Protocol Reverse Engineering

netspooky // May 11 2023

Who is netspooky? Tell 255.255.255.255

Hardware vulnerability research at $company

Spent several years doing protocol reverse engineer professionally

Founder of the Binary Golf Grand Prix

Co-Founder of tmp.0ut

Talk Goals

Provide context for people who want to understand protocols and their implementations

Share tips n’ tricks for people who want to reverse engineer protocols

Talk Overview

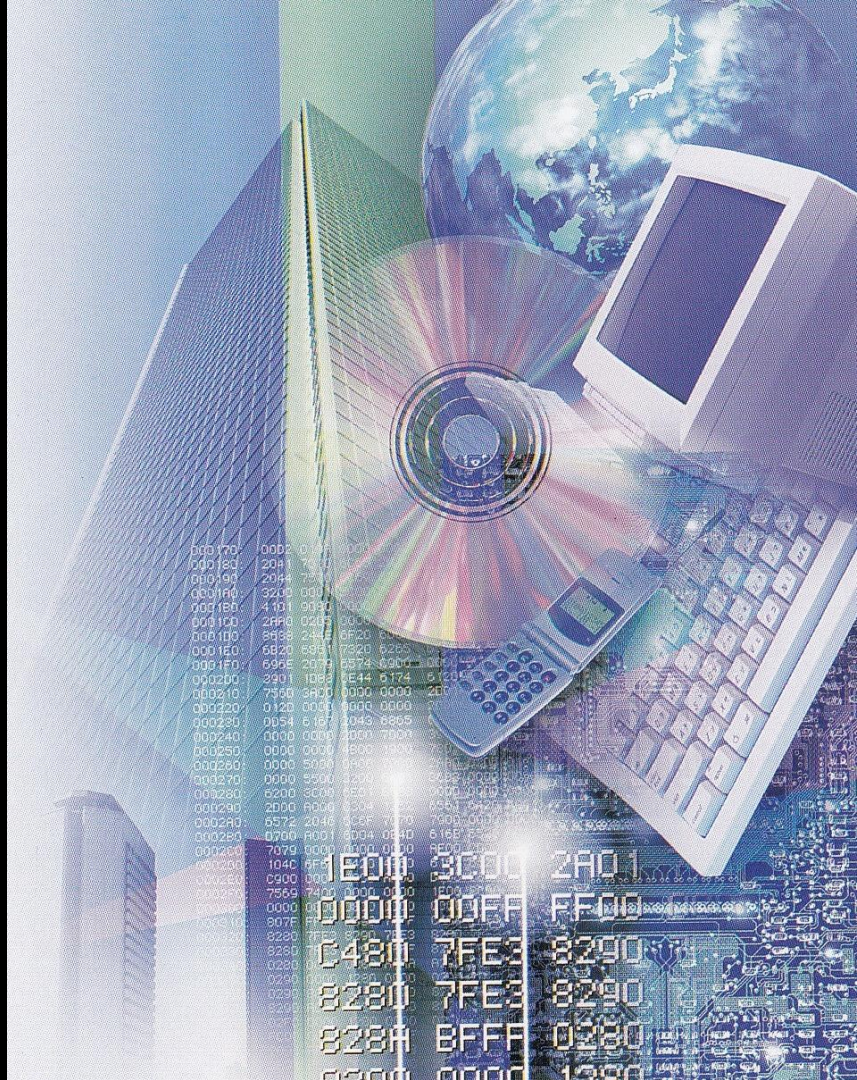
Protocol Fundamentals

Preparing to Reverse Engineer

Protocol Reverse Engineering Techniques

- Packet Analysis
- Software RE
- Hardware RE
- Specification Review

Documenting Your Findings



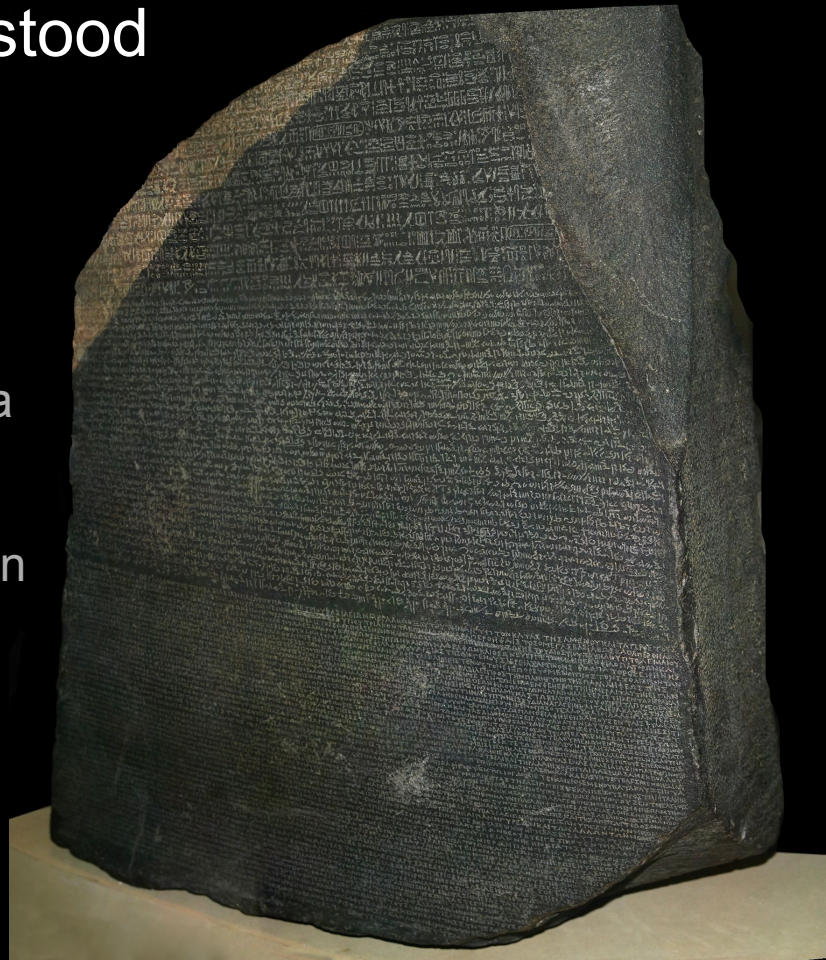
Protocol RE Fundamentals

Protocols Are Meant To Be Understood

It may not seem that way when looking at unknown data

Protocols define syntax and give meaning to arbitrary values and their order, very much like a language

They typically follow patterns that are common in other protocols



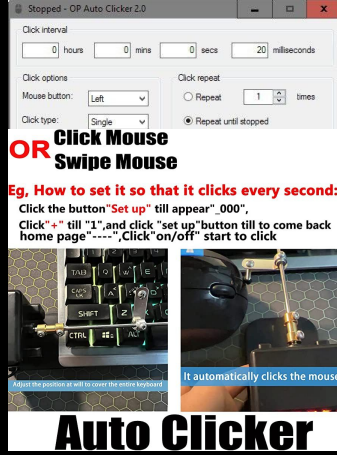
Dresden Codex - Mayan Writing System

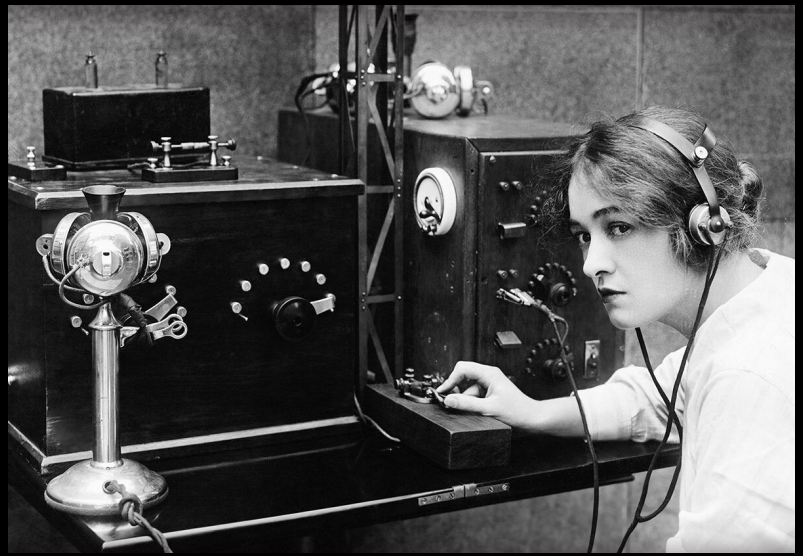
https://www.youtube.com/watch?v=YvLs3gDLCOI

Electronic Messages Are Just Pulses Of Energy

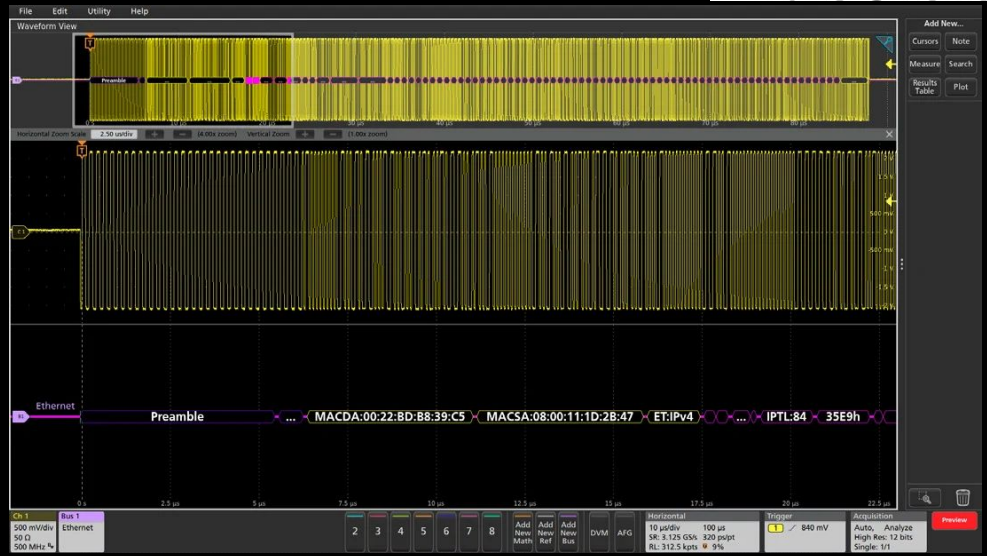
Imagine flipping a light switch or clicking a mouse really fast

Humans have gotten better at sending pulses of energy quickly





Morse Code Operator



An ethernet frame in an oscilloscope

Protocols Are A Set Of Agreements

For two devices to communicate, they must agree on several things

Logic Levels: What counts as a 1? What counts as a 0? Thresholds

Timing: How often data is sent

Framing: How to tell where one message begins & ends

Syntax: The structure of the data in the frame

Semantics: What the data means

Protocols Are A Set Of Agreements

As more devices wanted to talk to each other, some low-level agreements needed to be standardized.

Some worked extremely well for interoperability and ease of use, such as 8 bit bytes.

Many protocol components became de facto standards for backwards compatibility reasons, or ease of use.

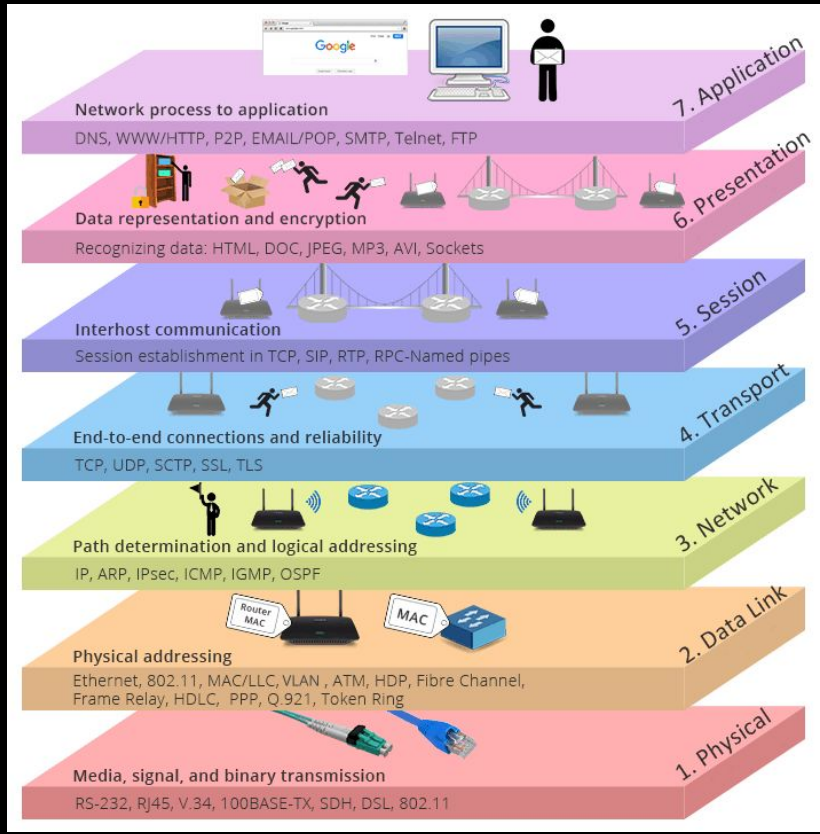
This can happen with entire protocols, resulting in *Protocol Ossification*

AT&T Engineer working on a 1ESS Switch

People Don’t Usually Reinvent The Wheel

Most things are done using common data structures and design paradigms

Once you learn some of the basic building blocks, you can use those as a guide



It's Still Real to Me Dammit!

