The Janus attack

The Janus attack is a low-level CAN protocol attack where a single CAN frame contains two different payload contents.

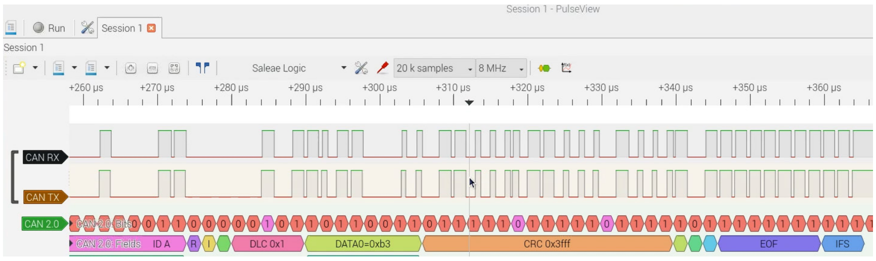


Figure 1: Logic analyzer trace of a Janus frame (Source: Canis Automotive Labs)

With the Janus Attack, a targeted device sees a different payload than other devices. This attack could be used to transmit a frame to evade an intrusion detection system (IDS), or it could put two different actuators into inconsistent states (e.g. moving a pair of motors in different directions). It breaks the atomic multicast feature of CAN (where every device sees the same frame) - an important property that lots of systems rely on (often implicitly).

The attack works by exploiting the CAN protocol synchronization rules and targets devices that have different sample points. The CAN specification defines the following rules:

a) Only one synchronization within one bit-time (between two sample points) shall be allowed. After an edge was detected, synchronizations shall be disabled until the next time the bus state, detected at the sample point, is recessive.

b) An edge shall cause synchronization only if the bus state detected at the previous sample point (previous read bus state) was recessive.

The attack can be mounted purely in software that takes control of the GPIO port connected to the CAN Tx pin of a CAN transceiver, so a hijacked device using a remote code execution vulnerability could be used to mount the attack.

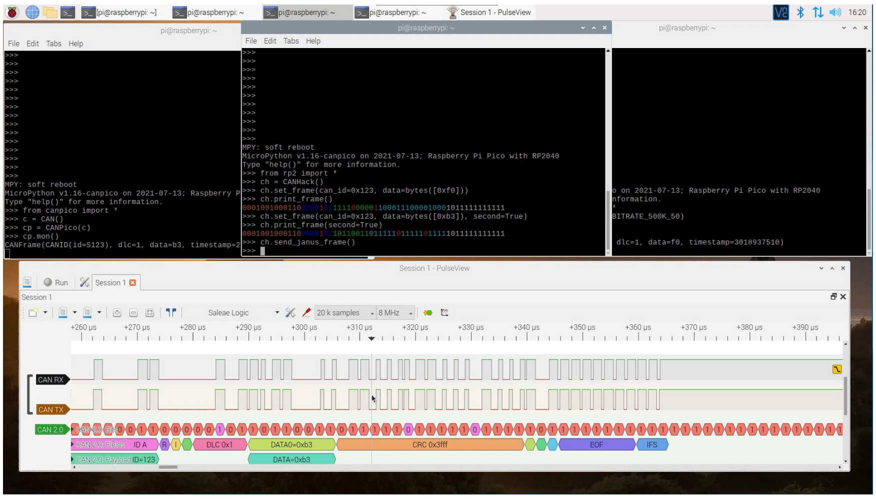
In a demonstration video of the attack, two CANPico boards (that contain the Microchip MCP2517/18FD CAN controller) are attacked by a CANHack board. The latter is a cut-down version of the CANPico that does not have a CAN controller, neatly proving that the attack can be mounted in pure software. The logic analyzer is running the Sigrok Pulseview CAN2 protocol decoder to show how the Janus signal is decoded into a CAN frame.

How does the attack work?

