Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

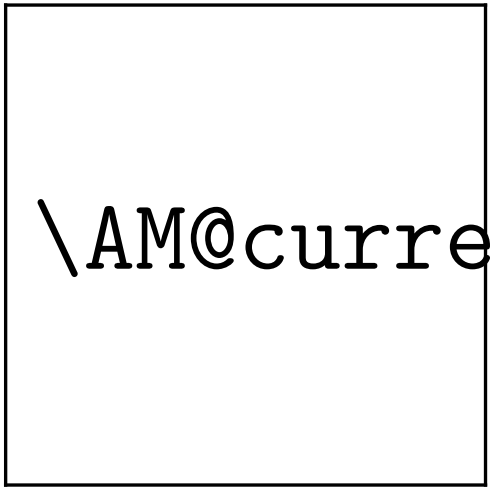
Projektarbeit Informatik

BlueBorne War Driving

Autoren	Benjamin Gehring Béla Horváth
Hauptbetreuung	Prof. Dr. Bernhard Tellenbach
Nebenbetreuung	Thomas Sutter

Datum	20.12.2019

Inhaltsverzeichnis



.pdf

2 Zusammenfassung

3 Abstract

4 Vorwort

Hier folgt eine Danksagung sowie eine Erwähnung aller beteiligten Personen welche zum Erfolg der Arbeit geführt haben.

5 Einleitung

5.1 Ausgangslage

Das Thema „BlueBorne“ ist keinesfalls neu. Trotzdem finden sich noch immer diverse IoT Geräte welche genau durch diesen Angriff infiltriert werden können. „BlueBorne“ setzt sich aus den Wörtern „Bluetooth“ und „Airborne“ zusammen und beschreibt einen möglichen, sehr infektiösen Angriff über das Bluetooth Protokoll. Um die Sicherheitslücken testen zu können werden Geräte benötigt, welche auf einem veralteten Softwarestand betrieben werden können. In dieser Arbeit werden ein Google Nexus 5x[1] und ein Raspberry Pi Model 3b+[2] als Testgeräte verwendet. Die zugrundeliegenden Whitepaper von Armis [3, 2] werden als Einstiegspunkt in die Materie gewählt. Um die technischen Auswirkungen zu sehen, werden die von Armis Labs zur Verfügung gestellten Skripte verwendet. Diese dienen nur zur Veranschaulichung und sind teils auf bestimmte Geräte zugeschnitten. Der aktuelle Stand der Technik zeigt, dass nur eine sehr begrenze Anzahl an Blueborne Scannern existiert. Alle diese Scanner haben die Gemeinsamkeit, dass sie die Geräte nur nach der MAC-Adresse und nicht nach der installierten Softwareversion überprüfen. Dies ist zwar weit einfacher, jedoch liefert es auch viel weniger Informationen über das zu testende Gerät. Das Team interessiert sich für Softwaresicherheit und Informatik Sicherheit, wodurch sich dieses Thema sehr gut eignete.

5.2 Zielsetzung

Das Ziel dieser Arbeit ist es, ein besseres Verständnis der BlueBorne Sicherheitslücke zu erlangen, sowie ein BlueBorne Testing Modul [3]zu entwickeln um potentiell gefährdete Geräte zu erkennen und die Benutzer dieser zu warnen. Dies wird erreicht durch eine vertiefte Einarbeitung in die BlueBorne Thematik mit den korrelierenden CVE Einträgen welche die Angriffe überhaupt ermöglichen. Der Fokus wird dabei auf den Betriebssystemen Android und Linux liegen. Diese Systeme sind am leichtesten in eine unsichere Version zu bringen wodurch sie sich als Testobjekte eignen. Ausserdem sind diese beiden Systeme die mit am weitesten verbreiteten Systeme im IoT Bereich.

5.3 Aufgabenstellung

Die Definierte Aufgabenstellung finden Sie im Punkt 12.1.