Protocol Characteristics Can Be Explained By Context

Protocol constraints are defined by what they are built on

Protocols must account for the requirements of lower level protocols

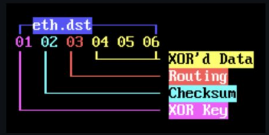
May avoid certain values with encoding - Example uuencode, base64

Other examples

- Pure binary protocols are often used when speed is a necessity
- Human readable data formats like JSON & XML are often used by application protocols
- Checksums are used when data integrity is required

Example of designing a protocol that fits in ethernet header

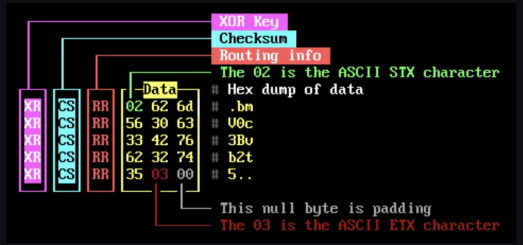
Our `eth.dst` field will now look like this:



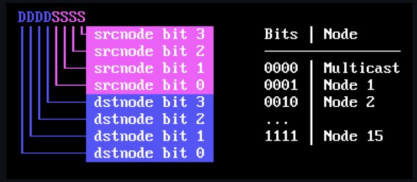
The data would be transformed like so:

- Plain Text: netspooky
- Base64 Encoded: bmV0c3Bvb2t5
- Afterwards, this would be XOR'd like in the previous scheme

When encoded, it will look like this (without XOR encoding)



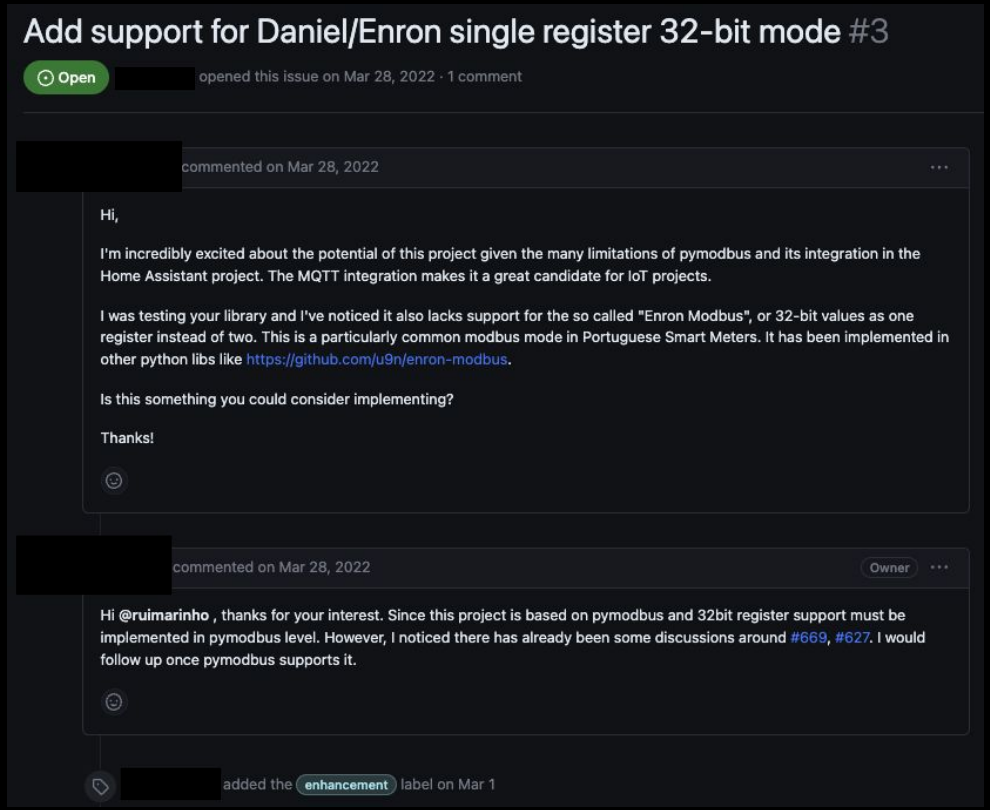
The RR routing info field looks like this:



Protocols Are A Product Of Their Era

- Different constraints are relevant at different times
- Many early programs and transport protocols were character oriented
- Used 7 bit ASCII with control characters, top bit would be special or ignored
- Protocols built on top of them would need to encode data and design around this
- Some have ASCII/Binary mode selectors like FTP
- *“If it ain’t broke don’t fix it”*

Protocols Can Outlive Their Authors



Add support for Daniel/Enron single register 32-bit mode #3

Open [redacted] opened this issue on Mar 28, 2022 · 1 comment

[redacted] commented on Mar 28, 2022

Hi,

I'm incredibly excited about the potential of this project given the many limitations of pymodbus and its integration in the Home Assistant project. The MQTT integration makes it a great candidate for IoT projects.

I was testing your library and I've noticed it also lacks support for the so called "Enron Modbus", or 32-bit values as one register instead of two. This is a particularly common modbus mode in Portuguese Smart Meters. It has been implemented in other python libs like https://github.com/u9n/enron-modbus.

Is this something you could consider implementing?

Thanks!

[redacted] commented on Mar 28, 2022 Owner

Hi **@ruimarinho** , thanks for your interest. Since this project is based on pymodbus and 32bit register support must be implemented in pymodbus level. However, I noticed there has already been some discussions around #669, #627. I would follow up once pymodbus supports it.

[redacted] added the enhancement label on Mar 1

Preparing To Reverse Engineer

Why Are You Reverse Engineering?

Compatibility

Detection Engineering and Network Monitoring

Malware Research

Vulnerability Research

The protocol may be "lost"

DRM / Server Check Bypass

Building your own implementation

Curiosity

What kinds of protocols do people RE?

Extensions of a known protocol - Vendor specific, custom commands, new structures etc.

Undocumented/Poorly Documented Protocols - Quite common

Proprietary Protocols - Also quite common

Specific Implementations - May have own quirks depending on the architecture, history, or other requirements

Protocol RE Techniques

Protocol RE Techniques

Packet / PCAP Analysis

Software RE

Hardware RE

Specification Review

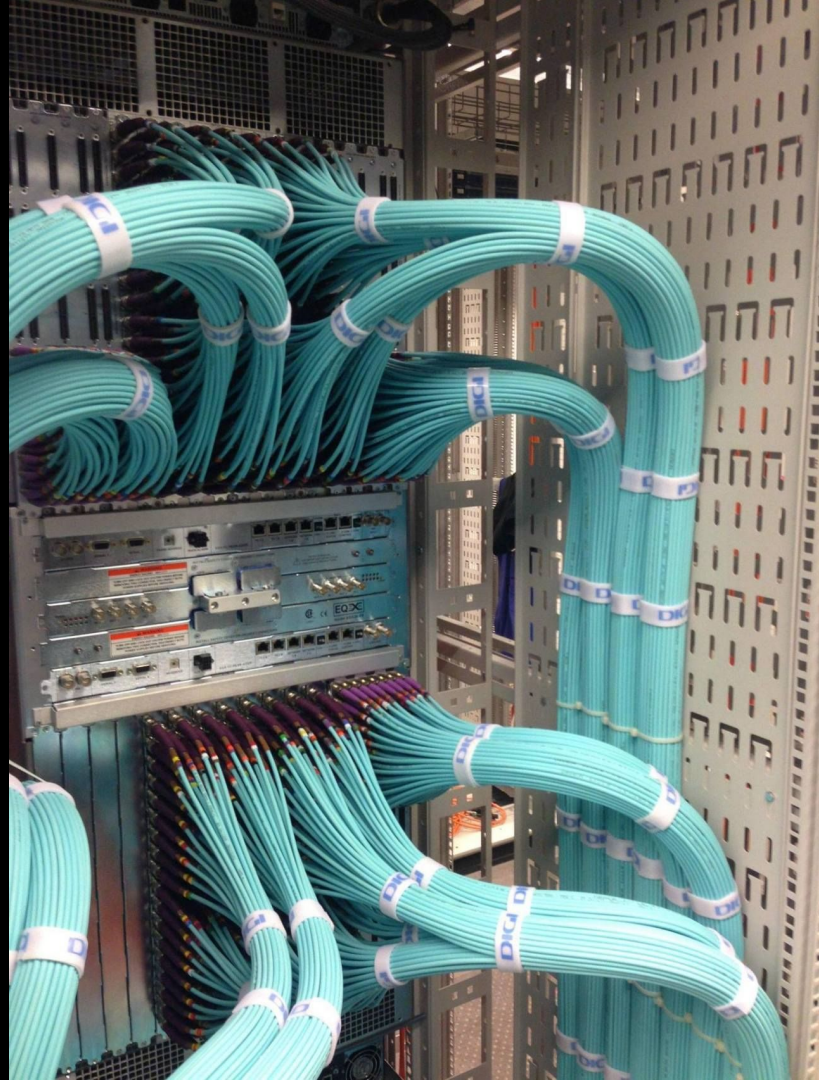
Packet Analysis

Packet Analysis

Often a good way to really understand a protocol

Can be done with a live capture or using a packet capture file (pcap)

Many of these same techniques can be done on any binary communication (and also file formats!)



SPOOKY FACT #1: PCAPs Or You Didn’t Capture It

People often say "PCAPs or it didn't happen", but there are *problems* with this

PCAPs are limited by your device's ability to capture packets and record them at the rate at which they appear on the wire

There are also limitations based on your software's parsing ability

Dissector & parser bugs exist

Trust, but verify

PCAPs Are Just A Snapshot

An inherent limitation of a single PCAP file is that it's only a snapshot.

PCAPs can’t show us what may have happened before or after

Filters can remove important details

Some protocols use two or more ports for sending/receiving data (ex. FTP)

Messages can be truncated too

****snap**Len**…this one’s going to be truncated to 68 bytes

```
$ xxd telnet-cooked.pcap
00000000: d4c3 b2a1 0200 0400 0000 0000 0000 0000  ................
00000010: ea05 0000 0100 0000 968f 4038 83e8 0500  ..........@8....
00000020: 4a00 0000 4a00 0000 0000 c09f a097 00a0  J...J...........
00000030: cc3b bffa 0800 4510 003c 463c 4000 4006  .;....E..<F<@.@.
00000040: 731c c0a8 0002 c0a8 0001 060e 0017 99c5  s...............
00000050: a0ec 0000 0000 a002 7d78 e0a3 0000 0204  ........}x......
00000060: 05b4 0402 080a 009c 2724 0000 0000 0103  ........'$......
00000070: 0300 968f 4038 60f2 0500 4a00 0000 4a00  ....@8`...J...J.
00000080: 0000 00a0 cc3b bffa 0000 c09f a097 0800  .....;..........
00000090: 4500 003c 51e3 0000 4006 a785 c0a8 0001  E..<Q...@.......
000000a0: c0a8 0002 0017 060e 17f1 633d 99c5 a0ed  ..........c=....
000000b0: a012 43e0 fbb7 0000 0204 05a8 0103 0300  ..C.............
000000c0: 0101 080a 0025 a62c 009c 2724 968f 4038  .....%.,..'$..@8
000000d0: 8ff2 0500 4200 0000 4200 0000 0000 c09f  ....B...B.......
000000e0: a097 00a0 cc3b bffa 0800 4510 0034 463d  .....;....E..4F=
000000f0: 4000 4006 7323 c0a8 0002 c0a8 0001 060e  @.@.s#..........
00000100: 0017 99c5 a0ed 17f1 633e 8010 7d78 edd7  ........c>..}x..
00000110: 0000 0101 080a 009c 2724 0025 a62c 968f  ........'$.%.,..
00000120: 4038 c3f8 0500 5d00 0000 5d00 0000 0000  @8....]...].....
00000130: c09f a097 00a0 cc3b bffa 0800 4510 004f  .......;....E..O
00000140: 463e 4000 4006 7307 c0a8 0002 c0a8 0001  F>@.@.s.........
00000150: 060e 0017 99c5 a0ed 17f1 633e 8018 7d78  ..........c>..}x
00000160: 6e67 0000 0101 080a 009c 2724 0025 a62c  ng........'$.%.,
00000170: fffd 03ff fb18 fffb 1fff fb20 fffb 21ff  ........... ..!.
00000180: fb22 fffb 27ff fd05 fffb 2396 8f40 38c2  ."..'.....#..@8.
00000190: 3308 0045 0000 0045 0000 0000 a0cc 3bbf  3..E...E......;.
000001a0: fa00 00c0 9fa0 9708 0045 1000 373d 8e00  .........E..7=..
```

