An Introduction to the CANard Toolkit

Eric Evenchick

March 9, 2015

Abstract

The Controller Area Network (CAN) protocol is used for networking controllers in industrial and automotive applications. Interfacing with a CAN network requires specialized hardware and software.

This paper introduces CANard, a Python library for communicating with CAN bus systems. It is capable of sending and receiving frames on a network, and includes support for protocols that are commonly used on CAN.

1 Introduction

Controller Area Network (CAN) is used as a networking protocol in industrial control systems and automotive applications. CAN has become popular due to the low cost of implementation, and built in reliability features. It is ubiquitous in the automotive industry, and became the mandatory protocol for automotive diagnostics in the United States in 2008.

CAN based systems typically assume that anyone with physical access to the network is trusted. Once a device is placed on a CAN network it is able to read all traffic, send fraudulent messages, or perform a denial of service attack.

The CANard [1] toolkit is a Python library which aims to make it easy to interact with CAN networks. This toolkit has a few goals:

1. Hardware Abstraction
2. Protocol Implementation
3. Ease of Automation
4. Sharing of Information

2 Basic CAN Communication

All communications on a CAN network are encapsulated as a **frame**. A CAN frame consists of:

- Identifier

- Control flags
  - Remote Request Flag
  - Extended ID Flag
- A data length code
- 0 to 8 bytes of data

CANard encapsulates CAN frames as Python objects. These frame objects can be sent, received, logged, and inspected. Listing 1 creates a standard CAN frame with identifier `0x123`, data length code `5` and data bytes `1, 2, 3, 4, 5`.

```
from canard import can

f = can.Frame(0x123)
f.dlc = 5
f.data = [1,2,3,4,5]
```

Listing 1: Creating a Frame in CANard

This example frame can now be sent using a hardware device. This simple interface makes it easy to generate and send payloads, or analyze frames received from the bus.

3 Hardware Abstraction

Since traditional PCs lack a CAN bus interface, an external adapter is required. A variety of adapters exist to provide a CAN bus interface over USB. Each one has its own drivers and tools.

The CANard library currently supports Linux's SocketCAN [2]. Any CAN interface supporting SocketCAN will therefore work with CANard on Linux. CANard also directly supports the CANtact [3] interface on Windows, OS X, and Linux.

```
from canard.hw import socketcan

dev = socketcan.SocketCanDev('can0')

dev.start()

while True:
    f = dev.recv()
    dev.send(f)
```

Listing 2: A CAN Echo Script

The CAN echo example will repeat any message that it receives. In the example, the a SocketCAN device named `can0` is used.

Implementing a new hardware interface is simple. The developer only needs to write methods for starting and stopping communications, and for sending and receiving messages. With those in place, a new CAN device can be used with the library.

3.1 Message Queuing

While the CAN echo example works, a common issue in dealing with CAN is blocking IO. When calling `dev.recv()`, the program will be blocked until a message is received. Most CAN interfaces will block while waiting for frames.

To prevent our script from being blocked, the CanQueue object is used.

```
from canard.hw import socketcan

# create a SocketCAN device
dev = socketcan.SocketCanDev('can0')
# wrap the device in a CanQueue
cq = CanQueue(dev)

cq.start()

# receive a frame, timing out after 10 seconds
print cq.recv(timeout=10)

cq.stop()
```

Listing 3: A CAN Queue Example

This example sends waits 10 seconds for a frame to be received. If no frame is received, we can handle the timeout as an error.

A common pattern used in CAN systems is a request/response. One device sends a frame to request data or an action from a remote device. That device then responds accordingly.

```
from canard.hw import socketcan

# create a SocketCAN device
dev = socketcan.SocketCanDev('can0')
# wrap the device in a CanQueue
cq = CanQueue(dev)

cq.start()

# create a request frame
req = can.Frame(0x6A5)
req.dlc = 3
req.data = [0x10, 0xFF, 0xFF]

# send the request
cq.send(req)

# receive a response, timing out after 10 seconds
print cq.recv(filter=0x6A5, timeout=10)

cq.stop()
```

Listing 4: A Request/Response Example

Our request/response example makes use of the CanQueue's filter. Since we know the response frame will have identifier `0x6A5`, we ignore all other messages. This is very useful for implementing protocols such as CANOpen, OBD-II, and Unified Diagnostic Services.

4 Protocols

A variety of standard protocols are used for CAN communications. We will focus on the automotive industry, which conforms to standards published by the International Organization for Standardization (ISO).

In automotive systems, there is a class of communications known as "diagnostics." These communications are not active during normal operation, but can be used by manufacturers and service technicians to get device status, run tests, read memory, and update firmware.

CANard aims to implement the these protocols, so developers don't have to deal with the underlying CAN message structure. From a security prospective, this is helpful for writing fuzzers and exploits targeting automotive devices.

