

Cryptography Session: "How Crypto Gets Broken (by you)" 0x4352415348th Ed.

Cybertruck 2021



About Me

- ❑ Senior Cybersecurity Research Engineer contractor at NMFTA
- ❑ Attack Tree Fanatic
- ❑ Enjoys CTFs (maybe a little too much)
- ❑ Enjoys Teaching
- ❑ >10 years of professional experience in embedded systems design
- ❑ MSc. Eng. in Applied Math & Stats from Queen's University
- ❑ Member of and contributor to SAE TEVEES18A1 Cybersecurity Assurance Testing TF (drafting J3061-2) and some ATA TMC committees
- ❑ HHV and CHV volunteer



[linkedin.com/in/Ben0L0Gardiner](https://www.linkedin.com/in/Ben0L0Gardiner)



github.com/BenGardiner



[@BenLGardiner](https://twitter.com/BenLGardiner)

Thanks to:

@Sagefault + @KennethSalt + Dr. Jeremy Daily

And the *CyberTruck Challenge™* Event

previously presented at:



CyberTruck Challenge™ 2018 & 2019 /
HF2020 / NSec 2021

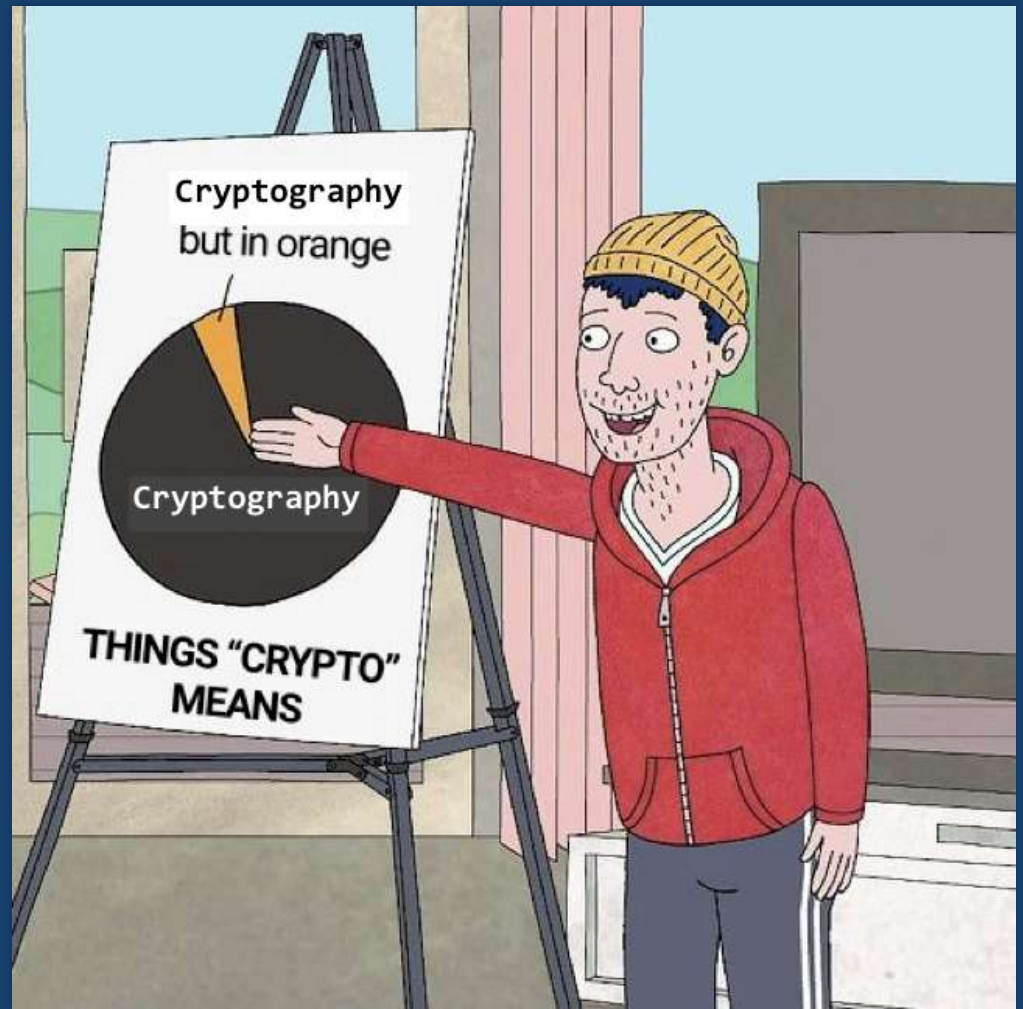
Agenda

- ❑ We will **be taking a brisk pace**
 - ❑ But also feel free to ask questions anytime
- ❑ Much material from the following reference:
Anderson, Ross. *Security engineering*. John Wiley & Sons, 2008.
 - ❑ Buy this book!
 - ❑ Prev. editions are also free!
www.cl.cam.ac.uk/~rja14/book.html

Challenge: Decrypt 'Crypto'	2
Building Blocks	10
☑ Attacking Building Blocks	10
Challenge: Break Hashes	5
Protocols	5
☑ Attacking Protocols	10
Protocol: UDS Seed-Key Exchange (☑ Attacks)	10
Challenge: Derive the UDS Routines	5
<hr/>	
57 mins	
<hr/>	

Highly compressed. See UNABRIDGED version in backmatter for follow-up details.

‘Crypto’



Crypto Building Blocks

Encryption

- **Encryption** – an encoding which can be reversed (given a **key**)
 - A **plaintext** (M) message is encrypted by a **cipher** ($\{\}$) to a **ciphertext** (E) using a **key** (K)

$$E = \{M\}_K$$

- Decryption is possible with the **cipher**, the **ciphertext**, **and the key**
- e.g. **AES, RSA, ECC, 3DES, ...**

- Something that's **not encryption**: **base64** (e.g. `ZS5nLiB0aGlzIGJhbG9uZXkgcm1naHQgaGVyZQ==`)

Hands-On: 2 Minute Challenge

'Decrypt' these (you're actually decoding):

d2VsY29tZSB0byBIRjIwMjA=

c2VudGluZWw=

These are base64 encoded (not encrypted).

This might seem obvious to some – but it is not uncommon to encounter base64 'encryption' in the wild.

Here's a handy set of tools for this:

- <http://rumkin.com/tools/>
- CyberChef: <https://gchq.github.io/CyberChef/>
- in python/jupyter: `import base64; base64.b64decode('xxx')`

Hashes

- ❑ (Cryptographic) Hashes – not an encoding & not reversible
 - ❑ Different than the larger, general class of hash functions
 - ❑ For a crypto. hash function f : given $f(x)$ you can't find (guess or calculate) x
 - ❑ i.e. shouldn't be able to find input x for:
3947cdf52a551de4983746545a1affdb2b04f4a2 or
21232f297a57a5a743894a0e4a801fc3
(actually, this one is easy)
- aka *Random Functions*
 - aka *Shortcut Functions*
 - aka *One-way Compression Functions*
 - aka *Digests*
- ❑ e.g. **SHA-1, SHA-256, BLAKE, ...**
 - ❑ **not a cryptographic hash: MD5**

➤ aka *One-way Functions*

'Classic' vs Modern Crypto

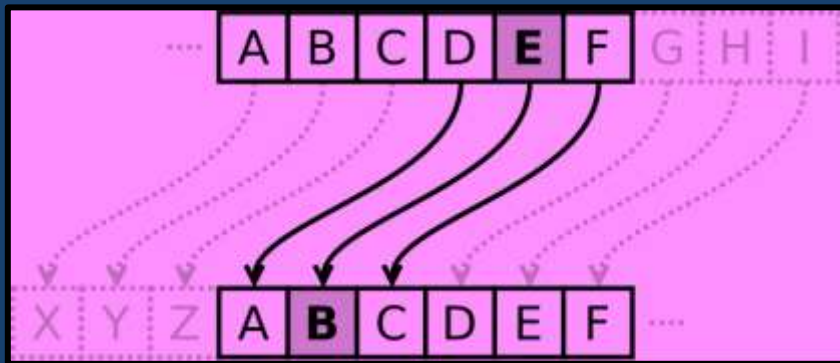
❑ 'Classic' Crypto

- ❑ Mostly pre-20th century
- ❑ Deals with alphabets: input & output
- ❑ e.g. shift cipher (Cesar cipher)

AB	AB	AB	...	AB	AB	AB
CD	CD	CD		CD	CD	CD

 `qbag qrpvcure guvf`


- ❑ e.g. substitution cipher, polyalphabetic substitutions, transpositions etc.
- ❑ It is still encryption – the 'key' is the knowledge of the mapping (shift, letter-map etc.)
- ❑ Relevance today: puzzles, challenges and easy reverse engineering



* Matt_Crypto, wikipedia, Public Domain

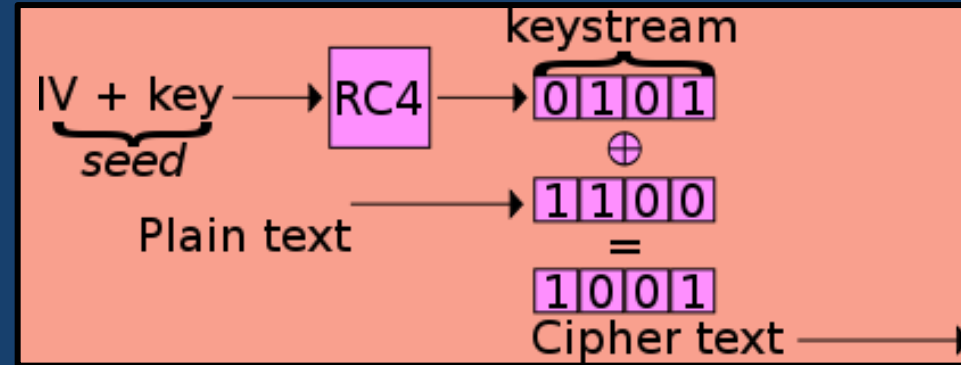
❑ Modern Crypto

- ❑ Deals with numbers: input & output
 - ❑ Text is treated as numbers via encodings – ASCII or UTF-8 is the most likely encoding
- e.g.

`646f6e742064656369706865722074686973` \oplus `(00...10)` 
`646e6c77246163646179626e7e2d7a677962`

Stream Ciphers

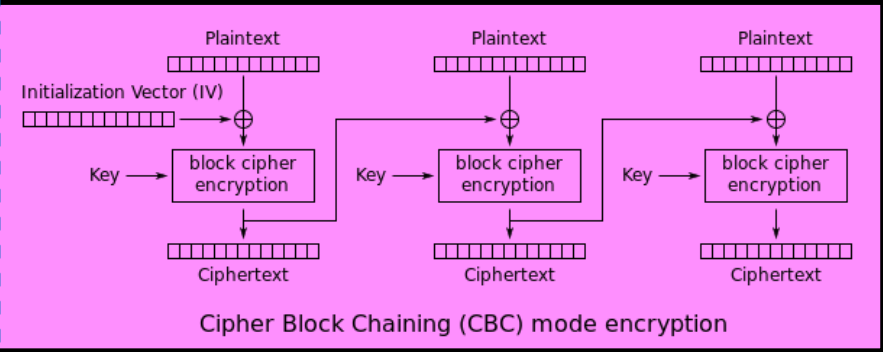
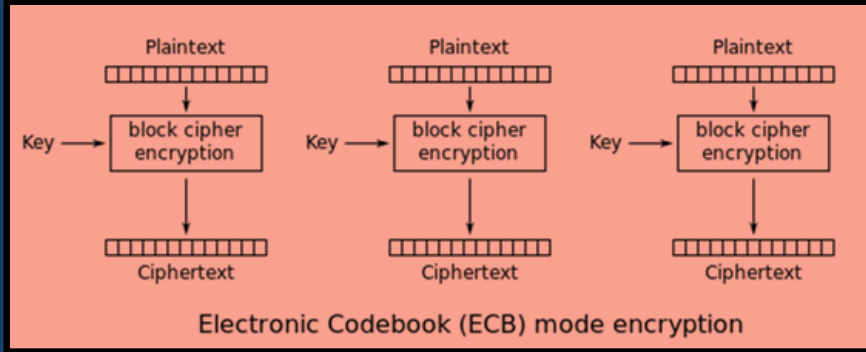
- ❑ One-Time Pad (OTP) – the only proven secure encryption scheme
 - ❑ Uses random **key-stream**, of length equal to or greater than the message
 - ❑ Then combine **key-stream** with message (assume **XOR**)
- ❑ Stream Ciphers – approximate the OTP
 - ❑ Expand short **key** into pseudo-random **keystream**
 - ❑ Then **XOR** (\oplus) (\wedge)
 - ❑ e.g. RC4, Salsa20, FISH
- ❑ note: IV – initialization vector. It shouldn't need to be secret



Di Kyle Siehl - Self-made, based on raster w:Image:Wep-crypt.png, which was taken with permission from The Final Nail in WEPs Coffin, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=1806804>

Block Ciphers

- ❑ Block Ciphers – different approach
 - ❑ Uses a **key** and **fixed-length inputs (blocks)**
 - ❑ Combined with previous outputs and more fixed-length inputs in various modes:
 - ❑ **ECB, CBC, PCBC, CFB, OFB, CTR ... GCM(!)**



By WhiteTimberwolf (SVG version) - PNG version, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=26434116>

By WhiteTimberwolf (SVG version) - PNG version, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=26434096>

Symmetric / Asymmetric Crypto

- ❑ **Symmetric Crypto** – can be encrypted + decrypted by any party with the **SAME** key
 - ❑ e.g. any of the crypto we've discussed so far
- ❑ **Asymmetric Crypto** – can be encrypted by any party for a specific recipient
 - ❑ aka public-key cryptography
 - ❑ Leverages certain problems that are hard in one way & easy in the other: **prime factorization** and **discrete logarithms**
 - ❑ Keys exist as pairs of **public** & **private** halves -- **key-pairs**
 - ❑ The party with the **private** key can **decrypt & sign** (more on signatures later)
 - ❑ Any parties with the **public** key can **encrypt & verify**
 - e.g. RSA, ECC
 - ❑ e.g.

```
-----BEGIN RSA PRIVATE KEY-----
izfrNTmQLnfsLzi2Wb9xPz2Qj9fQYGgeug3N2MkDuVHwpPcgkhHkJgCQuuvT+qZI
...
```

Crypto Building Blocks

Section Summary

- ❑ **Encryption**... it hides information, binds it – protects confidentiality, but not integrity (without additional effort)

$$E = \{M\}_K$$

- ❑ (Crypto) **Hashes** – one-way functions. With $f(x)$ you cannot get x
- ❑ 'Classic' **Crypto** – involves alphabets not numbers
- ❑ **Stream Cipher** – combine a sequence of key bits with a sequence of cleartext bits with XOR (\oplus) (\wedge)
- ❑ **Block Ciphers** – have a limited key stream, but extend to larger cleartext sequences
 - ❑ Not all block cipher modes are created equal (e.g. **Electronic Coloring Book (ECB)**)
- ❑ **Symmetric Crypto** – all parties share the same key
- ❑ **Asymmetric Crypto** – only one party has the decryption key (private key)

Attacks on Building Blocks

Attacking Hashes

- ❑ [Google](#).
 - ❑ Seriously... google this [21232f297a57a5a743894a0e4a801fc3](#) (from before) now
- ❑ Identifying what type of hash you have in-hand will be useful – the length gives it away
 - ❑ If you don't know lengths yet, use hash detector tools; e.g. [cothan/hashdetector](#)
- ❑ Hash Crack sites
- ❑ hashcat tool
 - ❑ (ab)uses your GPU for rapid hash cracking
- ❑ Rainbow Tables
 - ❑ 'halves' / parts-of hashes pre-built and ready to go
 - ❑ For things like MD5 these are trivial
 - ❑ For things like SHA-256 these are huge (multi-TB)
 - ❑ You can pick-up pre-generated tables at DEFCON Data Duplication Village. Bring a 6 TiB HDD.
- ❑ And cooler things like hash-length extension attacks

More on Attacking Hashes

- ❑ Salts
 - ❑ Because it's pretty easy to lookup or build a table of known inputs for hashes; designers tend to follow the best practice of 'salting' their inputs
 - ❑ `D033e22ae348aeb5660fc2140aec35850c4da997 = SHA1('admin')`
 - ❑ `3947cdf52a551de4983746545a1affdb2b04f4a2 = SHA1('saltadmin')`
 - ❑ Salts are usually pre-prepended onto the input; sometimes with a separator like `.'` or `'+'`
 - ❑ `hashcat` can find a salt for a given hash and input pair.
 - ❑ `hashcat` can also find inputs for hashes with a given salt as a parameter.
 - ❑ Find the salt with one known hash first.
 - ❑ OR find the salt with research (some systems' password salts are well-known)

Still More on Attacking Hashes



- ❑ Password lists

- ❑ Brute-forcing (all possible character combinations) for inputs to hashes is possible
- ❑ 'password lists' are more useful. There are hundreds of these to choose from, most from data breaches over the past years.
- ❑ In CTFs the [rockyou](#) list is the most common – but for applied hash cracking: YMMV.
- ❑ This is more generally known as a [dictionary attack](#)

Hands On: 5 Minute Challenge

Reverse these hashes:

- 5f4dcc3b5aa765d61d8327deb882cf99
- 5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
- ecadec2924e86bf88d622ceb0855382d
- ff4827739b75d73e08490b3380163658
- ~~6ce3bb6eb450df7d6345151ec00e4a4e~~

We've mentioned the tools you need for this.

Some are easy. ~~One is not (hint: 2 character salt)~~

Attacking 'Classic' Crypto

- ❑ Historically, **frequency analysis** was the undoing of classic crypto
 - ❑ Letter use in a language (e.g. English) has a predictable # occurrences (frequency)
 - ❑ Count the **number of occurrences** of a symbol in ciphertext; match to expected rate in language
 - ❑ Requires medium-large ciphertext for analysis to work
- ❑ Today (challenges/puzzles/RE):
 - ❑ Try shift ciphers (start with **ROT13**)
 - ❑ Then try a substitution cipher
 - ❑ Then have 'fun' : <http://rumkin.com/tools/cipher/>

Stream Cipher Attacks

□ Re-used Key Attack

□ Recall: it's all about XOR (\oplus) (\wedge)

□ If I know $A \wedge B$ and I know A or B , I can get the other

□ Anytime a stream cipher re-uses keys, it's a problem

□ if I have $E1 = A \wedge K$ and $E2 = B \wedge K$ I can get $A \wedge B$

□ this is a big deal if:

□ A, B are natural language (use running key cipher attacks on $A \wedge B$) or if

□ A, B are different lengths or if






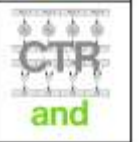



□ we can control A or B or if

□ we can make any guesses about A or B

Block Cipher Attacks

- Getting impractical now...
- Goals: *forgery* or *key-recovery*
- Block Cipher Attack Models
 - **known plaintext**: attacker is given a set of pairs of *cleartext*+*ciphertext*
 - **chosen plaintext**: attacker has the ability to query *cleartext* and receive *ciphertext*
 - **chosen ciphertext**: attacker has the ability to query *ciphertext* and receive *cleartext*
 - **chosen plaintext/ciphertext**: attacker has the ability to query either
 - **related key**: attacker has the ability to query with *key* related to specified *key*, K (e.g. $K+1$ $K+2$, ...)

