# Supply-Chain-Attack – immer das gleiche Problem

von Thomas Kestler, 28.01.2023

# Einleitung

heise.de berichtete[1] am 21.01.2023 von einer Vielzahl von Angriffen auf IoT-Geräte mit Realtek SoC, insbesondere Router.  Die Lücke selbst bestand schon länger und wurde (u.a.) von Onekey in einem Advisory[2] vom 16.08.2023 detailliert beschrieben.

Die Lücke ist Teil des Realtek SDK und dass seit Jahren über mehrere Evolutionen des SDK hinweg. Dadurch, dass dieser SDK von vielen OEMs kritiklos verwendet wird verbreitete sich diese Lücke auf 190 Geräte von 66 Herstellern. Dabei haben einige OEMs sogar den Sourcecode von Realtek erhalten um in für ihre Plattform zu integrieren.

Die im Advisory gezeigten Sourcecode-Snippets zeigen das übliche Problem: schlampige Programmierung, keine Kontrollen wie Code-Review, Code-Analyse, unzureichende Tests.

Dies ist keineswegs nur ein Problem von Realtek, sondern nahezu alle Software ist gespickt mit solchen potenziellen Angriffsvektoren. Dies gilt sowohl für Betriebssysteme, Bibliotheken/SDKs und Anwendungen. Es existiert einfach entweder keine Kompetenz oder keine Bereitschaft zu Besserung.

1 https://www.heise.de/news/Massive-Supply-Chain-Attacke-auf-Router-von-Asus-D-Link-Co-beobachtet-7471324.html
2 https://onekey.com/blog/advisory-multiple-issues-realtek-sdk-iot-supply-chain/

## Einige Details

Im Advisory von Onekey werden eine Stellen im Sourcecode analysiert. Ich nehme hier mal ein Beispiel:

```
48.       line += 1;
49.       start = (unsigned long)line;
50.       while ((unsigned long)line < buffer_end) {
51.         if (*line == '/') {
52.           end = (unsigned long)line;
53.           if (end <= start)
54.             return UPNP_E_INVALID_PARAM;
55.           else {
56.             char port[10];                          // stack buffer
57.             memset(port, 0, 10);                    // zeroed out
58.             memcpy(port, (char *)start, end-start); // VULN: end-start can be > 10
59.             sub->port = atoi(port);
60.             GotIPandPort = 1;
61.             start = (unsigned long)line;
62.             end = 0;
63.             break;
64.           }
65.         }
66.         line++;
67.       }
68.   get_callback:
69.     if (!GotIPandPort)
70.       return UPNP_E_INVALID_PARAM;
71.     while ((unsigned long)line < buffer_end) {
```

*Schaubild 1: Screenshot Sourcecode aus Onekey Advisory*

Es geht hier um die Funktion GetIPandPortandCallBack(), welche IP-Adresse und Port aus der Callback-URL ermittelt.

Zum einen wird die Variable port am Stack angelegt und ist damit natürlich gefährdet für Überschreiben, wenn mehr als 10 Zeichen dort hin geschrieben werden (memcpy() prüft keine Grenzen!). Und zum anderen findet dann eben keine Prüfung der Übergabeparameter statt. Somit kann ein Angreifer dies nutzen, um den Stack gezielt zu überschreiben.

Es geht hier um eine Funktion für UPnP SUBSRIBE, mit der sich ein Client als Subscriber registrieren kann. Warum muss man hier mit Lowlevel-Funktionen wie memcpy() operieren? Auf dieser Eben sollten abgesicherte Funktionen für das Handling von URL-Parametern genutzt werden, niemals memcpy()! Hätte der Programmierer wenigstens die Variable port vom Heap mit malloc() angefordert, dann wäre wenigstens kein Überschreiben des Stack möglich gewesen. Aber Generationen von Programmierern übernehmen solche Praktiken aus HelloWorld-Beispielen.

# Lösungen

Im Grunde sind die Lösungen einfach, aber eben unbequem. Die Ansätze sind auch seit Jahrzehnten bekannt und füllen Regale. Nur leider werden sie ständig und wiederholt ignoriert.