[1] Getestet mit verschiedene Versionen von Android 7.1.2 und älter
[2] Installiert mit einem Ubuntu Mate 16.04
[3] Wird später auch als „Scanner Applikation“ bezeichnet

5.4 Anforderungen

Als Ergebniss der Arbeit soll ein Scanner hervorgehen der möglichst viele der 8 BlueBorne Sicherheitslücken auf Geräten, innerhalb eines bestimmten Suchbereichs, erkennt. Der Scanner versucht zuerst über passives mithören oder aktives Anfragen die Geräte in der Umgebung zu lokalisieren. Der grössere Teil sollte hier jedoch passiv geschehen. Nachdem die Geräte entdeckt wurden, werden diese nach Hersteller und anderen Merkmalen klassifiziert, um den richtigen „Exploit“ für dieses Gerät auswählen zu können. Danach hat man die Möglichkeit diese Geräte anzugreifen, wodurch ermittelt wird ob dieses Gerät die gesuchte Sicherheitslücke besitzt oder nicht. Dies kann dann auf die Installierte Softwareversion zurückgeführt werden und es wird eine entsprechende Meldung ausgegeben. Das Tool wird in Python entwickelt.

6 Theoretische Grundlagen

6.1 BlueBorne

BlueBorne ist der Überbegriff einiger Sicherheitslücken in der Bluetooth Implementation von Android, IOS, Linux und Windows Geräten. Alle Geräte welche Bluetooth aktiviert haben sind potentiell gefährdet. Die Sicherheitslücke wurde von der Firma Armis entdeckt und im September 2017 öffentlich präsentiert. BlueBorne umfasst total acht Sicherheitslücken auf verschiedenen Plattformen welche alle zu funktionalen Exploits umgesetzt werden konnten. Alle in dieser Arbeit gezeigten Vorgehen sind stets konzeptuelle Veranschaulichungen.

6.2 Ubertooth

Da viele Bluetooth Geräte im öffentlichen Raum als „unsichtbar“ konfiguriert werden, kann man sie nicht mit Hilfe eines normalen Bluetooth-Adapters erkennen. Um solche Geräte zu entdecken und auch anzugreifen, benötigt man normalerweise extra Hardware wie z.B. einen Ubertooth Adapter. Mit dem Ubertooth ist es möglich genau diese Geräte zu finden und anzusprechen.

6.3 Linux Kernel RCE vulnerability - CVE-2017-1000251

Um diese Sicherheitslücke zu misbrauchen benötigt man zwei bluetooth Geräte. Das Gerät welches ängegriffen”wird muss für diesen Exploit in den ”PENDING”Modus versetzt werden. Dies erreicht man durch das folglich, vereinfacht beschriebene, Verfahren:

Angreifer sendet und empfängt:

1. Start: Der Angreifer stellt eine Verbindung zum Opfer über den L2CAP-Socket her. Dies muss geschehen um Daten zu schicken und um die DCID des Opfers zu erhalten.
2. Sendet: Configuration Request (DCID = 0x0040 / Destination Channel ID of L2CAP)
3. Sendet: Configuration Request mit dem EFS Element mit
   stype = L2CAP_SERV_NOTRAFIC
4. Empfängt: Configuration Request von Opfer mit DCID = 0x0040

5. Sendet: Configuration Response mit Result Feld = L2CAP_CONF_PENDING und den wiederholten Parametern (im Beispiel die MTU) um den Stackoverflow zu generieren.

Bei dieser Attacke ist es sehr wichtig das alle gesendeten Pakete welche den Stackoverflow generieren sollen L2CAP konforme Pakete darstellen.

6.4 BlueZ information leak vulnerability- CVE-2017-1000250

Die BlueZ information leak Sicherheitslücke beschreibt einen Fehler in der Implementation des BlueZ Servers auf Linux Basis. Der Fehler liegt hierbei in der Abhandlung von fragmentierten Paketen welche nicht als ganzes übermittelt werden können. Zu finden ist er in der Funktion: “service_search_attr_req„.

```
typedef struct {
        uint32_t timestamp;
        union {
                uint16_t maxBytesSent;
                uint16_t lastIndexSent;
        } cStateValue;
} sdp_cont_state_t;
```

Abbildung 1: cState-struct welcher maxBytesSent enthält. Wird vom Angreifer kontrolliert.

Der in Abbildung 1 gezeigte „Continuation State“ sollte eigendlich nur vom Server verwaltet werden können, wird jedoch jedes mal an den Client mitgeschickt. Der Server geht hier davon aus, dass der Client den „Continuation State“ unverändert, an seine Response gepackt, zurücksendet. Verändert man jedoch die Werte „maxBytesSent“ und „lastIntexSent“ so kann man den Server dazu bringen aus einem falschen Index (von einer falschen Adresse im Speicher) zu lesen. Um dies erfolgreich durchzuführen muss vom Client nur die „If-Abfrage“, gezeigt in Abbildung 2 umgehen, in dem er „maxBytesSent“ anpasst.