Global Header

```
4 magic_number   magic number - byte order - a1b2c3d4 (BE) or d4c3b2a1 (LE)
2 version_major  major version number
2 version_minor  minor version number
4 thiszone       GMT to local correction
4 sigfigs        accuracy of timestamps
4 snaplen        max length of captured packets, in octets
4 network        data link type
```

Packet Header

```
4 ts_sec     timestamp seconds
4 ts_usec    timestamp microseconds
4 incl_len   number of octets of packet saved in file
4 orig_len   actual length of packet
```

```
$ xxd telnet-cooked-ng.pcapng
00000000: 0a0d 0d0a 1c00 0000 4d3c 2b1a 0100 0000  ........M<+..... Section Hdr. Blk
00000010: ffff ffff ffff ffff 1c00 0000            ............
                                   0100 0000                    .... Interface Desc. Blk
00000020: 1400 0000 0100 0000 ea05 0000 1400 0000  ................

00000030: 0600 0000 6c00 0000 0000 0000 575a 0300  ....l.......WZ.. Packet 1
00000040: 037a f81e 4a00 0000 4a00 0000 0000 c09f  .z..J...J.......
00000050: a097 00a0 cc3b bffa 0800 4510 003c 463c  .....;....E..<F<
00000060: 4000 4006 731c c0a8 0002 c0a8 0001 060e  @.@.s...........
00000070: 0017 99c5 a0ec 0000 0000 a002 7d78 e0a3  ............}x..
00000080: 0000 0204 05b4 0402 080a 009c 2724 0000  ............'$..
00000090: 0000 0103 0300 0000 6c00 0000            ........l...
                                   0600 0000                    .... Packet 2
000000a0: 6c00 0000 0000 0000 575a 0300 e083 f81e  l.......WZ......
000000b0: 4a00 0000 4a00 0000 00a0 cc3b bffa 0000  J...J.....;.....
000000c0: c09f a097 0800 4500 003c 51e3 0000 4006  ......E..<Q...@.
000000d0: a785 c0a8 0001 c0a8 0002 0017 060e 17f1  ................
000000e0: 633d 99c5 a0ed a012 43e0 fbb7 0000 0204  c=......C.......
000000f0: 05a8 0103 0300 0101 080a 0025 a62c 009c  ...........%.,..
00000100: 2724 0000 6c00 0000                      '$..l...
                                   0600 0000 6400 0000              ....d...
00000110: 0000 0000 575a 0300 0f84 f81e 4200 0000  ....WZ......B...
00000120: 4200 0000 0000 c09f a097 00a0 cc3b bffa  B............;..
00000130: 0800 4510 0034 463d 4000 4006 7323 c0a8  ..E..4F=@.@.s#..
00000140: 0002 c0a8 0001 060e 0017 99c5 a0ed 17f1  ................
00000150: 633e 8010 7d78 edd7 0000 0101 080a 009c  c>..}x..........
00000160: 2724 0025 a62c 0000 6400 0000 0600 0000  '$.%.,..d.......
00000170: 8000 0000 0000 0000 575a 0300 438a f81e  ........WZ..C...
00000180: 5d00 0000 5d00 0000 0000 c09f a097 00a0  ]...]...........
00000190: cc3b bffa 0800 4510 004f 463e 4000 4006  .;....E..OF>@.@.
000001a0: 7307 c0a8 0002 c0a8 0001 060e 0017 99c5  s...............
```

[Section Header Block]

```
4 Block Type
4 Block Total Length
4 Byte Order Magic
2 Version Major
2 Version Minor
8 Section Length
n Options (Variable Len)
4 Block Total Length
```

[Interface Description Block]

```
4 Block Type
4 Block Total Length
2 Link Type
2 Reserved
4 Snap Len
n Options (Variable Len)
4 Block Total Length
```

[Enhanced Packet Block]

```
4 Block Type
4 Block Total Length
4 Interface ID
4 Time Stamp High
4 Time Stamp Low
4 Captured Length
4 Original Length
n Payload
n Options (Variable Len)
4 Block Total Length
```

Getting Packet Captures

There are a lot of places where people put PCAPs.

Uploaded to forums, analysis sites, Github, or they fell off a truck

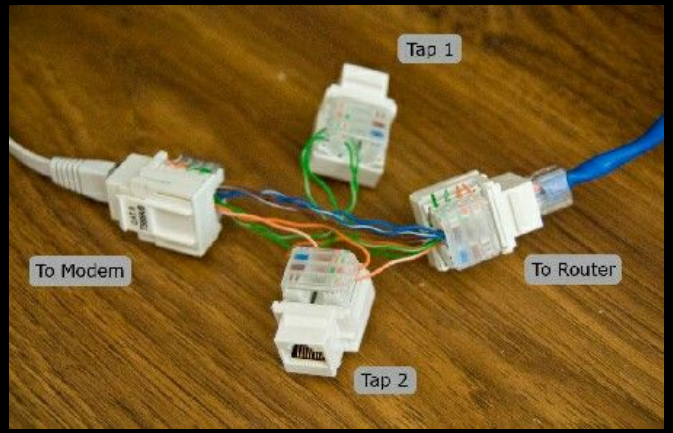
If you have hardware or software that implements the protocol, make your own

Wireshark and tcpdump are two standard tools

Many hardware devices support the PCAP format

Raw data is better than no data

Capturing Your Own



Wireshark/tcpdump/tshark/other tools

LAN Taps

WiFi Capture

BLE Capture - Using built in card, dongle, or features such as HCI Snoop Log

USB Capture

Radio capture software/hardware

You can fake PCAPs by directing crafted data over data channels

Can also edit pcaps

Try to collect as many packets as possible!

Analysis First Steps

You are examining the Wire Image

Identify all known traffic

Identify unknown messages

Identify encapsulation and transport mechanisms used

How you can isolate unknown traffic from known traffic?

- Ports
- Addresses
- Other identifiers

Start characterizing unknown traffic

SPOOKY FACT #2: Everything Is Made Up And The Ports Don’t Matter

Port numbers are just pointers to where to send data

While there are assigned port numbers, nothing stops a protocol from running over arbitrary ports

You don't have to memorize every port number

Your computer probably has a list of them somewhere

https://man7.org/linux/man-pages/man5/protocols.5.html

Different OSes consider different sets of ports to be ephemeral

Characterizing Unknown Traffic

Frequency Analysis

- How often are messages sent?
- How often are the same messages sent?
- What values are common in the packet?

Differential Analysis

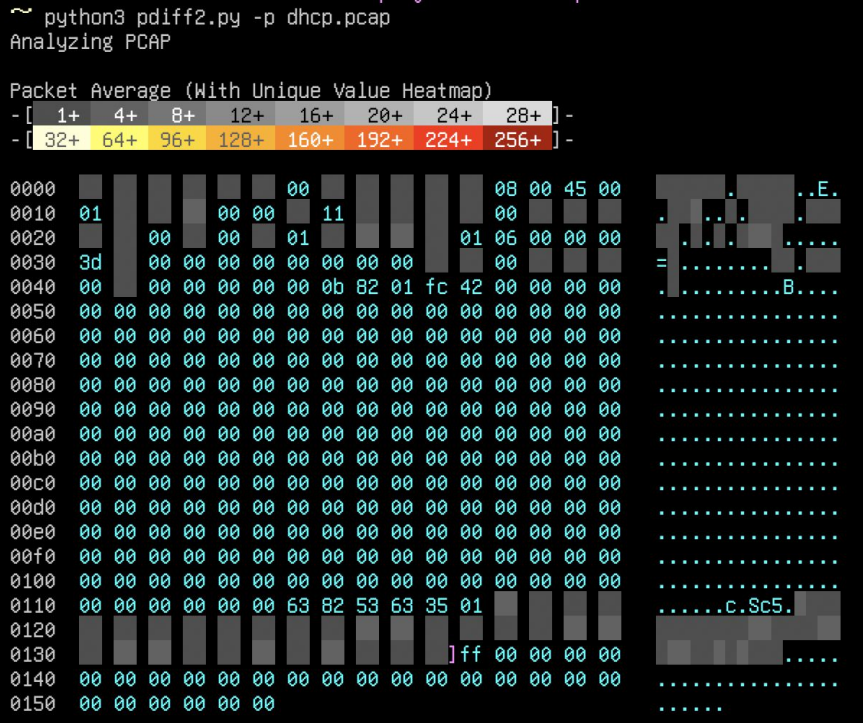
- How often do packets differ?
- Where do they differ?
- How do they differ?

Timing Analysis

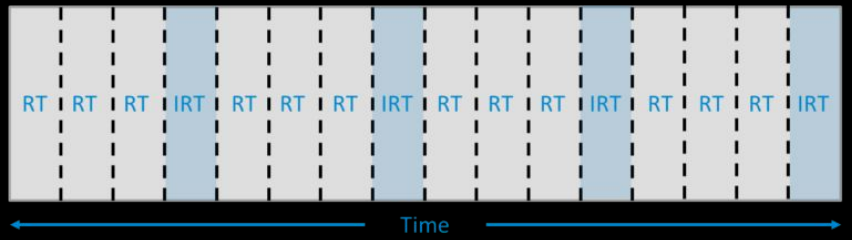
- How often are packets sent and by whom?
- What is the time delta between packets?

How long are packets? How does the length correlate with message content?

Sketch out an average packet, identify data structures and message types



Averaged frame - DHCP traffic



Isochronous Real-Time Profinet - Time Slicing

Common Header Components

Things that commonly appear in protocol headers

Magic Values - Usually at the start of the packet or section

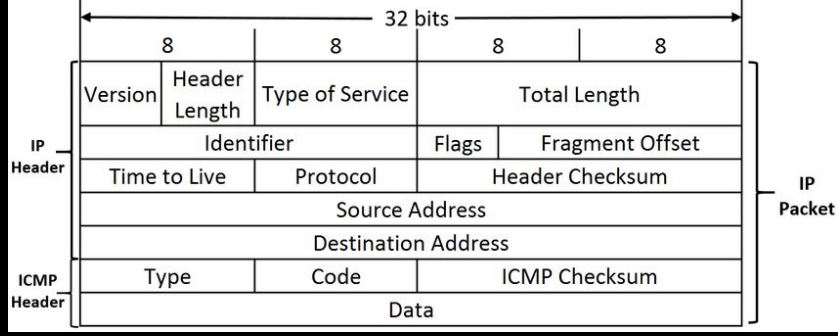
Sequence numbers - These increment over time

Transaction IDs - Value repeated by both the sender and receiver

Node IDs/Addresses - Are there any values associated with a specific sender?

Length Fields - Examine the packet's length in hex, do any values in the header look similar?

Commands and Message Types - These will usually appear at specific locations, related to overall packet content



IP Header with ICMP message

Common Protocol Components

Byte Order - Protocol features (such as values in the header) are typically big endian (network order) while packet data may be little endian

Time Stamps - https://github.com/netspooky/notes/blob/main/re/timestamps.md

Bitfields and Flags - Look for consistent 01 02 04 08 10 20 40 80 etc..

