Initialisierung der Verbindung verschickt wurden. Also das erste Symbol des ersten Kommandos. Die Daten des keys, hashes und der Konfigurationen wurden von der virtuellen Maschine erhalten. In den restlichen Zeilen werden die Funktionen zum Ändern der Konfiguration und des Payloads, sowie zur Entschlüsselung aufgerufen.

```
1  void ssh_decrypter()
2  {
3          u_char *key = "\x00\x00\x00\x21\x00... \xdb\x6a";
4          u_char *hash = "\xb4\x22\xb9\x0e\x18... \x3a\x55";
5          u_int hashlen = 32;
6          u_int key_len = 37;
7
8          char* enc_name = NULL;
9          char* mac_name = NULL;
10         char* comp_name = NULL;
11
12         char *rawpacket = "\x00\x00\x00\x10\xdd... \xe0\xdc";
13         int paketlen = 36;
14         int paket_seqnr = 11;
15
16         /*Change important values from the Configuration*/
17         lion_change_conf(key, hash, key_len, hashlen, enc_name,
               mac_name, comp_name, kex);
18
19         /*Change active_state->input to the raw paketdata*/
20         lion_change_input(rawpacket, paketlen, paket_seqnr);
21
22         /*Decryption*/
23         lion_decrypt();
24
25         /*Plaintext*/
26         lion_print_plain();
27
28         exit(1);
29 }
```

Listing 5.2: Codeausschnitt des SSH-Daemons

Die Funktion LION_CHANGE_CONF ist in dem Listing 5.3 zu sehen. Zunächst werden die kryptografischen Verfahren festgelegt. Wird einem Verfahren der Wert NULL zugewiesen, werden Standardwerte benutzt. In diesem Fall werden *aes128-ctr*, *hmac-md5-etm@openssh.com* und *none* als Standard gesetzt. Danach werden diese Werte durch die LION_KEX_CHOOSE_CONF-Funktion anderen Architekturebenen sichtbar gemacht. Dabei werden unter anderem Funktionszeiger auf Funktionen der SSL-Bibliothek definiert. Der Quellcode dieser Funktion befindet sich im Anhang A.4.
Anschließen wird der hash definiert und das shared_secred, was den Schlüssel darstellt, in einer Bufferstruktur gespeichert. Die Sitzungs ID ist zu diesem Zeitpunkt das selbe, wie der Hash und wird so definiert. Durch den Aufruf der KEX_DERIVE_KEYS-Funktion werden die sechs Hashes erzeugt, welche eine Zeile weiter durch SET_NEWKEYS den Funktionen der *cipher.c* und der *packet.c* Dateien sichtbar gemacht werden.

```
1  void lion_change_conf(u_char* lion_key, u_char* lion_hash,
       u_int lion_key_len, u_int lion_hashlen, char* lion_enc_name,
        char* lion_mac_name, char* lion_comp_name, Kex* lion_kex)
2  {
3          /*Change Configuration*/
4          char* enc_name = lion_enc_name;
5          char* mac_name = lion_mac_name;
6          char* comp_name = lion_comp_name;
7
8          lion_kex_choose_conf(lion_kex, enc_name, mac_name,
               comp_name);
9
10         /*Change Key, Hash and session ID*/
11         u_char *hash = lion_hash;
12         u_int hashlen = lion_hashlen;
13
14         Buffer shared_secret;
15         u_char *buf_shared_secret = lion_key;
16         u_int end = lion_key_len;
17         lion_buffer_init(buf_shared_secret, end);
18
19         lion_kex->session_id_len=hashlen;
20         lion_kex->session_id=xmalloc(lion_kex->session_id_len);
21         memcpy(lion_kex->session_id, hash, lion_kex->
               session_id_len);
22
23         kex_derive_keys(lion_kex, hash, hashlen, buffer_ptr(&
               shared_secret), buffer_len(&shared_secret));
24         set_newkeys(0);
25 }
```