4.1 CAN-TP

CAN frames are limited to 8 bytes of data. To overcome this limitation, the ISO 15765-2 standard, often called ISO-TP is used. This standard provides a way of packaging longer data into multiple frames.

While the details of ISO-TP are beyond the scope of this paper, CANard can be used to automatically generate and parse ISO-TP. This functionality is provided by the `IsoTpProtocol` class.

4.2 OBD-II

The OBD-II standard is used for basic vehicle diagnostics. This standard uses a subset of CAN-TP. CANard provides an `ObdInterface` class which facilitates sending requests for OBD-II data and receiving the responses.

While OBD-II is useful for reading basic vehicle data, it does not provide much functionality beyond that. For more complex diagnostics operations, Unified Diagnostic Services is used.

4.3 Unified Diagnostic Services

Unified Diagnostic Services, or UDS, is used for manufacturer specific diagnostics. The UDS standard is published in ISO 14229-1 This protocol provides a wide variety of functionality for manufacturers and services technicians.

To access these services, a diagnostics tool is connected to the CAN bus. It then sends UDS requests to the various controllers on the bus. Each controller has a unique CAN identifier for receiving UDS requests and sending UDS responses. A table of the supported services is provided in table 1.

The variety of services available provides a large attack service on these controllers. For example, the ability to read and write arbitrary memory on a controller (services `0x23` and `0x3D`) in an active vehicle is a major concern. While these services should be limited, there are often implementation issues. Finding these issues results in controller exploits.

The CANard library provides a `UdsInterface` class that deals with packaging UDS messages, sending them, receiving a response, and parsing the response data. This makes it easier to write scripts to fuzz and exploit diagnostic systems.

5 Automation and Scripting

CANard allows developers to build utilities that deal with raw CAN data and standard protocols. Due to the hardware abstraction provided by the library, scripts can be used across various operating systems and with a multitude of CAN bus adapters.

A simple script using CANard is shown in listing 5. This script performs a Denial of Service attack by sending a message with identifier 0 at a high rate. In this example, a CANtact device is used.

Service ID	**Function**
0x10	DiagnosticSessionControl
0x11	ECUReset
0x27	Security Access
0x28	CommunicationControl
0x3E	TesterPresent
0x83	AccessTimingParameter
0x84	SecuredDataTransmission
0x85	ControlDTCSetting
0x86	ResponseOnEvent
0x87	LinkControl
0x22	ReadDataByIdentifier
0x23	ReadMemoryByAddress
0x24	ReadScalingDataByIdentifier
0x2A	ReadDataByPeriodicIdentifier
0x2C	DynamicallyDefineDataIdentifier
0x2E	WriteDataByIdentifier
0x3D	WriteMemoryByAddress
0x14	ClearDiagnosticInformation
0x19	ReadDTCInformation
0x2F	InputOutputControlByIdentifier
0x31	RoutineControl
0x34	RequestDownload
0x35	RequestUpload
0x36	TransferData
0x37	RequestTransferExit

Table 1: UDS Services

```
from canard import can
from canard.hw import cantact

# create and start device
dev = cantact.CantactDev('/dev/cu.usbmodem14514')
dev.start()

# create our payload frame
frame = can.Frame(id=0)
frame.dlc = 8

# spam!
while True:
    dev.send(frame)
```

Listing 5: Denial of Service

CANard's protocol features can be used to quickly build tools that talk over standard protocols. For example, listing 6 will attempt to discover UDS enabled devices by requesting a diagnostic session on a range of IDs.

```
import sys

from canard.proto.uds import UdsInterface
from canard.hw.cantact import CantactDev

d = CantactDev(sys.argv[1])
d.set_bitrate(500000)
d.start()

p = UdsInterface(d)

# DiagnosticSessionControl Discovery
for i in range(0x700, 0x800):
    # attempt to enter diagnostic session
    resp = p.uds_request(i, 0x10, [0x1], timeout=0.2)
    if resp != None:
        print("ECU response for ID 0x%X!" % i)
```

Listing 6: Controller Discovery

6 Conclusions

The CAN protocol is widely used in a number of industries, including automotive. The CANard library provides tools for rapid development of scripts that

interface with CAN bus systems.

Since CANard performs hardware abstraction for the CAN bus interface, scripts can be used on different platforms using a variety of CAN bus interfaces. The open source nature of this tool means that anyone can add support for a CAN interface.

CANard is able to communicate with controllers that use the Unified Diagnostic Services standard. This provides a variety of access to the controller, which provides an attack surface in cars.

References

[1] CANard project: `https://github.com/ericevenchick/canard`

[2] SocketCAN Documentation: `https://www.kernel.org/doc/Documentation/networking/can.txt`

[3] CANtact website: `http://cantact.io`