6.5 Android information Leak vulnerability - CVE-2017-0785

Diese Sicherheitslücke wurde im SDP Protokoll des Bluetooth Stacks gefunden und ermöglicht es an bestimmte Speicherdaten eines Gerätes zu gelangen. Das SDP Protokoll teilt anderen Geräten mit, welche Bluetooth Services das angefragte Gerät unterstützt. Dabei stellt das externe Gerät (Client) einen Request der vom eigenen Gerät (Server) mit einer Response beantwortet wird. Diese Response kann maximal

```
} else {
        /* continuation State exists -> get from cache */
        sdp_buf_t *pCache = sdp_get_cached_rsp(cstate);
        if (pCache) {
                uint16_t sent = MIN(max, pCache->data_size - cstate->cStateValue.maxBytesSent);
                pResponse = pCache->data;
                memcpy(buf->data, pResponse + cstate->cStateValue.maxBytesSent, sent);
                buf->data_size += sent;
                cstate->cStateValue.maxBytesSent += sent;
                if (cstate->cStateValue.maxBytesSent == pCache->data_size)
                        cstate_size = sdp_set_cstate_pdu(buf, NULL);
                else
                        cstate_size = sdp_set_cstate_pdu(buf, cstate);
        } else {
                status = SDP_INVALID_CSTATE;
                SDPDBG("Non-null continuation state, but null cache buffer");
        }
```

Abbildung 2: Betroffene Methode service_search_attr_req in der SDP-Server implementation.

die Grösse der MTU des Clients annehmen und muss ansonsten fragmentiert werden. Der Fehler liegt nun im Abhandeln dieser Responses, wobei der Server berechnet welche Fragmente noch gesendet werden müssen. Der Server schickt im Falle einer Fragmentierung einen Continuation State mit der Response zu dem Client. Der Client sendet darauf einen identischen Request nur mit diesem Continuation State als Zusatz. Der Server weiss nun welches Fragment er als nächstes schicken muss. Ist z.B. die MTU 50 Bytes gross und die Response 80 Bytes sind 2 Fragmente nötig. Die Variable „remaining_handles“ berechnet sich aus „number_response_handles“ und „continuation_offset“.

```
rem_handles =
    num_rsp_handles - cont_offset; /* extract the remaining handles */
```

Abbildung 3: Berechnung der rem_handels Variable

Schafft man es nun, dass number_response_handels kleiner ist als der continuation_offset erreicht man einen Underflow. Um dies zu erreichen benötigt man zwei Verbindungen man geht nun wie folgt vor :

1. Im ersten Schritt baut man eine Verbindung zu einem Service auf, der eine Response schickt die grösser als die angegebene MTU ist.

2. Den dabei erhaltenen Continuation State verwendet man für den zweiten Schritt, wo eine Verbindung zu einem Service aufgebaut wird der eine kleinere Response als die MTU zurückschickt.

3. Innerhalb der Berechnung der „remaining_handels Variable wird nun z.B. 2 - 1 gerechnet. Dies führt bei einem uint16 zu einem Underflow und der Server denkt nun er müsse sehr viele Responses zurückschicken.

4. All diese Responses werden mit einer for-Schleife abgefangen und enthält wichtige Daten des Speichers auf dem Gerät, welche man normalerweise nicht einsehen kann.

Dieser Leak kann in einem weiteren Vorgehen dazu genutzt werden den ASLR Schutzmechanismus gegen BufferOverflows zu umgehen. Dazu berechnet man ausgehend von einer geleakten Memory Adresse einen Offset zurück auf die Basisadresse der Bluetooth Liabrarys. Während des Angriffs wird ebenfalls dieser Leak genutzt um mithilfe des Offsets die Basisadresse zu berechnen.

6.6 Android RCE vulnerability Nr. 1 - CVE-2017-0781

Die Schwachstellen auf Android Geräten liegt im BNEP Protokoll von Bluetooth. Hierbei sei angemerkt dass Android auf dem Bluedroid Stack basiert und nich auf dem BlueZ Stack von Linux. Diese verwenden keine gemeinsamen Codeteile. Das BNEP Protokoll wird genutzt um IP Pakete zwischen zwei Geräten auszutauschen. Ein Anwendungsfall wäre z.B. die Nutzung eines gemeinsamen Mobilfunkverbindungshotspots über Bluetooth. BNEP fügt dazu einen Header vor der Ethernet Payload an. Zusätzlich werden auch Mechanismen, wie Flussteuerung über sogennante Control Messages ermöglicht. Solche Nachrichten sind durch ein Extension Header beschrieben, welcher wiederum mit einem Extension Bit angekündigt wird.