IP Addresses / MAC Addresses - Compare to previous protocol layers

Checksums - Often appear close to the data, look for packets that change slightly

Padding and Data Alignment - Can help determine the boundaries of data, look for 00s and FFs

Auth Data - Could be login, password, or token. Related to a session or node

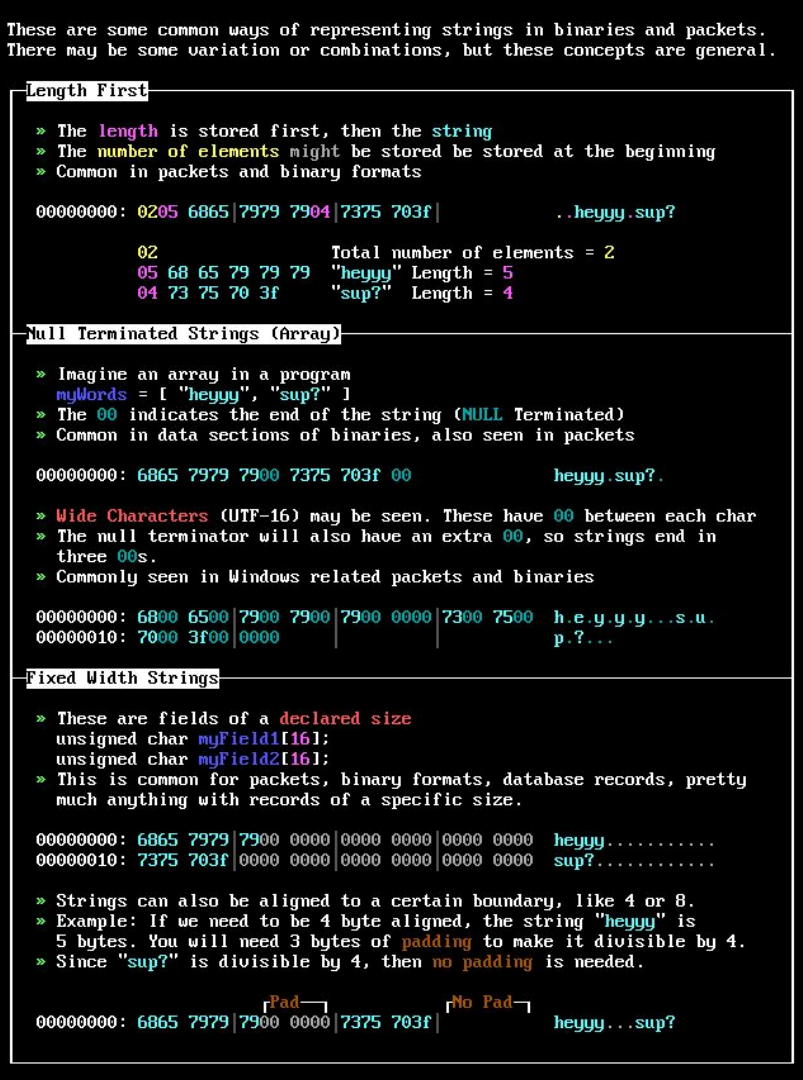
Strings

String representations are often a strong indicator of how other data is structured

Readable strings may also indicate what the protocol message is doing

Can be helpful to correlate with other RE and OSINT tasks

https://github.com/netspooky/notes/blob/main/re/string_representation.md

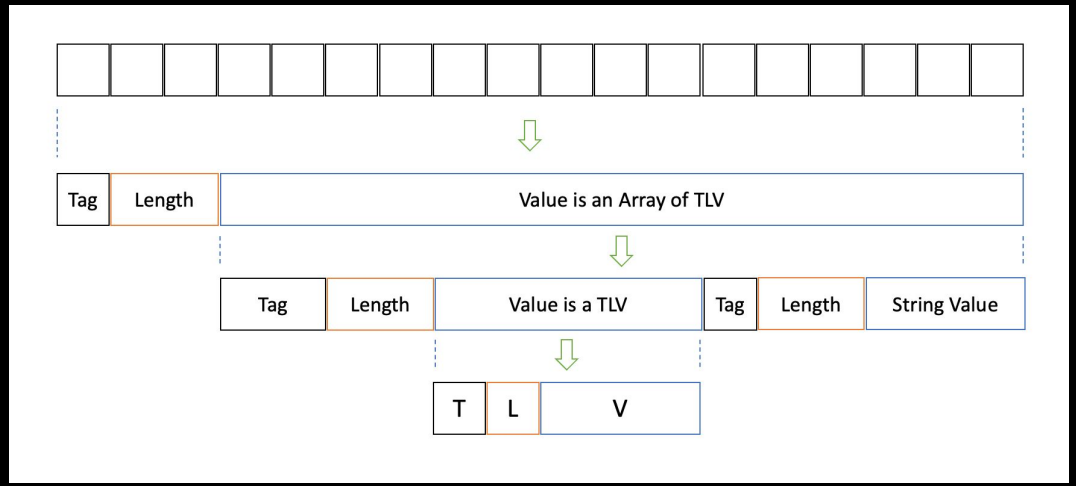


Common Protocol Components: TLV

TLV - Tag (or Type) Length Value

Can be nested

Can have more than one Tag, Length or Value



https://github.com/netspooky/xx/blob/main/examples/tls-clienthello.xx

```
~ cat tls-clienthello.xx
┌ TLSv1.3 Client Hello - 20220927 - @netspooky_
│ PCAP: tls13-firefox.nighty.51.0a-to-tls13.crypto.mozilla.org.pcapng (Frame 4)
│ From: https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=12779
│ Direct Link: https://bugs.wireshark.org/bugzilla/attachment.cgi?id=14839
│ Description: This is a TLS Version 1.3 Client Hello message.
│              It's what your computer sends when they want to use TLS.
│              I used wireshark to dissect the packet and then used the
│              dissections to guide the description.
│              NOTE: This is just the application layer,
│                    Ethernet/IP/TCP are not in here!
│ Generates: tls-clienthello.843c9527.bin
└ Transport Layer Security v1.3 Record Layer: Handshake Protocol: Client Hello
16    - Content Type: Handshake (22)
03 01 - Version: TLS 1.0 (0x0301)
02 27 - Length: 551
- Handshake Protocol: Client Hello
   01          - Handshake Type: Client Hello (1)
   00 02 23    - Length: 547
   03 04       - Version: TLS 1.3 (0x0304)
   f4 0e 41 64 - Random
   34 6d 03 4f
   17 dd 6b 17
   5b 63 e8 a3
   b9 1f f2 88
   60 75 81 fb
   5d 37 fd 50
   16 a4 67 53
   00          - Session ID Length: 0
   00 1e       - Cipher Suites Length: 30
   - Cipher Suites (15 suites)
      c0 2b - Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
      c0 2f - Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
      cc a9 - Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
      cc a8 - Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
      c0 2c - Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
      c0 30 - Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
      c0 0a - Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
      c0 09 - Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
      c0 13 - Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
      c0 14 - Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
      00 33 - Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA
      00 39 - Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA
      00 2f - Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA
      00 35 - Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA
      00 0a - Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA
   01 - Compression Methods Length: 1
   - Compression Methods (1 method)
      00 - Compression Method: null (0)
   01 dc - Extensions Length: 476
   - Extension: server_name (len=29)
      00 00 - Type: server_name (0)
      00 1d - Length: 29
      - Server Name Indication extension
         00 1b - Server Name list length: 27
         00    - Server Name Type: host_name (0)
         00 18 - Server Name length: 24
         "tls13.crypto.mozilla.org" - Server Name
```

Helpful Enumeration Tools

pdiff2 - A tool I made to do basic enumeration tasks. Works on pcaps and text files

pdiff-wasm - https://remyhax.xyz/tools/pdiffwasm/

NetworkMiner - https://www.netresec.com/?page=NetworkMiner

ngrep - https://github.com/jpr5/ngrep

binwalk - https://github.com/ReFirmLabs/binwalk

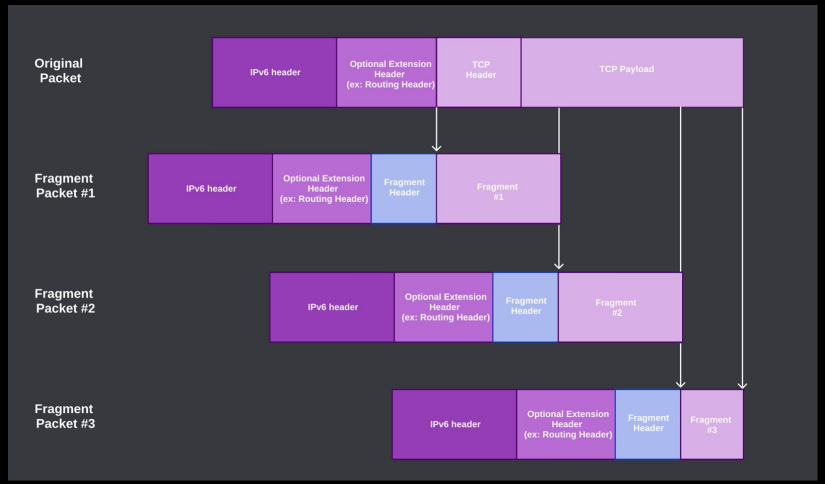
Wireshark has a lot of traffic analysis tools built in too!

Protocol Features To Know: Fragmentation and Pipelining

Single messages can be split up into fragments

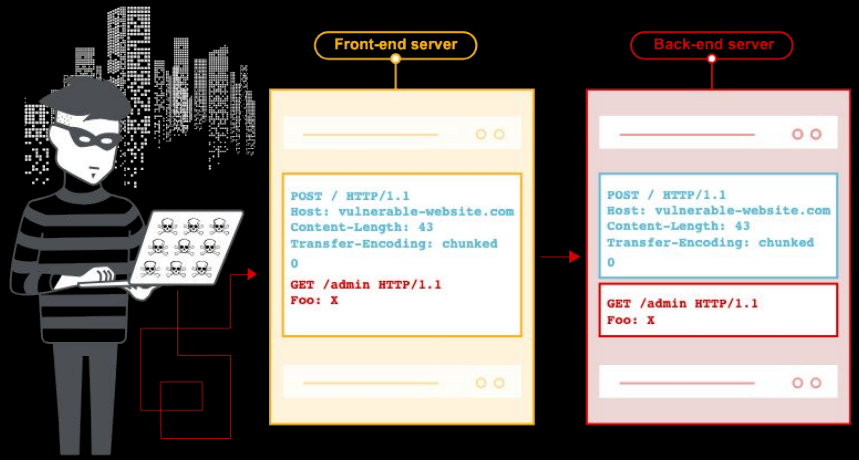
Multiple distinct protocol messages can be sent in one packet - pipelining

Multiple pipelined messages can be fragmented across multiple packets as well



IPv6 Fragmentation Diagram

https://securityintelligence.com/posts/dissecting-exploiting-tcp-ip-rce-vulnerability-evilesp/



HTTP Request Smuggling - Pipelining Example

https://portswigger.net/web-security/request-smuggling

Protocol Features To Know: Encryption

Many protocols use encryption

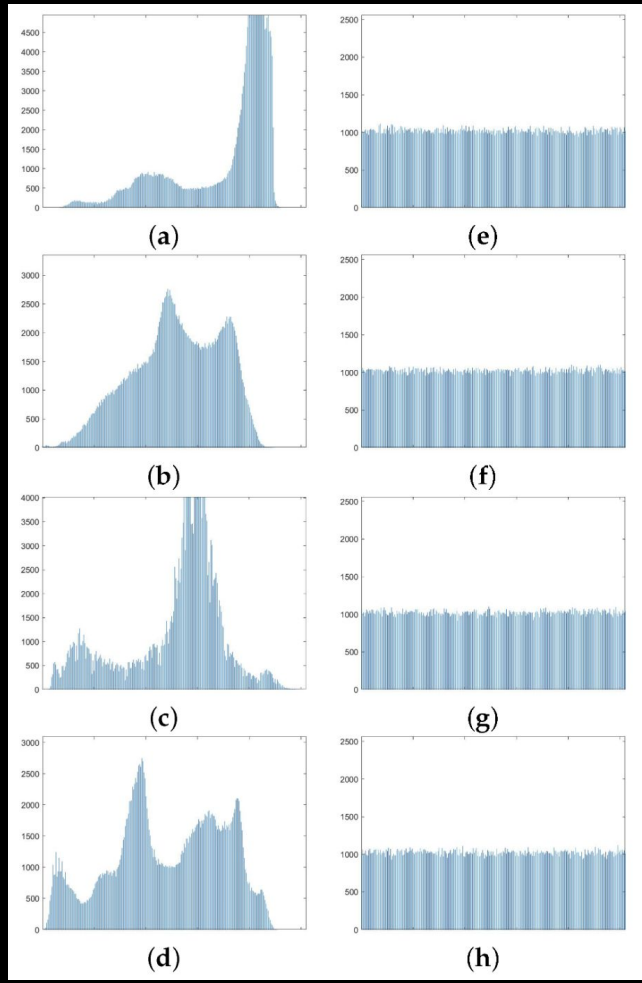
Look for areas containing high entropy data

Examine traffic for key negotiation

TLS is common for modern application protocols

Some protocols may roll their own

Compressed data and encoded data may look like encrypted data! Can confirm with Software RE



Left: Unencrypted Data // Right: Encrypted Data

Software RE

Put Yourself In The Developer’s Shoes

Think "What were they trying to do with this protocol?"

If looking for bugs, “What would be difficult to handle? Where would I mess it up?”

Consider any constraints they might have had

Principal engineer fixing a classic bit shifting error

OSINT

Are there any open source implementations?

Has any part of the source code been shared?

Does the vendor document the API?

Has anyone else reverse engineered this software?

Is there anywhere that users of the software chat?

