

Sharing Keys in Plain View

Using Elliptic-Curve Diffie-Hellman.

Lab6

Key management: Elliptic Curve Diffie-Hellman
with
Key Derivation Function (KDF)

Remember

There is no 100% security

Security, like all engineering, involves tradeoffs

Know what you are trying to secure

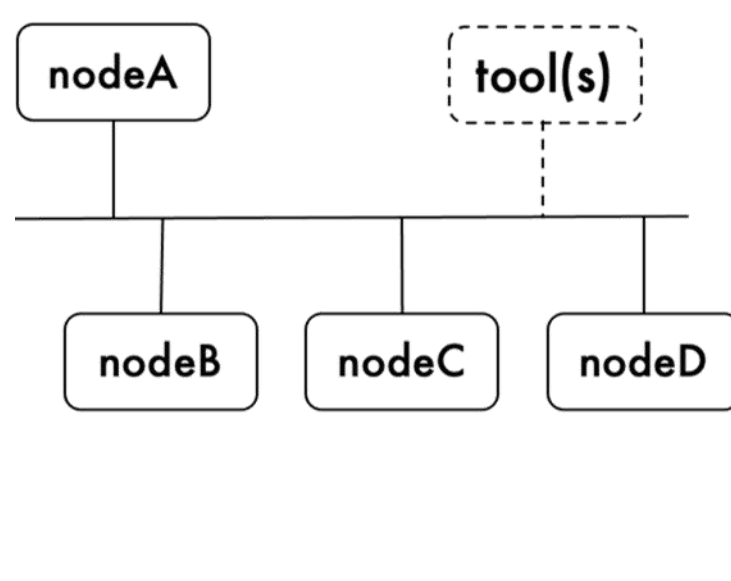
The adversary...



**State
Sponsored**

Network Configuration

Network for key demonstration



Simple Network for this Lab

1Hz : 1 Hz generators of J1939 messages

MitM : Man-in-the-Middle

ECU : ECU added (& controlled) by student

TIC : Text Instrument Cluster

tool(s): One or more of can-utils (canplayer, candump, cansniffer, etc.)

required

optional

adapter A : security adapter that validates secure messages before passing them to the TIC for decoding

adapter B : security adapter that secures messages before sending them on the bus

Elliptic-Curve Diffie-Hellman

Alice and Bob agree ahead of time to use:

- 1) **Curve25519** for their key agreement elliptic curve.
 - This gives them common parameters (G, p).
 - $p = 2^{255}-19$ (prime number; this is where the name comes from Curve25519)
- 2) A specific hash function (say SHA512/256)

ALICE

Generate random number, is it as private key: K_A

Calculate public key: $P_A = K_A \bullet G \pmod{p}$

Alice knows K_A and got P_B from Bob

Multiply to get shared secret = $K_A \bullet K_B \bullet G \pmod{p}$

Key, $S_{AB} = \text{hash}(\text{shared secret})$

BOB

Generate random number, is it as private key: K_B

Calculate public key: $P_B = K_B \bullet G \pmod{p}$

Bob knows K_B and got P_A from Alice

Multiply to get shared secret = $K_A \bullet K_B \bullet G \pmod{p}$

Key, $S_{AB} = \text{hash}(\text{shared secret})$

References

RFC for Curve25519

<https://www.rfc-editor.org/rfc/rfc7748>

Christof Paar (part of an entire semester course on cryptography)

<https://www.youtube.com/watch?v=vnpZXJL6QCQ>

Robert Pierce; Math intensive

<https://www.youtube.com/watch?v=F3zzNa42-tQ>

Bill Buchanan; Code in Python

https://www.youtube.com/watch?v=o9AdiGjOb_I

The Point

- Exchange public information
 - Eve can observe complete exchange
 - Arrive at a key that only Alice and Bob know
-
- One Hard Part: Knowing I have Alice's public key
 - Perhaps Eve is pretending to be Alice...

Notes – sometimes libraries are confusing

Mixing up
terms!?