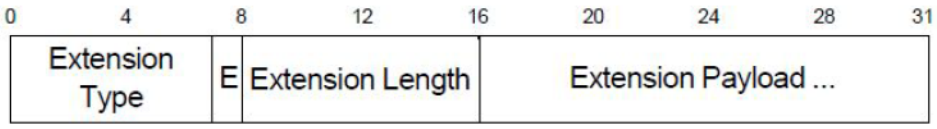


Abbildung 4: Der Extension Header des BNEP Protokolls

In der Implementation der Abhandlung solcher Control Message wurde nun ein entscheidender Fehler gefunden, der ein BlueBorne Angriff möglich macht:

Im obigen Codeabschnitt wird eine eingehende BNEP Control Message entgegengenommen und verarbeitet. Dabei wird innerhalb des if-Statments die unverarbeitete Nachricht für die spätere Nutzung in *p_pendig_data* zwischengespeichert. Innerhalb dieses Vorgangs (memcpy) verbirgt sich allerdings ein entscheidender Fehler: Die unverarbeitete Nachricht wird an die Stelle *p_pending_data+1* kopiert was natürlich zu einem Overflow führt da die Speicheradresse + 1 nach *p_pendig_data* nicht alloziert wurde. Der Overflow entspricht der Grösse rem_len. Um in diesen Codeabschnitt zu gelangen kann folgende Payload als BNEP Nachricht gesendet werden:

```
UINT8 *p = (UINT8 *)(p_buf + 1) + p_buf->offset;
...
type = *p++;
extension_present = type >> 7;
type &= 0x7f;
...
switch (type)
{
...
case BNEP_FRAME_CONTROL:
    ctrl_type = *p;
    p = bnep_process_control_packet (p_bcb, p, &rem_len, FALSE);
    if (ctrl_type == BNEP_SETUP_CONNECTION_REQUEST_MSG &&
        p_bcb->con_state != BNEP_STATE_CONNECTED &&
        extension_present && p && rem_len)
    {
        p_bcb->p_pending_data = (BT_HDR *)osi_malloc(rem_len);
        memcpy((UINT8 *)(p_bcb->p_pending_data + 1), p, rem_len);
        ...
    }
...
```

Abbildung 5: Der fehlerhafte Codeabschnitt im BNEP Protokoll

type	ctrl_type	len	Overflow payload (8 bytes)							
81	01	00	41	41	41	41	41	41	41	41

Abbildung 6: D
ie Payload welche nötig ist um den BufferOverflow auszulösen

Der Type Wert 81 entspricht dem Extension Bit für die Control Message und 01 unter ctrl_type setzt den Control Type der Message auf BNEP_FRAME_CONTROL, welcher uns ermöglich in den korrekten Switch Case zu gelangen.

Für einen erfolgreichen Angriff auf ein Android Gerät müssen noch diverse zusätzliche Schritte befolgt werden:

1. Umgehung des ASLR Schutzmechanismus von Android gegen BufferOverflows
2. Shell Code im Memory des Gerätes platzieren der durch den Overflow ausgeführt werden kann
3. Entsprechender Codeabschnitt überschreiben der diesen Shell Code ausführt

6.6.1 Schritt 1: ASLR umgehen

ASLR sorgt dafür, dass sich die Adressen für Bibliotheken und Services bei jedem Neustart des Gerätes ändern. Dies macht es natürlich schwierig einen bestimmten Speicherbereich anzusprechen, wie es im aktuellen Angriff nötig ist. Spezifisch ausgedrückt wird die Adresse der Systemfunktion benötigt, um die Programmausführung auf den Shell Code führen zu lassen und den Speicherort des Bluetooth Devices Names des Verbindungspartners, um den Shellcode überhaupt im Memory des Opfers platziern zu können. Um diese Ziele zu erreichen hilft der Memory Leak, beschrieben im Punkt 6.4, von Android. Die Abstände zwischen einzelnen Speicher-