Search for strings and other protocol characteristics you've found

Network/Communication APIs

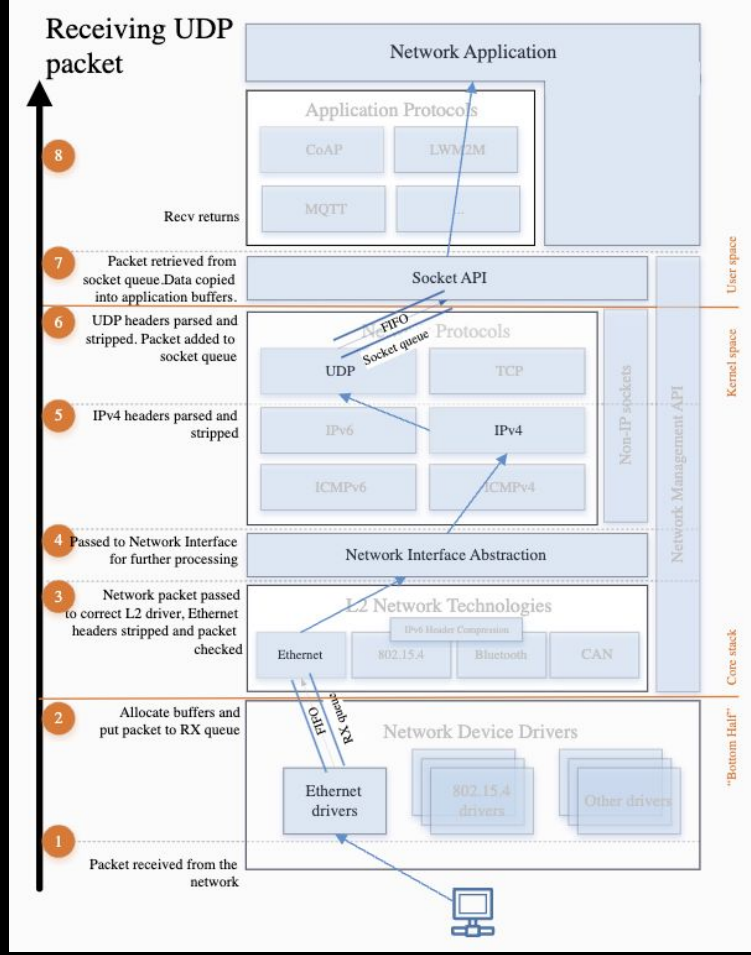
Many operating systems use well-defined APIs for communication

These abstract physical connections to a file descriptor or handle

RTOS or other firmware also use well-defined APIs, may wrap existing libraries

For non-network protocols (such as USB), platform specific functions for sending/receiving data are also defined.

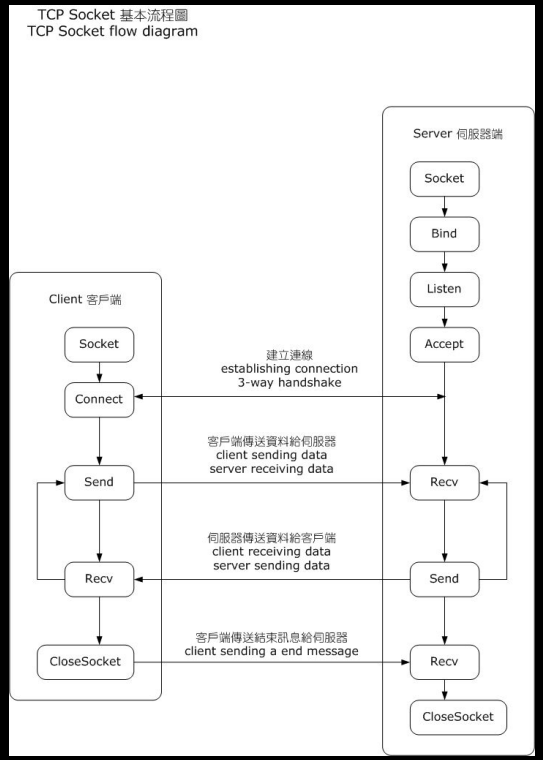
At the end of the day it's all just I/O



Example network stack with data flow

https://docs.zephyrproject.org/1.14.1/guides/networking/net-stack-architecture.html

Berkeley Sockets



TCP socket flow

https://en.wikipedia.org/wiki/Berkeley_sockets

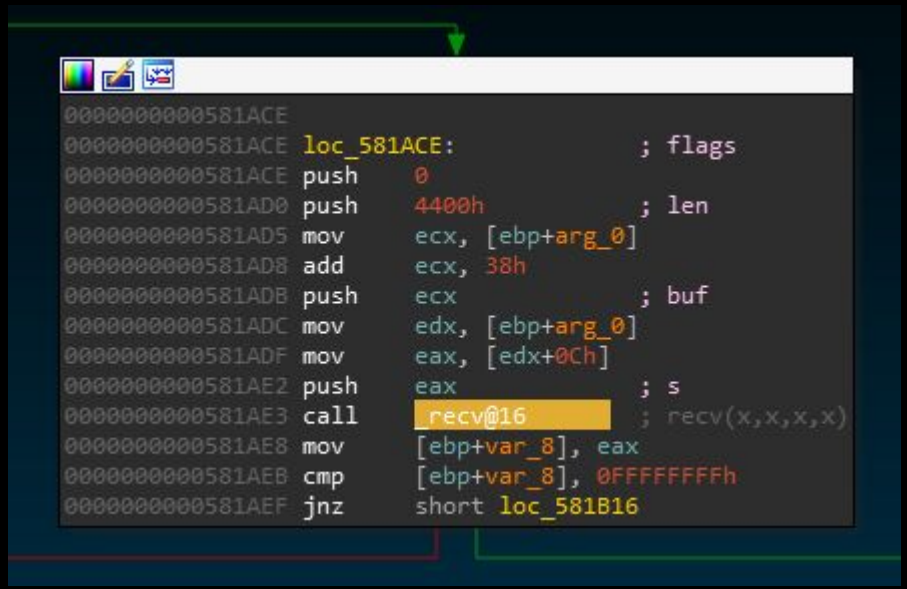
Very common socket API

Written in C, wrappers in higher level languages

Provides interface to set up a socket, handle connections, and send/receive data.

Will usually be imported from a library unless it's using syscalls directly (like on Linux)

Reverse Engineering: Static



Reference to the recv function in a program open in Ida

Once you enumerate network APIs, work backwards to locate where the protocol is parsed. Cross references to send/recv type functions etc.

The usual static reverse engineering techniques can be done, just focusing on message parsing and building.

Strings and error messages are a really solid way of finding where specific things happen. `rabin2 -zzz`

Reverse Engineering: Dynamic

Use tools like `netstat -ano` on Windows or `ss -tapu` on Linux to learn what processes are listening on what ports

Capture packets using Wireshark, monitor what is sent

Attach to the running process with a debugger, or run a new process

Set breakpoints on the functions you found during Static Analysis

Send packets and follow the data through the program's logic

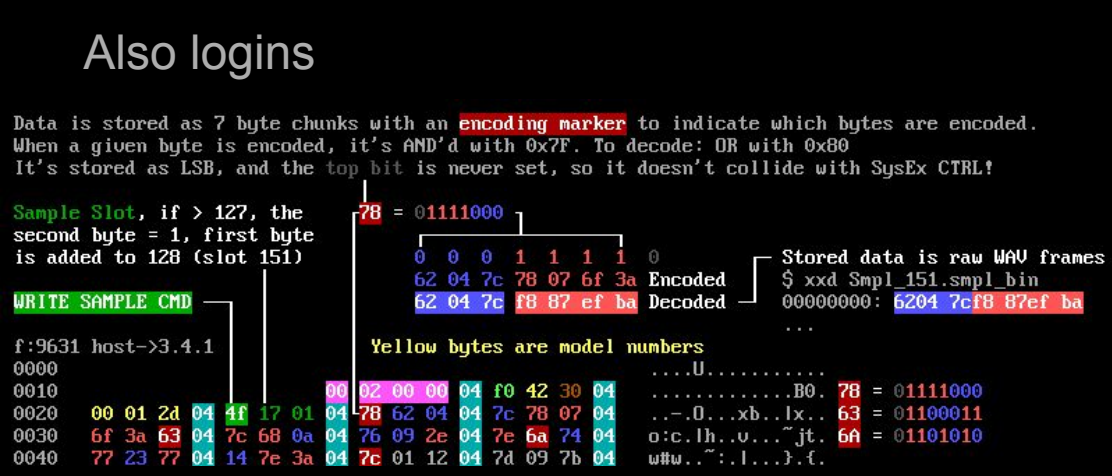
Set access breakpoints (read/write) on memory areas with packet data

Reverse Engineering: Dynamic

Correlate specific actions within the software to what you see on the wire

File transfers can be helpful because a lot of messages will need to be sent, and you can compare to known values (aka what the file looks like on the disk)

Also logins



Notes from RE of a custom MIDI SysEx extension

```
xxd mono_amen_snare_32khz_exported.wav
00000000: 5249 4646 4e29 0000 5741 5645 4a55 4e4b  RIFFN)..WAVEJUNK
00000010: 3400 0000 0000 0000 0000 0000 0000 0000  4...............
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 666d 7420 1000 0000  ........fmt ....
00000050: 0100 0100 127a 0000 24f4 0000 0200 1000  .....z..$.......
00000060: 736d 706c 3c00 0000 0000 0000 0000 0000  smpl<...........
00000070: 0000 0000 3c00 0000 0000 0000 0000 0000  ....<...........
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000a0: 0000 0000 6461 7461 aa28 0000 6204 7cf8  ....data.(..b.|.
000000b0: 87ef bafc e80a 7609 aefe 74f7 23f7 14fe  ......v...t.#...
000000c0: ba01 12fd 89fb a1fb 59f9 ccfa 6803 5108  ........Y...h.Q.
```

In the fmt header, the sample rate is little endian
In the data header, the data length is little endian
and the first frame is two bytes

This is saved in EEPROM

```
xxd Smpl_151.smpl_bin
00000000: 6204 7cf8 87ef bafc e80a 7609 aefe 74f7  b.|.......v...t.
00000010: 23f7 14fe ba01 12fd 89fb a1fb 59f9 ccfa  #...........Y...
00000020: 6803 5108 b8fc d0f4 d9fe 7706 92fa 93ed  h.Q.......w.....
```