Other Attacks on Block Ciphers

 All	 modes	 are
 beautiful	 not you	 and
 deserve	 to be	 used

- ❑ Recognizing ciphertext blocks can let you decrypt them:
 - ❑ maybe not to their contents, but to their meaning
 - ❑ (Sometimes also their contents; e.g. infer all-zeroes input)
- ❑ Use viz tools: vix, radare2, binvis.io, Veles, hobbits

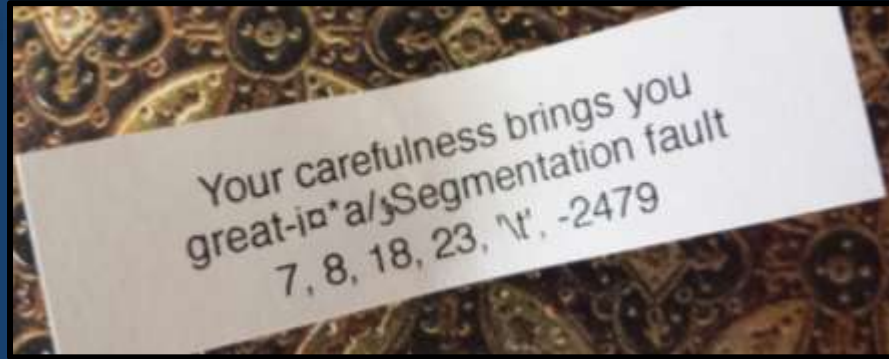
AES_ECB() =



<https://github.com/pakesson/diy-ecb-penguin>

Other Attacking Building Blocks

- ❑ Software Exploitation can yield both **control of the software** and also **information leaks**



- ❑ Access to process memory can be fruitful **key extraction** attacks
- ❑ Multiple tools are available to scour memory for **keys**:
 - ❑ e.g. aeskeyfind, radare2, volatility
- ❑ Reverse engineering of the program code in memory can yield pointers to the memory locations of **keys**
- ❑ Don't underestimate the downplayed Infoleak vulnerabilities
 - ❑ c.f. Heartbleed

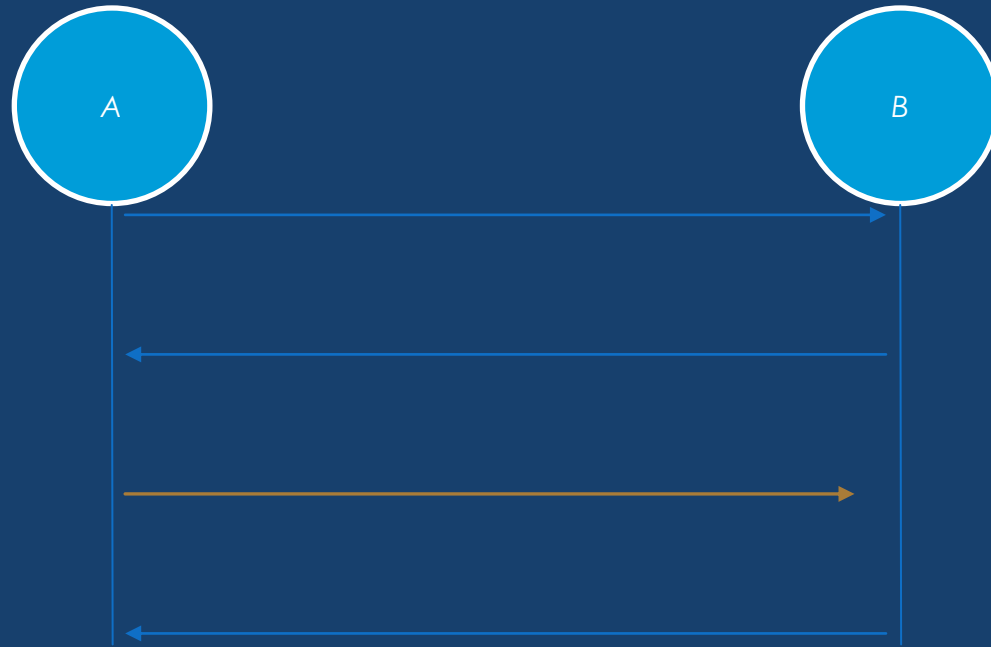


Attacks on Building Blocks **Section Summary**

- ❑ **Hash Attacks** – collisions, pre-image etc. use google. All other practical (for us mortals) attacks are in hashcat, use it.
- ❑ **Classic Crypto Attacks** – frequency analysis. Try simple things first, use cryptogram tools, ID the cipher and try cipher-specific attacks
- ❑ **Stream Cipher Attack** – Reused Key Attack. i.e. try XOR (^) things together, make guesses
- ❑ **Block Cipher Attack Models** – probably impractical but use the right search terms
 - ❑ Except ECB: recognize patterns
- ❑ Don't forget about software exploitation; in-memory attacks.
- ❑ Breaking protocols is more fruitful (next sections)
- ❑ Remember these tools:
 - ❑ <http://rumkin.com/tools/>
 - ❑ CyberChef: <https://gchq.github.io/CyberChef/>
 - ❑ Visualization tools: binwalk -E, radare2, binvis.io, Veles, hobbits

Protocols

Protocols



- ❑ **Protocols** – the rules that govern the communication between parties
 - ❑ What information is transmitted from party A to party B?
 - ❑ What steps must party B perform?
 - ❑ What information must be sent in reply (if any)?
 - ❑ etc.

Protocol: Simple Authentication

Simple Authentication:

- Source: wants to be authenticated by the target
- Target: decides if source is authentic
- The source sends to the target:
 - its ID (T) plus an encrypted concatenation of T and a nonce (N), with a key (KT) that could be specific to the ID and also is known to the target.



- The target looks-up encryption key K_T for ID T; decrypts the $\{\dots\}_{K_T}$ and checks the nonce N hasn't been seen before.
- Nonce : Number used ONCE
- E.g. older keyfobs / garage door openers – source is the fob, target is the car or garage door.

Protocol: Message Authentication Codes (MAC)

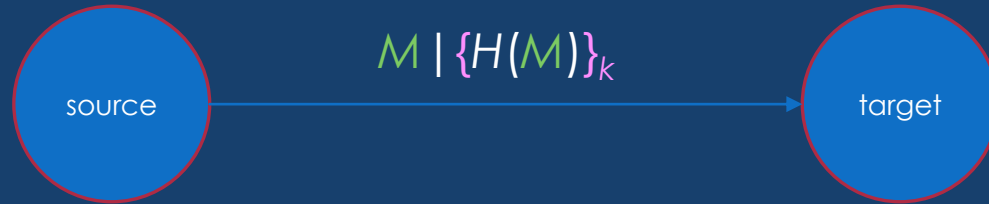
- ❑ **Message Authentication Codes:** for a **message**, create a **value** that can enable the message to be verified by any party with the **shared key** (the same **shared key** that is used to create the value). e.g.:
- ❑ CBC-MAC – build a MAC with CBC chaining mode of a block cipher
- ❑ CMAC – also uses a block cipher
- ❑ HMAC – build a MAC with a hash function
- ❑ CBC-MAC-AES128, HMAC-SHA1, etc.



- ❑ Parties receiving messages that don't verify against the **key** (**shared** in this case) shall discard messages
 - ❑ How the **shared keys** are distributed and how messages are discarded is additional protocol details (for the next layer of the protocol specification)
- aka **Message Integrity Code (MIC)**
- aka **protected checksums**
- **Not a MAC:** a message digest: $f(M)$ where f is a hash function.

Protocol: Digital Signatures

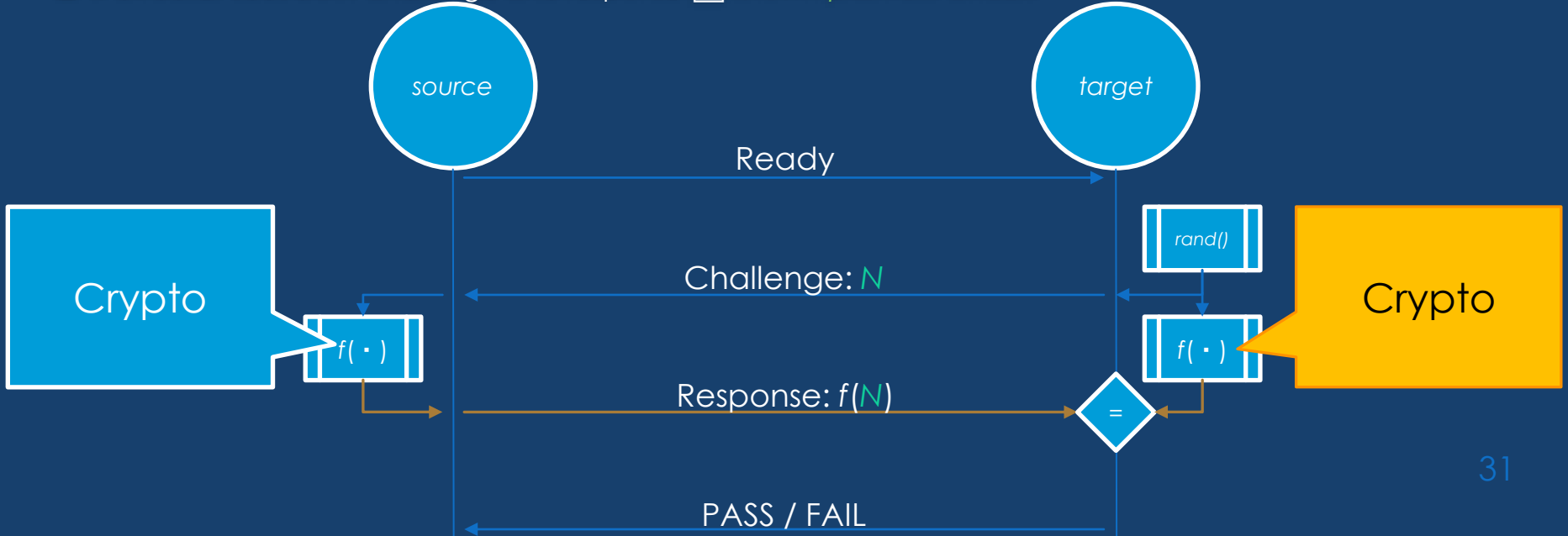
- **Digital Signatures**: using asymmetric crypto, for a message: create a value that can enable the message to be verified by any party with the **public key** but cannot be created by any party without the **private key**.
 - a signing party with a **private key** can **create** a signature
 - parties with the **public key** can **verify** that signature
- e.g. DSA, ECDSA. Let's consider a simple, older RSA signing:
 - Send message, M , and signature together



- To verify: Decrypt $\{H(M)\}_k$ and assert it is equal to $H(M)$, where H is a cryptographic hash and k is the RSA private key
- In both MAC and Signatures, parties receiving messages that don't verify against the **key** (**public** in this case) shall discard messages
 - How the **public keys** are distributed and how messages are discarded is additional protocol details (for the next layer of the protocol specification)
 - e.g. what if they sent: $K \mid M \mid \{H(M)\}_k$ where K is the **public key**?

Protocol: Challenge-Response (C-R)

- Source wants to be authenticated by the target
- Source receives a nonce as challenge
- Transforms it and replies as response
- An ideal C-R would make it impractical for an attacker to guess the secret by observing traffic of multiple C-R exchanges.
 - If attacker sees both challenge and response \rightarrow known plaintext attack



Protocols

Section Summary

- ❑ **Protocols** – the rules that govern the communications between parties
- ❑ **Digital Signatures** – can be **created** by parties with the **private key** but **verified** by anyone with the **public key** (built from asymmetric crypto)
- ❑ **Message Authentication Codes (MAC)** – can be created and verified by any party with the key (can be built from symmetric crypto)
- ❑ **Nonce** “number used once” – can be random or a counter ...
- ❑ **Simple Authentication** – source send its **ID** and an **encrypted ID+nonce** pair to a target for verification
- ❑ **Challenge Response** – target sends **nonce** to source; source replies with some proof that it has an **ID** known to the target
 - ❑ e.g. **nonce** encrypted with **key** known to source
 - ❑ e.g. **nonce** transformed with parameters known to source

Attacks on Protocols

Attacks on Protocols

- *Generally*: try to break the **assumptions** of the protocol
- This actually generalizes to “How to attack any specification”:
 - Anywhere the specification says **SHALL/SHOULD** – see what happens when it **DON'T**...

Attacks on Simple Authentication

- ❑ Simple Authentication assumes nonce N hasn't been seen before
- ❑ If the nonce is random:
 - ❑ Does it actually check? \rightarrow Send again (Replay Attack)
 - ❑ How many nonces does it store? \rightarrow Send +1 (Valet Attack)
- ❑ If the nonce is a counter:
 - ❑ How does it resynchronize? \rightarrow Try sending counter guesses (Bad counter resync attack)
- ❑ Simple Authentication assumes that the key K_T is associated with the ID T and
 - ❑ Are there other T that could associate with K_T ? \rightarrow Try sending to other target (Key collision attack)

Attacks on MAC

❑ For digests

- ❑ Recall: these aren't actually MACs – but they get used that way occasionally
- ❑ Recall: you will know the input, i.e. you will have at least one digest+message pair
- ❑ You need to identify digest algorithm – length usually gives it away; also see tools like cothan/hashdetector
- ❑ You may need to identify the salt also – hashcat can do this

❑ For HMAC- MD5, SHA1, ...:

- ❑ hashcat can crack the key or salt given a hmac+message pair

❑ Software exploitation, 'confused deputy'

- ❑ Software exploitation could enable control of what messages are sent by a piece of SW designed to send mac+message pairs.
- ❑ Yields a successful forgery attack unless other software-integrity measures are taken.

Attack on Digital Signatures

- Recall the RSA Signature example: Send message, M , and signature together

$$M \mid \{H(M)\}_k$$

- Agreeing on the K public key for the k private key is a critical part.

- What if the protocol includes the public key K ?

$$K \mid M \mid \{H(M)\}_k$$

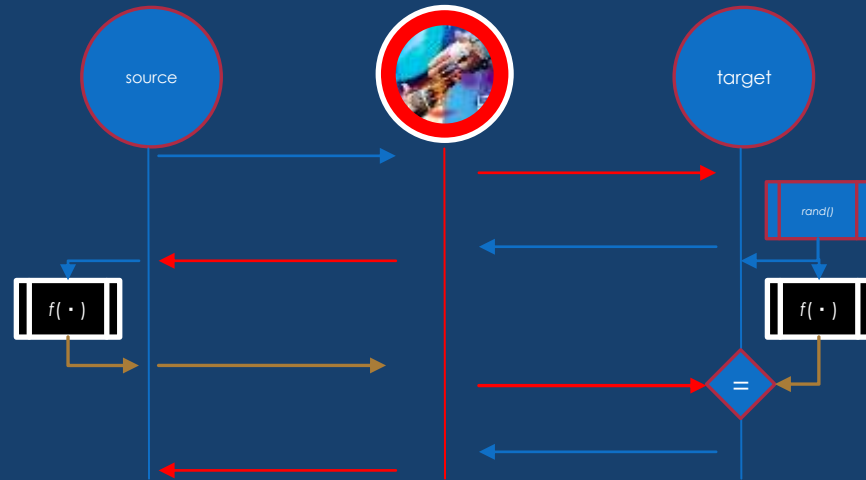
- Then an attack is to use your own private/public key pair a/A and send:

$$A \mid M \mid \{H(M)\}_a$$

- Watch out for this **broken protocol (sending the pubkey)**. It happens sometimes...
- More generally: try to find ways to substitute the expected public key K for your key, A
 - Stored in flash somewhere?

Attack on Challenge-Response: Middleperson Attack (in General)

- ❑ Interposing an actor **in-between** the source and target
 - aka **MITM**
- ❑ Enables tampering with the contents, ordering, timing etc.
- ❑ Good concept for **attacks** on specific **Challenge-Response protocols**
- ❑ Definitely applicable in **TLS/SSL attacks** when you can interpose
- ❑ Can even be effectively achieved without physical interposition if messages can be selectively denied (e.g. **CANT** or **CANHack attacks**)



Attacks on Protocols

Section Summary

- ❑ **Attacks on protocols** are more fruitful than attacks on building blocks
- ❑ Simple Authentication Attacks
 - ❑ **Key Collisions** – e.g. 16bit serial number used as input to key
 - ❑ **Key Extraction and Extension** – e.g. Keeloq
 - ❑ **Replay Attack** – capture one or more, replay selectively
 - ❑ **Valet Attack** – capture a large set during temporary but extended possession
 - ❑ **Bad Counter Resynchronization** – depends on resync behavior of protocol
- ❑ MAC
 - ❑ **Digests (broken)**, **Hash breaking** HMACs, **shared-key reuse** for MACs
- ❑ Digital Signature Attacks
 - ❑ **Public key** substitution
- ❑ Challenge-Response Attacks
 - ❑ Middleperson Attack
 - ❑ (and more coming up in later section)

Protocol: UDS Seed-Key Exchange

UDS

- ❑ Unified Diagnostic Services – ISO 14229 ; on CAN: ISO 15765
- ❑ Used for nearly ALL vehicle Diagnostic Protocols
- ❑ You will learn *a lot* about it in other sessions today and tomorrow

- ❑ There are actions in UDS that are protected. To execute the action requires *authorization*: e.g.
 - ❑ Read memory
 - ❑ Reflash ECUs
 - ❑ Perform potentially dangerous maintenance operations
 - aka '*the fun stuff*'

UDS Authorization

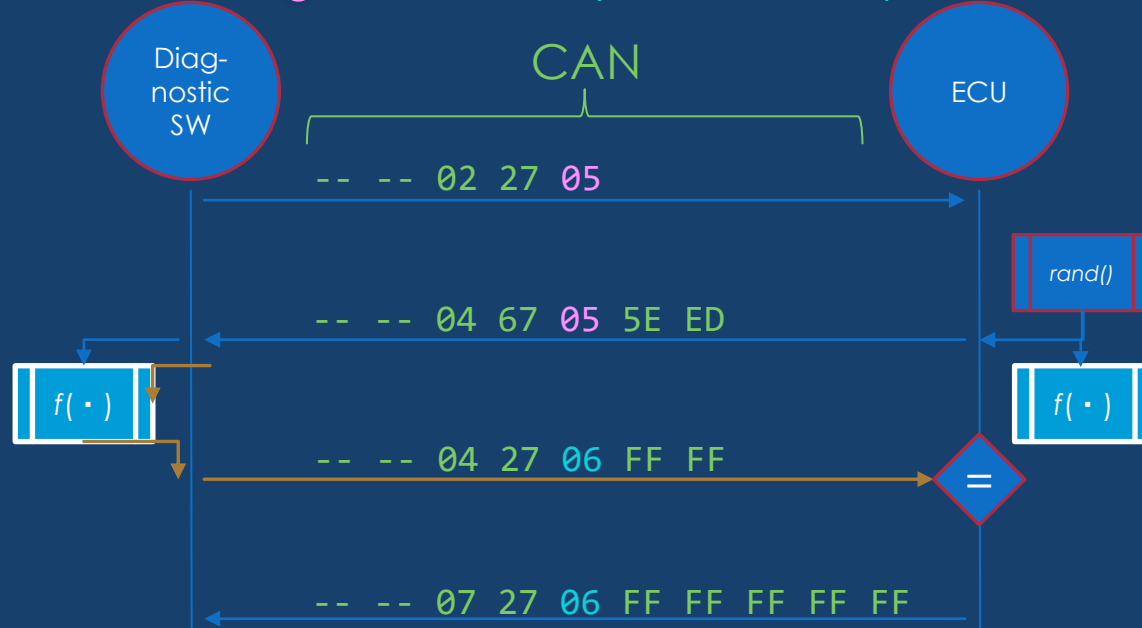
- ❑ Sometimes UDS is helpful; it will tell you that you need to authorize
 - ❑ Negative Response Code : `SecurityAccessDenied`
 - ~~❑ You'll learn about these~~
- ❑ To authorize; unlock the current session with `SecurityAccess Seed-Key Exchange`
 - ❑ 'Session holder' (server) emits a '`seed`'; 'session user' (client) returns a '`key`'
 - ❑ Service `0x27` (replies on `0x67`)
 - ❑ Subfunction `0x05` for `requestSeed` / `0x06` for `sendKey`
 - ❑ You'll know more about these soon