bereichen sind immer gleich gross. Ziel ist es, die Basisadressen der „libc.so“ Bibliothek und „bluetooth.default.so“ Bibliothek zu ermitteln. Innerhalb „libc.so“ befinden sich die System Funktionen welche anschliessend, kombiniert mit einem Offset zur libc.so Basisadresse führen. In der „bluetooth.default.so“ befindet sich der Speicherort des Gerätenamens der später ebenfalls durch einen Offset zur Basisadresse ermittelt werden kann. Um die Basisadressen zu berechnen wird im Memory Leak eine Adresse gesucht die Zwischen Anfangs und Endadresse der „libc.so“ beziehungsweise „bluetooth.default.so“ Bibliothek liegt. Von dieser Adresse wird nun die Basisadresse subtrahiert um den Offset zu erhalten. Bei einem nächsten Neustart muss über den Memory Leak der exakt gleiche Speicherteil nochmals ausgelsen werden. Dieser hat nun zwar eine andere absolute Adresse, besitzt jedoch zur Basisadresse noch immer den selben Abstand. Durch diese Gegebenheit kann nun der vorherig berechnete Offset von der ermittelten Adresse subtrahiert werden um die ebenfalls durch den Neustart veränderte Basisadresse zu erhalten. An diesem Punkt ist es wichtig anzumerken dass diese Offsets pro Gerät und Android Version unterschiedlich sind. Für einen erfolgreichen Angriff muss zuerst dieselbe Android Version, wie die des Opfers, auf einem eigenen, identischen Gerät installiert werden, um so die Offsets ermitteln zu können die für den Angriff nötig sind.

6.6.2 Schritt 2: Schadcode im Speicher des Opfers platzieren

Die Platzierung des Schadcodes ist relativ simpel. Bei jedem Bluetooth Verbindungsaufbau wird der Gerätename des Verbindungspartners ermittelt und lokal abgelegt. Der Angreifer muss nun seinen Bluetooth Adapter entsprechend des Schadcodes umbenennen. Anschliessend startet er einen Verbindungsaufbau um den Gerätenamen im Speicher des Opfers zu hinterlegen und so den Shellcode zu platzieren. Für einen erfolgreichen Angriff muss nun natürlich noch, wie im ersten Schritt beschrieben, die entsprechende Speicheradresse ermittelt werden.

6.6.3 Schritt 3: Schadcode ausführen

Unter Android 7.1.2 wird mit 80% Wahrscheinlichkeit beim Bufferoveflow ein Zeiger innerhalb einer list_node überschrieben. Ziel ist es nun diesen Zeiger so zu überschreiben, dass er auf den Schadcode zeigt und dieser ausgeführt wird. Solche List Nodes werden bei jedem Verbindungsversuch erzeugt, jedoch teilweise bevor der p_pending_data Buffer alloziert wird. In diesem Fall wäre ein Exploit natürlich nicht möglich. Um zu verhindern, dass die list_nodes vorher alloziert werden, können Löcher zwischen Buffer und list_nodes generiert werden. Die ersten 8 Bytes der Payload überschreiben das Loch mit A’s und der zweite Teil der Payload platziert dann die Adresse zum Shell Code in dem list_node.

7 Vorgehen / Methoden

7.1 Research

Um einen generellen Überblick über das Thema zu erhalten, analysierten wir in einem ersten Schritt die offiziellen WhitePaper der Entdecker von BlueBorne. Daraus erhielten wir die Erkenntnis, dass es sich um mehrere, verschiedene Sicherheitslücken handelt. Des weiteren waren Beispielangriffe mit möglichen betroffenen Geräten beschrieben. Wir entschieden uns daher, dieselben Geräte zu beschaffen, wodurch wir die Beispiele exakt replizieren konnten. Allerdings waren alle diese Geräte bereits gepatcht und die Sicherheitslücke geschlossen. Als weiteres Kriterium mussten die Geräte also herabstufbar sein. Unter diesen Umständen entschlossen wir uns für ein Android Telefon Nexus 5X und einen Raspberry Pi 3B+. Beide Geräte erfüllten unsere Voraussetzungen und es gab entsprechende Beispielangriffe dazu. In einem nächsten Schritt wollten wir nun diese Angriffe selbst durchführen um daraus neue Erkenntnisse zu gewinnen und die theoretischen Erklärungen der WhitePaper besser verstehen zu können.

