TBONE – A zero-click exploit for Tesla MCUs

Ralf-Philipp Weinmann and Benedikt Schmotzle

Comsecuris UG (haftungsbeschränkt)

2020-10-16
v1.0

Executive Summary

This document describes the Comsecuris exploit that was supposed to be used in the PWN2OWN 2020 submission to gain remote code execution over WiFi on the ICE of the Tesla Model 3. We exploit the fact that modern Tesla vehicles such as the Model 3 automatically connect to the “Tesla Service” WiFi, together with two vulnerabilities in two components of the ConnMan daemon, to gain remote code execution on the CID (Infotainment): a stack overflow in the DNS forwarder, and a stack infoleak in the DHCP component. Having control over ConnMan is much more powerful than most other non-root daemons on the CID, as it allows to shut down the firewall, change routing tables, and load and unload kernel modules [1]. For demonstration purposes, we shut down the firewall and send a message to the gateway to open the charge port of the car.

[1] Albeit these are signed, so there is no immediate escalation to root

Contents

1 Initial entry over WiFi

1.1 One SSID to rule them all

Tesla vehicles support a so-called “service WiFi”, for which hardcoded WPA2-PSK credentials are stored in the firmware. The SSID for this WiFi network is `Tesla Service`, the passphrase can either be found in the `.ssq` provided, or on random Twitter profiles: This network was already documented in Mahaffey

Figure 1: Twitter profile containing Tesla Service passphrase

and Rogers security research[1] on Tesla Model S vehicles in 2015, and is still in use. Contrary to what is reported in Keen Labs' Free-Fall paper[2], we found that not only Model 3s but also recent Model S and Model X vehicles connect to this SSID by default.

1.2 Parsing DNS replies with side effects

Looking for daemons that process input from the network without user interaction, we come across ConnMan. ConnMan is a lightweight manager for network connections primarily aimed at embedded Linux devices. It offers both a DHCP server and client as well as a DNS forwarder. While ConnMan has had a number of security issues including a stack-based buffer overflow in the DNS proxy in version 1.34 (CVE-2017-12865), the version used on the Model 3 is recent (1.37) and has no known public issues.

Since the DNS parsing looks really poor, we decide to write a little harness for the `forward_dns_reply()` function in `src/dnsproxy.c` and fuzz it with ASAN enabled using AFL[3]. In no time, we find crashes in this function. Replaying these crashes with gdb, we see the following pattern:

```
Reading symbols from src/connmand...done.
Starting program: ./src/connmand < out/crashes/id:000003,sig:06,src:006692,op:havoc,rep:8
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
=================================================================
==28665==ERROR: AddressSanitizer: negative-size-param: (size=-1)
    #0 0xf720f9b3  (/usr/lib32/libasan.so.4+0x779b3)
    #1 0x56936544 in uncompress src/dnsproxy.c:1841
    #2 0x5694ba34 in forward_dns_reply src/dnsproxy.c:2086
    #3 0x5694ba34 in fuzz src/dnsproxy.c:2200
```

[1] https://blog.lookout.com/hacking-a-tesla
[2] https://www.blackhat.com/docs/us-17/thursday/us-17-Nie-Free-Fall-Hacking-Tesla-From-Wireless-To-CAN-Bus-wp.pdf
[3] Please note that the crashes shown are from a 32-bit builds of Connman 1.36 that we initially used

```
    #4 0x5662b475 in main src/dnsproxy.c:2205
    #5 0xf6c88e80 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x18e80)
    #6 0x5662b76f  (/home/user/connman/src/connmand+0x3e76f)

Address 0xffe197e4 is located in stack of thread T0 at offset 1220 in frame
    #0 0x5694b01f in fuzz src/dnsproxy.c:2189

  This frame has 3 object(s):
    [32, 36) 'uptr'
    [96, 1121) 'uncompressed'
    [1184, 5280) 'buf' <== Memory access at offset 1220 is inside this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism
      or swapcontext (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: negative-size-param (/usr/lib32/libasan.so.4+0x779b3)
==28665==ABORTING
[Inferior 1 (process 28665) exited with code 01]
```