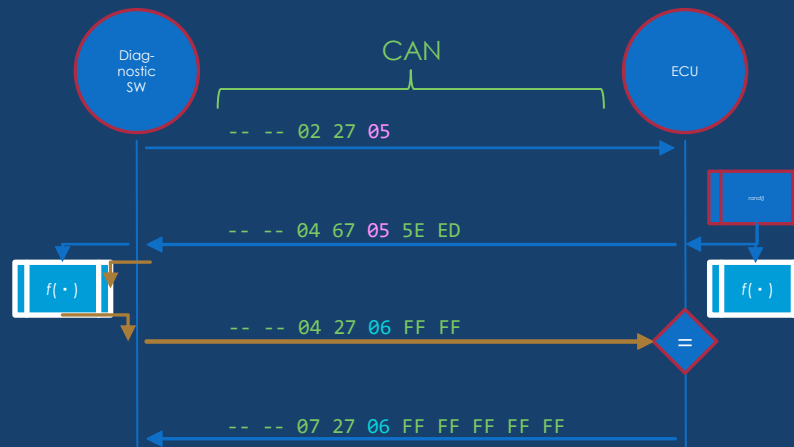
Seed-Key Exchange

⑥ Seed-key exchange is a Challenge-Response Protocol

⑥ Only 16-bit space; so it might not fit our ideal characteristics of resisting known plaintext forgery attacks

⑥ The 'seed' here is a challenge and the 'key' here is a response





PT	B1	B2	B3	B4	B5	B6	B7	B8
18DA00F1	2	10	3	0	0	0	0	0
18DAF100	6	50	3	0	14	0	C8	0F
18DA00F1	3	22	F1	0	0	0	0	0
18DAF100	7	62	F1	0	2	1	0E	3
18DA00F1	2	27	5	0	0	0	0	0
18DAF100	4	67	5	81	B7	1	0E	3
18DA00F1	4	27	6	16	98	0	0	0
18DAF100	2	67	6	81	B7	1	0E	3
18DA00F1	10	0D	2E	F1	5C	0	0	0

Daily J., COMVEC15, A Digital Forensics Perspective ...

NB: J1939 IDs 0x18DA00F1 and 0x18DAF100 are used for UDS over J1939

5 Minute Hands-On: Derive the Seed-Key Routines

1

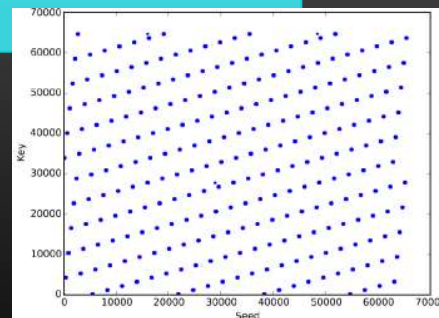
```
18DAF100#0467055b31
18DA00F1#0427065c31
18DAF100#0467053632
18DA00F1#0427063732
18DAF100#0467052c31
18DA00F1#0427062d31
18DAF100#0467053839
18DA00F1#0427063939
```

2

```
18DAF100#0467050100
18DA00F1#0427063435
18DAF100#0467050100
18DA00F1#0427063435
18DAF100#0467050100
18DA00F1#0427063435
18DAF100#0467050100
18DA00F1#0427063435
```

3

```
18DAF100#0467052c31
18DA00F1#0427060005
18DAF100#0467053132
18DA00F1#0427061d06
18DAF100#0467053732
18DA00F1#0427061b06
18DAF100#0467053137
18DA00F1#0427061d03
```



Daily J., COMVEC15, A Digital Forensics
Perspective ...

Attacking Seed-Key Exchange

Attack Goals:

- a) Get a security session (small-scale)
- b) 'Pirate' a security session (large-scale)

For a):

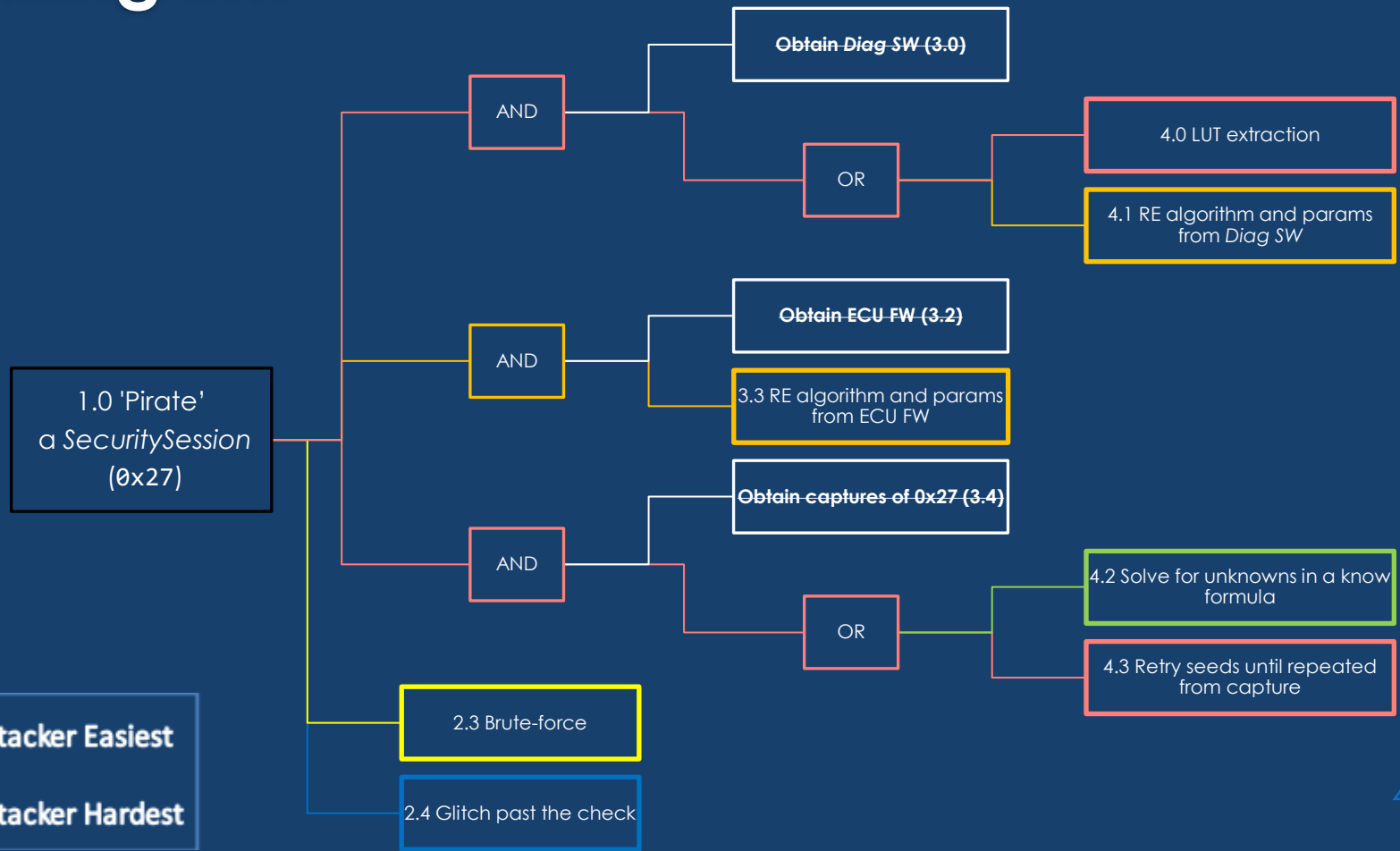
If you **do** have diagnostic SW:

- Use a middleperson attack; done.

If you **don't**:

- Then it is equivalent to *pirating a session* ; which we'll expand on...

b) Pirating it...



Protocol: Seed-Key Exchange

Section Summary

- ❑ J1939 IDs `0x18DA00F1` and `0x18DAF100` are used for UDS over J1939
- ❑ `SecurityAccess` service is `0x27` / sub requestSeed: `0x05` sendKey: `0x06`
- ❑ If you have **diagnostic software**:
 - ❑ Reverse the key algorithm & parameters from PC software
 - ❑ Black-box / Lift the key algorithm & parameters
- ❑ If you have **ECU firmware**:
 - ❑ Reverse the key algorithm & parameters from firmware image (NB: you might have the wrong direction of algorithm)
- ❑ If you have **some captures** of successful `SecurityAccess`:
 - ❑ Solve for unknowns in a known formula from related ECUs
 - ❑ Retry seeds until a match occurs with one in the captures
- ❑ If you have **only the ECU**:
 - ❑ Brute-force (can you control the seed?)
 - ❑ Get some captures (e.g. service center) – see above
 - ❑ Glitch past the check – be amazing



Any transformation

UDS Seed-Key
Exchange

Is this Crypto?

Closing

Summary

- ❑ 'Modern' crypto is about numbers / Classic 'crypto' is about alphabets
- ❑ 'Crypto is hard' → means correct crypto is hard to break, if you have only the capture of communications
- ❑ Crypto building blocks don't get broken very often (given only the capture of comms)
- ❑ Crypto *protocols* get broken
- ❑ Crypto gets broken via side-channels
- ❑ Crypto gets broken by compromise of execution environment
- ❑ You can middleperson-attack TLS/SSL
- ❑ You can lift/reverse/solve/brute-force Seed-Key Exchange

Resources for Continued Learning

- [*Cryptopals \(CTF\)*, T. Ptacek et. al.](#)
- [*Let's Play with Crypto \(Pres.\)*, Ange Albertini](#)
- Any and all [*SO answers*](#) by Thomas Pornin
- [*Security Engineering \(Book\)*, Ross Anderson](#)
- [*PotatoSec Crypto Puzzle Challenges*](#)
- POC | |GTF0 (Journal), [mirror](#)

***Cryptography Session:
"How Crypto Gets Broken (by you)"
-- UNABRIDGED --***

About Me

- ❑ Senior Cybersecurity Research Engineer contractor at NMFTA
- ❑ Attack Tree Fanatic
- ❑ Enjoys CTFs (maybe a little too much)
- ❑ Enjoys Teaching
- ❑ >10 years of professional experience in embedded systems design
- ❑ Masters of Engineering in Applied Math & Stats from Queen's University
- ❑ Member of and contributor to SAE TEVEES18A1 Cybersecurity Assurance Testing TF (drafting J3061-2) and some ATA TMC committees
- ❑ HHV and CHV volunteer



[linkedin.com/in/Ben0L0Gardiner](https://www.linkedin.com/in/Ben0L0Gardiner)



github.com/BenGardiner



[@BenLGardiner](https://twitter.com/BenLGardiner)

Thanks to:

@Sagefault + @KennethSalt + Dr. Jeremy Daily

And the *CyberTruck Challenge*[™] Event

previously presented at:



CyberTruck Challenge[™] 2018 & 2019 /
HF2020 / NSec 2021

Agenda

- ❑ We will **break regularly** for questions at section breaks
 - ❑ But also feel free to ask questions anytime
- ❑ Much material from the following reference:
Anderson, Ross. *Security engineering*. John Wiley & Sons, 2008.
 - ❑ Buy this book!
 - ❑ Prev. editions are also free!
www.cl.cam.ac.uk/~rja14/book.html

Challenge: Decrypt 'Crypto'	10
Building Blocks	20
Challenge: Break Hashes	15
☐ Attacking Building Blocks	20
Challenge: Break Crypto, others	25
☐ (More) Attacking Building Blocks	10
Challenge: Visualize Crypto	20
Protocols	15
☐ Attacking Protocols	10
Protocol: WPA2 (☐ Attacks)	5
Protocol: TLS / SSL (☐ Attacks)	20
Heavy Vehicle Networks Crash Course	5
Protocol: UDS Seed-Key Exchange (☐ Attacks)	30
Challenge: Derive the UDS Routines	15
210 mins	

‘Crypto’



Crypto Building Blocks

Encryption

- ❑ **Encryption** – an encoding which can be reversed (given a **key**)
 - ❑ A **plaintext** (M) message is encrypted by a **cipher** ($\{\}$) to a **ciphertext** (E) using a **key** (K)

$$E = \{M\}_K$$

- ❑ Decryption is possible with the **cipher**, the **ciphertext**, **and the key**
- ❑ e.g. **AES, RSA, ECC, 3DES, ...**
- ❑ Something that's **not encryption**: **base64** (e.g. `ZS5nLiB0aGlzIGJhbG9uZXkgcm1naHQgaGVyZQ==`)

Hands-On: 10 Minute Challenge

'Decrypt' these (you're actually decoding):

d2VsY29tZSB0byBIRjIwMjA=

c2VudGluZWw=

These are base64 encoded (not encrypted).

This might seem obvious to some – but it is not uncommon to encounter base64 'encryption' in the wild.

Here's a handy set of tools for this: <http://rumkin.com/tools/>

Also python/jupyter: `import base64; base64.b64decode('xxx')`

Hashes

- ❑ (Cryptographic) Hashes – not an encoding & not reversible
- ❑ Different than the larger, general class of hash functions
- ❑ For a crypto. hash function f : given $f(x)$ you can't find (guess or calculate) x
- ❑ i.e. shouldn't be able to find input x for:
3947cdf52a551de4983746545a1affdb2b04f4a2 or
21232f297a57a5a743894a0e4a801fc3
(actually, this one is easy)

➤ aka *One-way Functions*

- aka *Random Functions*
- aka *Shortcut Functions*
- aka *One-way Compression Functions*
- aka *Digests*
- ❑ e.g. **SHA-1, SHA-256, BLAKE, ...**
- ❑ **not a cryptographic hash: MD5**

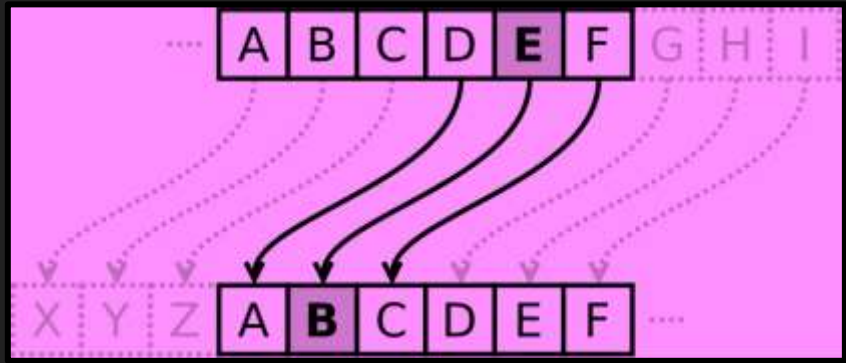
'Classic' vs Modern Crypto

❑ 'Classic' Crypto

- ❑ Mostly pre-20th century
- ❑ Deals with alphabets: input & output
- ❑ e.g. shift cipher (Cesar cipher)



- ❑ e.g. substitution cipher, polyalphabetic substitutions, transpositions etc.
- ❑ It is still encryption – the 'key' is the knowledge of the mapping (shift, letter-map etc.)
- ❑ Relevance today: puzzles, challenges and easy reverse engineering



* Matt_Crypto, wikipedia, Public Domain

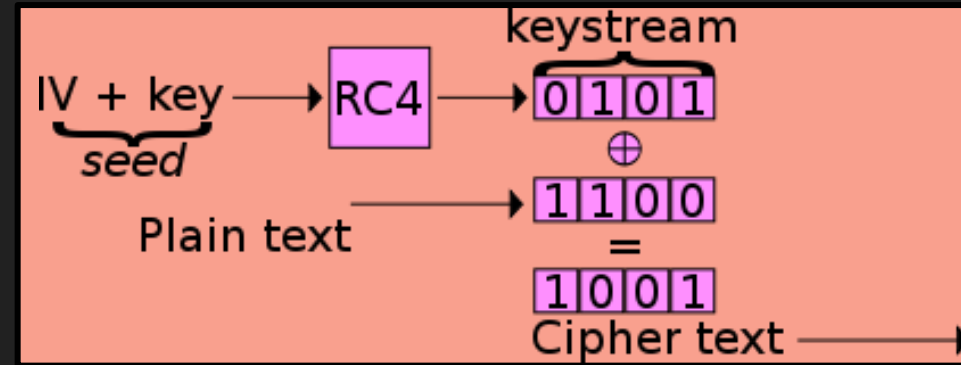
❑ Modern Crypto

- ❑ Deals with numbers: input & output
- ❑ Text is treated as numbers via encodings – ASCII or UTF-8 is the most likely encoding

e.g.
646f6e742064656369706865722074686973 \oplus (00...10) \rightarrow
646e6c77246163646179626e7e2d7a677962

Stream Ciphers

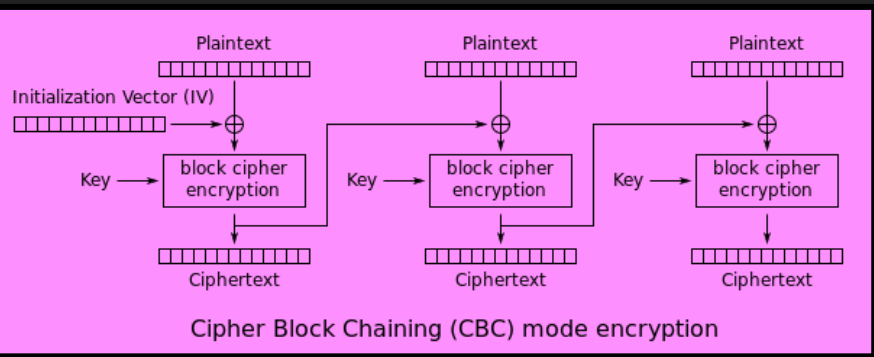
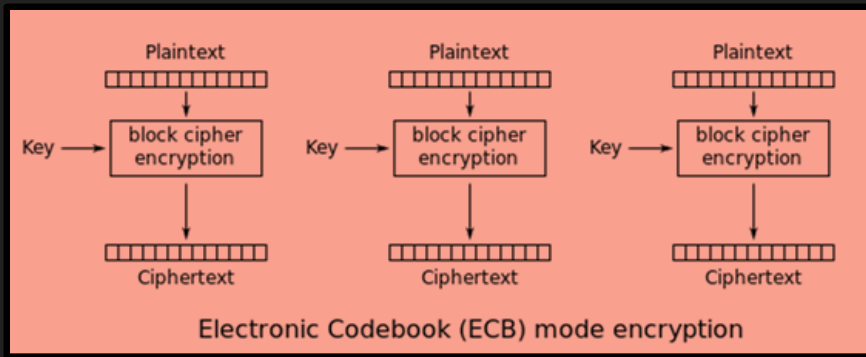
- ❑ One-Time Pad (OTP) – the only proven secure encryption scheme
 - ❑ Uses random **key-stream**, of length equal to or greater than the message
 - ❑ Then combine **key-stream** with message (assume **XOR**)
- ❑ Stream Ciphers – approximate the OTP
 - ❑ Expand short **key** into pseudo-random **keystream**
 - ❑ Then **XOR** (\oplus) (\wedge)
 - ❑ e.g. RC4, Salsa20, FISH
- ❑ note: IV – initialization vector. It shouldn't need to be secret