The attack forces CAN controllers to synchronize at the same time and then changes the CAN bus level after one controller has sampled the bus but before another. The bit sequences are set so that each device sees a valid frame, but the frames can have different payloads. The logic analyzer trace (Figure 1) shows how a Janus frame is made up of many more transitions than CAN bits but that form a valid CAN frame.

There are two restrictions on the bit sequences. First, the first and second CAN frame have to have the same length, so there must be the same number of stuff bits. The CANHack tool kit has a function to show the bit patterns of both halves of a Janus frame (Figure 2).

Second, if the Janus bit is ***10*** (i.e. the first sampled value in a CAN bit is a ***1*** but the second sampled value is a ***0***) then all controllers have to see the same subsequent bits (***00*** or ***11***) until they are brought back into sync (which happens after a ***11***).



```
MPY: soft reboot
MicroPython v1.16-canpico on 2021-07-13; Raspberry P
Type "help()" for more information.
>>> from canpico import *
>>> c = CAN()
>>> cp = CANPico(c)
>>> cp.mon()
CANFrame(CANID(id=S123), dlc=1, data=b3, timestamp=2
```

```
MPY: soft reboot
MicroPython v1.16-canpico on 2021-07-13; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> from rp2 import *
>>> ch = CANHack()
>>> ch.set_frame(can_id=0x123, data=bytes([0xf0]))
>>> ch.print_frame()
0001001000110000010111110000011000111000010001011111111111
>>> ch.set_frame(can_id=0x123, data=bytes([0xb3]), second=True)
>>> ch.print_frame(second=True)
0001001000110000010110110011011111011111011111011111111111
>>> ch.send_janus_frame()
>>>
```

```
o on 2021-07-13; Raspberry Pi Pico with RP2040
nformation.
*
BITRATE_500K_50)

 dlc=1, data=f0, timestamp=3018937510)
```

Figure 2: Setup of the two CANPico boards and the CANHack board in the middle. The CANHack tool kit has a function to show the bit patterns of both halves of a Janus frame. (Source: Canis Automotive Labs)

Updated CANPico firmware released

The CANPico firmware has been updated to the latest release of Micropython v. 1.16. The update fixes some bugs and adds robustness to SPI (serial peripheral interface) noise (for example, if SPI pins accidentally pick up noises from logic analyzer probes where the logic analyzer is not grounded). Thus, corrupted values could be written to or read from the CAN controller. The firmware keeps counters for the number of corrupted values with a new method ***get_diagnostics()***.

The documentation has been updated to reflect this API (application programming interface) call. There is also a new version of the ***canpico.py*** file of example functions that are useful when trying out different things on the CANPico board. This file can be copied to the CANPico board using the ***rshell*** utility.

There is a Janus bitstream test function called ***is_janus()*** included in the latest version of the Python CAN frame tool in the CANHack repository, plus a simple brute force algorithm to look for Janus payloads (no doubt, other smarter algorithms exist as well). This can be used to create CAN frames to show how the attack works. It would also be possible to attack devices with sample points that were more similar if the CANHack toolkit would use the output-compare-timer hardware present in most microcontrollers to make the CAN Tx transitions more accurately. But the goal with the CANHack toolkit is not to make it easy to attack a CAN bus but to prove that there is vulnerability that must be defended against.

How to defend against Janus attack?

Firstly, an intrusion detection system (IDS) with dedicated hardware should be used to detect these transitions. An IDS that uses a conventional CAN controller cannot detect this (it also cannot detect many other CAN protocol attacks). Secondly, devices should have sample points set as close to each other as possible: Ideally, this would be a part of an acceptance test when integrating devices together on to a CAN network. There are other protections too. Using the CAN-HG Bus Guardian hardware prevents a Janus frame from being sent and allows an IDS to shut down an attack. Protecting a payload with a cryptographic message authentication code (MAC) makes it much harder for an attacker to find a valid Janus payload, even if the attacker has the ability to sign messages with the necessary shared cryptographic key.

Author

Ken Tindell
Canis Automotive Labs
ken@canislabs.com
canislabs.com