Listing 5.3: Code zum Verändern der Konfiguration und des Schlüssels

Nachdem die Konfiguration geändert wurde, muss der Payload geändert werden. Dies passiert in der Funktion LION_CHANGE_INPUT, welche in dem Listing 5.4 zu sehen ist. Die Funktion befindet sich in der Datei *packet.c*. Damit hat sie Zugriff auf die *active_state* Struktur. Durch die Funktion BUFFER_APPEND wird der *input*-Buffer überschrieben. Außerdem muss die Sequenznummer der *active_state* Struktur neu gesetzt werden.

```
1 void lion_change_input(char *lion_rawpacket, int lion_packetlen
     , int lion_paket_seqnr)
2 {
3         char *rawpacket = lion_rawpacket;
4         int len = lion_packetlen;
5
6         buffer_append(&active_state->input, rawpacket, len);
7         active_state->p_read.seqnr = lion_paket_seqnr;
8 }
```

Listing 5.4: Code zum verändern des Payloads

5.4 Proof of Concept

In diesem letzten Kapitel wird gezeigt, dass die Modifizierungen von dem vorherigen Kapitel funktionieren. Dazu wird zuerst eine Verbindung zwischen dem Angreifer und dem Honeypot simuliert. Anschließend wird diese Verbindung entschlüsselt. Der Ablauf ist in Abbildung 5.4 visualisiert.
Als ersten Schritt wird das Programm *modified_SSH* installiert. Das Programm *modified_SSH* ist die Implementierung des in Kapitel 5.1 erläuterten Konzepts. Dies wird durch das Kommando *sudo make install* erreicht. Danach wird das Programm *honeypot_sim*[2] durch *make* installiert und anschließend ausgeführt. Nachdem das Programm, welches den geheimen Schlüssel empfangen soll, gestartet wurde kann die SSH-Sitzung beginnen. Dazu wird zuerst der modified_SSH mit der -d Option und darauf das Programm ssh ausgeführt.
Noch bevor der Client nach seinem Passwort gefragt wird, findet der Schlüsselaustausch statt. Deshalb empfängt der honeypot_sim den Schlüssel und gibt ihn aus. Jetzt wird noch das Paket und der *initialization vector* benötigt. Der initialization vector ist zwar im Schlüssel vorhanden, wird allerdings mit jedem Paket verändert. Diese Veränderungen sind noch nicht implementiert und werden deshalb für diesen Proof of Concept mit ausgegeben. Mit der Hilfe von Wireshark wird nun auf dem Loopbackinterface auf SSH-Pakete gewartet.
Nach dem der Angreifer sein Passwort eingetippt hat, beginnt für ihn eine normale SSH-Sitzung. Damit man später das erste Paket findet, wird nach dem Login der SSH-Sitzung in dem Programm Wireshark das letzte Paket durch *Mark Packet* markiert. Als erstes wird der Angreifer für dieses Testbeispiel auf dem Rechner nach Geld suchen, um reich zu werden. Dazu ruft er das Kommando *find money* auf.
Das erste Kommando ist damit versandt. In dem Terminal, in dem der *sshd* gestartet wurde, befindet sich jetzt der IV. Er wird kopiert und gespeichert. In dem Programm Wireshark sehen wir

[2] Server, welcher den Schlüssel empfängt und ausgibt

das zugehörige Paket. Der Hexstream des SSH-Payloads des Paketes, welches direkt nach dem markierten Paket kommt, wird kopiert[3] und gespeichert.
An dieser Stelle sind alle Informationen vorhanden und die Verbindung kann geschlossen werden.

Zur Entschlüsselung werden der OpenSSH-Variante *ssh_decrypter* die Daten übergeben. Dazu wird das Shared Secred und der Hash mit den jeweiligen Längen in die Datei *sshd.c* Zeile 2507 und folgende geschrieben.
Das Paket wird mithilfe des Programmes *hexumwandler*[4] in das richtige Format gebracht. Dazu wird das Programm durch *make* installiert und im Anschluss mit dem Paket und seiner Länge als Parameter ausgeführt. Das Ergebnis wird in den *sshd* Zeile 2516 geschrieben. Als letztes wird in der Datei *cipher.c* in der Zeile 365 der IV erneuert.
Zum Entschlüssel wird der *ssh_decrypter* installiert und gestartet. Dann wird der ssh gestartet. Nach der *Start lion_print_plain*-Ausgabe steht das Entschlüsselte Ergebnis[5]: *66*. Die *66* ist die ASCII-Codierung für das *f* des *find*-Kommandos.