`shared_key()`

[\[source\]](#)

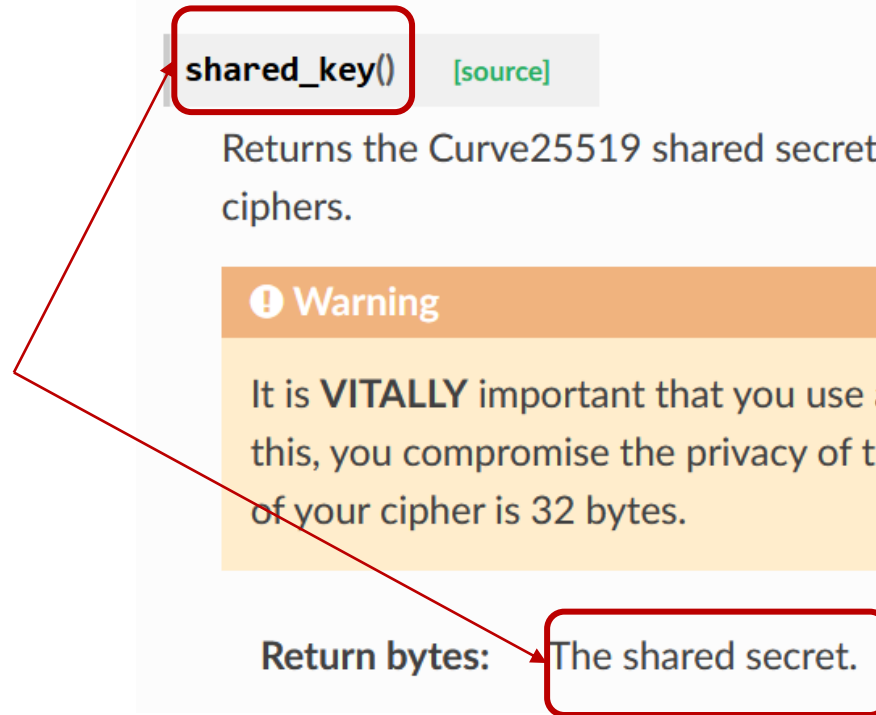
Returns the Curve25519 shared secret, that can then be used as a key in other symmetric ciphers.

⚠ Warning

It is **VITALLY** important that you use a nonce with your symmetric cipher. If you fail to do this, you compromise the privacy of the messages encrypted. Ensure that the key length of your cipher is 32 bytes.

Return bytes:

The shared secret.



Notes – The Curve25519 paper

A **hash** of the shared secret $\text{Curve25519}(a, \text{Curve25519}(b, \underline{9}))$ is **used as the key** for a secret-key authentication system (to authenticate messages), or as the key for a secret-key authenticated-encryption system (to simultaneously encrypt and authenticate messages).

Can use a Key Derivation Function (KDF) as the hash

Notes

! Danger

This is a “Hazardous Materials” module. You should **ONLY** use it if you’re 100% absolutely sure that you know what you’re doing because this module is full of land mines, dragons, and dinosaurs with laser guns.

Key derivation functions

Key derivation functions derive bytes suitable for cryptographic operations from passwords or other data sources using a pseudo-random function (PRF). Different KDFs are suitable for different tasks such as:

- Cryptographic key derivation

Deriving a key suitable for use as input to an encryption algorithm. Typically this means taking a password and running it through an algorithm such as `PBKDF2HMAC` or `HKDF`. This process is typically known as [key stretching](#).

- Password storage

When storing passwords you want to use an algorithm that is computationally intensive. Legitimate users will only need to compute it once (for example, taking the user’s password, running it through the KDF, then comparing it to the stored value), while attackers will need to do it billions of times. Ideal password storage KDFs will be demanding on both computational and memory resources.



Notes

ConcatKDF

```
class cryptography.hazmat.primitives.kdf.concatkdf.ConcatKDFHash(algorithm, length, otherinfo)
```

[\[source\]](#)

New in version 1.0.

ConcatKDFHash (Concatenation Key Derivation Function) is defined by the NIST Special Publication [NIST SP 800-56Ar2](#) document, to be used to derive keys for use after a Key Exchange negotiation operation.

Warning

ConcatKDFHash should not be used for password storage.