## *Ausbildung*

Man fragt sich, was an den Hochschulen gelehrt wird, den schon die X-te Generation von Programmierern, pardon EntwicklerInnen, macht immer wieder die gleichen Fehler. Meiner Meinung nach ist die Ausbildung oft zu oberflächlich. Auch nach der Hochschule geht die Ausbildung weiter (oder erst richtig los). Unternehmen glauben aber, sie stellen Absolventen ein und können denen sofort komplexe, verantwortungsvolle Aufgaben übertragen. Man muss allerdings schon unterscheiden im Niveau der Hochschulen weltweit im Vergleich (ich halte hier mal die Stange hoch für die deutschen Hochschulen).

In der Praxis verstehen z.B. nur ganz wenige Entwickler (auch Absolventen) wie mächtig, komplex und gefährlich C++ ist.

## *Saubere Architekturen*

Es klingt immer so verlockend, eine Programmiersprache für alle Schichten vom Betriebssystem bis zur App zu nutzen, aber in der Praxis zeigt es sich immer wieder, dass ein präzises Rasiermesser wie C++ eben nicht für alle gleichermaßen gut geeignet ist. In der Systemprogrammierung (Betriebssystemkern, Treiber), da wo es um Performance geht, ist C++ hervorragend geeignet. Auf Anwendungsebene kommt es nicht auf Performance an (und in dem Beispiel des Realtek-Codes schon dreimal nicht!).

Funktionen wie memcpy() sind gefährlich, wenn man Übergabeparameter ungeprüft hineingibt. Auf Anwendungsebene haben solche Lowlevel-Funktionen nichts zu suchen! Ein SW-Architekt muss sicherstellen, dass sie nicht eingesetzt werden. Dafür gibt es sicherere Funktionen und wo nicht, da sollte man sie als Basisbibliothek bereitstellen. In Verbindung mit lokalen Variablen (Stack!) schafft man damit eine Pforte zur Hölle.

## *Code-Review*

Wie das Beispiel von Realtek zeigt, kann eine kleine Schlamperei massiven Schaden verursachen. In der Zeit der ersten Großrechner kam kein Programm auch nur zur Kompilation, bevor es nicht am Schreibtisch überprüft wurde. Heute haben wir Möglichkeiten Code-Review in den Buildprozess zu integrieren.

Pair-Programming ist eine Möglichkeit eines kontinuierlichen Code-Reviews, aber kann solche Fehler nur bedingt verhindern. Ein zusätzliches Code-Review kann trotzdem nötig sein (z. B. in Abhängigkeit von Erfahrung/Kompetenz der Programmierer).

## *Code-Hygiene*

Ein Sourcecode, der nicht laufend durch Refactoring verbessert wird, ist tot. Leider bestehen SW-Systeme heute überwiegend aus solchen Leichen. Die damaligen Programmierer sind längst fort und keiner will den Code mehr anfassen. Entweder baut man neue Lösungen daneben an oder man nutzt Code, den man nicht mehr versteht.

Ein guter SW-Architekt sollte ständig überprüfen wie frisch einzelne Module des Sourcecode sind und was wirklich noch benötigt wird und was nicht. Die Tools dafür gibt es längst.

## *Testing*

Das klingt jetzt fast schon albern, aber ja, durch einen einfachen Test mit „1234AAAAAAAAAAAAA...AAAAAAA“ für den Port im obigen Beispiel hätte das Problem gefunden werden können (SIGSEV durch Überschreiben des Stack, siehe Advisory).

Das sind Grundlagen im ISTQB Buch Foundation Level[3].

3 z. B. https://www.amazon.de/Basiswissen-Softwaretest-Weiterbildung-Foundation-ISTQB%C2%AE-Standard/dp/3864905834/ref=sr_1_2

## Zusammenfassung

Supply-Chain-Attack ist ein schönes, neues Buzzword, aber es sind immer die gleichen Schlampereien. Die Branche lernt einfach nicht dazu. SDKs und Bibliotheken werden kritiklos übernommen und so setzt sich die Ausbreitung fort.

Dass solche Fehler über Jahre (Jahrzehnte) im Sourcecode verbleiben zeigt, das es an Code-Hygiene mangelt.

Am Rande: Das Advisory verweist auch auf die Attacken Kaseya[4] und Solar Winds[5]. Gerade im Falle Solar Winds haben hochspezialisierte Hacker (wohl staatlich unterstützt) gezielt einen Infrastrukturdienstleister ausgesucht, um ihren Schadcode weltweit zu installieren (lesenswert)

4 https://portswigger.net/daily-swig/revil-ransomware-attackers-demand-70m-following-kaseya-vsa-supply-chain-attack-nbsp

5 https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html