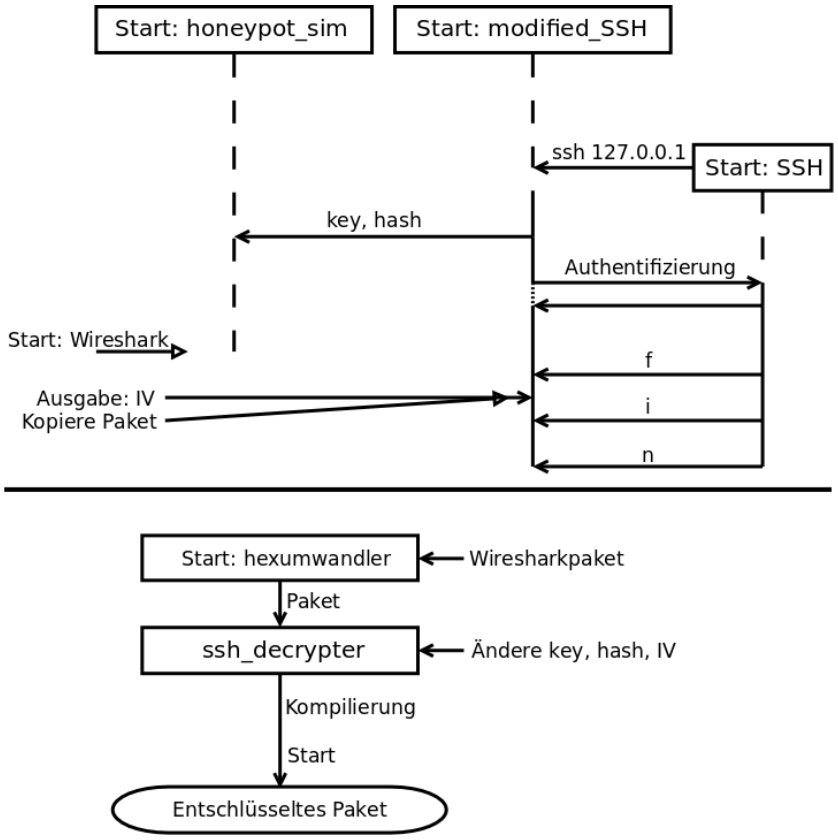

Abbildung 5.4: Ablauf des Proof of Concept

[3] Copy->Bytes->Hex Steam
[4] Programm zum umwandeln von Daten der Form "10b1.." in die Form "\x10\xb1"
[5] Die ersten Acht Byte gehören gehören nicht zum Kommando

6 Zusammenfassung

Die Aufgabenstellung dieser Bachelorarbeit ist gleichzeitig eine Zusammenfassung: “Finden Sie einen Weg, SSH-Verbindungen in dem Honeydv6 zu entschlüsseln und setzen Sie diesen um.“
Die Möglichkeiten SSH-Verbindungen in High-Interaction Honeypots zu entschlüsseln werden in dem Kapitel 4 gezeigt. Dabei stellt sich heraus, dass es unauffällig und zugleich einfach zu implementieren ist, den kryptografischen Schlüssel bei dem Verbindungsaufbau heraus zu senden und die Pakete im Nachhinein zu entschlüsseln.
Damit dieser Ansatz verfolgt werden kann, ist Grundlagenwissen über den Schlüsselaustausch, das SSH-Protokoll und das Diffie-Hellman-Verfahren notwendig. Außerdem ist es wichtig, ein tiefes Verständnis der OpenSSH-Software zu haben. Zu diesem Zweck wird in dem Kapitel 5 auf die Architektur der Software eingegangen.
Das erlangte Wissen der vorherigen Kapitel wird zur Modifizierung benötigt. Der OpenSSH wird für zwei unterschiedliche Aufgaben abgeändert. Zum einen als Server auf dem Honeypot, welcher den Schlüssel heraus sendet, bevor der Angreifer Zugriff hat. Zum anderen als Programm zum entschlüsseln der mitgeschnittenen Pakete.
Der “Proof of Concept“ zeigt, dass die Entschlüsselung prinzipiell funktioniert.

