TorComm - Secure P2P Communication

Documentation

Taha Canturk
kibnakanoto@protonmail.com

2024-05-20

Contents

1 Networking

1.1 one on one

1.2 groupchat

2 Cryptography

The key communication protocol used is Elliptic Cryptography Diffie Hellman ($ECDH$)

The encryption protocol used is Elliptic Cryptography Integrated Encryption Scheme ($ECIES$)

There are currently 140 cryptographic protocols to choose from. the protocol consists of all cryptographic algorithms required for a secure communication

i.e. Hashing algorithm, cipher algorithm, cipher mode, verification algorithm, elliptic curve

2.1 Cipher Suites

SECP256K1_ECIES_ECDSA_AES256_CBC_SHA256	SECP256K1_ECIES_ECDSA_AES256_CBC_SHA512
SECP256K1_ECIES_ECDSA_AES192_CBC_SHA256	SECP256K1_ECIES_ECDSA_AES192_CBC_SHA512
SECP256K1_ECIES_ECDSA_AES128_CBC_SHA256	SECP256K1_ECIES_ECDSA_AES128_CBC_SHA512
SECP256K1_ECIES_AES256_GCM_SHA256	SECP256K1_ECIES_AES256_GCM_SHA512
SECP256K1_ECIES_AES192_GCM_SHA256	SECP256K1_ECIES_AES192_GCM_SHA512
SECP256K1_ECIES_AES128_GCM_SHA256	SECP256K1_ECIES_AES128_GCM_SHA512
SECP256K1_ECIES_HMAC_AES256_CBC_SHA256	SECP256K1_ECIES_HMAC_AES256_CBC_SHA512
SECP256K1_ECIES_HMAC_AES192_CBC_SHA256	SECP256K1_ECIES_HMAC_AES192_CBC_SHA512
SECP256K1_ECIES_HMAC_AES128_CBC_SHA256	SECP256K1_ECIES_HMAC_AES128_CBC_SHA512
SECP256K1_ECIES_HMAC_AES128_GCM_SHA512	SECP256K1_ECIES_ECDSA_CHACHA20_SHA256
SECP256K1_ECIES_ECDSA_CHACHA20_SHA512	SECP256K1_ECIES_HMAC_CHACHA20_SHA256
SECP256K1_ECIES_HMAC_CHACHA20_SHA512	SECP256R1_ECIES_ECDSA_AES256_CBC_SHA256
SECP256R1_ECIES_ECDSA_AES256_CBC_SHA512	SECP256R1_ECIES_ECDSA_AES192_CBC_SHA256
SECP256R1_ECIES_ECDSA_AES192_CBC_SHA512	SECP256R1_ECIES_ECDSA_AES128_CBC_SHA256
SECP256R1_ECIES_ECDSA_AES128_CBC_SHA512	SECP256R1_ECIES_AES256_GCM_SHA256
SECP256R1_ECIES_AES256_GCM_SHA512	SECP256R1_ECIES_AES192_GCM_SHA256
SECP256R1_ECIES_AES192_GCM_SHA512	SECP256R1_ECIES_AES128_GCM_SHA256
SECP256R1_ECIES_AES128_GCM_SHA512	SECP256R1_ECIES_HMAC_AES256_CBC_SHA256
SECP256R1_ECIES_HMAC_AES256_CBC_SHA512	SECP256R1_ECIES_HMAC_AES192_CBC_SHA256
SECP256R1_ECIES_HMAC_AES192_CBC_SHA512	SECP256R1_ECIES_HMAC_AES128_CBC_SHA256
SECP256R1_ECIES_HMAC_AES128_CBC_SHA512	SECP256R1_ECIES_HMAC_AES128_GCM_SHA512
SECP256R1_ECIES_ECDSA_CHACHA20_SHA256	SECP256R1_ECIES_ECDSA_CHACHA20_SHA512
SECP256R1_ECIES_HMAC_CHACHA20_SHA256	SECP256R1_ECIES_HMAC_CHACHA20_SHA512
SECP521R1_ECIES_ECDSA_AES256_CBC_SHA256	SECP521R1_ECIES_ECDSA_AES256_CBC_SHA512
SECP521R1_ECIES_ECDSA_AES192_CBC_SHA256	SECP521R1_ECIES_ECDSA_AES192_CBC_SHA512
SECP521R1_ECIES_ECDSA_AES128_CBC_SHA256	SECP521R1_ECIES_ECDSA_AES128_CBC_SHA512