Looking at the corresponding source code, we see the culprit in the `uncompress()` function:

```
1        static char *uncompress(int16_t field_count, char *start, char *end,
2                                char *ptr, char *uncompressed, int uncomp_len,
3                                char **uncompressed_ptr)
4        {
5                char *uptr = *uncompressed_ptr; /* position in result buffer */
6
7                debug("count %d ptr %p end %p uptr %p", field_count, ptr, end, uptr);
8
9                while (field_count-- > 0 && ptr < end) {
10                       int dlen;                /* data field length */
11                       int ulen;                /* uncompress length */
12                       int pos;                /* position in compressed string */
13                       char name[NS_MAXLABEL]; /* tmp label */
14                       uint16_t dns_type, dns_class;
15                       int comp_pos;
16
17                       if (!convert_label(start, end, ptr, name, NS_MAXLABEL,
18                                          &pos, &comp_pos))
19                               goto out;
20
21                       /*
22                        * Copy the uncompressed resource record, type, class and \0 to
23                        * tmp buffer.
24                        */
25
26                       ulen = strlen(name);
27                       strncpy(uptr, name, uncomp_len - (uptr - uncompressed));
28
29                       debug("pos %d ulen %d left %d name %s", pos, ulen,
30                               (int)(uncomp_len - (uptr - uncompressed)), uptr);
31
32                       uptr += ulen;
33                       *uptr++ = '\0';
34
35                       ptr += pos;
```

```
36
37                          /*
38                           * We copy also the fixed portion of the result (type, class,
39                           * ttl, address length and the address)
40                           */
41                          memcpy(uptr, ptr, NS_RRFIXEDSZ);
42
43                          dns_type = uptr[0] << 8 | uptr[1];
44                          dns_class = uptr[2] << 8 | uptr[3];
45
46                          if (dns_class != ns_c_in)
47                                  goto out;
48
49                          ptr += NS_RRFIXEDSZ;
50                          uptr += NS_RRFIXEDSZ;
```

On line 27, the `memcpy()` is performed to the destination `uptr` for a fixed size of 10 bytes (`NS_RRFIXEDSZ`) without checking whether this copy operation writes outside of the result buffer that is passed to the function.

1.2.1 Jumping over the stack canary

Looking at the `connmand` binary in the firmware image provided for the PWN2OWN contest, we see that it is compiled with stack canaries. While the connmand binary restarts after a crash, and even though we could partially overwrite the stack canary with byte-level precision, bruteforcing this stack canary is not possible as it is a random 64-bit value that is rerandomized every time. Henceforth, the only option is to either find a way to leak the stack canary or to "jump" over it, meaning that we have to find a way for the pointer `uptr` to increase without writing data.

Inspecting the `uncompress()` function a plan begins to form on how to achieve this jump: As we see in line 27 of the listing for the function `uncompress()` above, a `strncpy()` is performed from `name` to `uptr` with an maximum allowed target length that is decreased in each loop iteration – at the same time, the pointer `uptr` is always increased by the actual string length of `name`. This means that we can engineer a situation in which we write outside of the buffer right up to the stack canary, but then jump forward by 8 or more bytes before continuing to write to `uptr` in line 41.

1.2.2 Triggering the stack overflow

Trying to trigger the bug, we run into the next issue: The stack overflow is only exploitable in a scenario in which the forwarding proxy (i.e. the CID) appends a domain name (i.e. when the `append_domain` field of the `request_data` struct `req` is set) to an unqualified host name (see line 20 of the listing below). We had commented out the `req->append_domain` check when fuzzing. All of the DNS queries made by the daemons running on the CID and the AutoPilot (which are routed through the CID), however, use qualified hostnames.

Pondering our options, we notice that ConnMan implements the Web Proxy Autodiscovery Protocol (WPAD) and itself issues a query for `wpad.<domain>` itself right after acquiring a DHCP lease – where `<domain>` is the domain name provided by the DHCP server to the ConnMan.

```
static int forward_dns_reply(unsigned char *reply, int reply_len, int protocol,
                                struct server_data *data)
{
...
        if (hdr->rcode == ns_r_noerror || !req->resp) {
                unsigned char *new_reply = NULL;

                /*
                 * If the domain name was append
                 * remove it before forwarding the reply.
                 * If there were more than one question, then this
                 * domain name ripping can be hairy so avoid that
                 * and bail out in that that case.
                 *
                 * The reason we are doing this magic is that if the
                 * user's DNS client tries to resolv hostname without
                 * domain part, it also expects to get the result without
                 * a domain name part.
                 */
                if (req->append_domain && ntohs(hdr->qdcount) == 1) {
```