A Anhang

```
1         u_int key_len = buffer_len(&shared_secret);
2         u_char *key = xmalloc(key_len);
3         u_char *h = hash;
4         u_int h_len = hashlen;
5
6
7         int sockfd, port;
8         struct sockaddr_in serv_addr;
9         struct hostent *server;
10
11        port = 101475;
12
13        sockfd = socket(AF_INET, SOCK_STREAM, 0);
14        if (sockfd < 0) exit(0);
15
16        server = gethostbyname("127.0.0.1");
17        if (server == NULL) exit(0);
18
19        bzero((char *) &serv_addr, sizeof(serv_addr));
20        serv_addr.sin_family = AF_INET;
21        bcopy((char *)server->h_addr, (char *)&serv_addr.
              sin_addr.s_addr, server->h_length);
22        serv_addr.sin_port = htons(port);
23
24        if (connect(sockfd,(struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0) exit(0);
25        if (send(sockfd, h, h_len, 0) != h_len) exit(0);
26        sleep(1);
27
28        lion_buffer_get(&shared_secret, key, key_len);
29
30        if (send(sockfd, key, key_len, 0) != key_len) exit(0);
```

Listing A.1: Codeausschnitt des Schlüsselversandes

```
1  void ssh_decrypter()
2          {
3          fprintf(stderr," |-----------------------------|\n");
4          fprintf(stderr," |-------- SSH_DECRYPTER ------|\n");
5
6          u_char *key = "\x00\x00\x00\x21\x00\xc9\xca\xef\x05\x27
               \xf2\xb9\x0c\x15\x0f\xdc\xd9\xd9\xa7\x24\x4f\x70\x31
               \xf7\x10\xd8\xe6\x95\xa3\x00\xbc\x9e\x06\x17\x19\xd2
               \x16";
7          u_char *hash = "\x11\xef\x89\x42\x7a\x33\xc1\x0f\xa0\
               x6e\xce\x4c\x21\x7f\x3a\x8a\xf9\x75\xdb\x31\xbe\x29\
               xbf\xaf\xce\xd8\xfa\xd5\x5d\x7d\x94\x16";
8          u_int hashlen = 32;
9          u_int key_len = 37;
10
11         char* enc_name = NULL;
12         char* mac_name = NULL;
13         char* comp_name = NULL;
14
15         char *rawpacket = "\x00\x00\x00\x10\xb1\x97\xd5\xb1\xfe
               \x05\x80\x12\x8f\xf0\x03\x85\x84\xa7\x62\xb4\xb5\x98
               \xf6\x8a\xab\x23\x52\x5e\xd2\x41\x01\x62\xba\x82\x94
               \x59";
16         int paketlen = 36;
17         int paket_seqnr = 11;
18
19         /*Change every important values from CipherContext*/
20         lion_change_conf(key, hash, key_len, hashlen, enc_name,
               mac_name, comp_name, kex);
21
22         /*Change active_state->input zum raw paketdata*/
23         lion_change_input(rawpacket, paketlen, paket_seqnr);
24
25         /*decrypt*/
26         lion_decrypt();
27
28         /*Plaintext anzeigen */
29         lion_print_plain();
30         fprintf(stderr," |----------------- DECRYPTED
               ----------------------------|\n");
31         fprintf(stderr,"exit(1)\n\n");
32         exit(1);
33         }
34         ssh_decrypter();
```

Listing A.2: Codeausschnitt des SSH-Daemons

```
1  void lion_change_conf(u_char* lion_key, u_char* lion_hash,
       u_int lion_key_len, u_int lion_hashlen, char* lion_enc_name,
        char* lion_mac_name, char* lion_comp_name, Kex* lion_kex){
2          fprintf(stderr,"-- Start lion_change_conf.\n");
3          /*Simuliere die Funktion kex_kexinit_finisch() */
4          char* enc_name = lion_enc_name;
5          char* mac_name = lion_mac_name;
6          char* comp_name = lion_comp_name;
7          lion_kex_choose_conf(lion_kex, enc_name, mac_name,
               comp_name);
8
9          u_char *hash = lion_hash;
10          u_int hashlen = lion_hashlen;
11
12         Buffer shared_secret;
13         buffer_init(&shared_secret);
14          u_char *buf_shared_secret = lion_key;
15         u_int alloc = 4096;
16         u_int offset = 0;
17          u_int end = lion_key_len;
18         lion_buffer_init(&shared_secret, buf_shared_secret,
               alloc, offset, end);
19
20         lion_kex->session_id_len = hashlen;
21         lion_kex->session_id = xmalloc(lion_kex->session_id_len
               );
22         memcpy(lion_kex->session_id, hash, lion_kex->
               session_id_len);
23
24         fprintf(stderr,"-- Start kex_derive_keys.\n");
25         kex_derive_keys(lion_kex, hash, hashlen, buffer_ptr(&
               shared_secret), buffer_len(&shared_secret));
26
27         fprintf(stderr,"-- Start set_newkeys(0).\n");
28         set_newkeys(0);
29 }
```