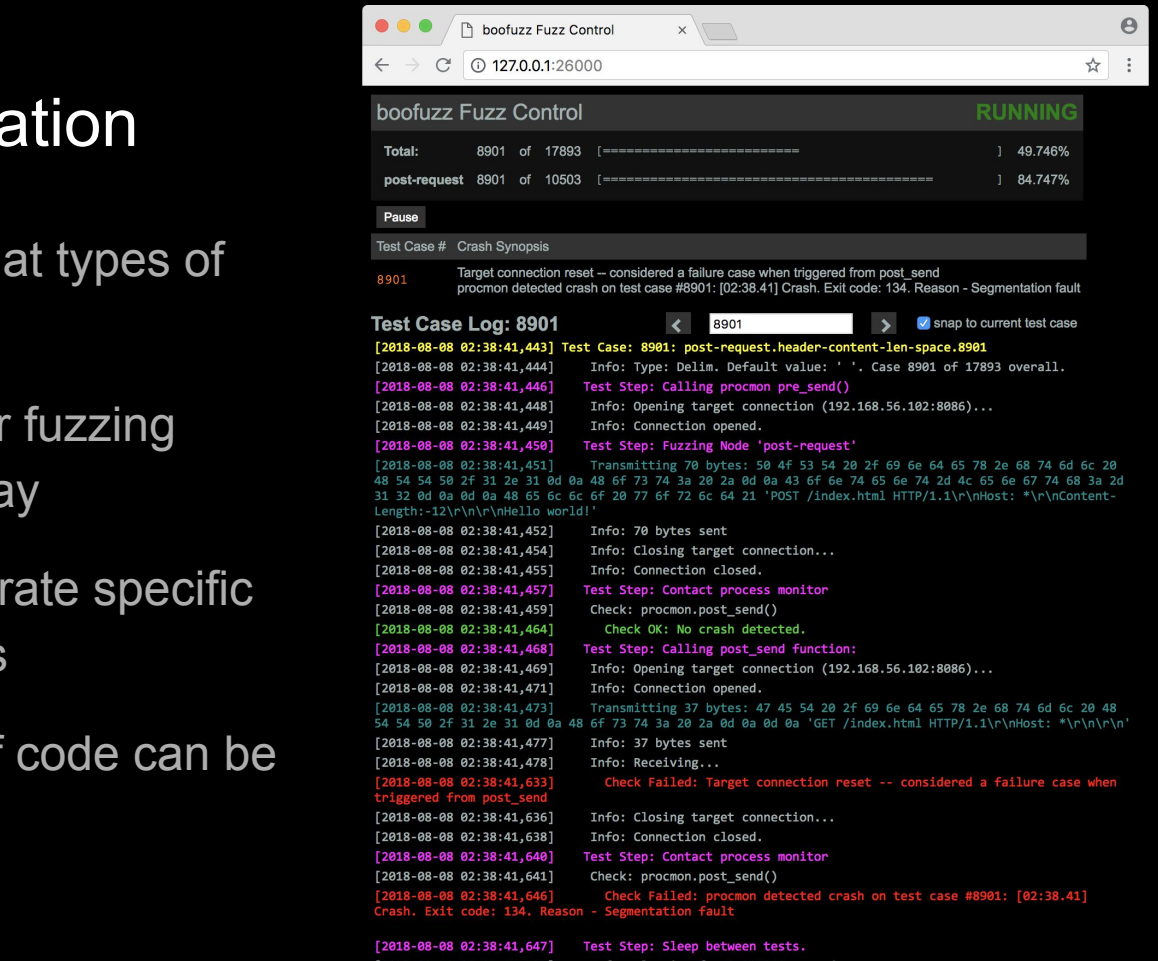
Fuzzing and Enumeration

A good way to understand what types of messages are supported

boofuzz is my personal fav for fuzzing protocols in a deterministic way

Can write your own to enumerate specific components such as opcodes

Emulating specific sections of code can be helpful too



boofuzz control panel

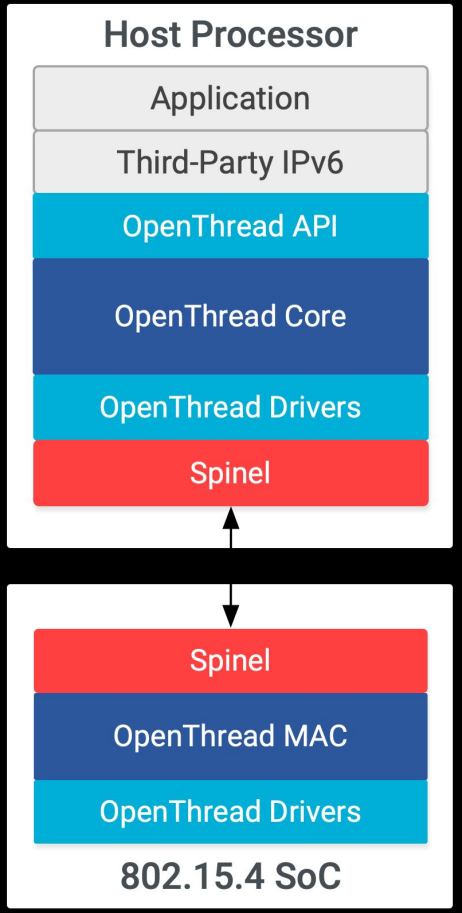
Drivers

Sometimes the software doesn’t implement the full protocol, or only part of it.

It can be offloaded to a device using a driver

Drivers usually interface with the kernel directly to access hardware devices

It can be hard to figure out exactly where the messages are constructed



Co-processor interface for Thread

https://openthread.io/platforms/co-processor

Hardware RE

Enumerating Hardware

OSINT

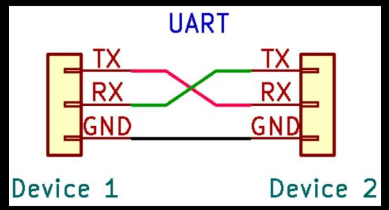
FCC IDs

Antennas

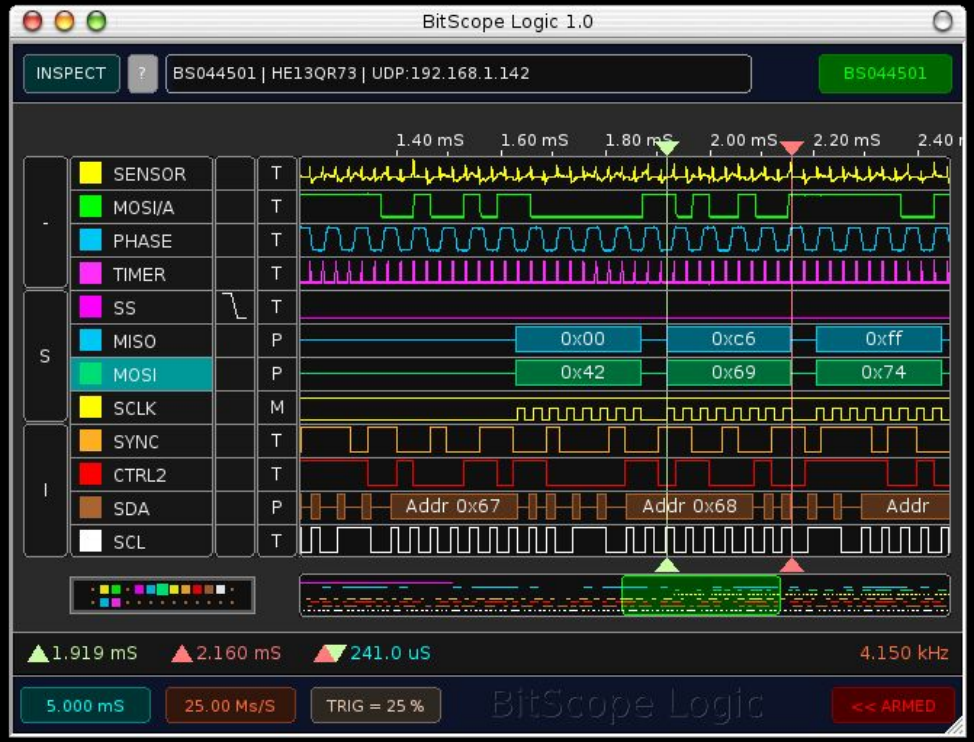
Chips on board - Datasheets

Debug Interfaces

- UART, JTAG, SPI, I2C, USB, SWD



Serial Protocols



Logic analyzer software examining SPI, I2C, and other protocols

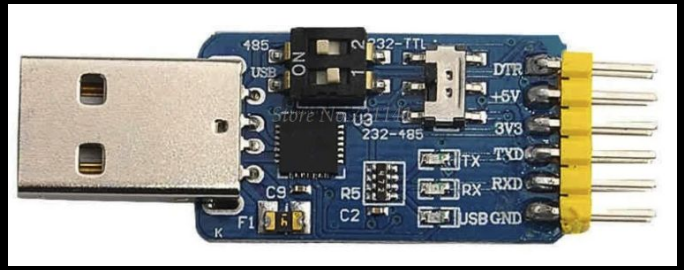
If you can't find any information on what protocols are supported, you can use a logic analyzer or an oscilloscope to analyze serial protocols and inter-chip protocols

Logic analyzer software can decode common protocols as well as custom

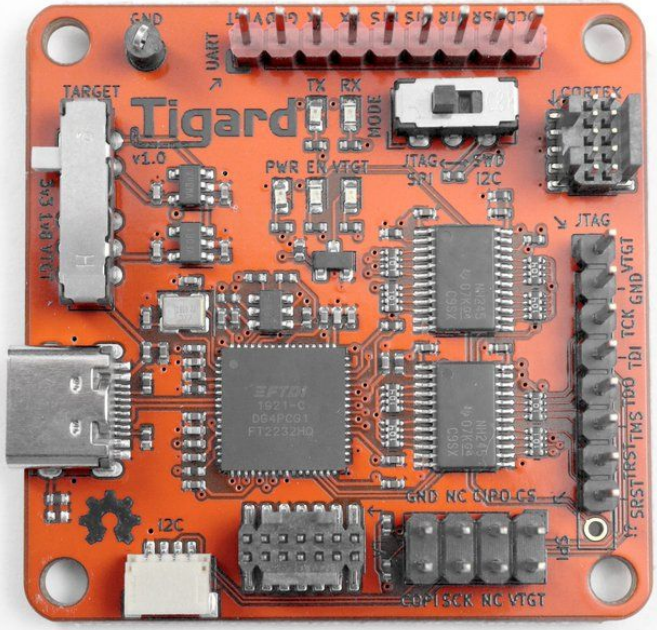
Interfacing With Serial Protocols

You can use Serial/USB interfaces or other tools to speak directly to the hardware

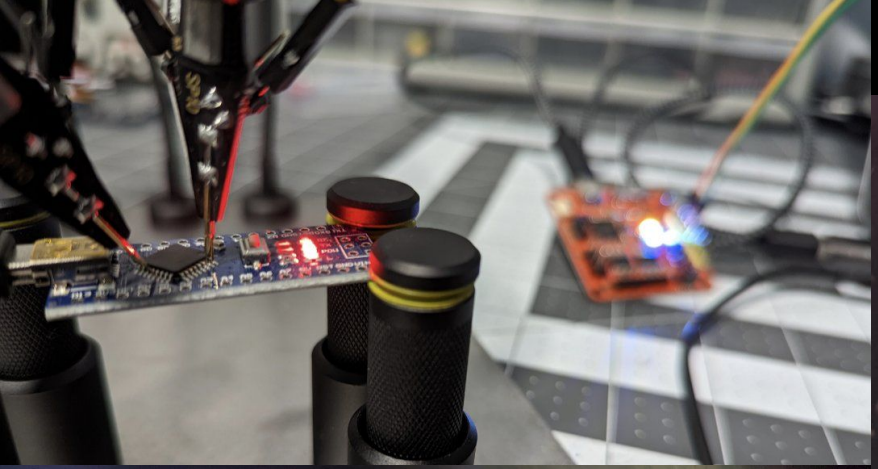
Can use boards like Raspberry Pi, Arduinos, or other dev boards as well

