Automotive Security - Übung

Diese Übung soll einen Überblick über die Möglichkeiten geben lang etablierte Automotive Protokolle wie CAN kryptografisch zu sichern. In diese Übung werden die Schutzziele nacheinander betrachtet und Schritt für Schritt passende Maßnahmen ergriffen um Angriffen auf das System zu eliminieren. Nach Abschluss dieser Übung werden wir eine Client/Server Infrastruktur aufgebaut haben, welche sehr nahe an SecOC, der im AUTOSAR Standard festgehaltenen Security Empfehlung, herankommt.

Anzumerken ist, dass diese Übung für ein besseres Verständnis/ eine bessere Umsetzbarkeit eine vereinfachte Form des Standards implementiert. Details können aus dem SecOC Modul des offenen AUTOSAR Standards entnommen werden.
https://www.autosar.org/fileadmin/standards/R20-11/FO/AUTOSAR_PRS_SecOcProtocol.pdf

Installation

Die Netzwerkteilnehmer in diese Übung werden in C++ implementiert werden. Eine passende Entwicklungsumgebung sowie Programme zum Testen des Codes werden vorausgesetzt.

Es wird benötigt:

- make
- g++ Compiler
- die cryptopp Bibliothek
- can-utils
- ein Editor/IDE der Wahl (z.B. VC-Code, vim, ...)

Für die Installation kann folgender Befehl verwendet werden:

```
1 sudo apt install can-utils build-essential libcrypto++-dev
```

Vorbereitung

Es wird empfohlen die vorangegangene Übung, welche einen Überblick über den CAN-Bus und möglich Angriffe gibt, vor dieser auszuführen um eine einfachen Einstieg in die Materie zu erhalten. Falls bereits Grundkenntnisse in diesen Bereichen vorhanden sind ist es möglich auch direkt mit diesem Abschnitt zu beginnen.

Es werden einige nützliche Code Beispiele in den Dateien bereitgestellt. Falls Unklarheit über die Verwendung auftreten verweise ich auf die Dokumentation der Crypto++ Bibliothek in welcher die verwendeten Klassen implementiert werden.
https://www.cryptopp.com

Um mit der Übung beginnen zu können muss ein virtueller CAN-Bus erzeugt werden, welcher als Interface in Linux ersichtlich ist. Es sind minimale Anpassungen zum virtuellen Bus der letzten Übung nötig, es sollte ein neuer Bus erzeugt werden oder der bestehende entsprechend konfiguriert.

```
# load the kernel module
sudo modprobe vcan

# create an interface called "vcan0"
sudo ip link add dev vcan0 type vcan

# activate the interface
sudo ip link set up vcan0 mtu 72

# verify the interface is there
ip a
```

Für das einfache Compilieren der Quelldateien enthält das Projekt ein Makefile, welches die Angabe der Parameter abnimmt. Werfen Sie zunächst einen Blick in die Datei und machen Sie sich klar welche Build-Optionen verwendet werden können.

Das Vorgehen sieht in etwa so aus:

```
# example to build all files
make all

# example to only build the crypto_intro
make crypto
```

Viel Erfolg!

Übungen

Aufgabe 1:

Ziel der ersten Aufgabe ist es sich mit den bereits implementierten Klassen „canfd-send.cpp“ und „canfd-recv.cpp“ vertraut zu machen. Diese stellen eine Erweiterung der in der vorangegangenen Übung verwendeten Klassen dar um nun größere CAN-FD Nachrichten senden zu können.

Öffnen Sie die Dateien und machen Sie sich mit ihnen vertraut.

Um die Verwendung der Klassen zu erproben gib es die unvollständigen Implementierungen „canfd_send_main.cpp“ und „canfd_recv_main.cpp“. Verfollständigen Sie diese um einige Nachrichten auf dem CAN-Bus zu senden und zu empfangen.

Verwenden Sie außerdem „candump“ bzw. „cansend“ um die Funkionalität zu testen. Die Flag „help“ verrät, welche Modifikationen gemacht werden müssen um CAN-FD Nachrichten senden zu können.

Aufgabe 2:

Für die folgenden Aufgaben werden Funktionalitäten aus der Crypto++ Bibliothek verwendet. Diese Aufgabe stellt eine Einführung für die benötigten Features dar.

Verschaffen sie sich einen Überblick über die Datei „crypto_intro.cpp“. Compilieren Sie die Datei und führen sie diese aus. Gibt es eine einfache Möglichkeit mehrere Byte-Arrays konkateniert zu Hashen?

Aufgabe 3:

Ein Problem, mit welchem das Protokoll CAN zu kämpfen hat, ist die Manipulierbarkeit von Nachrichten auf dem Bus und das unbemerkte Einschleusen von neuen Nachrichten. Um die Integrität der Nachricht sicherzustellen, soll in dieser Aufgabe der Payload mit einem HASH versehen werden. Der Hash soll an den Payload angehangen werden und ebenfalls auf den Bus gesendet werden. Stellen Sie hierbei sicher, dass der Payload die maximale Größe von 64 Byte nicht überschreitet.