Listing A.3: Code zum verändern der Konfiguration und des Schlüssels

```
1  static void lion_kex_choose_conf(Kex *kex, char* enc_name, char
       * mac_name, char* comp_name){
2
3          Newkeys *newkeys;
4          char **my, **peer;
5          char **cprop, **sprop;
6          int nenc, nmac, ncomp;
7          u_int mode, ctos, need, dh_need, authlen;
8          int first_kex_follows, type;
9
10                 /* Algorithm Negotiation */
11         for (mode = 0; mode < MODE_MAX; mode++) {
12                 newkeys = xcalloc(1, sizeof(*newkeys));
13                 kex->newkeys[mode] = newkeys;
14                 ctos = (!kex->server && mode == MODE_OUT) ||
15                     (kex->server && mode == MODE_IN);
16                 nenc  = ctos ? PROPOSAL_ENC_ALGS_CTOS  :
                       PROPOSAL_ENC_ALGS_STOC;
17                 nmac  = ctos ? PROPOSAL_MAC_ALGS_CTOS  :
                       PROPOSAL_MAC_ALGS_STOC;
18                 ncomp = ctos ? PROPOSAL_COMP_ALGS_CTOS :
                       PROPOSAL_COMP_ALGS_STOC;
19         lion_choose_enc(&newkeys->enc, enc_name); /*LION*/
20                 /* ignore mac for authenticated encryption */
21                 authlen = cipher_authlen(newkeys->enc.cipher);
22                 if (authlen == 0)
23                         lion_choose_mac(&newkeys->mac, mac_name
                           );/*LION*/
24                 lion_choose_comp(&newkeys->comp, comp_name);/*
                       LION*/
25                 debug("kex: %s %s %s %s",
26                     ctos ? "client->server" : "server->client",
27                     newkeys->enc.name,
28                     authlen == 0 ? newkeys->mac.name : "<
                           implicit>",
29                     newkeys->comp.name);
30         }
31
32         ...
33 }
```

Listing A.4: Code der Funktion lion_kex_choose_conf

```
1  void
2  lion_choose_enc(Enc *enc, char* name){
3          if (name == NULL)
4                  name = "aes128-ctr";
5          if ((enc->cipher = cipher_by_name(name)) == NULL)
6                  fatal("matching cipher is not supported: %s",
                       name);
7          enc->name = name;
8          enc->enabled = 0;
9          enc->iv = NULL;
10         enc->iv_len = cipher_ivlen(enc->cipher);
11         enc->key = NULL;
12         enc->key_len = cipher_keylen(enc->cipher);
13         enc->block_size = cipher_blocksize(enc->cipher);
14
15 }
```

Listing A.5: Code der Funktion lion_choose_enc

```
1  int main(int argc, char** argv) {
2          int i;
3          printf("Usage: Arg1 is Hexstream, Arg2 is length of the
               Hexstream\n");
4          char* alt_hex = argv[1];
5          int len = atoi(argv[2]);
6          printf("\n");
7          printf("Result: \n");
8          for (i=0;i<len*2;i=i+1){
9                  if(!(i%2)) printf("\\x");
10                         printf("%c",alt_hex[i]);
11         }
12         printf("\n");
13
14         return 0;
15 }
```