Notes

NaCl implement Curve25519

- Use NaCl to derive **sharedsecret**
- Use concatKDF() to derive **sharedkey**
 - Use 'otherinfo' with jointly created nonce – allows same Public keys to create new symmetric keys.

Notes

```
> cat -n using_kdf.py
1  #
2  # reference:
3  # https://pynacl.readthedocs.io/en/latest/public/#nacl.public.Box
4  # https://cryptography.io/en/latest/hazmat/primitives/key-derivation-functions/#concatkdf
5
6  import nacl.utils
7  from nacl.public import PrivateKey, Box
8
9  from cryptography.hazmat.primitives import hashes
10 from cryptography.hazmat.primitives.kdf.concatkdf import ConcatKDFHash
11
12 # notes on key naming
13 # P/Kx  -- Public/Private keypair for entity x
14 # Sx    -- Symmetric key for entity x
15
16 # Generate Bob's private key (!! keep this secret !!)
17 Kb = PrivateKey.generate()
18
19 # Use the private key to create the public key -- this must be shared
20 Pb = Kb.public_key
21
22 # Alice does the same thing
23 Ka = PrivateKey.generate()
24 Pa = Ka.public_key
```

Notes

secret → key

```
26 # By sharing just their public keys with each other,
27 # Alice and Bob have enough to derive a shared secret
28
29 # Alice knows her private key and Bob's public key
30 boxerAB = Box(Ka, Pb)
31 sharedsecret = boxerAB.shared_key() # CONFUSING... the library docs say
32                                     # we are getting both the shared secret and a key.
33                                     # We are going to assume it is just the shared secret
34                                     # and that we still need to generate the key.
35
36 # since we want an AES-128 key we still need to do a KDF and get 16 bytes
37 ckdf = ConcatKDFHash(algorithm=hashes.SHA256(), length=16, otherinfo=b"00000000")
38 sharedkey = ckdf.derive(sharedsecret)
39
40 print("[Alice] the shared key is: %s" % (sharedkey.hex(" ", 4)), flush=True)
41
42 # Bob knows his private key and Alices's public key
43 boxerBA = Box(Kb, Pa)
44 sharedsecret = boxerBA.shared_key()
45 # since we want an AES-128 key we still need to do a KDF and get 16 bytes
46 ckdf = ConcatKDFHash(algorithm=hashes.SHA256(), length=16, otherinfo=b"00000000")
47 sharedkey = ckdf.derive(sharedsecret)
48
49 print(" [Bob] the shared key is: %s" % (sharedkey.hex(" ", 4)), flush=True)
```

otherinfo

```
> python3 using_kdf.py
[Alice] the shared key is: 3bf536db 46788954 56450fb0 6e27514a
[Bob] the shared key is: 3bf536db 46788954 56450fb0 6e27514a
```

Notes

entropy

key
derivation

```

1  from cryptography.hazmat.primitives import hashes                # for SHA256 hash
2  from cryptography.hazmat.primitives.kdf.concatkdf import ConcatKDFHash # for kdf
3  import secrets                                                    # for randbits
4
5  x = secrets.randbits(8)
6  print(x)
7
8  x = secrets.randbits(64)
9  xb = x.to_bytes(8, "big")    # converts 64 bits into 8 bytes
10 print(xb.hex(" ", 4))        # example: 61e015f7 920a8f0a
11
12 y = secrets.randbits(64)
13 yb = y.to_bytes(8, "big")    # example: 13a56ccd bed1a7c8
14
15 zb = xs + ys
16 print(zb.hex(" ", 4))        # example: 61e015f7 920a8f0a 13a56ccd bed1a7c8
17
18 Sn = bytes.fromhex("00000000 11111111 22222222 33333333")
19
20 keymaterial = Sn + zb
21 print(keymaterial.hex(" ", 4)) # example: 00000000 11111111 22222222 33333333 61e015f7 920a8f0a 13a56ccd bed1a7c8
22
23 sv = kdf.derive(keymaterial)
24 print(sv.hex(" ", 4))        # example: fb2e494f df1c65a1 01e8649f f59f7bd6

```

Lab

- The provided nodeA.py and nodeB.py **MUST BE FIXED**.
 - They need to include a KDF.