When multiple DNS servers have been provided to ConnMan over DHCP, the internal resolver library (implemented in `gweb/gresolv.c`) will issue DNS requests for `wpad.<domain>` simultaneously. By providing `127.0.0.1` as a second DNS server in our DHCP reply, we make sure that the request by the ConnMan WPAD code is sent through the ConnMan DNS forwarder. Moreover, by sending a string of zero bytes as domain name in the DHCP options we force the query issued by `g_resolv_lookup_hostname` to be `wpad.` while at the same time making the DNS forwarder append the zero-byte domain name, causing `req->append_domain` to become true and hence making the DNS stack overflow exploitable.

1.3 Using DHCP queries to leak DWORDs from the stack

Next, we need to circumvent DEP and ASLR – the stack is marked non-executable and the stack, text and library bases addresses are randomized. We see this both in our emulated target as well as in the kernel configuration which we extracted from the kernel binary which can be found in `deploy/iasImage-bank_a`:

```
CONFIG_ARCH_MMAP_RND_BITS_MIN=28
CONFIG_ARCH_MMAP_RND_BITS_MAX=32
```

Hence, we want to to determine where the stack is located in memory and where either the `connman` or the dynamic libraries it pulls in are mapped. Note that is is sufficient to be able to leak one stable library address to compute the address of any function in any of the libraries as there is no randomized memory gap between mappings of the different libraries, i.e.:

```
root@ice-unknown:~# head -n 20 /proc/$(pgrep connmand)/maps
55c442c92000-55c442d61000 r-xp 00000000 fe:00 9077                       /usr/sbin/connmand
55c442f61000-55c442f64000 r--p 000cf000 fe:00 9077                       /usr/sbin/connmand
55c442f64000-55c442f6e000 rw-p 000d2000 fe:00 9077                       /usr/sbin/connmand
55c442f6e000-55c442f70000 rw-p 00000000 00:00 0
55c44427c000-55c4442fc000 rw-p 00000000 00:00 0                          [heap]
```

```
7f5928940000-7f5928af0000 r-xp 00000000 fe:00 2614                       /lib/libc-2.29.so
7f5928af0000-7f5928cf0000 ---p 001b0000 fe:00 2614                       /lib/libc-2.29.so
7f5928cf0000-7f5928cf4000 r--p 001b0000 fe:00 2614                       /lib/libc-2.29.so
7f5928cf4000-7f5928cf6000 rw-p 001b4000 fe:00 2614                       /lib/libc-2.29.so
7f5928cf6000-7f5928cfa000 rw-p 00000000 00:00 0
7f5928cfa000-7f5928cfd000 r-xp 00000000 fe:00 2618                       /lib/libdl-2.29.so
7f5928cfd000-7f5928efc000 ---p 00003000 fe:00 2618                       /lib/libdl-2.29.so
7f5928efc000-7f5928efd000 r--p 00002000 fe:00 2618                       /lib/libdl-2.29.so
7f5928efd000-7f5928efe000 rw-p 00003000 fe:00 2618                       /lib/libdl-2.29.so
7f5928efe000-7f5928f09000 r-xp 00000000 fe:00 6466                       /usr/lib/libxtables.so.12.2.0
7f5928f09000-7f5929109000 ---p 0000b000 fe:00 6466                       /usr/lib/libxtables.so.12.2.0
7f5929109000-7f592910a000 r--p 0000b000 fe:00 6466                       /usr/lib/libxtables.so.12.2.0
7f592910a000-7f592910b000 rw-p 0000c000 fe:00 6466                       /usr/lib/libxtables.so.12.2.0
7f592910b000-7f592910e000 rw-p 00000000 00:00 0
7f592910e000-7f5929115000 r-xp 00000000 fe:00 2641                       /lib/librt-2.29.so
root@ice-unknown:~#
```