SECP521R1_ECIES_AES256_GCM_SHA256	SECP521R1_ECIES_AES256_GCM_SHA512
SECP521R1_ECIES_AES192_GCM_SHA256	SECP521R1_ECIES_AES192_GCM_SHA512
SECP521R1_ECIES_AES128_GCM_SHA256	SECP521R1_ECIES_AES128_GCM_SHA512
SECP521R1_ECIES_HMAC_AES256_CBC_SHA256	SECP521R1_ECIES_HMAC_AES256_CBC_SHA512
SECP521R1_ECIES_HMAC_AES192_CBC_SHA256	SECP521R1_ECIES_HMAC_AES192_CBC_SHA512
SECP521R1_ECIES_HMAC_AES128_CBC_SHA256	SECP521R1_ECIES_HMAC_AES128_CBC_SHA512
SECP521R1_ECIES_HMAC_AES128_GCM_SHA512	SECP521R1_ECIES_ECDSA_CHACHA20_SHA256
SECP521R1_ECIES_ECDSA_CHACHA20_SHA512	SECP521R1_ECIES_HMAC_CHACHA20_SHA256
SECP521R1_ECIES_HMAC_CHACHA20_SHA512	BRAINPOOL256R1_ECIES_ECDSA_AES256_CBC_SHA256
BRAINPOOL256R1_ECIES_ECDSA_AES256_CBC_SHA512	BRAINPOOL256R1_ECIES_ECDSA_AES192_CBC_SHA256
BRAINPOOL256R1_ECIES_ECDSA_AES192_CBC_SHA512	BRAINPOOL256R1_ECIES_ECDSA_AES128_CBC_SHA256
BRAINPOOL256R1_ECIES_ECDSA_AES128_CBC_SHA512	BRAINPOOL256R1_ECIES_AES256_GCM_SHA256
BRAINPOOL256R1_ECIES_AES256_GCM_SHA512	BRAINPOOL256R1_ECIES_AES192_GCM_SHA256
BRAINPOOL256R1_ECIES_AES192_GCM_SHA512	BRAINPOOL256R1_ECIES_AES128_GCM_SHA256
BRAINPOOL256R1_ECIES_AES128_GCM_SHA512	BRAINPOOL256R1_ECIES_HMAC_AES256_CBC_SHA256
BRAINPOOL256R1_ECIES_HMAC_AES256_CBC_SHA512	BRAINPOOL256R1_ECIES_HMAC_AES192_CBC_SHA256
BRAINPOOL256R1_ECIES_HMAC_AES192_CBC_SHA512	BRAINPOOL256R1_ECIES_HMAC_AES128_CBC_SHA256
BRAINPOOL256R1_ECIES_HMAC_AES128_CBC_SHA512	BRAINPOOL256R1_ECIES_HMAC_AES128_GCM_SHA512
BRAINPOOL256R1_ECIES_ECDSA_CHACHA20_SHA256	BRAINPOOL256R1_ECIES_ECDSA_CHACHA20_SHA512
BRAINPOOL256R1_ECIES_HMAC_CHACHA20_SHA256	BRAINPOOL256R1_ECIES_HMAC_CHACHA20_SHA512
BRAINPOOL512R1_ECIES_ECDSA_AES256_CBC_SHA256	BRAINPOOL512R1_ECIES_ECDSA_AES256_CBC_SHA512
BRAINPOOL512R1_ECIES_ECDSA_AES192_CBC_SHA256	BRAINPOOL512R1_ECIES_ECDSA_AES192_CBC_SHA512
BRAINPOOL512R1_ECIES_ECDSA_AES128_CBC_SHA256	BRAINPOOL512R1_ECIES_ECDSA_AES128_CBC_SHA512
BRAINPOOL512R1_ECIES_AES256_GCM_SHA256	BRAINPOOL512R1_ECIES_AES256_GCM_SHA512
BRAINPOOL512R1_ECIES_AES192_GCM_SHA256	BRAINPOOL512R1_ECIES_AES192_GCM_SHA512
BRAINPOOL512R1_ECIES_AES128_GCM_SHA256	BRAINPOOL512R1_ECIES_AES128_GCM_SHA512
BRAINPOOL512R1_ECIES_HMAC_AES256_CBC_SHA256	BRAINPOOL512R1_ECIES_HMAC_AES256_CBC_SHA512
BRAINPOOL512R1_ECIES_HMAC_AES192_CBC_SHA256	BRAINPOOL512R1_ECIES_HMAC_AES192_CBC_SHA512
BRAINPOOL512R1_ECIES_HMAC_AES128_CBC_SHA256	BRAINPOOL512R1_ECIES_HMAC_AES128_CBC_SHA512
BRAINPOOL512R1_ECIES_HMAC_AES128_GCM_SHA512	BRAINPOOL512R1_ECIES_ECDSA_CHACHA20_SHA256
BRAINPOOL512R1_ECIES_ECDSA_CHACHA20_SHA512	BRAINPOOL512R1_ECIES_HMAC_CHACHA20_SHA256