Di Kyle Siehl - Self-made, based on raster w:Image:Wep-crypt.png, which was taken with permission from The Final Nail in WEPs Coffin, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=1806804>

Block Ciphers

- Block Ciphers – different approach
 - Uses a **key** and **fixed-length inputs (blocks)**
 - Combined with previous outputs and more fixed-length inputs in various modes:
 - ECB, CBC, PCBC, CFB, OFB, CTR ... **GCM(!)**



By WhiteTimberwolf (SVG version) - PNG version, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=26434116>

By WhiteTimberwolf (SVG version) - PNG version, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=26434096>

Symmetric / Asymmetric Crypto

- ❑ **Symmetric Crypto** – can be encrypted + decrypted by any party with the **SAME** key
 - ❑ e.g. any of the crypto we've discussed so far
- ❑ **Asymmetric Crypto** – can be encrypted by any party *for* a specific recipient
 - ❑ aka public-key cryptography
 - ❑ Leverages certain problems that are hard in one way & easy in the other: **prime factorization** and **discrete logarithms**
 - ❑ Keys exist as pairs of **public** & **private** halves -- **key-pairs**
 - ❑ The party with the **private** key can **decrypt & sign** (more on signatures later)
 - ❑ Any parties with the **public** key can **encrypt & verify**
 - e.g. RSA, ECC
 - ❑ e.g.

```
-----BEGIN RSA PRIVATE KEY-----
izfrNTmQLnfsLzi2Wb9xPz2Qj9fQYGgeug3N2MkDuVHwpPcgkhHkJgCQuuvT+qZI
...
```


Crypto Building Blocks

Section Summary

- ❑ **Encryption**... it hides information, binds it – protects confidentiality, but not integrity (without additional effort)

$$E = \{M\}_K$$

- ❑ (Crypto) **Hashes** – one-way functions. With $f(x)$ you cannot get x
- ❑ 'Classic' **Crypto** – involves alphabets not numbers
- ❑ **Stream Cipher** – combine a sequence of key bits with a sequence of cleartext bits with XOR (\oplus) (\wedge)
- ❑ **Block Ciphers** – have a limited key stream, but extend to larger cleartext sequences
 - ❑ Not all block cipher modes are created equal (e.g. **Electronic Coloring Book (ECB)**)
- ❑ **Symmetric Crypto** – all parties share the same key
- ❑ **Asymmetric Crypto** – only one party has the decryption key (private key)

Attacks on Building Blocks

Attacking Hashes

- ❑ [Google](#).
 - ❑ Seriously... google this `21232f297a57a5a743894a0e4a801fc3` (from before) now
- ❑ Identifying what type of hash you have in-hand will be useful – the length gives it away
 - ❑ If you don't know lengths yet, use hash detector tools; e.g. [cothan/hashdetector](#)
- ❑ Hash Crack sites
- ❑ hashcat tool
 - ❑ (ab)uses your GPU for rapid hash cracking
- ❑ Rainbow Tables
 - ❑ 'halves' / parts-of hashes pre-built and ready to go
 - ❑ For things like MD5 these are trivial
 - ❑ For things like SHA-256 these are huge (multi-TB)
 - ❑ You can pick-up pre-generated tables at DEFCON Data Duplication Village. Bring a 6 TiB HDD.
- ❑ And cooler things like hash-length extension attacks

Cooler Attacks on Hashes

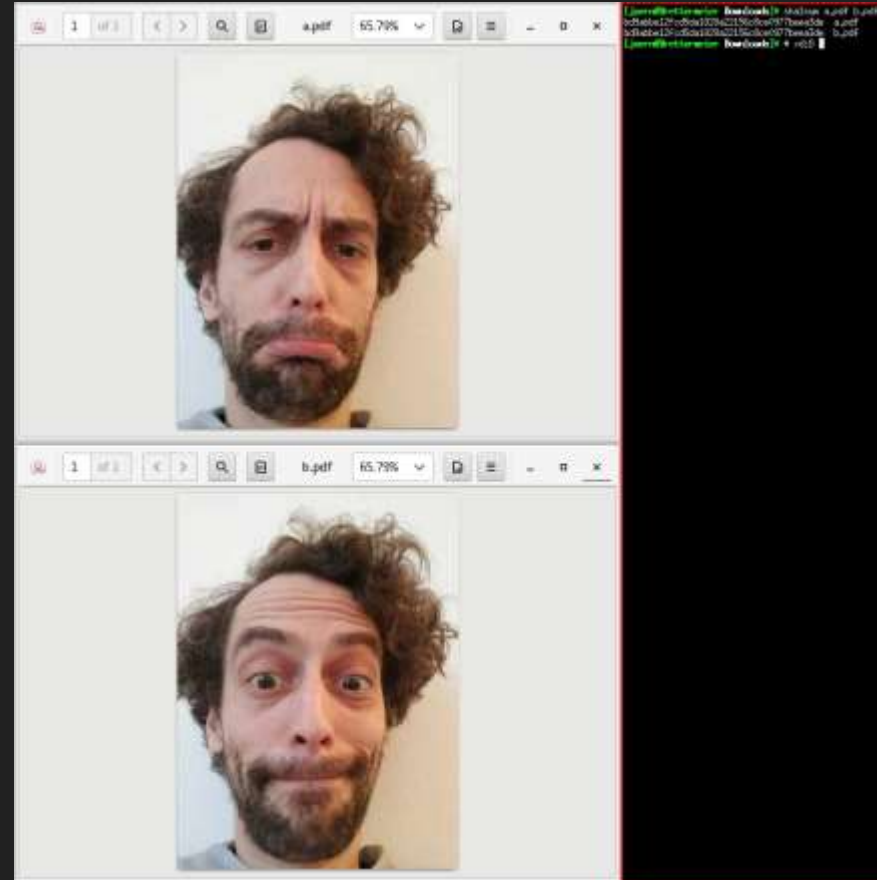
❑ Hash-Length Extension Attacks

- ❑ Take a known $H(\text{'start'})$ and add to it to get: $H(\text{'start'} + \text{junk})$
- ❑ Get to a known identical hash for 'start' and 'start' + junk

❑ Taking Advantage of File Formats

- ❑ PDF has lots of place to hide information
- ❑ See Ange Albertini's work on PDF polyglots
- ❑ This can be leveraged to create PDFs with the same SHA-1

❑ <https://shattered.io/>



More on Attacking Hashes

❑ Salts

- ❑ Because it's pretty easy to lookup or build a table of known inputs for hashes; designers tend to follow the best practice of 'salting' their inputs

 - ❑ `D033e22ae348aeb5660fc2140aec35850c4da997` = `SHA1('admin')`

 - ❑ `3947cdf52a551de4983746545a1affdb2b04f4a2` = `SHA1('saltadmin')`

- ❑ Salts are usually pre-prepended onto the input; sometimes with a separator like `.'` or `'+'`

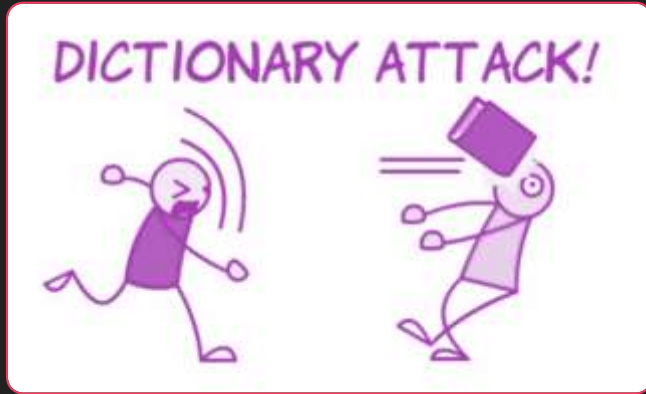
- ❑ `hashcat` can find a salt for a given hash and input pair.

- ❑ `hashcat` can also find inputs for hashes with a given salt as a parameter.

 - ❑ Find the salt with one known hash first.

 - ❑ OR find the salt with research (some systems' password salts are well-known)

Still More on Attacking Hashes



- ❑ Password lists
 - ❑ Brute-forcing (all possible character combinations) for inputs to hashes is possible
 - ❑ 'password lists' are more useful. There are hundreds of these to choose from, most from data breaches over the past years.
 - ❑ In CTFs the [rockyou](#) list is the most common – but for applied hash cracking: YMMV.
 - ❑ This is more generally known as a [dictionary attack](#)

Hands On: 15 Minute Challenge

Reverse these hashes:

- 5f4dcc3b5aa765d61d8327deb882cf99
- 5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
- ecadec2924e86bf88d622ceb0855382d
- ff4827739b75d73e08490b3380163658
- 6ce3bb6eb450df7d6345151ec00e4a4e

We've mentioned the tools you need for this.

Some are easy. One is not (hint: 2 character salt)

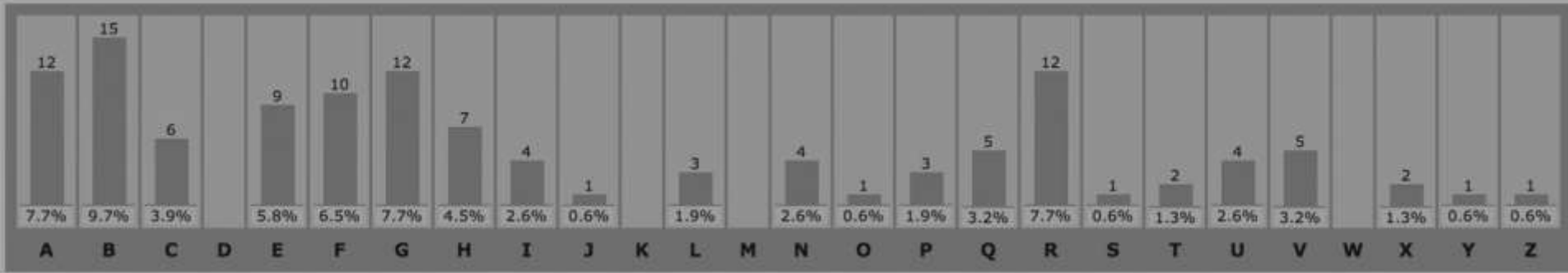
Attacking 'Classic' Crypto

- ❑ Historically, **frequency analysis** was the undoing of classic crypto
 - ❑ Letter use in a language (e.g. English) has a predictable # occurrences (frequency)
 - ❑ Count the **number of occurrences** of a symbol in ciphertext; match to expected rate in language
 - ❑ Requires medium-large ciphertext for analysis to work
- ❑ Today (challenges/puzzles/RE):
 - ❑ Try shift ciphers (start with **ROT13**)
 - ❑ Then try a substitution cipher
 - ❑ Then have '**fun**' : <http://rumkin.com/tools/cipher/>

Hands On: 10 min Classic Crypto Attack Example

Ploregehpx 2018 sbe gur jva!

*N uhtr gunax lbh arrqf gb tb bhg gb bhe fcbafbef. Guvf ceb-vaqhfge
rirag qrcraqf ba npgvir fcbafbe vaibyrzrag naq fhccbeg.*



❑ E→B? E→A ? E→R ?

❑ Try them all: <http://rumkin.com/tools/cipher/caesar.php>

❑ Shift 13 (aka ROT-13):

❑ “Cybertruck 2018 for the win! A huge thank you needs to go out to our sponsors. This pro-industry event depends on active sponsor involvement and support.”

Stream Cipher Attacks

□ Re-used Key Attack

□ Recall: it's all about XOR (\oplus) (\wedge)

□ If I know $A \wedge B$ and I know A or B , I can get the other

□ Anytime a stream cipher re-uses keys, it's a problem

□ if I have $E1 = A \wedge K$ and $E2 = B \wedge K$ I can get $A \wedge B$

□ this is a big deal if:

□ A, B are natural language (use running key cipher attacks on $A \wedge B$) or if

□ A, B are different lengths or if

□ we can control A or B or if

□ we can make any guesses about A or B

Hands On: 15 Minute Challenge

Break these stream-ciphertexts

And get the key

ad9bc999b790c281

b69895ddecce86cc

- it's all about XOR (^)
- gchq.github.io/CyberChef/ is great for playing around with XOR
- Key is 32 bits / 4 bytes
- I'm lazy (I re-use things)
- 'sentinel' ^ 0xDEFEA7ED // 'hf2020!!' ^ 0xDEFEA7ED

Block Cipher Attacks

- Getting impractical now...
- Goals: forgery or key-recovery
- Block Cipher Attack Models
 - **known plaintext**: attacker is given a set of pairs of cleartext+ciphertext
 - **chosen plaintext**: attacker has the ability to query cleartext and receive ciphertext
 - **chosen ciphertext**: attacker has the ability to query ciphertext and receive cleartext
 - **chosen plaintext/ciphertext**: attacker has the ability to query either
 - **related key**: attacker has the ability to query with key related to specified key, K (e.g. $K+1$ $K+2$, ...)

Padding Oracle Attacks

- ❑ An example **chosen ciphertext attack**:
- ❑ **Padding oracle attack**: attacker supplies ciphertext, detects 'incorrect padding' error conditions – can use this **oracle** to ultimately decrypt messages
- ❑ Surprisingly common

Cryptanalysis and More

□ Linear Cryptanalysis

- solving for **linear relationships** between cleartext (input) and ciphertext (outputs)
- at fractional likelihoods
- using the likelihoods to *sometimes predict ciphertext from cleartext*
- 'correct' crypto is designed to resist these attacks

□ Differential Cryptanalysis

- solving for **sensitivity relationships** of changes to cleartext bits (input) onto ciphertext bits (outputs)
 - at fractional likelihoods
 - then use any high likelihoods to guide attacks with chosen inputs
 - Modern 'correct' crypto is design to resist these attacks too
-
- Other Cool Stuff: Slide Attack, XSL Attack, Impossible Differential, Boomerang, ...

Reality Check

- ❑ We talked about attack **models** & attack **goals**; some families of attacks
- ❑ No simple attacks after 'Classic' crypto
- ❑ Few practical attacks
- ❑ Attacking Crypto *these ways* is hard, for '**correct**' crypto:
 - ❑ e.g. **SHA-256, AES-128, RSA-2048, ECC w/ curve 25519**
- ❑ For *incorrect* crypto (e.g. anything else)
 - ❑ Is it **XOR 'Crypto'**? → Try XOR ciphertexts together; try XORing it with good guesses too
 - ❑ OR Are there **repetitions of data patterns** in the ciphertext? Maybe it is ECB mode or maybe it is key-reuse in a stream cipher
 - ❑ OR If you know the name of the crypto, use google – maybe you will find tool or PoC to break it
- ❑ But it's not impossible
 - ❑ People build protocols out of these building blocks – **protocols get broken more often**
 - ❑ (and don't forget **side-channel attacks** and **software exploitation**)

Hands-On: 10 Minute Challenge

Decipher the following strings:

Lqydo1g#Sdvvzrug\$

Sdvvzrug#RN\$\$\$#=,

Hints:

- ❑ from the IOLI crackme challenges:

`pof.eslack.org/tmp/IOLI-crackme.tar.gz`

- ❑ 'Sdvvzrug' shows up in both strings, this tells you something






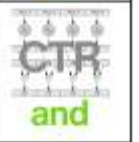



- ❑ '#' is 0x35 and '\$' is 0x36

- ❑ ' ' is 0x32 and '!' is 0x33

- ❑ `rumkin.com/tools/cipher/caesar.php`

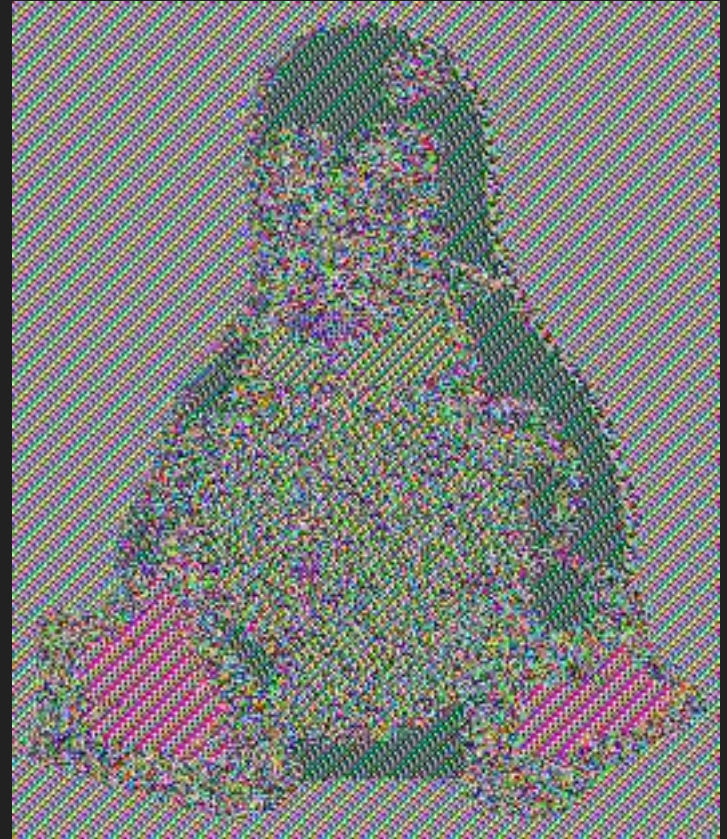
Done? Already? Do a 'beginner' challenge at potatop1a.net/crypto/

Other Attacks on Block Ciphers

 GCM All	 OFB modes	 GCM are
 XEX beautiful	 ECB not you	 CTR and
 OCB deserve	 CFB to be	 XTS used

- ❑ Recognizing ciphertext blocks can let you decrypt them:
 - maybe not to their contents, but to their meaning
 - (Sometimes also their contents; e.g. infer all-zeroes input)
- ❑ Use viz tools: vix, radare2, binvis.io, Veles, hobbits

AES_ECB() =



<https://github.com/pakesson/diy-ecb-penguin>

Other Attacking Building Blocks

- ❑ Software Exploitation can yield both **control of the software** and also **information leaks**



