Solution to Exercise #2

|  |  |
| --- | --- |
| Authors | |
| **Name** | **Matriculation Number** |
| Beshoy Saad | 2572741 |
| Vitalij Minor | 2565427 |

# Teach Me How to Drive

1. We wrote a python script to send the provided CAN frames to the Arduino one by one (with one second delay) and noticed the cluster’s response. We observed that the cluster turned on after the frame 35C#F4C60F93AE3F6224 and turned off after a while. Sending the aforementioned frame once every 500 ms (2 Hz) kept the cluster on.
2. Again trying each frame we observe that the speedometer displays the speed value 87 km/h after the message 5D7#1D13036BF150FE (ID 0x5D7). By trial and error, we discovered that the first three nibbles of the frame with ID 0x5D7 are used to display the speed information. Trying different values for these three nibbles we discovered that the speed is calculated from these 12 bits using the formula (), where is the value of the 12 bits. Applying this formula we were able to calculate the 12-bit value corresponding to 136 km/h: . Sending the frame 5D7#3363036BF150FE to the cluster displayed the speed value of 136 km/h. To discover which bit is responsible for the “ready” status, we wrote another script to flip one bit at a time of the payload of the frame with ID 0x652 and send it to the Arduino. Doing so revealed that the 7th bit of the 2nd byte of the payload was responsible for that status (0 = “ready”). So the frame that turns off the “ready” status is 652#2F4FFF1540055551.

# Baby you can drive my car!

1. Buffer overflow happens when a piece of software writes beyond allocated memory (or buffer). This can easily happen in C and C++ which don’t have built-in bound checks in the language. Buffer overflows can happen unintentionally due to a bug in the code or an attacker can exploit poorly designed software to cause a buffer overflow and write some data to memory that enables them to compromise the system.
2. A good starting point to look for buffer overflow vulnerabilities is in code that handles input from outside the system, like the data reception part of networking code, textual user input, and reading files from storage media to give some examples. If proper care is not taken when developing this code, it can be very easy to overlook buffer overflow vulnerabilities.
3. In order to prevent buffer overflows care must be taken when handling data from external sources (outside the program code). For example, never rely on the presence of the string termination character “\0” when reading a string, but limit the number of characters to be read to the size of the allocated buffer. Also, use safe functions when manipulating strings (strncpy instead of strcpy and so on). So in general, good coding practices can prevent buffer overflow vulnerabilities. Using a language that enforces boundary checks like Java can also be a solution but only on platforms that can run it.

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Payload Changes? | Tx Period (ms) | Frequency (Hz) |
| 0x050 | No | 10 | 100 |
| 0x0C2 | Yes | 80 | 12.5 |
| 0x1A0 | Yes | 10 | 100 |
| 0x1AC | No | 10 | 100 |
| 0x280 | Yes | 10 | 100 |
| 0x288 | Yes | 80 | 12.5 |
| 0x3D0 | No | Twice per 80 | 12.5 × 2 |
| 0x470 | No | 80 | 12.5 |
| 0x531 | No | 80 | 12.5 |
| 0x5A0 | Yes | 10 | 100 |
| 0x5C1 | No | 80 | 12.5 |
| 0x635 | No | 80 | 12.5 |

1. By analyzing the logs and excluding the frames with fixed payload, we can see that the ID 0x5A0 is responsible for the speed value as the payload variation follows the pattern seen on the cluster’s speedometer. This is confirmed by sending the frame 5A0#0000000000000000 repeatedly and observing that the speedometer stays close to 0 km/h.
2. Excluding the frames with varying payload (as the airbag indicator is always off), we tested each frame ID with an all 1s payload to see when the airbag indicator would come on. We found out that the frame ID 0x050 was the one responsible for airbag status.
3. * The program is vulnerable to buffer overflows at line 69. Specifically, parseIntoCanBuf is called with the len argument taking on a value that can be larger than the size of canBuf, namely packetSize which is the size of the received UDP packet - 4. This doesn’t cause problems as long as the size of the payload of the sent UDP buffer is less than or equal to the legal maximum of 8 bytes, but once it exceeds that limit then the extra bytes get written to the memory positions after canBuf, which in this case happens to be the allowed buffer.
   * Our attack goes as follows: we send a CAN frame to the Arduino over UDP using one of the allowed IDs. The size of the payload of this frame, however, is more than the maximum legal size of 8 bytes. By trial and error, we discovered that we needed 4 extra bytes before starting to overwrite the allowed buffer; after that, each extra byte overwrites one byte of the allowed buffer. By overwriting the first allowed ID with the ID of the frame we want to send (0x288), we can send the spoofed engine temperature frame and achieve our goal.
   * The frame we sent to cause the buffer overflow is:

280#DEADBEEFDEADBEEF000000008802

The last two bytes are the ID of the message we want to send (0x288) in little endian, which is the endianness of the AVR chip. We were able to verify that the attack had the intended effect by sending another frame with a different disallowed ID (0x123 for example) and observing the error message returned by the Arduino, which contains the overwritten contents of the allowed buffer. Indeed, 0x288 was now the first allowed ID in the list.

* + Having 0x288 as an allowed ID, we were able to send the spoofed frame 288#33ed000000585200 in a loop and the engine temperature light came on.

# The Bus CAN Do

1. To simulate a physical circuit connection between sender and receiver, so that if the link is broken (wire damaged for example) the receiver can detect the breakage.
2. We were sending 1 frame with 96 bits (11 ID bits + 4 length bits + 8 \* 8 payload bits + 16 CRC bits + 1 ACK bit) every 500 ms. This results in a bit rate of . The utilization therefore is .
3. Simply because this frame is a sensor reading with the sole purpose of displaying information to the driver; it doesn’t affect the car’s actuators in any way, thus it doesn’t have an effect on the car’s speed or brakes.
4. In the referenced work[[1]](#footnote-1), integrity of the messages is secured using a light-weight keyed-Hash Message Authentication Code (HMAC) that is sent in a separate CAN frame after the message to be authenticated. The chosen underlying hash function is SHA-3.
5. The following table summarizes some advantages and disadvantages of using different key establishment protocols. We used the paper1 referenced in the exercise as a source for this material.

|  |  |  |
| --- | --- | --- |
|  | Advantage | Disadvantage |
| Asymmetric Cryptography | * + Assuming each ECU has its own private-public key pair; it would be more secure than static shared keys since one leaked pair doesn’t compromise all the other ECUs, which is the case with static shared keys. | * + Encrypting and verifying messages is computationally expensive for an automotive ECU, which might introduce latency and violate the hard real-time requirements of automotive communication   + Storing the public keys of all other ECUs might be prohibitive in terms of available memory for an automotive ECU |
| Key Exchange | * + Faulty ECUs can be replaced without needing further work to redistribute static keys | * + Protocol has to be rerun every time an ECU wakes up   + Man-In-The-Middle attacks are still possible since installing a certificate on each ECU is impractical   + Multiparty key exchange is too demanding for an automotive ECU |

1. Nürnberger, Rossow (2016): VatiCAN - Vetted, Authenticated CAN Bus. Springer [↑](#footnote-ref-1)