The DNS forwarding code turns out to be unsuitable for information leaks as we cannot send queries to `connmand` from the external (WiFi) interface, but only replies. Henceforth, we decide to look at the DHCP code and luckily for us, the central listener `listener_event()`[4] allocates a stack buffer for receiving DHCP packets:

```
1  static gboolean listener_event(GIOChannel *channel, GIOCondition condition,
2                                                       gpointer user_data)
3  {
4          GDHCPClient *dhcp_client = user_data;
5          struct sockaddr_in dst_addr = { 0 };
6          struct dhcp_packet packet;
7          struct dhcpv6_packet *packet6 = NULL;
8          uint8_t *message_type = NULL, *client_id = NULL, *option,
9                  *server_id = NULL;
10         uint16_t option_len = 0, status = 0;
11         uint32_t xid = 0;
12         gpointer pkt;
13         unsigned char buf[MAX_DHCPV6_PKT_SIZE];
14         uint16_t pkt_len = 0;
15         int count;
16         int re;
17
18         if (condition & (G_IO_NVAL | G_IO_ERR | G_IO_HUP)) {
19                 dhcp_client->listener_watch = 0;
20                 return FALSE;
21         }
22
23         if (dhcp_client->listen_mode == L_NONE)
24                 return FALSE;
25
26         pkt = &packet;
27
28         dhcp_client->status_code = 0;
29
30         if (dhcp_client->listen_mode == L2) {
31                 re = dhcp_recv_l2_packet(&packet,
32                                          dhcp_client->listener_sockfd,
33                                          &dst_addr);
```

[4] in `gdhcp/client.c`

```
34                  xid = packet.xid;
```

Further down in this function we see the following switch statement for processing replies from the DHCP server:

```
1  ...
2          switch (dhcp_client->state) {
3          case INIT_SELECTING:
4                  if (*message_type != DHCPOFFER)
5                          return TRUE;
6
7                  remove_timeouts(dhcp_client);
8                  dhcp_client->timeout = 0;
9                  dhcp_client->retry_times = 0;
10
11                 option = dhcp_get_option(&packet, DHCP_SERVER_ID);
12                 dhcp_client->server_ip = get_be32(option);
13                 dhcp_client->requested_ip = ntohl(packet.yiaddr);
14
15                 dhcp_client->state = REQUESTING;
```

We note that in the `INIT_SELETING` state – reached when client has received a DHCP offer from the server – the `DHCP_SERVER_ID` option is parsed from the packet buffer and set to `dhcp_client->server_ip`. This value of this very field subsequently is encoded into the BOOTP portion of the DHCP packet that is then sent out to the DHCP server in `send_request()`:

```
1  static int send_request(GDHCPClient *dhcp_client)
2  {
3          struct dhcp_packet packet;
4
5          debug(dhcp_client, "sending DHCP request (state %d)",
6                  dhcp_client->state);
7
8          init_packet(dhcp_client, &packet, DHCPREQUEST);
9
10         packet.xid = dhcp_client->xid;
11         packet.secs = dhcp_attempt_secs(dhcp_client);
12
13         if (dhcp_client->state == REQUESTING || dhcp_client->state == REBOOTING)
14                 dhcp_add_option_uint32(&packet, DHCP_REQUESTED_IP,
15                                 dhcp_client->requested_ip);
16
17         if (dhcp_client->state == REQUESTING)
18                 dhcp_add_option_uint32(&packet, DHCP_SERVER_ID,
19                 dhcp_client->server_ip);
```

We take note of this fact; it will be useful later for correlating queries to responses containing the leaked memory contents. The above function is called through `start_request()` which is called at the end of the `INIT_SELECTING` above. The function `start_request()` also switches the client state to `REQUESTING`.

We then have a closer look at the function `dhcp_get_option` that is used for parsing options and notice that it does not use the length of the packet, but rather `rem = sizeof(packet->options)` as the initial number of bytes it will parse:

```
uint8_t *dhcp_get_option(struct dhcp_packet *packet, int code)
{
        int len, rem;
        uint8_t *optionptr;
        uint8_t overload = 0;

        /* option bytes: [code][len][data1][data2]..[dataLEN] */
        optionptr = packet->options;
        rem = sizeof(packet->options);

        while (1) {
                if (rem <= 0)
                        /* Bad packet, malformed option field */
                        return NULL;

                if (optionptr[OPT_CODE] == DHCP_PADDING) {
                        rem--;
                        optionptr++;

                        continue;
                }

                if (optionptr[OPT_CODE] == DHCP_END) {
                        if (overload & FILE_FIELD) {
                                overload &= ~FILE_FIELD;

                                optionptr = packet->file;
                                rem = sizeof(packet->file);

                                continue;
                        } else if (overload & SNAME_FIELD) {
                                overload &= ~SNAME_FIELD;

                                optionptr = packet->sname;
                                rem = sizeof(packet->sname);

                                continue;
                        }

                        break;
                }

                len = 2 + optionptr[OPT_LEN];

                rem -= len;
                if (rem < 0)
                        continue; /* complain and return NULL */

                if (optionptr[OPT_CODE] == code)
                        return optionptr + OPT_DATA;

                if (optionptr[OPT_CODE] == DHCP_OPTION_OVERLOAD)
                        overload |= optionptr[OPT_DATA];

                optionptr += len;
```

```
56          }
57
58          return NULL;
59  }
```