- ❑ Access to process memory can be fruitful **key extraction** attacks
- ❑ Multiple tools are available to scour memory for **keys**:
 - e.g. aeskeyfind, radare2, volatility
- ❑ Reverse engineering of the program code in memory can yield pointers to the memory locations of **keys**
- ❑ Don't underestimate the downplayed Infoleak vulnerabilities
 - ❑ c.f. Heartbleed



Aside: Entropy Visualization

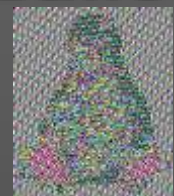
- Entropy (in the sense of C. Shannon) is a metric of information-density in message/value/bit-sequence
 - It turns out (thanks also to Shannon) that information is maximized when the likelihood of 1/0 are equal
 - i.e. 'completely random' IS highest entropy.
- The entropy of a bitsequence can be estimated
- Estimated entropy approaches 1.0 for random number sequences
 - Next-closest to 1.0 is 'correct' crypto
 - Then compressed data
- Estimated entropy is not high for other data (structured data)

Aside: Entropy Visualization (cont'd)

- ❑ The entropy estimates can be broken-up over a large input and visualized
- ❑ You can identify and distinguish between
 - ❑ encrypted (correct) content
 - ❑ Other encrypted (incorrect) content
 - ❑ Compressed content
- ❑ Rules of thumb:
 - ❑ Compression looks like pretty high entropy
 - ❑ Encryption looks like really high entropy

Aside: Entropy Visualization (cont'd)

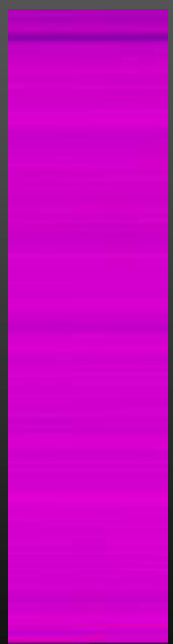
Image



AES ECB

AES CBC

binvis.io
entropy



Hands-On: 15 Minute Challenge

Use Entropy Visualization (and anything else) to Identify:

- a) A compressed file
- b) An ECB-mode encrypted file
- c) A 'correct' encrypted file

In the set: <https://goo.gl/LbzMbE>

Use these (or any other tools):

- <http://binvis.io>
- `binwalk -E`
- `Veles`
- `hobbits`
- `radare2`
- hex editors

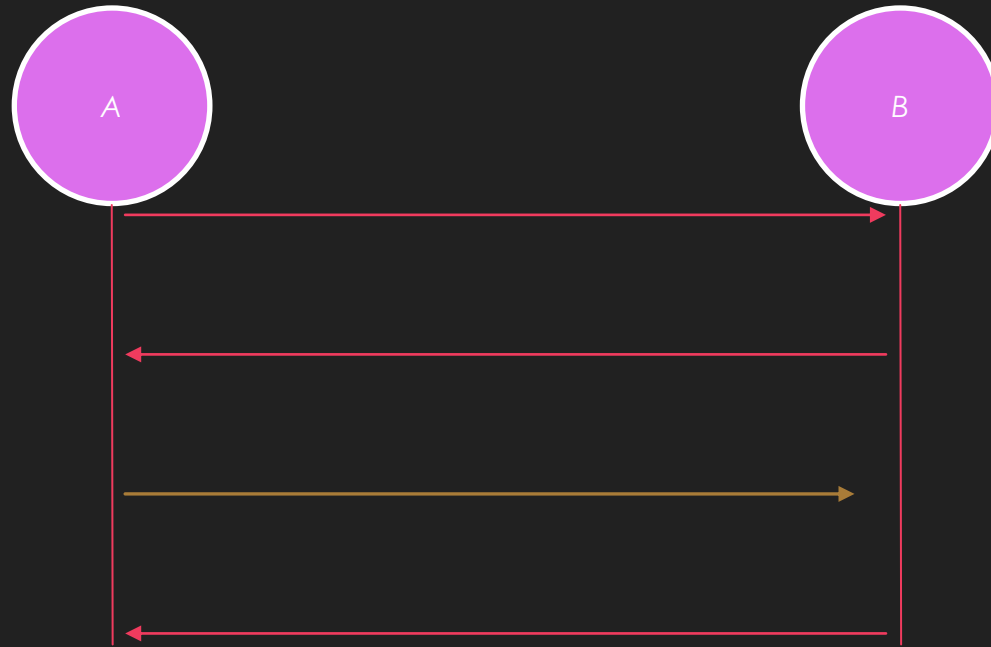
I corrupted some headers to break file magic 

Attacks on Building Blocks **Section Summary**

- ❑ **Hash Attacks** – collisions, pre-image etc. use google. All other practical (for us mortals) attacks are in hashcat, use it.
- ❑ **Classic Crypto Attacks** – frequency analysis. Try simple things first, use cryptogram tools, ID the cipher and try cipher-specific attacks
- ❑ **Stream Cipher Attack** – Reused Key Attack. i.e. try XOR (^) things together, make guesses
- ❑ **Block Cipher Attack Models** – probably impractical but use the right search terms
 - Except ECB: recognize patterns
- ❑ Don't forget about software exploitation; in-memory attacks.
- ❑ **Breaking protocols is more fruitful** (next sections)
- ❑ Remember these tools:
 - <http://rumkin.com/tools/>
 - CyberChef: <https://gchq.github.io/CyberChef/>
 - Visualization tools: binwalk -E, radare2, binvis.io, Veles, hobbits

Protocols

Protocols



- ❑ **Protocols** – the rules that govern the communication between parties
 - ❑ What information is transmitted from party A to party B?
 - ❑ What steps must party B perform?
 - ❑ What information must be sent in reply (if any)?
 - ❑ etc.

Protocol: Simple Authentication

□ Simple Authentication:

□ **Source**: wants to be authenticated by the **target**

□ **Target**: decides if **source** is authentic

□ The source sends:

- its ID (**T**) plus an encrypted concatenation of **T** and a **nonce** (**N**), with a **key** (**KT**) that could be specific to the ID and also is known to the target.



□ The target:

□ looks-up **encryption key** **KT** from given ID **T**;

□ decrypts the $\{...\}_{KT}$ and checks the **nonce** **N** hasn't been seen before.

▶ **Nonce** : Number used ONCE

(e.g. older keyfobs / garage door openers – source is the fob, target is the car or garage door.)

Protocol: Message Authentication Codes (MAC)

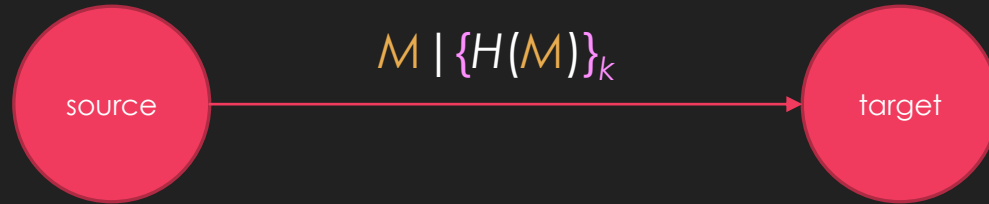
- Message Authentication Codes: for a message, create a value that can enable the message to be verified by any party with the shared key (the same shared key that is used to create the value). e.g.:
- CBC-MAC – build a MAC with CBC chaining mode of a block cipher
- CMAC – also uses a block cipher
- HMAC – build a MAC with a hash function
- CBC-MAC-AES128, HMAC-SHA1, etc.



- Parties receiving messages that don't verify against the key (shared in this case) shall discard messages
 - How the shared keys are distributed and how messages are discarded is additional protocol details (for the next layer of the protocol specification)
- aka Message Integrity Code (MIC)
- aka protected checksums
- **Not a MAC:** a message digest: $f(M)$ where f is a hash function.

Protocol: Digital Signatures

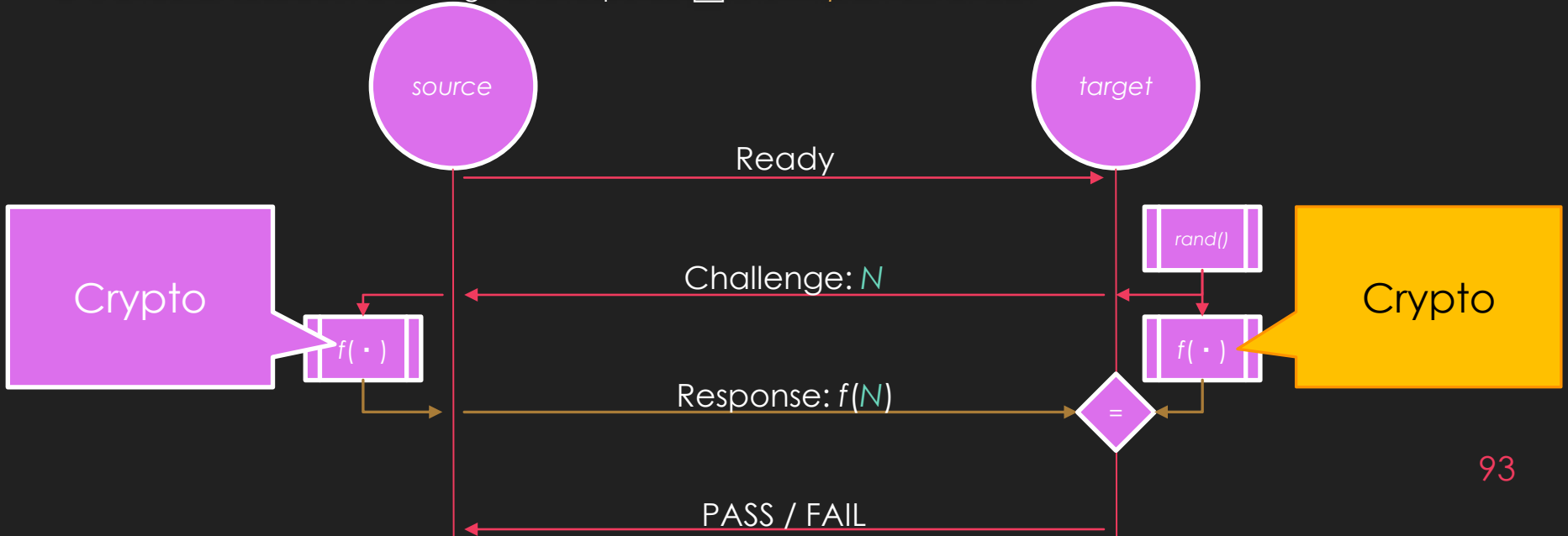
- **Digital Signatures**: using asymmetric crypto, for a message: create a value that can enable the message to be verified by any party with the **public key** but cannot be created by any party without the **private key**.
 - a signing party with a **private key** can **create** a signature
 - parties with the **public key** can **verify** that signature
- e.g. DSA, ECDSA. Let's consider a simple, older RSA signing:
 - Send message, **M**, and signature together



- To verify: Decrypt $\{H(M)\}_k$ and assert it is equal to $H(M)$, where H is a cryptographic hash and k is the RSA private key
- In both MAC and Signatures, parties receiving messages that don't verify against the **key** (**public** in this case) shall discard messages
 - How the **public keys** are distributed and how messages are discarded is additional protocol details (for the next layer of the protocol specification)
 - e.g. what if they sent: $K \mid M \mid \{H(M)\}_k$ where K is the **public key**?

Protocol: Challenge-Response (C-R)

- Source wants to be authenticated by the target
- Source receives a nonce as challenge
- Transforms it and replies as response
- An ideal C-R would make it impractical for an attacker to guess the secret by observing traffic of multiple C-R exchanges.
 - If attacker sees both challenge and response \rightarrow *known plaintext attack*



Protocols

Section Summary

- **Protocols** – the rules that govern the communications between parties
- **Digital Signatures** – can be **created** by parties with the **private key** but **verified** by anyone with the **public key** (built from asymmetric crypto)
- **Message Authentication Codes (MAC)** – can be created and verified by any party with the key (can be built from symmetric crypto)
- **Nonce** “number used once” – can be random or a counter ...
- **Simple Authentication** – source send its **ID** and an **encrypted ID+nonce** pair to a target for verification
- **Challenge Response** – target sends **nonce** to source; source replies with some proof that it has an **ID** known to the target
 - e.g. **nonce** encrypted with **key** known to source
 - e.g. **nonce** transformed with parameters known to source

Attacks on Protocols

Attacks on Protocols

- *Generally*: try to break the **assumptions** of the protocol
- This actually generalizes to “How to attack any specification”:
 - Anywhere the specification says **SHALL/SHOULD** – see what happens when it **DON'T**...

Attacks on Simple Authentication

- Simple Authentication assumes nonce N hasn't been seen before
- If the nonce is random:
 - Does it actually check? \rightarrow Send again (Replay Attack)
 - How many nonces does it store? \rightarrow Send +1 (Valet Attack)
- If the nonce is a counter:
 - How does it resynchronize? \rightarrow Try sending counter guesses (Bad counter resync attack)
- Simple Authentication assumes that the key K_T is associated with the ID T and
 - Are there other T that could associate with K_T ? \rightarrow Try sending to other target (Key collision attack)

Attacks on MAC

○ For digests

- Recall: these aren't actually MACs – but they get used that way occasionally
- Recall: you will know the input, i.e. you will have at least one digest+message pair
- You need to identify digest algorithm – length usually gives it away; also see tools like cothan/hashdetector
- You may need to identify the salt also – hashcat can do this

○ For HMAC- MD5, SHA1, ...:

- hashcat can crack the key or salt given a hmac+message pair

○ Software exploitation, 'confused deputy'

- Software exploitation could enable control of what messages are sent by a piece of SW designed to send mac+message pairs.
- Yields a successful forgery attack unless other software-integrity measures are taken.

Attack on Digital Signatures

- Recall the RSA Signature example: Send message, M , and signature together

$$M \mid \{H(M)\}_k$$

- Agreeing on the K public key for the k private key is a critical part.

- What if the protocol includes the public key K ?

$$K \mid M \mid \{H(M)\}_k$$

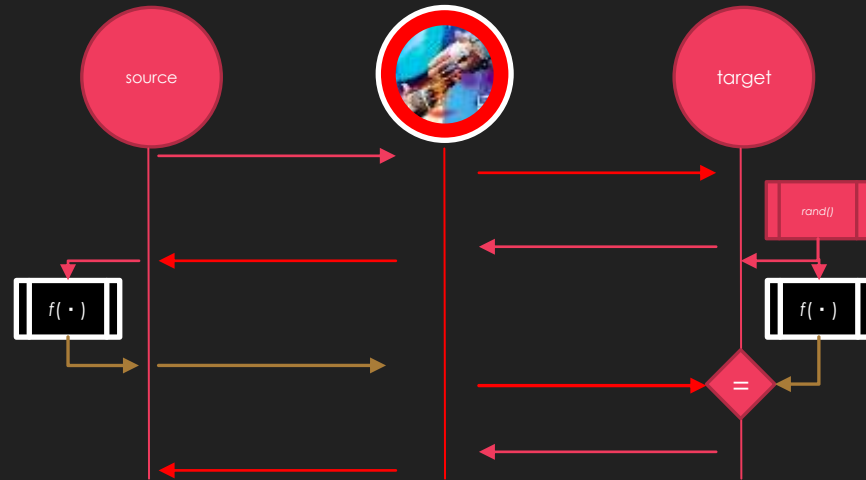
- Then an attack is to use your own private/public key pair a/A and send:

$$A \mid M \mid \{H(M)\}_a$$

- Watch out for this **broken protocol (sending the pubkey)**. It happens sometimes...
- More generally: try to find ways to substitute the expected public key K for your key, A
 - Stored in flash somewhere?

Attack on Challenge-Response: Middleperson Attack (in General)

- Interposing an actor **in-between** the source and target
 - aka **MiTM**
- Enables tampering with the contents, ordering, timing etc.
- Good concept for **attacks** on specific **Challenge-Response protocols**
- Definitely applicable in **TLS/SSL attacks** when you can interpose
- Can even be effectively achieved without physical interposition if messages can be selectively denied (e.g. **CANT** or **CANHack attacks**)



Attacks on Protocols

Section Summary

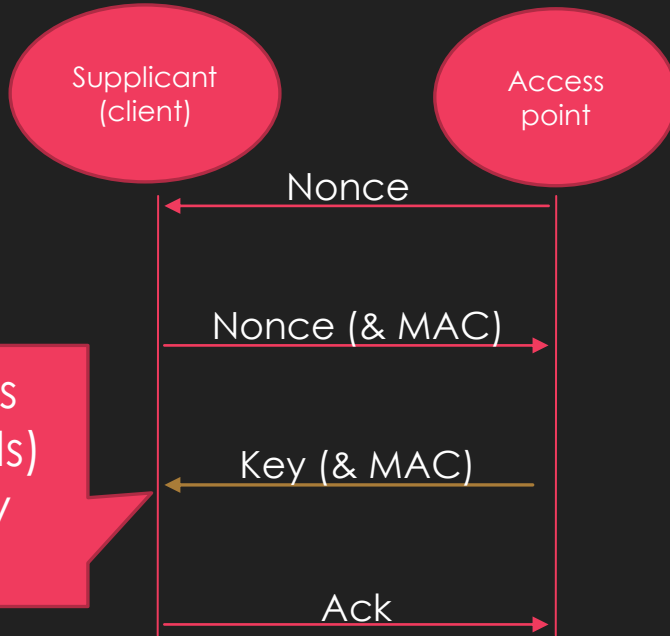
- **Attacks on protocols** are more fruitful than attacks on building blocks
- Simple Authentication Attacks
 - **Key Collisions** – e.g. 16bit serial number used as input to key
 - **Key Extraction and Extension** – e.g. Keeloq
 - **Replay Attack** – capture one or more, replay selectively
 - **Valet Attack** – capture a large set during temporary but extended possession
 - **Bad Counter Resynchronization** – depends on resync behavior of protocol
- MAC
 - **Digests (broken)**, **Hash breaking** HMACs, **shared-key reuse** for MACs
- Digital Signature Attacks
 - **Public key** substitution
- Challenge-Response Attacks
 - Middleperson Attack
 - (and more coming up in later section)

Protocol: WPA2

WPA2

- **Wi-Fi Protected Access 2**
- Wi-Fi confidentiality measure
- Supersedes WEP (which was a very broken protocol)
- WPA2-Personal (-PSK)
 - uses a pre-shared key.
 - Each client (supplicant in WPA-speak) gets its own session key
 - Setup of the key is visible at different levels.
- WPA2-Enterprise
 - Enables authentication of the Access-point
 - All communication with the Access-point is done with individualized keys
- Let's discuss WPA2-Personal

WPA2 Handshake



4-way handshake

- A nonce
- Then another nonce with MAC
- Then a global key with MAC
- Then an ACK

Grossly over-simplified

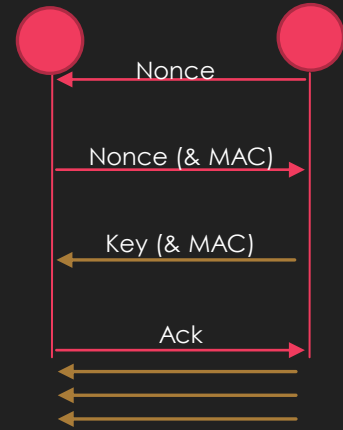
Attacks on WPA2

There is a MAC, implemented as a HMAC which is sent by supplicants and derived from the pre-shared key

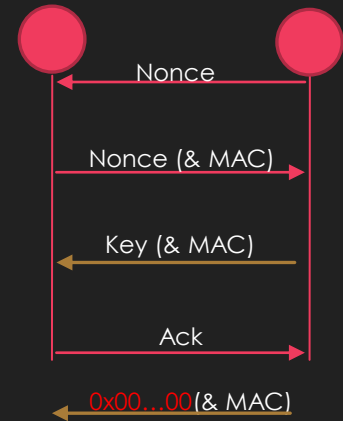
- Hash attacks to reverse this
- There are advantages to having observed multiple nonce & MAC -- so the attack starts with causing the target to deauthorize from the Wi-Fi (repeatedly)
- `hashcat` can do the cracking, but not the de-auth
- `airocrack-ng` can do both

Attacks on WPA2 (cont'd)

- There is a key reuse vulnerability in some client software, dubbed **KRACK**
 - When the key is 'installed', the client resets its communication counters
 - By **replaying message 3** in the handshake, counters can be reset repeatedly – **key reuse attack**



- Some systems were even vulnerable to installing a **null-key** by sending a tampered 3rd message
- Fun-fact: WPA2 had been **formally-proven secure**.
 - The spec of the formal proof did not include "keys must be 'installed' once and only once"



Hands-On: 1 Minute Challenge

Capture as many users of the Cybertruck Wi-Fi as you can in 1 minute.