7.1.1 Angriff auf Android v.7.1.2 durchführen

ArmisLab haben auf GitHub ein Repository mit Code bereitgestellt. Dieses Repository umfasst Python Skripte, mithilfe deren, der Angriff für bestimmte Geräte direkt durchführbar wird. Für den Android Exploit ist ein Nexus 5X mit Android 7.1.2 nötig. Wir installierten auf unserem Nexus 5X also die Android Version 7.1.2. Im ersten Teil folgten führten wir nur den Angriff aus, ohne speziell den Sourcecode zu studieren um die „Proof of Concept“ Attacke in Aktion zu sehen.

7.1.2 Angriff auf beliebigen Android Versionen durchführen

In höheren Versionen als Android 7.1.2 wurde die Sicherheitslücke bereits behoben, wir konnten den Angriff also nur auf Geräten mit tieferen Versionen durchführen. Im Internet haben wir eine Beschreibung eines BlueBorne Angriffs auf die Version 6.0.1 gefunden allerdings für das Gerät Nexus 5. Dank Schritt für Schritt Anleitung sollte es allerdings trotzdem möglich sein, mit demselben Vorgehen einen Angriff auf unser Nexus 5X mit Android 6.0.1 und anderen Versionen erfolgreich durchzuführen. Leider wurde uns schnell bewusst, dass die Suchfunktion (memsearch) unerwartet keine Ergebnisse lieferte. Wir versuchten daraufhin dasselbe Vorgehen nochmals auf der Android Version 7.1.1 die näher an der Version 7.1.2 lag und mit der Version 6.0.1 wie beschrieben.

7.1.3 Debugging auf Android Geräten

Um den Speicher des Telefons zu durchsuchen benötigt man einen Debugger welcher sich an das Telefon anhängen lässt. Mit PEDA-ARM[4] kann man sich auf einen „gdbserver“, welchen man auf dem Telefon startet, verbinden. Ist man verbunden mit dem Debugger Server, so hat man vollen Zugriff auf den aktuellen Speicher des Gerätes sowie debugging Funktionalität wie z.B. Breakpoints. Um den „gdbserver“ auf dem Telefon auführen zu können, muss dieser zuerst auf das Telefon kopiert werden und mit allen Rechten verliehen werden. Hierbei verweisen wir gerne auf ein Manual zu GDB Debugging, welches uns von Bernhard Tellenbach zu Verfügung gestellt wurde [4].

7.1.4 Sicherheitslücke in Linux verstehen

Auf dem Raspberry Pi 3b+ wurde eine veraltete Ubuntu Mate Version 16.04 installiert. Diese Version verwendet einen Linux Kernel v. 4.1.19-v7+ und Bluethooth(BlueZ) v. 5.37. Beides davon ist genügend alt und somit noch von den beschriebenen Sicherheitslücken im Punkt 6.2 und 6.3 betroffen. Um das Prinzip des Exploits nachzuvollziehen wird nur die Stackoverflow Sicherheitslücke intensiver getestet. Sobald der Stackoverflow ohne Stack Canaries durchgeführt werden kann, kann auch der Vollzugriff auf das Gerät nach weiteren Schritten erfolgen.

[4] Das offizielle PEDA-ARM Plugin kann hier gefunden werden: https://github.com/alset0326/peda-arm

7.2 Entwicklung

7.2.1 Scanner App

In einem ersten Schritt wird eine Blueborne Scanner App in Python entwickelt. Diese App klassifiziert die entdeckten Bluetooth Geräte nach deren MAC-Adressen. Weiter wird versucht über Dienste wie „nmap“ Informationen über das Betriebssystem oder die Version zu erlangen.

7.2.1.1 Unsichtbare Geräte erkennen

Dank dem Ubertooth One können Unsichtbare Geräte entdeckt werden. Der Vorgang wird besser in der Abbildung 7 dargestellt. Wir beziehen uns hierbei auf einen Blogpost von Michael Ossmann [1] einem der Entwickler von Ubertooth und anderen Gadgets.