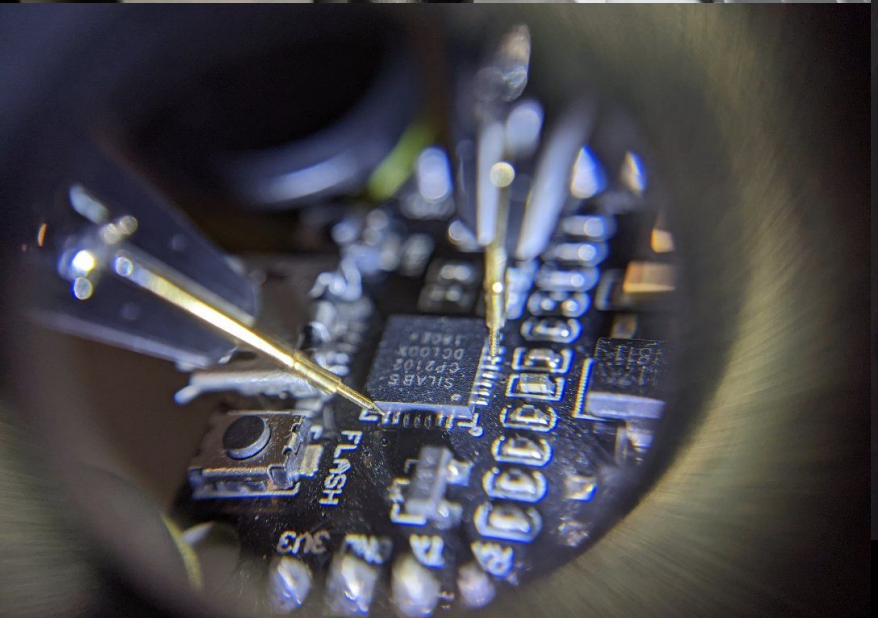


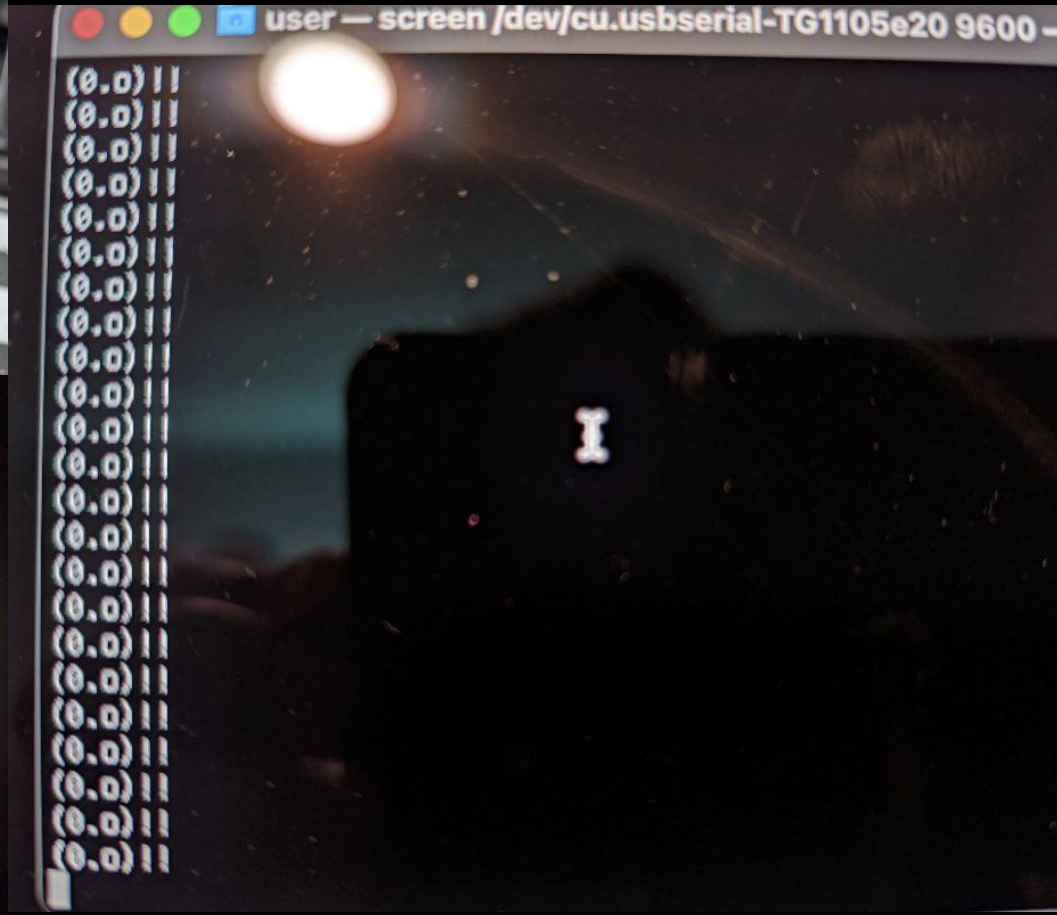
A classic USB to TTL Serial Dongle



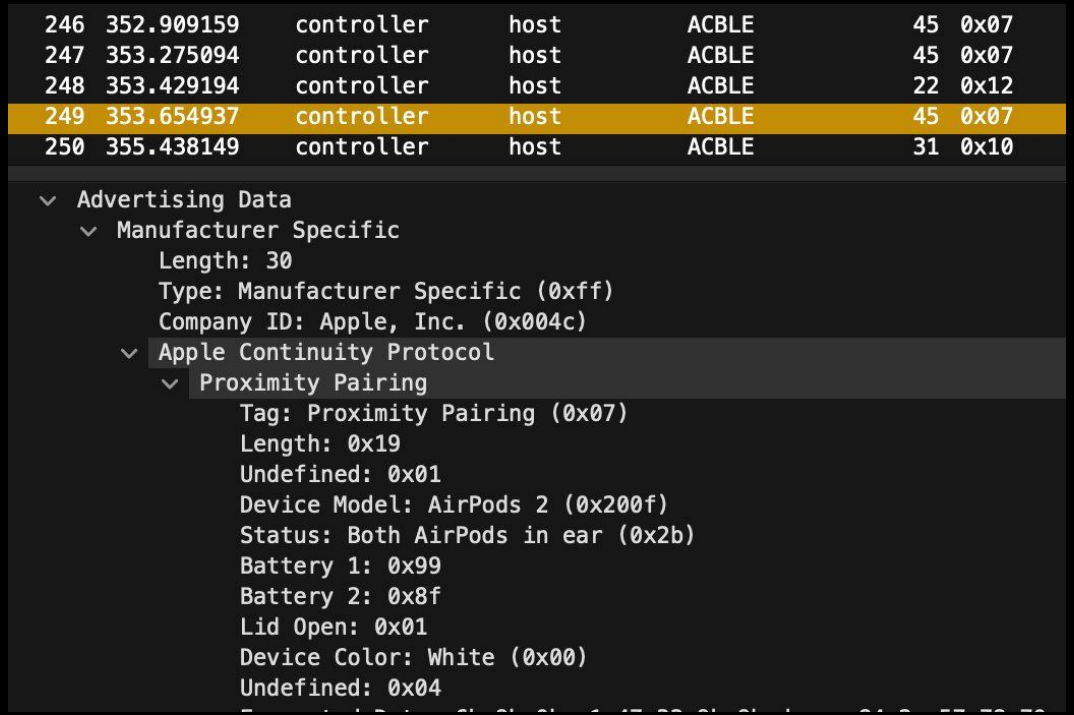
Attaching to UART with Needle Probes







Wireless Protocols



Dissector for the Apple Continuity Protocol

Can be challenging due to hardware requirements

Many protocols require custom dongles or radios to transmit and decode properly

You can monitor raw WiFi and Bluetooth frames with Wireshark

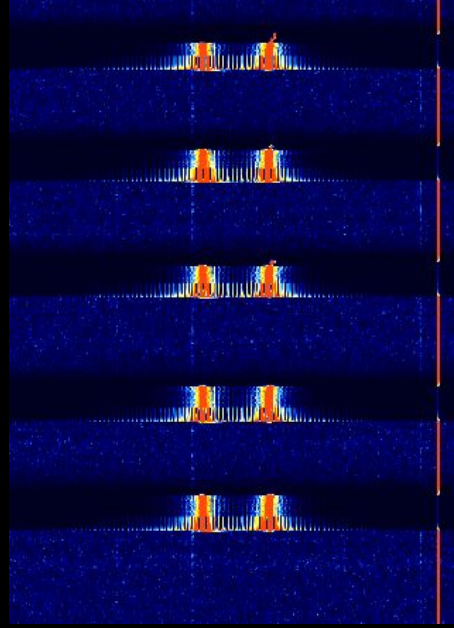
SDR - Software Defined Radio

Open source SDR software can help visualize and decode many radio protocols

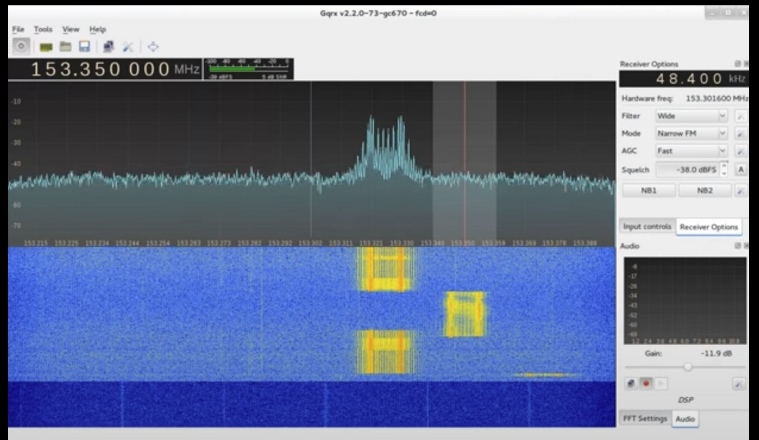
There are plenty of good, cheap SDRs

You can pipe many wireless protocols into Wireshark too

There is a Signal ID Wiki!



2006 Kia Grand Carnival Key Fob



POCSAG pager data in gqrx

RTL-SDR

Specifications

Specifications

Specifications are the design document for the protocol. Like a blueprint.

They can be a blessing and a curse

Some are extremely thorough, may even provide a reference implementation

Others leave much to the imagination, may only consider a specific use-case

Reading Specifications

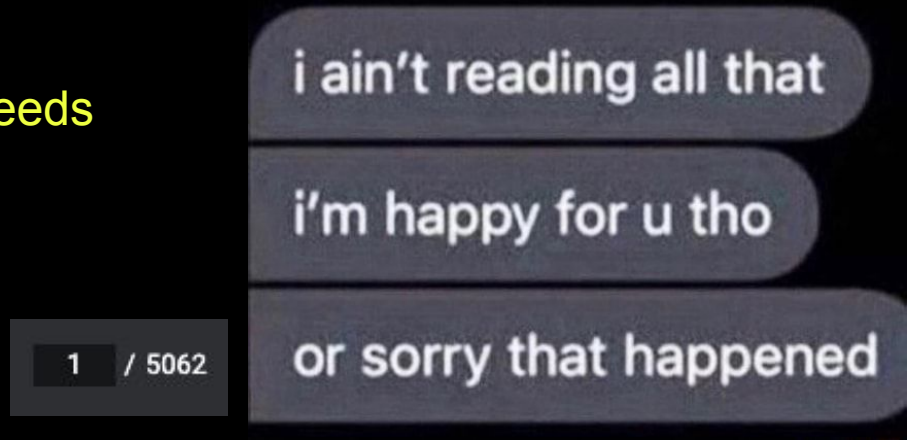
RFCs tend to follow a similar writing style

Vendor specs or standards may be extremely long and may not present the data in a format that's easy to parse like a table, diagram, or structure

Look for document conventions page - Explains how they will present data

Look for appendixes

Take notes in a format that works for your needs



Caution

Specs are subject to change

They don’t reference future versions of themselves so it can be hard to know if what you’re looking at is outdated unless you look it up.

Specifications can be paywalled and *very* expensive.

Specifications can have mistakes

Beware of *Rose Colored Glasses*

I’ve wasted many hours trying to fit data into a protocol spec or header file struct that didn’t match what was actually being transmitted.

Specification vs Implementation

Different implementations of a protocol can vary greatly in terms of features and reliability

The less explicit the spec is, the more liberty can be taken with the protocol overall

This can make it hard to compare a written spec with a particular implementation

Details of a spec should be used as a reference, but if it differs from what you are seeing, the differences should be examined

Two implementations of the same thing

Specification Holes

If something isn’t defined, it’s up to the programmer to figure out what to do with it

Implementations can get things wrong due to misunderstandings and programming mistakes

Implementation Bug: 2 Byte Remote DOS in Telnet (2022)

Telnet (1969) is older than TCP/IP

Telnet was designed to interface with a teletype

FF F7 or FF F8 crashes multiple telnet versions

FF = 255 = Interpret As Command

F7 = 247 = Erase Character

F8 = 248 = Erase Line

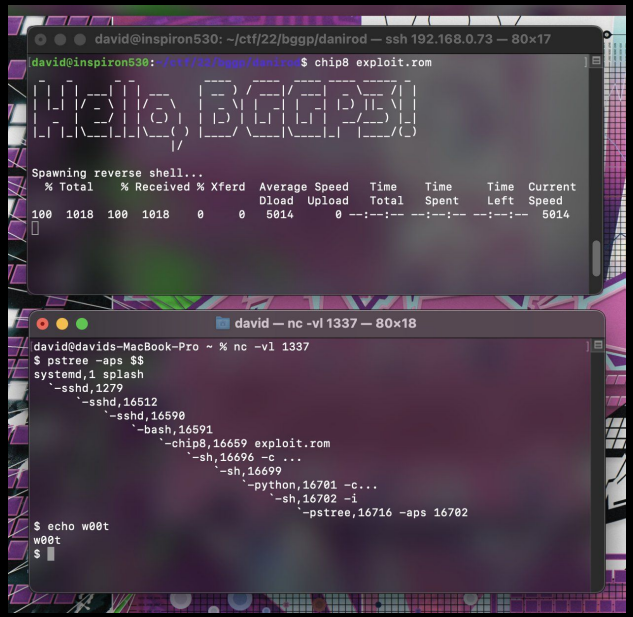
Reads memory that wasn't allocated and crashes

https://pierrekim.github.io/blog/2022-08-24-2-byte-dos-freebsd-netbsd-telnetd-netkit-telnetd-inetutils-telnetd-kerberos-telnetd.html

The following are the defined TELNET commands. Note that these codes and code sequences have the indicated meaning only when immediately preceded by an IAC.

NAME	CODE	MEANING
SE	240	End of subnegotiation parameters.
NOP	241	No operation.
Data Mark	242	The data stream portion of a Synch. This should always be accompanied by a TCP Urgent notification.
Break	243	NVT character BRK.
Interrupt Process	244	The function IP.
Abort output	245	The function AO.
Are You There	246	The function AYT.
Erase character	247	The function EC.
Erase Line	248	The function EL.
Go ahead	249	The GA signal.
SB	250	Indicates that what follows is subnegotiation of the indicated option.
WILL (option code)	251	Indicates the desire to begin performing, or confirmation that you are now performing, the indicated option.
WON'T (option code)	252	Indicates the refusal to perform, or continue performing, the indicated option.
DO (option code)	253	Indicates the request that the other party perform, or confirmation that you are expecting the other party to perform, the indicated option.
DON'T (option code)	254	Indicates the demand that the other party stop performing, or confirmation that you are no longer expecting the other party to perform, the indicated option.
IAC	255	Data Byte 255.