As the packet buffer is not zeroized before reading the packet, we can send a packet with an option code but no option data: This causes the pointer `option` in the `INIT_SELECTING` case shown above to point to a stack position with residual contents and to remotely read uninitialized stack area, one 4 bytes at a time: We can shift the position of the `option` pointer with each DHCP offer, for instance by sending a different length domain name in each offer (we call this "padding" in the exploit). This allows us to iteratively read stack contents.

Luckily for us, when testing we find both return addresses from library function calls and stack addresses in the chunk of memory we are able to leak.

1.4 Putting the bugs together for remote code execution

Since our primitives are powerful, putting them together is a pretty standard exploit development job: We simply use the stack information leak to iteratively leak both a return address from a library call and a stack address. From these two addresses we calculate the libc base address in memory and the stack pointer at entry to the `forward_dns_reply()` function. We use this information to build a ROP chain from very simple gadgets in libc that copy our shellcode[5] to a page on the stack and set the page permissions on this page to readable and executable using `mprotect()`. This shellcode is small, but its purpose is to connect to the attacker's IP on port 80, and read more code from there to a heap-allocated buffer, `mprotect()` that as well and execute it. Our demo stage 3 payload disables all packet filters. This then allows us to interact with the REST endpoint on port `4070/tcp` to display a message on the screen. Additionally, we have added a masquerading rule to forward IP packets addressed port `3500/udp` to the gateway IP `192.168.90.102`.

[5] we call this stage 2; in contrast to stage 1, which is the ROP chain

2 Instructions for and notes on running the exploit

You will need a physical or virtual machine running either Linux and an external WiFi adapter that is supported by the Linux distribution you are using. Debian Buster and Ubuntu 18.04 are recommended and tested distributions. We have used a TP-Link WN-722N in our tests. The following packages need to be installed:

- hostapd
- dnsmasq
- python3
- python3-scapy
- gcc (gcc 5.x or higher)
- net-tools [for ifconfig]
- sudo

To use the exploit, simply run `make` in subdirectory `code/` of the directory you extracted the `tbone-code-v1.0.tar.gz` tarball in. You will need to be able to elevate to root using sudo. The exploit uses hostapd to provide a WiFi access point with the "Tesla Service" SSID. Scapy is uses to construct DNS and DHCP packets, gcc is used to build the shell code with custom linker scripts. Please note that the `tbone.py` Python script starts `hostapd` as a sub-process but performs minimal error checking; if you don't see a message `AP-ENABLED` after start-up, something went wrong. We have included an asciinema recordings for both 2018.42.3 and 2020.4.1 of a successful exploit run.

We are also shipping a Dockerfile that sets up an appropriate environment and allows running the exploit with minimal fuss. Use the `docker-build.sh` script to build a Docker image and `docker-run.sh` to be dropped into the environment; please note that this runs a privileged container with the host network.

Our exploit supports versions 2020.4.1 (the version that was published for PWN2OWN 2020) and 2018.42.3 (installed on a salvaged Model 3 MCU we obtained off eBay in 2019). The exploit is not 100% reliable as this was not our objective; also, our exploit does not use version fingerprinting but requires manual adaption of the `tbone.py`[1].

A successful exploit run will take approximately 30–45 seconds, however, when unexpected values are encountered during the stack leaking, the MCU will disconnect and subsequently reconnect to the hotspot. Only 6 values can be leaked in each round before the MCU disconnects.

[1] the `MCU_VERSION` variable at the top of the script needs to be changed.

3 Privilege escalation on the MCU

REDACTED.