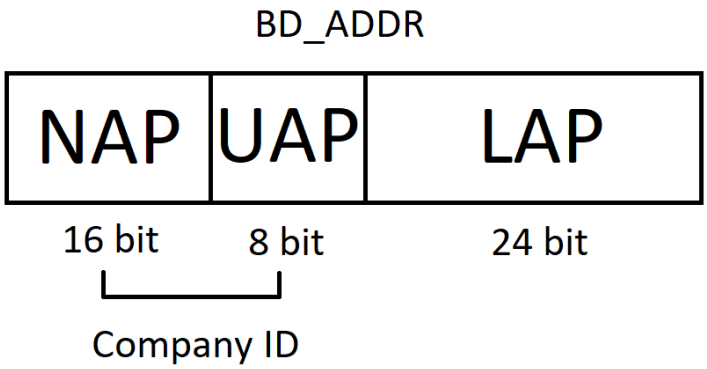


Abbildung 7: Der Aufbau einer Bluetooth Adresse mit eigenen Bezeichnungen

Hierbei ist wichtig zu verstehen, dass nur der LAP Teil der MAC-Adresse, also der „Lower Address Part“ vom Ubertooth erkannt wird. Der UAP, der „Upper Address Part“ kann, sofern man möchte, vom Ubertooth, durch passives mithören, errechnet werde. Hat man den UAP errechnet, so reicht diese fast komplette MAC-Adresse in der Form von „00:00:XX:XX:XX:XX:XX“ für die meisten Dienste aus und es kann mit dem Gerät kommuniziert werden. Um jedoch ein Gerät genau einem Hersteller zuordnen zu können, benötigt man auch den richtigen NAP, auch „Non-significant Address Part“ genannt. Wir verwenden hierfür eine MAC-Adressen Lookup Tabelle (QUELLE!!). Die Idee besteht darin, den UAP Teil der zu findenden Adresse mit dem Ubertooth zu errechnen. Nachdem wir diesen Errechnet haben, wenden wir Bruteforce an, um alle möglichen MAC-Adressen aus der Lookup Tabelle durchzutesten, welche das Format

„XX:XX:UAP:UAP:LAP:LAP:LAP:LAP“ besitzen. UAP und LAP sind hier nur Platzhalter für die richtigen MAC-Adressen Teile. Hier wird jedoch die Annahme getroffen, dass sich das anzugreifende Gerät auch in einem Verbindungsmodus befindet und auf Anfragen antwortet. Ist dies nicht der Fall, so kann die Geräte MAC-Adresse nicht genauer bestimmt werden.

8 Resultate

8.1 Angriff auf Nexus 5X mit Android v.7.1.2

Mit unseren Vorbereitungen gelang es uns sehr schnell den Angriff selbst durchzuführen und wir erhielten den kompletten Zugriff auf unser Testgerät. Dieser Erfolg half uns die Erkenntnisse aus den WhitePaper zu bestätigen und besser zu verstehen wie der Angriff strukturiert wird. Bereits zu diesem Zeitpunkt war es uns möglich erste Überlegungen, über die spätere Scanner Applikation, zu machen. In einem nächsten Schritt wird der Angriff auf andere Android Geräte erweitert um so noch bestehende Unklarheiten der Funktionsweise aus dem Weg schaffen zu können. Der Angriff war also wie erwartet ein Erfolg.

8.2 Angriff auf Nexus 5X mit Android v.6.0.1

Funktionierte bis auf funktion Memsearch

8.3 Angriff auf Nexus 5X mit Android v.7.1.1

Auf dieser Version gelang es uns den BufferOverflow durchzuführen die Suchfunktion für den Speicher[5] gab uns ebenfalls entsprechende Ergebnisse zurück. Wir konnten erfolgreich den Schadcode im Speicher platzieren, allerdings gelang es uns nicht diesen Auszuführen. Die Vermutung liegt nahe, dass die Funktion, welche wir auf Android 7.1.2 dafür verwendet hatten, um den Code auszuführen, eine andere war als in der Android Version 7.1.1. Weiterführend kann man sicherlich eine alternative Systemfunktion suchen, welche die Ausführung ermöglicht. Dies war zeitlich jedoch nicht mehr möglich. Wir genügten uns an diesem Punkt mit dem Fortschritt, da wir der Meinung waren, dass ein erfolgreicher BufferOverflow ausreicht um eine Scanner Applikation entwickeln zu können.

8.4 Angriff auf Raspberry Pi 3B+

8.5 Scanner App mit Python

[5] Funktion memsearch um bestimmte Werte im Speicher eines Gerätes zu lokalisieren.

9 Diskussion und Ausblick

10 Verzeichnisse

Literatur

[1] M. Ossmann. Discovering the bluetooth uap, Juni 2014.

[2] B. Seri and A. Livne. Exploiting blueborne in linux based iot devices, 2019.