Listing A.6: Code zum umwandeln von Hexstrings

```
1  #define RCVBUFSIZE 1000
2  #define PORT 101475
3  u_char* dump_recive(int client_socket)
4  {
5          u_char echo_buffer[RCVBUFSIZE];
6          int recv_size, i;
7          if((recv_size = recv(client_socket, echo_buffer,
               RCVBUFSIZE,0)) < 0) printf("Fehler bei recv()");
8          printf("Dump recieved Data: \n"); printf("\\x");
9          for (i = 0; i < recv_size; i++) {
10                 printf("%02x", echo_buffer[i]);
11                 if(i!=recv_size-1)
12                         printf("\\x");      }
13         printf("\n");
14         printf("Hashlen: %d\n",recv_size);
15         return echo_buffer;
16 }
17 u_char* dump_recive2(int client_socket)
18 {
19         u_char echo_buffer2[RCVBUFSIZE];
20         int recv_size = 0, j;
21         if((recv_size = recv(client_socket, echo_buffer2,
               RCVBUFSIZE,0)) < 0) printf("Fehler bei recv()");
22         printf("\\x");
23         for (j = 0; j < recv_size; j++) {
24                 printf("%02x", echo_buffer2[j]);
25                 if(j!=recv_size-1)
26                         printf("\\x");          }
27         printf("\n");
28         printf("Secredlen: %d\n",recv_size);
29         return echo_buffer2;
30 }
31
32 int main( ) {
33         struct sockaddr_in server, client;
34         int sock, fd;
35         unsigned int len;
36         sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
37         if (sock < 0) error_exit("Fehler beim Anlegen eines
               Sockets");
38         memset( &server, 0, sizeof (server));
39         server.sin_family = AF_INET;
40         server.sin_addr.s_addr = htonl(INADDR_ANY);
41         server.sin_port = htons(PORT);
```

```
42          if(bind(sock,(struct sockaddr*)&server, sizeof( server)
               ) < 0)
43                  error_exit("Kann das Socket nicht \"binden\"");
44          if(listen(sock, 5) == -1 )
45                  error_exit("Fehler bei listen");
46          printf("Server bereit - wartet auf Anfragen ...\n");
47          for (;;) {
48                  len = sizeof(client);
49                  fd = accept(sock, (struct sockaddr*)&client, &
                       len);
50                  if (fd < 0) error_exit("Fehler bei accept");
51                  /* Daten vom Client auf dem Bildschirm ausgeben
                        */
52                  printf("Hash: \n");
53                  dump_recive( fd );
54                  printf("\n\n");
55                  printf("Secred: \n");
56                  dump_recive2( fd );
57                  close(fd);
58                  exit(0);
59          }
60          return 0;
61  }
```

Listing A.7: Der simulierte Honeypot

Literaturverzeichnis

[BSI06] BSI. Manipulation von arp-tabellen. 2006. URL retrieved: Aug 20, 2014. Available from: https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/_content/g/g05/g05112.html.

[BSI07] BSI. Man-in-the-middle-angriff. 2007. URL retrieved: Aug 20, 2014. Available from: https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/_content/g/g05/g05143.html.

[DXS01] Xuqing Tian Dawn Xiaodong Song, David Wagner. Timing analysis of keystrokes and timing attacks on ssh. August 2001.

[Gut11] Peter Gutmann. Do users verify ssh keys? August 2011.

[HON08] Honeyd, 2008. URL retrieved: Aug 10, 2014. Available from: http://honeyd.org/.

[OPE99] Openssh, January 1999. URL retrieved: Jun 14, 2014. Available from: http://www.openssh.com/.

[OSS99a] Openssh - history, January 1999. URL retrieved: Jun 14, 2014. Available from: http://www.openssh.com/history.html.

[OSS99b] Openssh - windows, January 1999. URL retrieved: Jun 14, 2014. Available from: http://www.openssh.com/windows.html.

[QEM14] Qemu, 2014. URL retrieved: Aug 12, 2014. Available from: http://qemu.org/.

[Sch12] Sven Schindler. Honeydv6: A low-interaction ipv6 honeypot. Technical report, Department of Computer Science, University of Potsdam, German, 2012.

[Sta09] William Stallings. Protocol basics: Secure shell protocol. *The Internet Protocol Journal, Volume 12, No.4*, December 2009. URL retrieved: Jun 13, 2014. Available from: http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_12-4/124_ssh.html.

[TY06] Cisco Systems Inc. T. Ylonen, C. Lonvick Ed. Rfc 4253 - the secure shell (ssh) transport layer protocol. *RFC*, January 2006. URL retrieved: Jun 14, 2014. Available from: http://tools.ietf.org/html/rfc4253.