Implementation Bug: CHIP-8 Sandbox Escape (2022)



12 bit address size defined by spec

Non-standard size means it needs to use a bit mask to prevent overflows

The spec only speaks in terms of the virtual machine, and can't know what considerations need to be made to run it on specific hardware.

This ambiguity lead to multiple implementations having out of bounds reads and writes within the emulator from a rom.

https://www.da.vidbuchanan.co.uk/blog/bggp3.html

Parser/Dissector Challenges

Operating systems don’t check the ei_data (endianness) field in an ELF when they run them

If the machine type matches, it will try to run whatever code is in the binary

File analysis tools need to know the correct the endianness to parse it correctly

These tools can be tricked by giving them incompatible parameters and letting the program trust the input

https://tmpout.sh/2/3.html

The file itself has 0xFF for ei_data, with many other fields maxed out in terms of values. It doesn't contain any overlays, it's just a regular ELF.

```
▶ ./0xFFtactics
▶ echo $?
6
```

objdump also has problems with this binary:

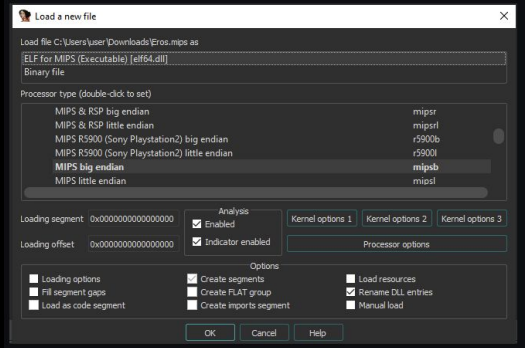
```
▶ objdump -d 0xFFtactics
objdump: 0xFFtactics: file format not recognized
```

As does GDB:

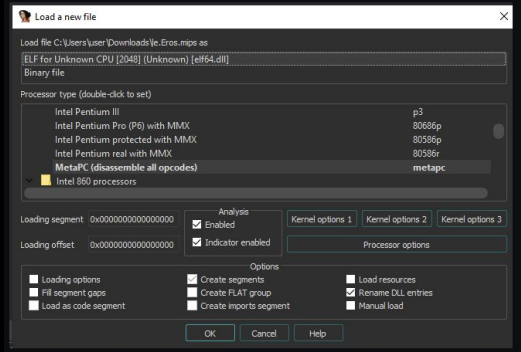
```
▶ gdb ./0xFFtactics
...snip...
"/home/user/./0xFFtactics": not in executable format: file format not recognized
```

As for Ida Pro, similar behavior between the two MIPS ELFs is also achieved. Ida will see le.Eros.mips as an ELF, but it won't know what CPU architecure to process it as. These tests were done on Ida Pro 7.6.210427

Here is the unmodified MIPS ELF:



Here is le.Eros.mips. Note that the CPU is listed as "2048", which corresponds to the e_machine field being swapped to 0x800 when read in little endian format.



With 0xFFtactics, it also doesn't know what architecture it is, but does successfully characterize it as an ELF.

Documenting Your Findings

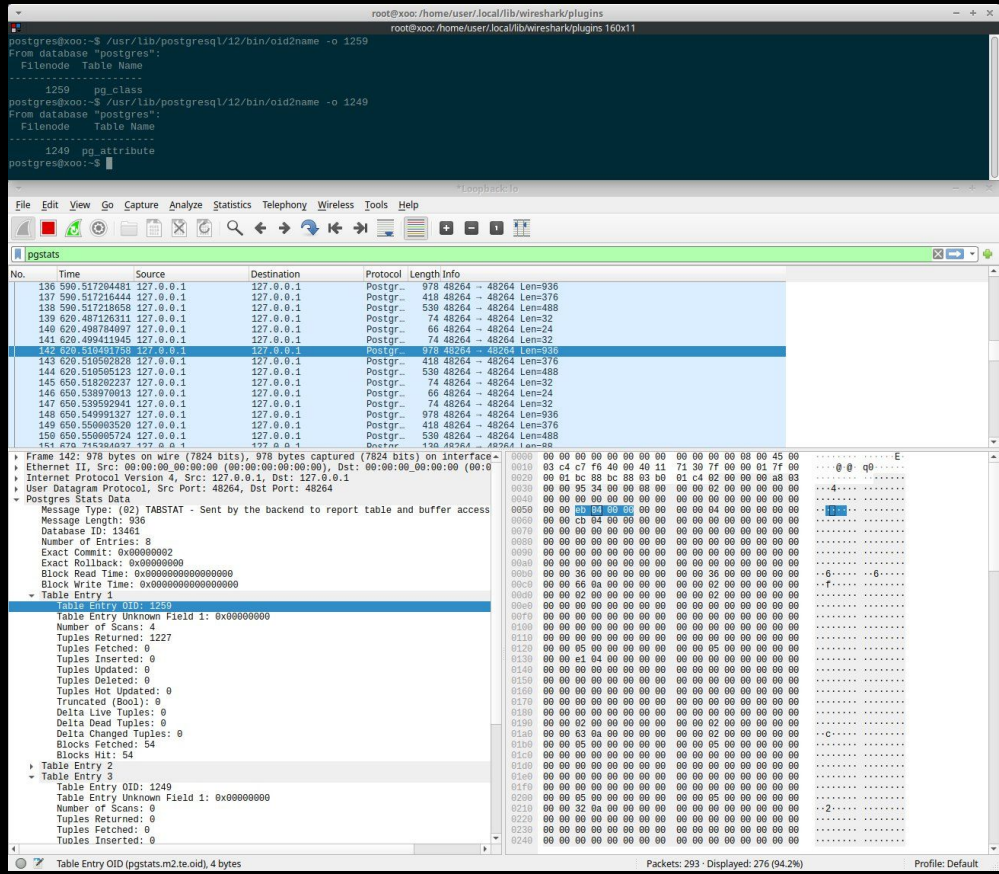
Dissectors

Writing Wireshark dissectors is very handy for tracking protocol messages and finding data you haven't seen before

You can use dissectors to add filtering capability as well

Lua dissectors are good for quickly tracking your findings

Dissectors can also be written in C, but it can be a lot harder to do rapid prototyping in C



Custom dissector for Postgres internal stats protocol

Parsers / Implementations

```
22:50:46 ~/Projects/Korg_RE/client
▸ python3 volcaclient.py -q -p 8
:: Found device: volca sample:volca sample volca sample _ OUT 24:0
Pattern Name: Sci-2Step
```

Slot	Inst.	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	
01	010	■		■		■		■				■						vth2_vnyl_kick_proces_c
02	150								■		■						■	sample_amen_amensnare1
03	151													■				sample_amen_amensnare2
04	152						■											sample_amen_amensnarrid
05	153	■	■	■		■						■				■		sample_amen_amenride
06	154							■	■				■	■				sample_amen_amencrash1
07	156																	sample_amen_amenkickcras
08	110																	Upstrum Pad
09	107	■		■														Holy Stab
10	107																	Holy Stab

```
22:50:48 ~/Projects/Korg_RE/client
▸ python3 volcaclient.py -q --all
:: Found device: volca sample:volca sample volca sample _ OUT 24:0
SLOT: 000 SIZE: 7210    TITLE: Kick Thick
SLOT: 001 SIZE: 16474   TITLE: Kick Dusty
SLOT: 002 SIZE: 10866   TITLE: BD_LELL_24_01
SLOT: 003 SIZE: 19532   TITLE: Kick Vintage
SLOT: 004 SIZE: 7282    TITLE: BD_LELL_26_01
SLOT: 005 SIZE: 1307    TITLE: DHitB-Kick01
SLOT: 006 SIZE: 10856   TITLE: HIPOPTIC
SLOT: 007 SIZE: 13622   TITLE: JJ - Kick6
SLOT: 008 SIZE: 10283   TITLE: kick_get_short
SLOT: 009 SIZE: 14407   TITLE: MIAMI_KICK_04
SLOT: 010 SIZE: 15228   TITLE: vth2_vnyl_kick_proces_c
SLOT: 011 SIZE: 19826   TITLE: vth2_vnyl_kick_sub_warbl
SLOT: 012 SIZE: 8230    TITLE: vth2_vnyl_kick_process_s
SLOT: 013 SIZE: 29553   TITLE: wa_808tape_kick_13_sat
```

Client for custom MIDI extension, result of RE work

Scripts/Programs for parsing input data can also be helpful to codify the protocol, even if you're just analyzing a small piece

Writing a partial (or full if you're ambitious) implementation can be used to research the protocol, giving bi-directional capability

Other Documentation

Structs and header files

Tables

Kaitai

010 Structs

xx

ASCII diagrams

TCP segment header

Offsets		0										1										2										3	
Octet	Bit	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0 0				CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20	160	Options (if *data offset* > 5. Padded at the end with "0" bits if necessary.)																															
⋮	⋮																																
56	448																																

Table with TCP segment header layout

```
~ cat arp.xx
arp.xx -- Creates arp.4cc3ae7e.bin -- 20220926 -- @netspooky_
ARP packet from https://github.com/netspooky/protocols/tree/main/broadcast_brujeria
                    ┌─ Ethernet Layer ─┐
ff ff ff ff ff ff   eth.dst             Destination Mac Address: ff:ff:ff:ff:ff:ff
00 11 22 33 44 55   eth.src             Source MAC Address: 00:11:22:33:44:55
08 06               eth.type            Packet Type: ARP
                    ┌─ ARP Layer ──────┐
00 01               arp.hw.type         Hardware Type: Ethernet
08 00               arp.proto.type      Protocol Type: IPv4
06                  arp.hw.size         Hardware Size: 6
04                  arp.proto.size      Protocol Size: 4
00 01               arp.opcode          Opcode: Request
00 11 22 33 44 55   arp.src.hw_mac      Source MAC Address: 00:11:22:33:44:55
0a 00 01 0a         arp.src.proto_ipv4  Source IP: 10.0.1.10
00 00 00 00 00 00   arp.dst.hw_mac      Target MAC Address: 00:00:00:00:00:00
0a 00 01 0b         arp.dst.proto_ipv4  Target IP: 10.0.1.11
                    └──────────────────┘
```

xx file documenting an ARP packet

Healthy Research Mindset

Abyss Gazer Tips

It can be a rabbit hole to look at protocols and implementations, it can feel like you were playing Tetris for a long time

Make sure you take breaks and take care of your IRL needs

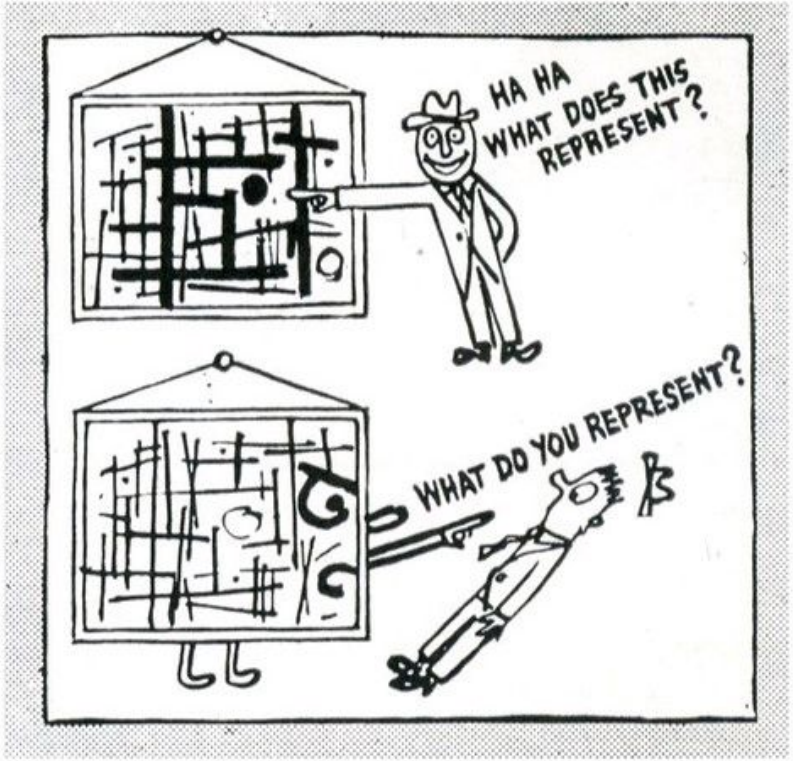
Have things that exercise other parts of your brain in your life

Talk to other people about your project whenever possible (vent/rubber ducky)

Give yourself enough time to document findings/clean up your notes/write down what you want to do next

Figure out what works best for your style

You're only human!



Thanks!

Interesting protocols are *everywhere*! Happy hunting.

References and Further Reading

https://github.com/netspooky/protocols/blob/main/protocol_re/pre_talk_2023_rsrc.md

My website: https://n0.lol