[3] B. Seri and G. Vishnepolsky. Blueborne, September 2017.

[4] B. Tellenbach. *Exploitation – Assembly and GDB Refresher*. ZHAW/SOE/INIT, 2019.

10.1 Glossar

Begriff	**Erklärung**
CVE	Eine Liste, gefüllt mit Einträgen für öffentlich bekannte Sicherheitslücken
Exploit	Eine Sicherheitslücke (Vulnerabilität) wird auch Exploit genannt. Ein Exploit ist ein nicht absichtlich eingeführter und unbehobener Fehler im Softwarecode, welcher von Angreifern ausgenutzt werden kann, um das System zu infiltrieren oder Malware einzuschleusen.
MAC-Adresse	Die MAC-Adresse ist die Eindeutige Hardware Adresse eines Gerätes. Sie ist zwingend einzigartig. Sie setzt sich aus 3 Byte Herstellernummer und 3 Byte Seriennummer zusammen.
Malware	Malware beschreibt Bösartige Software / Schadsoftware. Es beschreibt Software welche ausschliesslich dazu entwickelt wurde, schädliche oder böswillige Funktionen auf einem System auszuführen.
MTU	die Maximum transmission unit beschreibt die maximale grösse eines Datenpakets welche von einem Empfänger verarbeitet werden kann. Ist die MTU kleiner wie das empfangene Paket so muss das Paket fragmentiert werden und in Teilen übermittelt werden.

Abbildungsverzeichnis

10.2 Tabellenverzeichnis

10.3 Symbolverzeichnis

10.4 Abkürzungsverzeichnis

10.5 Stichwortverzeichnis

11 Anhang

12 Projektmanagement

12.1 Aufgabenstellung

12.1.1 Research-Ziel

1. Analyse der Bluetooth BlueBorne Sicherheitslücken und der Sicherheitspatches
2. Durchführen einer empirische Analyse
3. Suchen nach anfälligen Geräten an der ZHAW oder allenfalls im Raum Winterthur

12.1.2 Engineering-Ziel

1. Einarbeiten in die Bluetooth-Architektur
2. Entwickeln eines BlueBorne Testing Modules (Für oder ohne das Malware App)

12.2 Zeitplan

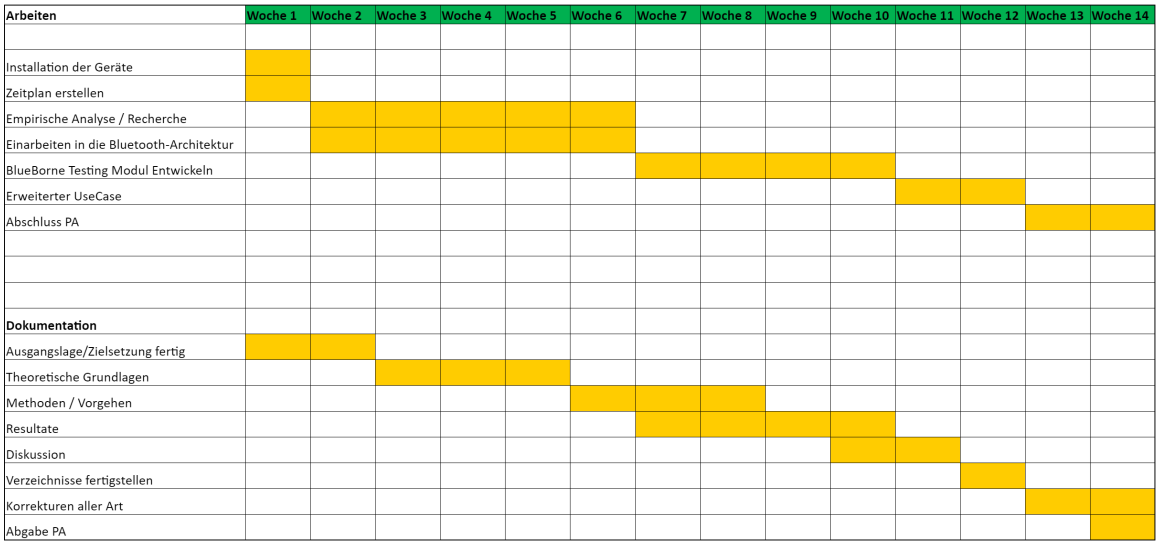


Abbildung 8: Zeitplan der Arbeit und Dokumentation

13 Weiteres