Analysieren Sie die Datei „hash_frame.cpp“, diese ermöglicht das Senden und Empfangen vom Nachrichten je nach Parameter.

1. Implementieren Sie die Methode „hash_payload“, diese ermöglicht es den übergeben Payload zu Hashen und gibt das Ergebnis im Parameter „digest“ zurück.
2. Implementieren Sie die Logik des Senders, welche die eben implementierte Methode verwendet, um die Integrität sicherzustellen. Für eine Vereinfachung kann ein statischer CAN-Frame (id & data) verwendet werden.
3. Implementieren Sie die Methode „verify_frame“, welche verwendet werden kann um die Integrität des Frames sicherzustellen.
4. Implementieren Sie die Logik zum Empfangen eines Paketes und der Prüfung der Integrität mittels der eben implementierten Methode.

Hilft diese Herangehensweise das Problem der Manipulation zu beheben? Wie könnte ein Angriff auf diese Implementierung aussehen? Gibt es eine Möglichkeit dieses Problem zu beheben?

Aufgabe 4:

Wie bereits angedeutet behebt die Implementierung aus Aufgabe 3 das Problem der Manipulation nicht, da ein Angreifer den kompletten Inhalt der Nachricht austauschen kann und so auch den errechneten Hash durch einen gültigen ersetzen kann. Um diesen Mangel zu beheben werden in dieser Aufgabe HMAC's eingesetzt, welche mittels eines privaten, symmetrischen Schlüssels die Generierung eines gültigen Hashes zu einer manipulierten Nachricht unmöglich machen.

Verwenden Sie die Datei „hmac_frame.cpp“ als Template, der Aufbau der Datei orientiert sich stark an Aufgabe 2.

1. Implementieren Sie die Methode „hmac_payload“, diese ermöglicht es einen Hash Message Authentication Code für den übergeben Payload zu erzeugen.
2. Implementieren Sie die Logik des Senders, welche die eben implementierte Methode verwendet um die Integrität wirklich sicherzustellen. Für eine Vereinfachung kann ein statischer CAN-Frame (id & data) verwendet werden.
3. Implementieren Sie die Methode „verify_frame“, welche verwendet werden kann um eine die Integrität des Frames mittels des symmetrischen Schlüssels sicherzustellen.
4. Implementieren Sie die Logik zum Empfangen eines Paketes und der Prüfung der Integrität mittels der eben implementierten Methode.

Ist diese Herangehensweise immer noch anfällig für Attacken? Unter welchem Umständen könnte man ein Fehlverhalten auf dem Bus verursachen? Gibt es eine Möglichkeit dies zu Beheben?

Aufgabe 5:

Die gerade implementierten HMAC's schützen die Integrität der Nachricht zuverlässig, was eine Manipulation des Inhaltes ohne Kenntnis des Schlüssels unmöglich macht. Zeichnet man jedoch Nachrichten auf und spielt diese zu einem späteren Zeitpunkt wieder ab, kann es immer noch zu ungewolltem Fehlverhalten auf dem Bus kommen. Um Replay-Angriffe zu verhindern verwendet SecOC einen Wert, um die Frische der Nachricht zu garantieren, dieser besteht entweder aus einem Zeitstempel oder einem Nachrichtenzähler, welcher beim jedem Sende/Empfangen inkrementiert wird. In dieser Aufgabe wird der Code aus Aufgabe 4 um einen Zähler erweitert, um die Frische der Nachricht sicherzustellen.

Achtung: Die HMAC muss auch über den Zähler gebildet werden, da er sonst wieder manipuliert werden kann.

Verwenden Sie die Datei „hmac_frame.cpp“ als Template, der Aufbau der Datei orientiert sich stark an Aufgabe 2 & 3.

1. Implementieren Sie die Methode „fresh_frame“, welche eine HMAC über den Payload und den Counter bilder.
2. Implementieren Sie die Logik des Senders, welche die eben implementierte Methode verwendet, um die Integrität sicherzustellen und Replay-Angriffe verhindert. Für eine Vereinfachung kann ein statischer CAN-Frame (id & data) verwendet werden.
3. Implementieren Sie die Methode „verify_frame“, welche verwendet werden kann um die Integrität des Frames und die Korrektheit des Counters sicherzustellen.
4. Implementieren Sie die Logik zum Empfangen eines Paketes und der Prüfung der Frische mittels der eben implementierten Methode.

Was würde passieren, wenn in der Übertragung ein Paket verloren geht? Welche Änderungen müssten im Code vorgenommen werden, um diesen Randfall abzufangen?

Überlegen Sie welche Notwendigkeit die Verwendung von symmetrischen Schlüsseln, Zeitstempeln oder Countern mit sich bringen? Welcher Prozess müsste angestoßen werden, wenn ein Steuergerät ersetzt wird?