I'm kidding – please don't attack the Wi-Fi. I'm using it.



- ❑ KRACK is unnecessary – your systems all know the WPA2 password already (it is a pre-shared key)
- ❑ How this *would* work :
 - ❑ 'de-auth' other clients so you could witness their handshake with the Access Point.
 - ❑ At which point you would have their session key and could decrypt all their traffic.

Protocol: WPA2

Section Summary

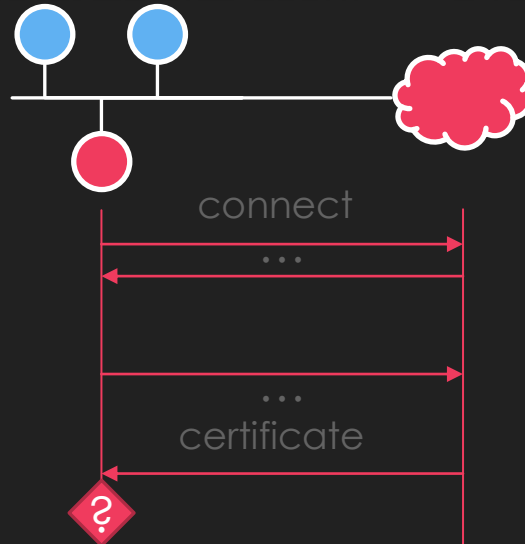
- WPA2 Passwords can be cracked, indirectly, via the hashes exposed in the handshake
 - The process is accelerated by capturing multiple 4-way handshakes, so the attack usually also includes a flood of de-authenticating the clients
- WPA2 keys can be reinstalled (KRACK)
 - Re-installing a key resets counters – this gives a key reuse attack
- Sometimes WPA2 keys can be nulled (KRACK)
 - Then follow up with known-key attack (v. simple in this case)
- These attacks on Wi-Fi require clients are connected



**Protocol:
TLS / SSL**

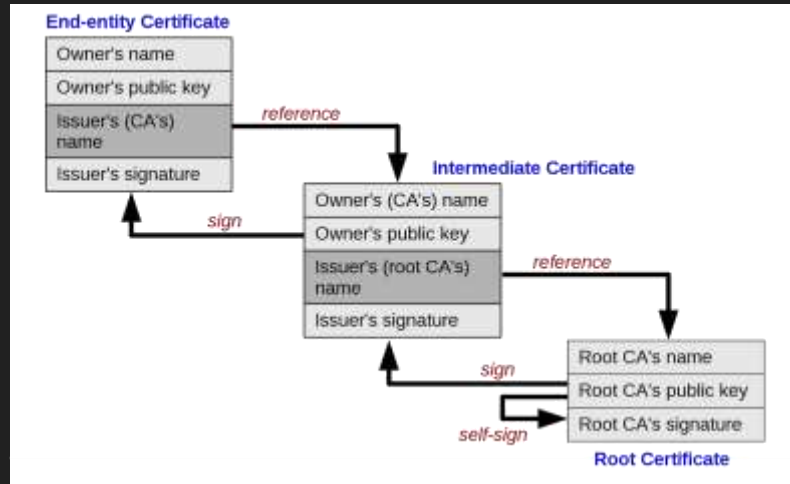
Protocol: TLS / SSL

- Transport Layer Security (TLS). Was SSL, now that name is deprecated
- Used in HTTPS – but can be found without HTTP
- Provides both confidentiality and authentication of endpoints
 - typically client authenticates server
 - Sometimes server also authenticates client -- we're not going cover this

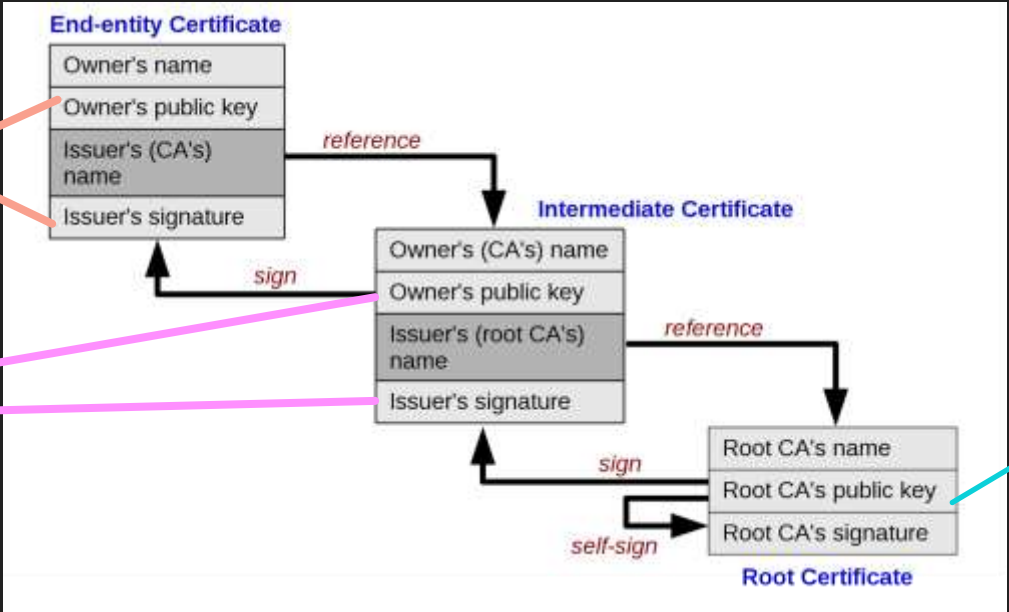
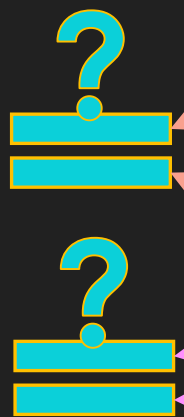


Certificates?

- Chains of Digital Signatures (asymmetric crypto)
- Recall: only the owner of the private part of a public key-pair can:
 - decrypt traffic encrypted to the public key
 - create a signature verifiable by anyone with the public key


















How Clients Are Supposed to Authenticate Servers



in?

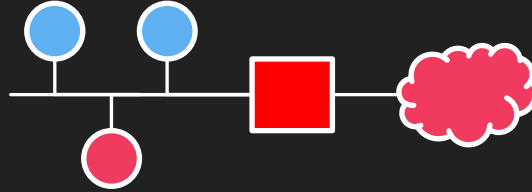
By Yanpas - Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=46369922>

Client Implementations of Server Authentication

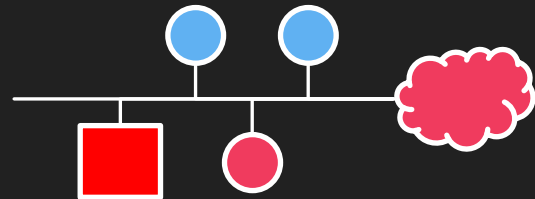
"Type"	Trust
Type 1  	Trust anything (no SSL/TLS)
Type 2  	Trust any valid certificate
Type 3  	Trust any root-CA in OS Trust Store
Type 4  	Trust only (pin) the pub key of certificate
Type 5  	Trust only (pin) the pub key of cert. signer
Type 6     	Pinning and <u>Integrity Verification</u>

NB: The proxies will work out-of-the-box on Type 1 and Type 2

Middleperson (aka MiTM) Attacks

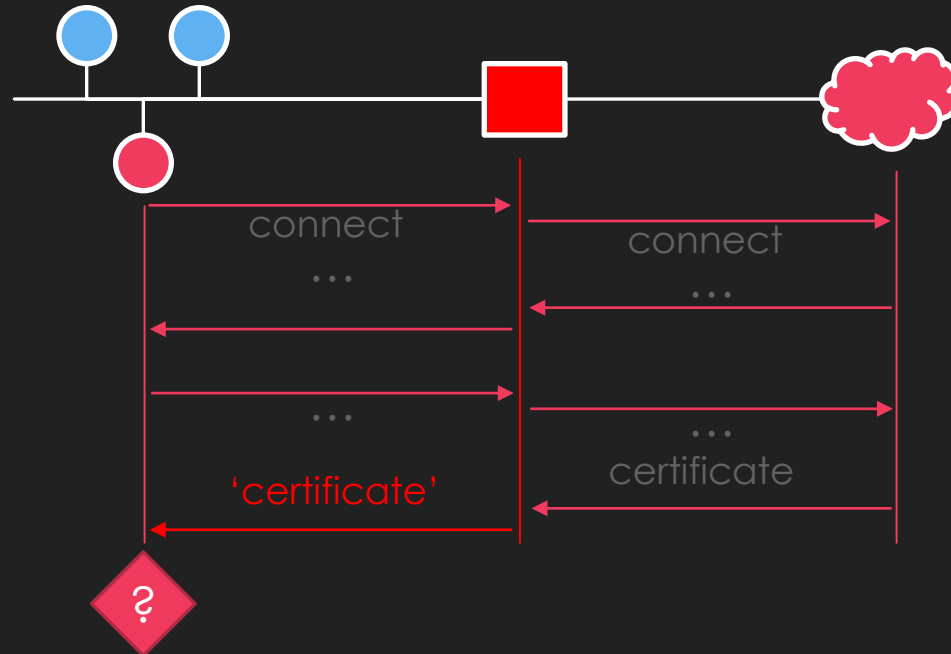


- HTTP Proxies: `mitmproxy`, Burp, ZAP, martian
- Non-HTTP: MiTMF, ettercap, bettercap, SSLSplit
- Some require that you setup the proxy as a gateway -- some can work as a sibling (leveraging ARP poisoning)



Certificate Substitution Attacks (on Type 2)

- Proxy creates two TLS connections
 - Upstream, client connection to server – normal, valid, nothing to see here
 - Downstream, served to client – supplies some other certificate
- Type 2 client sees 'a cert' and is happy



Trust Store Attacks (on Type 3)

- Can you add a root certificate authority to the trust store?
 - If you have UI access to an Android device the answer is probably yes
- Can you use a compromised root certificate that is already in the trust store?
 - There have been several compromised root certificates over the years (Komodo, Symantec)
 - If the device is old enough, the compromised root certificate might be in its trust store
 - Forge a server certificate signed with the secret from the compromise root; install that in the proxy (e.g. mitmproxy, Burp, etc.)
 - Getting the compromised secret is... the tricky part

Types 4-5 Attacks

- Recall: types 4-5 use certificate pinning – they will **only** accept a connection from a server with a **particular expected public key**
 - If a different public key is supplied they abort connections
- **Software Exploitation** is the only remote attack
- If you have superuser privileges on the systems executing the type 4-5 app then there are simple ways to replace the expected pub key or bypass the abort connection response:
 - Patch the **pubkey** from the software
 - Runtime hooking: e.g. **Universal Android SSL Pinning Bypass with Frida**
<https://codeshare.frida.re/@pcipolloni/universal-android-ssl-pinning-bypass-with-frida/>

Types 6 Attacks

- The runtime integrity checks will prevent most patches, hooks and exploits.

Other Attacks (on all types)

- SWEET32 – monitor long-lived Triple-DES and recover cookies
- DROWN – break confidentiality of some TLS (downgrade)
- Logjam – break confidentiality and integrity of some TLS (downgrade)
- POODLE – break confidentiality and integrity of some TLS (downgrade)

- Not very practical – only PoCs available : `poodle-PoC` , `Tim---/drown` , `drownAttackDemo`

- There are even passive differential cryptanalysis attacks – working only at large-scale and long time periods
 - *Recover a RSA private key from a TLS Session with Perfect Forward Secrecy – Marco Ortisi*

- Other 'other attacks' (not confidentiality or integrity compromising):
 - Heartbleed – exploit memory leak in some OpenSSL versions to view 64K of server memory (in theory could yield a server secret)

Protocol: TLS / SSL

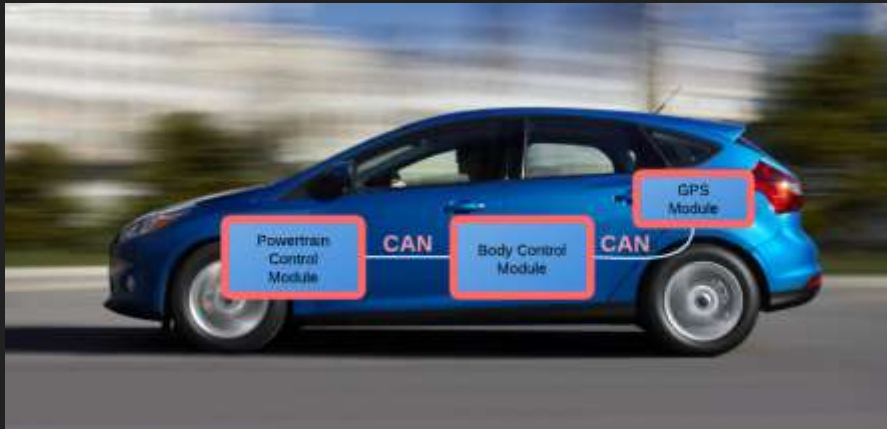
Section Summary

- TLS (SSL is deprecated) sets up a channel with confidentiality and authentication
 - Confidentiality is established with key-exchange
 - Authentication is established with certificate chain verification – the chain ultimately ending in an authority in a trust store of the endpoint
- TLS/SSL middleperson attacks require a network interposition and include:
 - Abuse of endpoints not checking certificate chains
 - Abuse of trust-stores – adding new authorities into them, or convincing users to do it
 - (rare) crypto breaks to obtain session or master keys
 - (less rare) forced downgrade to TLS/SSL version with publicly broken crypto
- Other TLS/SSL Attacks (some are aforementioned rare crypto breaks):
 - SWEET32, DROWN, logjam, POODLE, Heartbleed
- Tools:
 - mitmproxy, Burp, ZAP, MITMf
 - poodle-PoC , Tim---/drown , drownAttackDemo

Heavy Vehicle Networks Crash Course

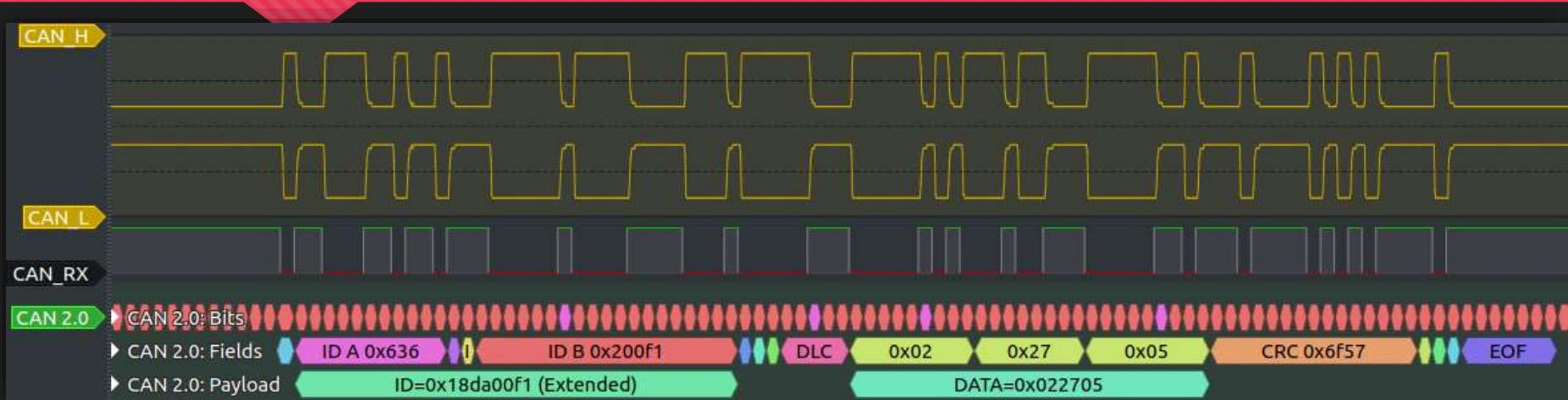
1/3 Rapid Review: CAN

- A very common bus in Automotive
- 2 wires. Serial. 250-500kbps.



2/3 Rapid Review: CAN

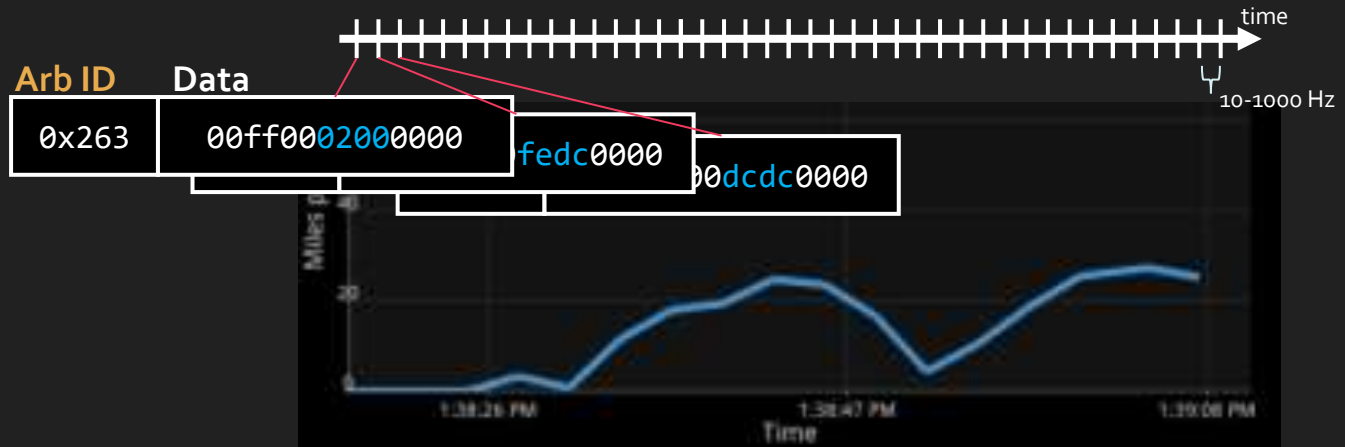
It is a ISO Levels 0-2 (ish) framing/signaling protocol.



- For GORY details, watch [When CAN CANT](#) - Tim Brom and Mitchell Johnson @ GRIMM

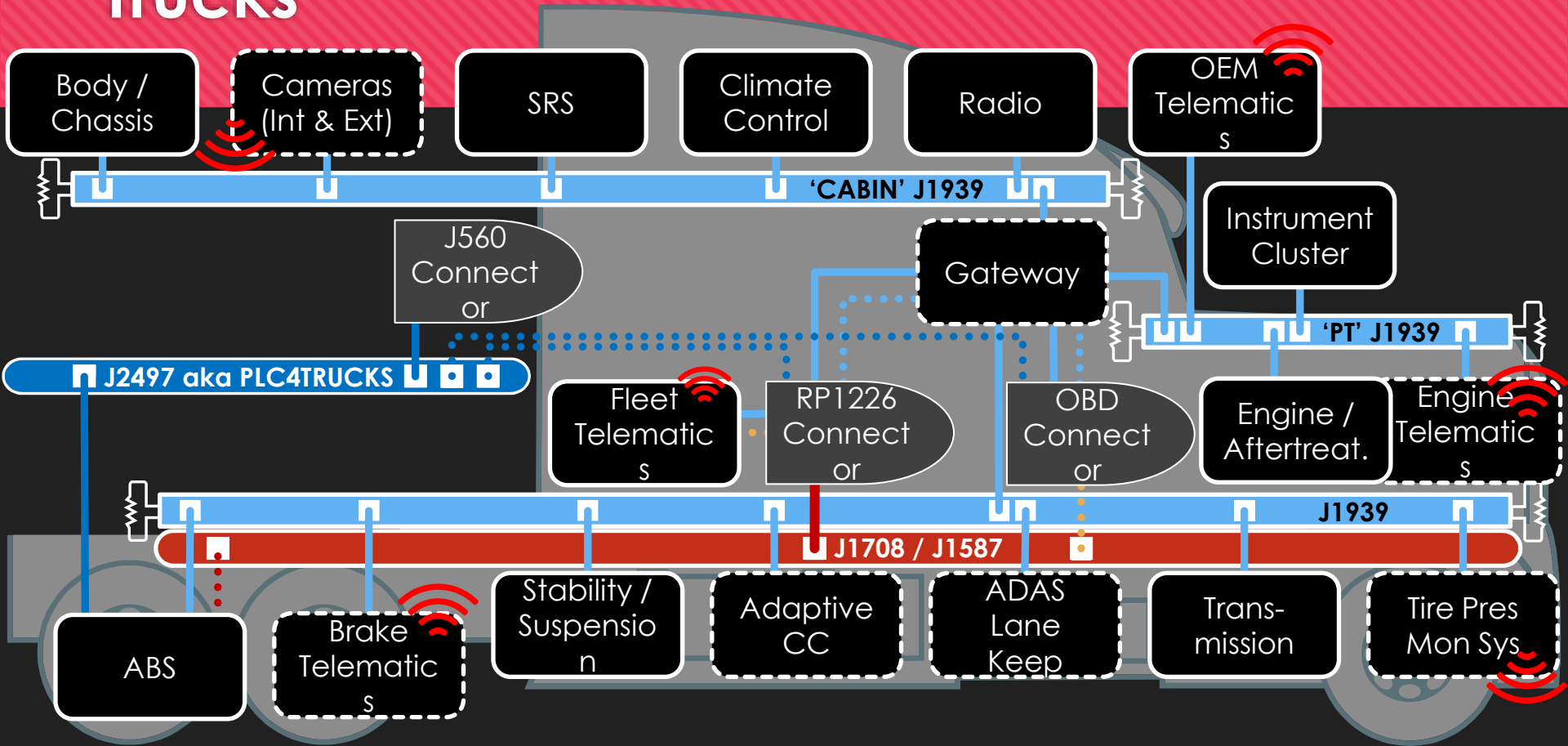
3/3 Rapid Review: CAN

- Applications commonly pack signals into bitfield locations of a frame 'type' (Arbitration ID)
 - Encoding time-varying signals by changing those bitfield contents over time

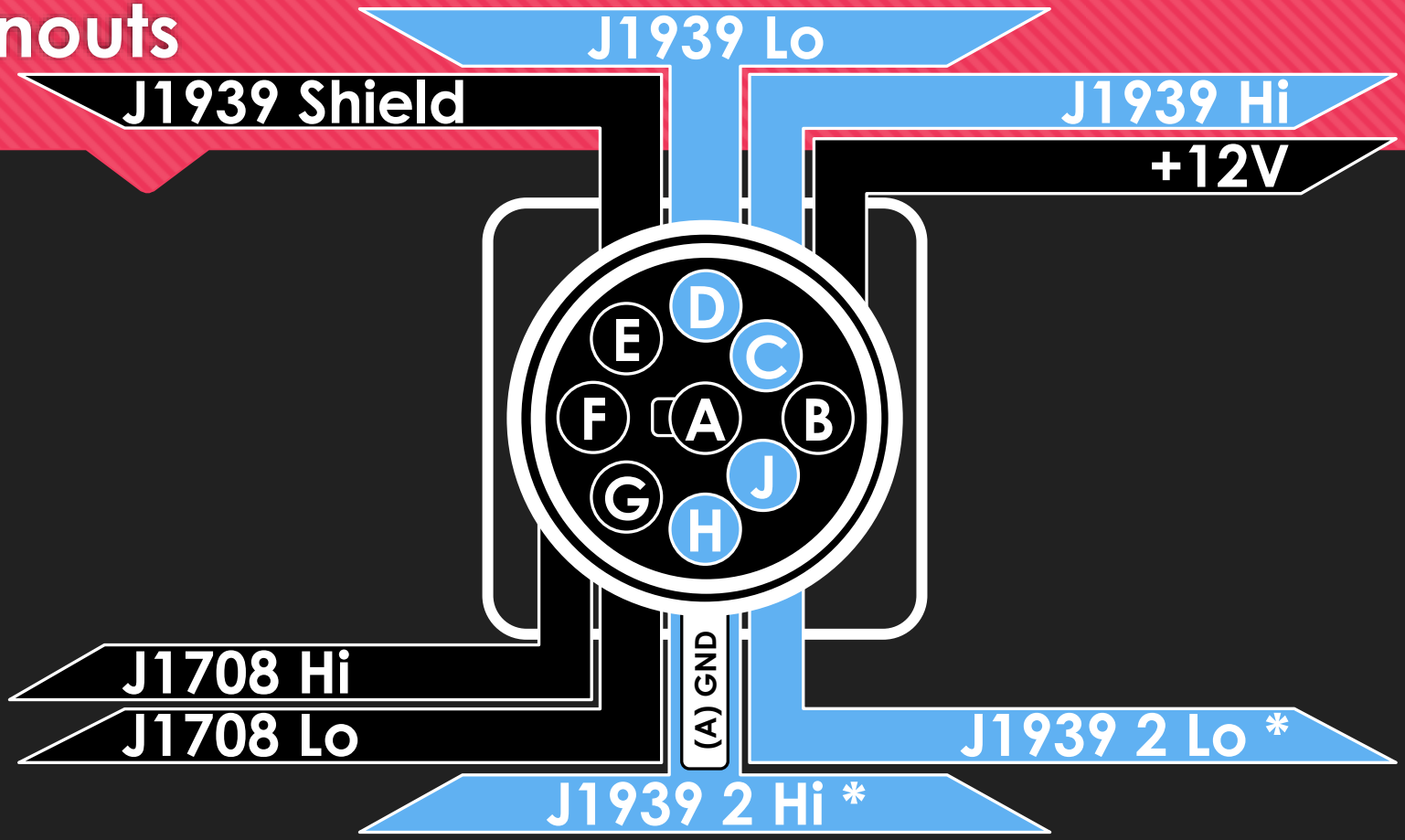


About Trucks

aka Tractors aka Power Units aka 'The things that roll'.
If it isn't moving then the fleet is losing money.



Pinouts



J1939 in relation to CAN in Passenger Cars

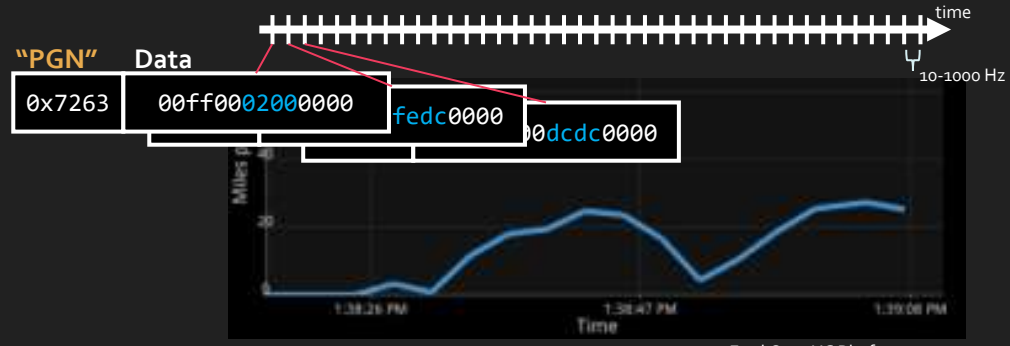
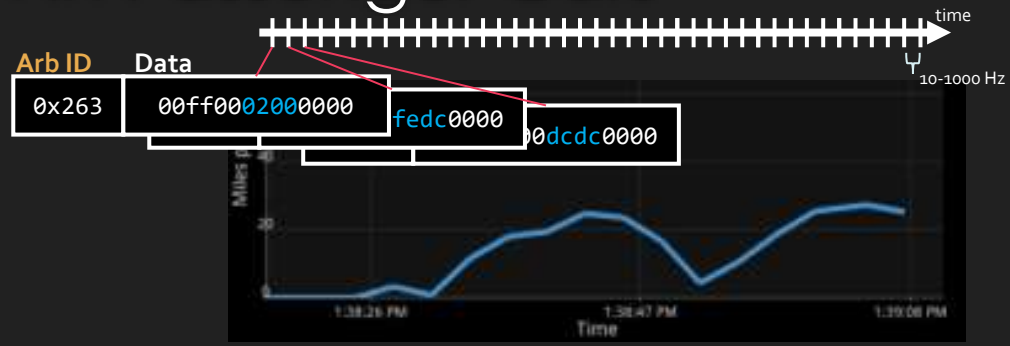
○ Both: encoding time-varying signals into bitfield locations and diagnostics

○ Passenger cars:

1. proprietary Arbitration ID,
2. proprietary bitfield locations,
3. standard diagnostics (mostly)

○ J1939:



1. standard PGNs (mostly),
2. standard SPNs (mostly),
3. proprietary diagnostics



J1939 Features

- Both unicast (PGNs < 0xF000) & broadcast (>= 0xF000)
- Transport fragmentation and reassembly (PGNs 0xEC00 and 0xEB00)
- Address claiming (PGN 0xEE00)
- Request of PGNs (PGN 0xEA00)
- Proprietary messages:
 - destination-specific (propA 0xEF00 and propA2 0x1EF00) and
 - broadcast (propB0 0xFF00-0xFFFF and propB1 0x1FF00-0x1FFFF)
- Dump, reconfigure, reflash (i.e. 'the fun stuff') is all protected by an authentication and authorization challenge-response system called *Seed-Key Exchange*
 - ISO 15765-2 aka 'ISO-TP' (PGN 0xDA00) (for UDS)

Other Things J1939 is for...

- ECU Firmware Reads
- ECU Parameter Reconfiguration
- ECU Diagnostics
-  The FUN stuff
- And of course also TRUCK stuff 
Body Control functions, Immobilizer Functions, ADAS, trailers, brakes etc.

Protocol: UDS Seed-Key Exchange

UDS

- Unified Diagnostic Services – ISO 14229 ; on CAN: ISO 15765
- Used for nearly ALL vehicle Diagnostic Protocols
- ~~○ You will learn a lot about it in other sessions today and tomorrow~~

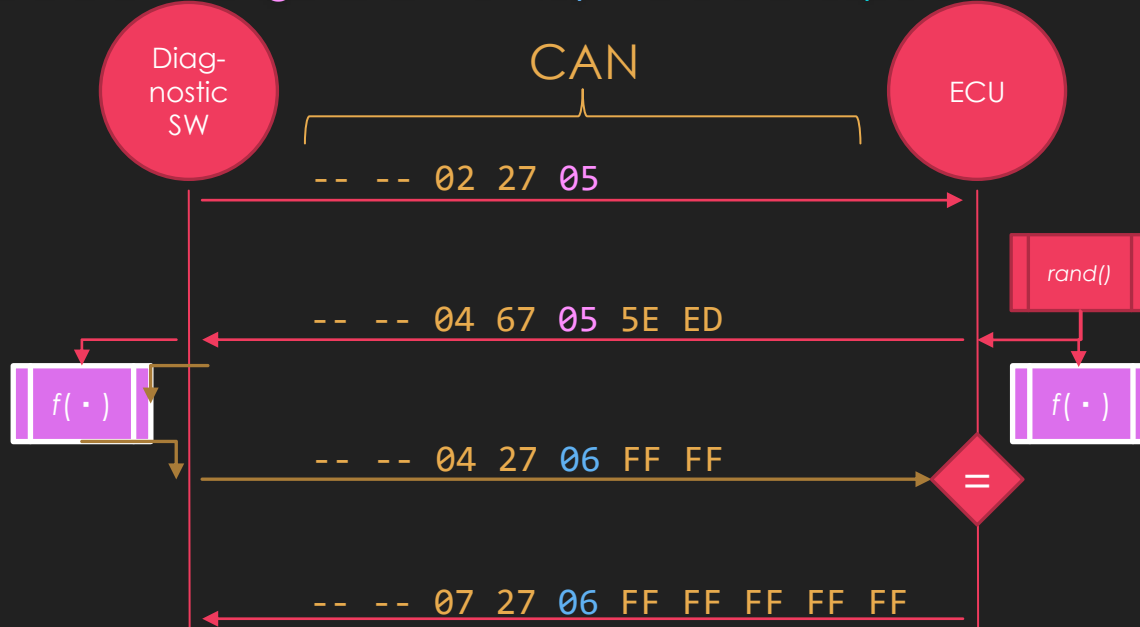
- There are actions in UDS that are protected. To execute the action requires authorization: e.g.
 - Read memory
 - Reflash ECUs
 - Perform potentially dangerous maintenance operations
 - aka 'the fun stuff'

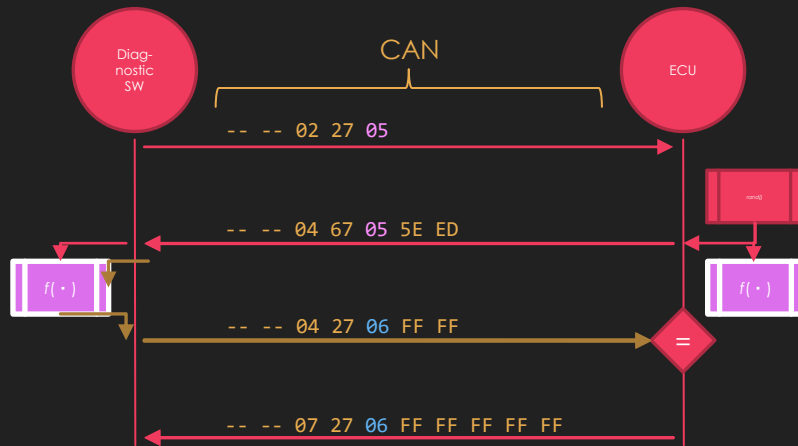
UDS Authorization

- Sometimes UDS is helpful; it will tell you that you need to authorize
 - Negative Response Code : `SecurityAccessDenied`
 - ~~○ You'll learn about these~~
- To authorize; unlock the current session with `SecurityAccess Seed-Key Exchange`
 - 'Session holder' (server) emits a '`seed`'; 'session user' (client) returns a '`key`'
 - Service `0x27` (replies on `0x67`)
 - Subfunction `0x05` for `requestSeed` / `0x06` for `sendKey`
 - You'll know more about these soon

Seed-Key Exchange

- 🔑+Seed-key exchange is a Challenge-Response Protocol
- 🔑+Only 16-bit space; so it might not fit our ideal characteristics of resisting known plaintext forgery attacks
- 🔑+The 'seed' here is a challenge and the 'key' here is a response





PT	B1	B2	B3	B4	B5	B6	B7	B8
18DA00F1	2	10	3	0	0	0	0	0
18DAF100	6	50	3	0	14	0	C8	0F
18DA00F1	3	22	F1	0	0	0	0	0
18DAF100	7	62	F1	0	2	1	0E	3
18DA00F1	2	27	5	0	0	0	0	0
18DAF100	4	67	5	81	B7	1	0E	3
18DA00F1	4	27	6	16	98	0	0	0
18DAF100	2	67	6	81	B7	1	0E	3
18DA00F1	10	0D	2E	F1	5C	0	0	0

Daily J., COMVEC15, A Digital Forensics Perspective ...

NB: J1939 IDs 0x18DA00F1 and 0x18DAF100 are used for UDS over J1939

15 Minute Hands-On: Derive the Seed-Key Routines

1

18DAF100#0467055b31
18DA00F1#0427065c31
18DAF100#0467053632
18DA00F1#0427063732
18DAF100#0467052c31
18DA00F1#0427062d31
18DAF100#0467053839
18DA00F1#0427063939

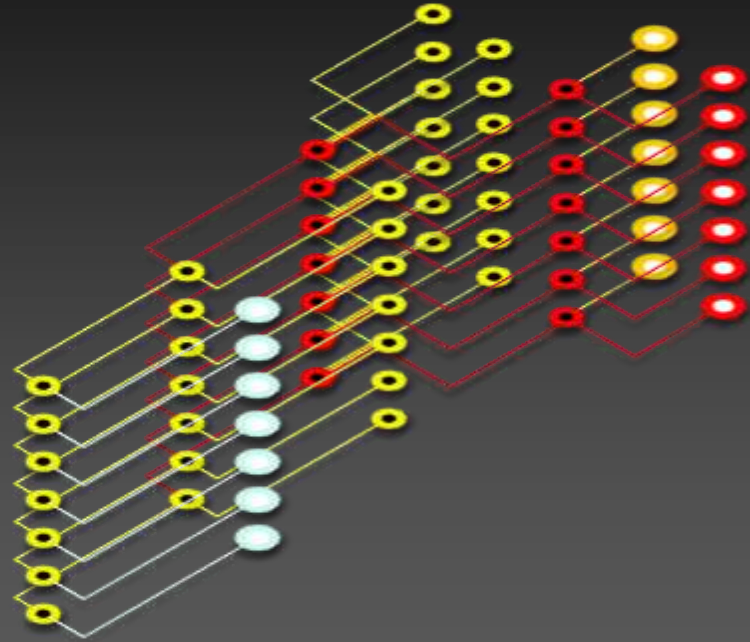
2

18DAF100#0467050100
18DA00F1#0427063435
18DAF100#0467050100
18DA00F1#0427063435
18DAF100#0467050100
18DA00F1#0427063435
18DAF100#0467050100
18DA00F1#0427063435

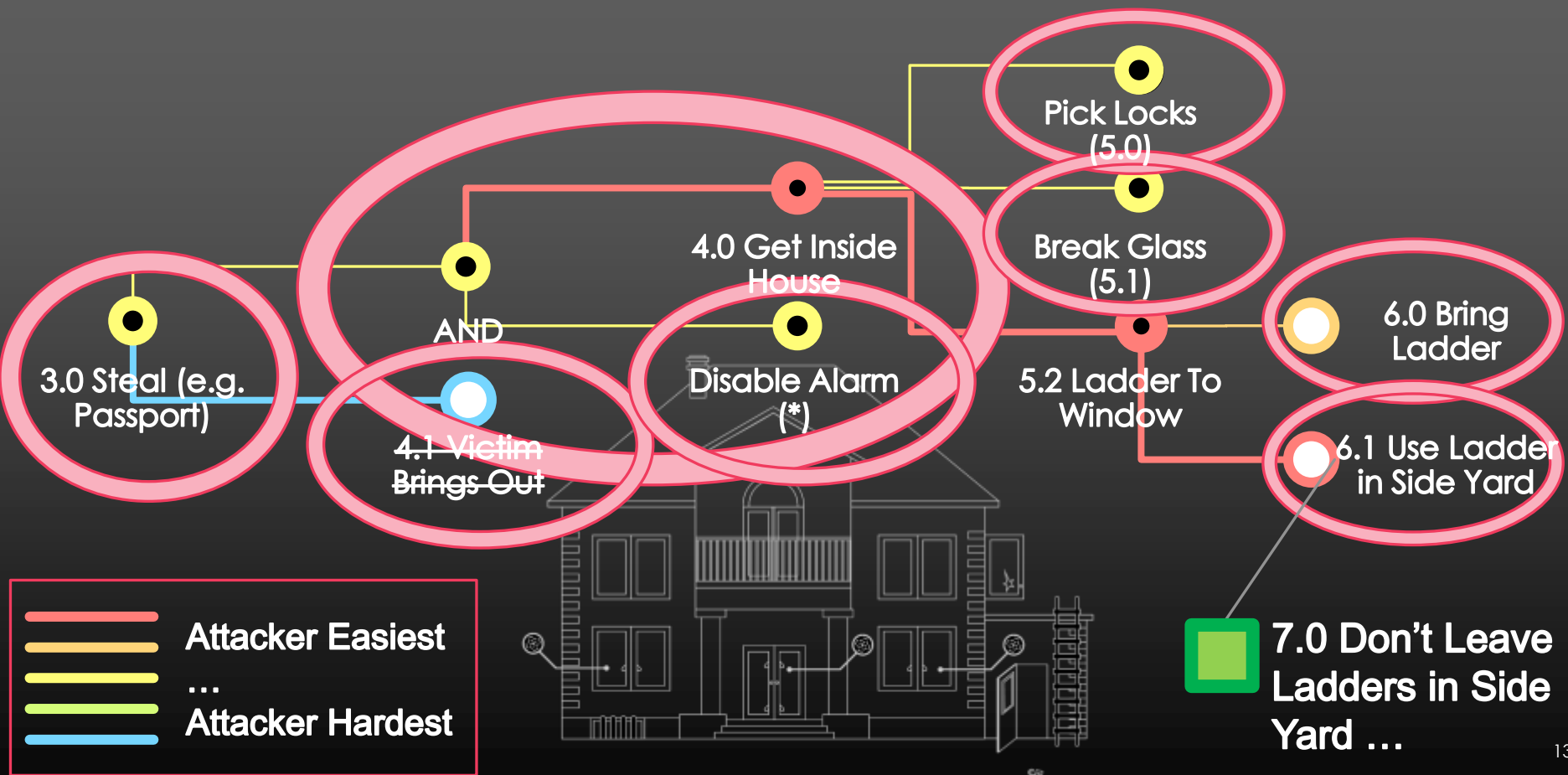
3

18DAF100#0467052c31
18DA00F1#0427060005
18DAF100#0467053132
18DA00F1#0427061d06
18DAF100#0467053732
18DA00F1#0427061b06
18DAF100#0467053137
18DA00F1#0427061d03

Aside: Attack Trees

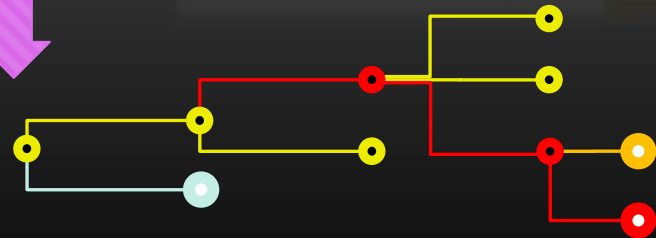
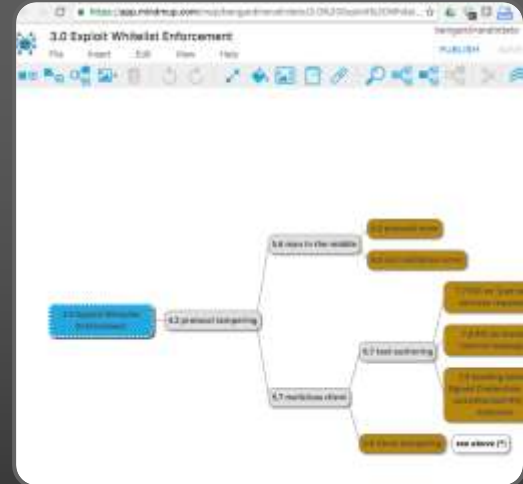


Aside: Attack Tree Notation



Aside: Tools For Attack Trees (& TARAs)

- \$\$:
 - Word Smart-Art
 - Visio
 - OmniGraffle
 - Also purpose-built commercial products
- Free:
 - Graphviz DOT: e.g. [security-decision-trees-graphviz](#) by Kelly Shortridge 🤖
 - Any indented text
 - Any mind-mapping SW; e.g. *Mindmup* (at right) using [mindmup-as-attack-trees](#)



Recall: Attacking Seed-Key Exchange

Attack Goals:

- a) Get a security session (small-scale)
- b) 'Pirate' a security session (large-scale)

For a):

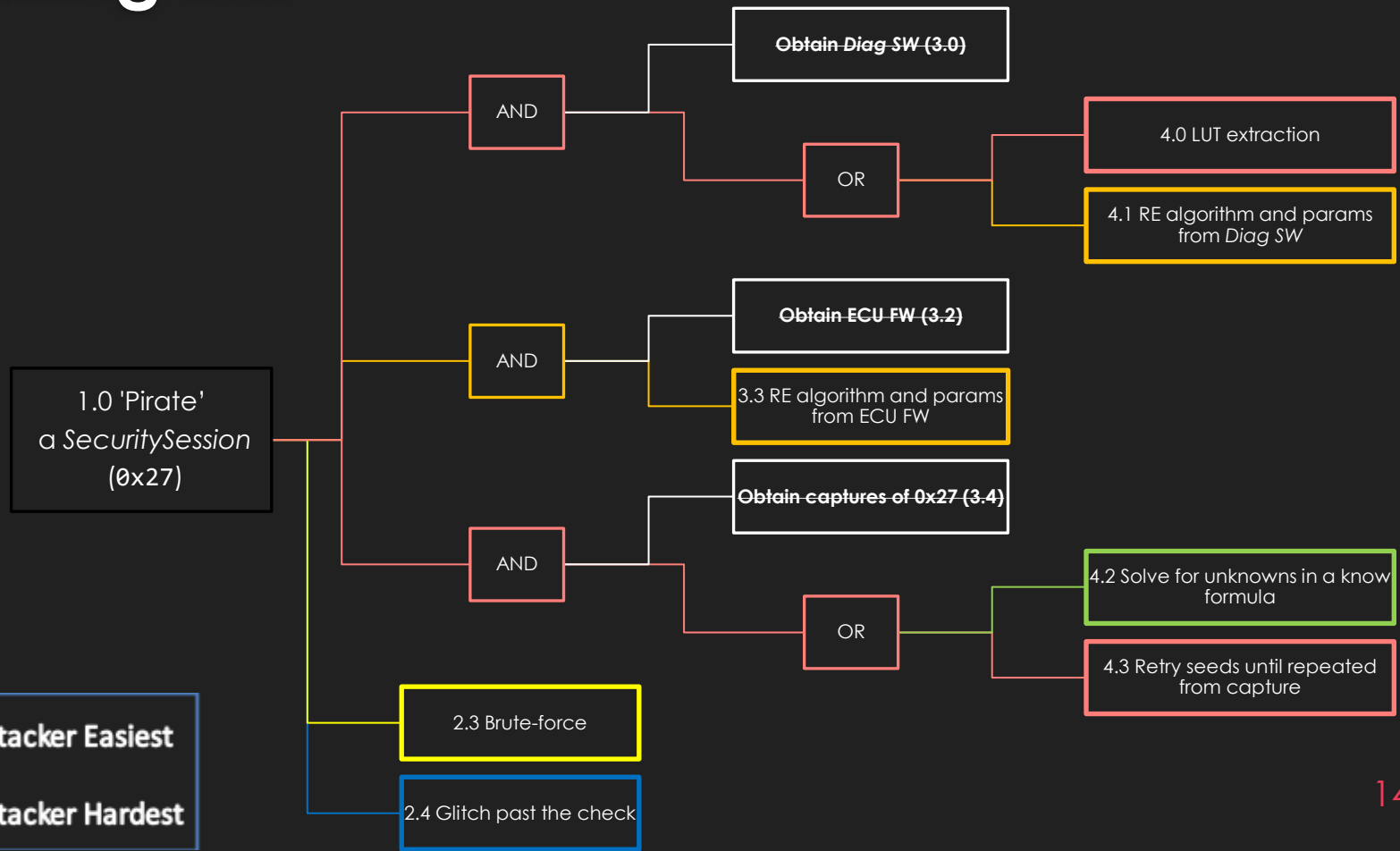
If you **do** have diagnostic SW:

- Use a middleperson attack; done.

If you **don't**:

- Then it is equivalent to *pirating a session* ; which we'll expand on...

b) Pirating it...



Look Up Table (LUT) Extraction (4.0)

16-bit seed-key exchange means a 128KiB LUT.

- Great example by John Maag of how to *lift* a challenge-response routine.

“

- Impersonate brake module using previously logged exchanges
 - Specifically, respond over CAN to software requests how the module would
- Give brake software all possible seeds and record the corresponding keys

“



John Maag, NMFTA HVCS Nov 2017

Reverse Engineering the Algorithm & Key from Maintenance SW (4.1)

```
public static string DecryptFileName(string pathName)
{
    string path2 = FileEncryptionProvider.CodeChars(FileEncryptionProvider.RearrangeCodeChars(pathName));
    return Path.Combine(Path.GetDirectoryName(pathName), path2);
}

private static byte SwitchNibbles(byte source)
{
    byte num = Convert.ToByte((int) Convert.ToByte((int) source & 15) << 4);
    source = Convert.ToByte((int) Convert.ToByte((int) source >> 4));
    source = Convert.ToByte((int) source | (int) num);
    return source;
}

private static byte ApplyKey(byte source)
{
    return source ^ (byte) 83;
}

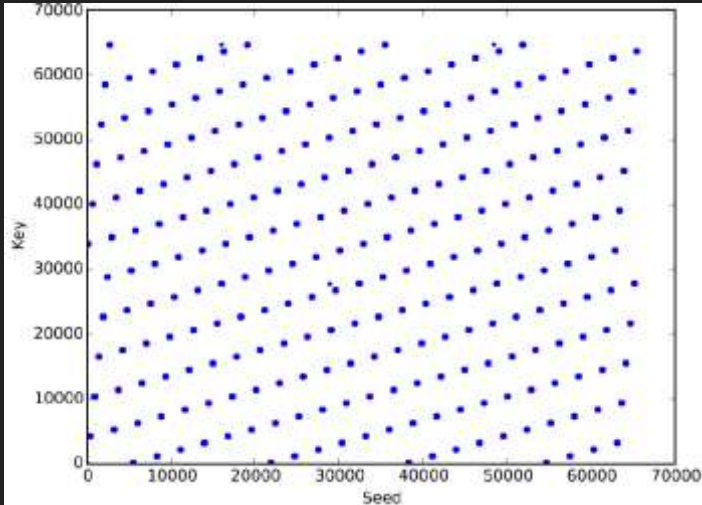
private static byte[] CodeBytes(byte[] source, int offset, bool encrypt)
{
    byte[] numArray = new byte[source.Length - offset];
    for (int index = offset; index < source.Length; ++index)
        numArray[index - offset] = !encrypt ? FileEncryptionProvider.SwitchNibbles(source[index]) : source[index];
    return numArray;
}

private static byte CalculateChecksum(byte[] data, int offset)
{
    ushort num = (ushort) 0;
    for (int index = offset; index < data.Length; ++index)
        num += (ushort) data[index];
}
```

Daily J., COMVEC15, A Digital Forensics Perspective ...

🔍† Use a decompiler or disassembler to identify the code responsible for computing the key sent in response to a seed

'Brute-Force' (2.3)



Daily J., COMVEC15, A Digital Forensics Perspective ...

- If you have no diagnostic SW, you can't lift the correct algorithm through LUT or RE.
- You can try every possible 16-bit value for each seed received.
- Worst case is 16bit x 16bit (700 weeks at 10 requests per second)
- But many seed generators won't cover all 16bits
- Can you control the seed? (e.g. hard-reset)
- single seed: (110 minutes at 10 requests per second)

'Crypto'



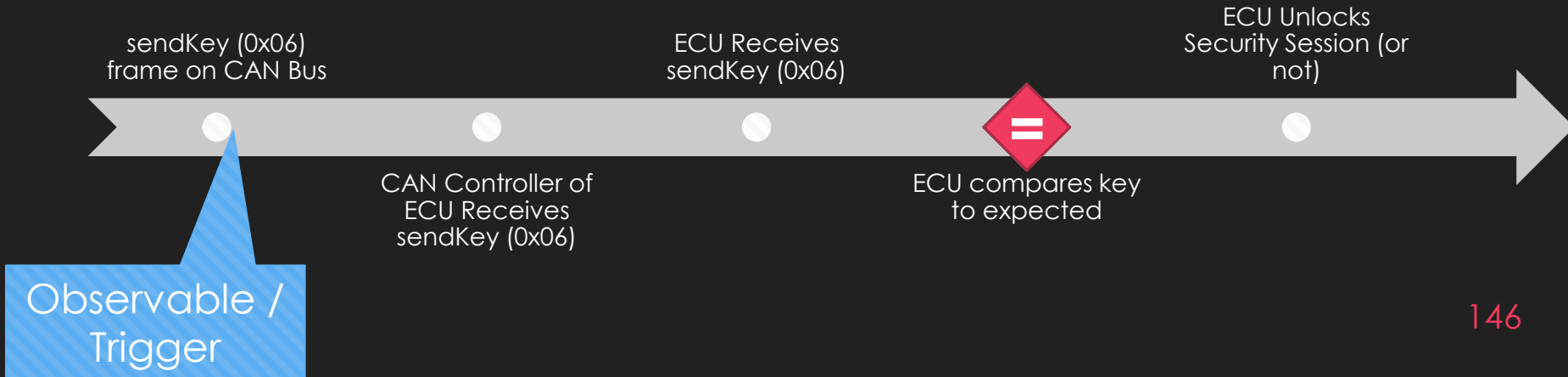
Solve for Unknown Constants in a Known Algorithm (4.2)

- If you have diagnostic software for a related ECU and access to ~10 captures of seed-key exchange
- Solve for unknown constants in known formula.

```
def sctrl(seed, key_constant_1, key_constant_2):  
    tmpSeed = 0  
    tmpSeed |= ((seed >> 16) & 0xff) << 24;  
    tmpSeed |= ((seed >> 24) & 0xff) << 16;  
    tmpSeed |= ((seed >> 8) & 0xff) << 0;  
    tmpSeed |= ((seed >> 0) & 0xff) << 8;  
    shiftSeed = ((tmpSeed << 11) + (tmpSeed >> 22)) & 0xffffffff  
    return shiftSeed ^ key_constant_1 ^ (seed & key_constant_2)
```

Glitch Past (2.4)

- If you have no diagnostic SW...
- If you can setup low-jitter triggering based on key reply CAN frame
- You can try glitching the module at delays from that trigger.
 - Maybe use simple power analysis to refine the delay
- Large search space; tricky setup (removing caps etc.)
- c.f. Riscure's recent work on demoing this at RSA/CHV



Protocol: Seed-Key Exchange

Section Summary

- J1939 IDs `0x18DA00F1` and `0x18DAF100` are used for UDS over J1939
- SecurityAccess service is `0x27` / sub requestSeed: `0x05` sendKey: `0x06`
- If you have **diagnostic software**:
 - Reverse the key algorithm & parameters from PC software
 - Black-box / Lift the key algorithm & parameters
- If you have **ECU firmware**:
 - Reverse the key algorithm & parameters from firmware image (NB: you might have the wrong direction of algorithm)
- If you have **some captures** of successful SecurityAccess:
 - Solve for unknowns in a known formula from related ECUs
 - Retry seeds until a match occurs with one in the captures
- If you have **only the ECU**:
 - Brute-force (can you control the seed?)
 - Get some captures (e.g. service center) – see above
 - Glitch past the check – be amazing

Protocol: Seed-Key Exchange

Section Summary

- J1939 IDs `0x18DA00F1` and `0x18DAF100` are used for UDS over J1939
- `SecurityAccess` service is `0x27` / sub requestSeed: `0x05` sendKey: `0x06`
- If you have **diagnostic software**:
 - Reverse the key algorithm & parameters from PC software
 - Black-box / Lift the key algorithm & parameters
- If you have **ECU firmware**:
 - Reverse the key algorithm & parameters from firmware image (NB: you might have the wrong direction of algorithm)
- If you have **some captures** of successful `SecurityAccess`:
 - Solve for unknowns in a known formula from related ECUs
 - Retry seeds until a match occurs with one in the captures
- If you have **only the ECU**:
 - Brute-force (can you control the seed?)
 - Get some captures (e.g. service center) – see above
 - Glitch past the check – be amazing

Closing

Summary

- 'Modern' crypto is about numbers / Classic 'crypto' is about alphabets
- 'Crypto is hard' → means correct crypto is hard to break, if you have only the capture of communications
- Crypto building blocks don't get broken very often (given only the capture of comms)
- Crypto *protocols* get broken
- Crypto gets broken via side-channels
- Crypto gets broken by compromise of execution environment
- You can middleperson-attack TLS/SSL
- You can lift/reverse/solve/brute-force Seed-Key Exchange

Resources for Continued Learning

- *Cryptopals (CTF)*, T. Ptacek et. al.
- *Let's Play with Crypto (Pres.)*, Ange Albertini
- Any and all *SO answers* by Thomas Pornin
- *Security Engineering (Book)*, Ross Anderson
- *PotatoSec Crypto Puzzle Challenges*
- POC | |GTF0 (Journal), *mirror*