3 Errors

Everything about errors and error codes are stored in errors.h file.

networking related errors are stored in *log/network.log* and other bugs (mainly cryptographic) is stored in *log/errors.log*

4 Blocking

4.1 Algorithm

blocking an ip address code is in comm.cpp, the algorithm is as follows:

```

Generate(iv) # 12-byte iv
key = 32-byte key from keys file
pepper = 32-byte pepper from keys file

# encrypt ip
encrypted = AES-256-CBC(key=key, data=ip, iv=iv)

# store encrypted ip in blocked file
write(encrypted + " " + iv, "blocked")
```

Figure 1: Block

5 Key Protector

5.1 What Is It

The Key protector app in security folder is used to secure a 32-byte symmetric key, 2-byte port key, 32-byte pepper. The output is in a file named *keys*. The data in this file is used for securing the local data. It needs a 4-32 byte password generated and stored by you.

To set the password, execute the *key* file which would generate the *get_keys* executable which is the key protector program. Store a copy of *get_keys* in somewhere secure if you don't want to lose it. If you lose the *get_keys* and don't have the *keys* file, then your key is forever lost.

5.1.1 What is the keys used for

The key is used for securing any personal data stored on the device. Such as the configuration file for each session. If you're texting somebody and want to save their ip address so you can conviniently text them again without re-entering the ip address and reconfiguring the communication session, the ip and other data needs to be encrypted and stored. The 2-byte port key is used for encrypting the ports in configuration files

5.2 Algorithm

The C++ code is in *security/key.cpp*, but the basic idea is as following:

5.3 Security

since 2/3-bytes of the pepper is not stored in the *get_keys* file, they need to be guessed with every password that is entered. If we say 3 bytes of the data needs to be guessed. then the number of combinations in password is multiplied with 256^3.

e.g. if you have a 4-digit pin as your password, then there are 10^4 combinations in your password. Then the total number of combinations in password is $(256^3)(10^4) = 167772160000$.

This doesn't mean that your password needs to be smaller, it should still be 6-16 characters of numbers, small/capital letters, and symbols.

```

Generate key, pepper, iv
Ask user for 4-32 byte password
# chacha cipher function: chacha(data, key, iv)
$result = pepper \oplus password$
# Use sha256(result) as symmetric key to encrypt key using chacha
encrypted key = chacha(key, sha256(result), iv)
Store sha256(result) as sha256(sha256(result))
Generate exe for getting key ($get\_keys$):
    Store sha256(sha256(result)), iv, encrypted key, pepper (excluding 3-bytes)

    Ask user for password:
        Guess 3 bytes of password of unknown pepper
        $result = pepper \oplus password$
        Compute sha256(sha256(result)) and compare with stored sha256(sha256(
   result)).
        if no match:
            Continue guessing all possible 3-bytes

        if user guessed more than once:
            If guessed 3 or 6 times and while guess count is smaller than 7:
                Pause for 10s
            Else if Every 5 guesses:
                Pause for 30s
        Sleep(random(1s,5s)) # make it a random range so that timing attacks aren
   't possible

        If not valid match:
            If more than 10 password inputs made:
                Delete everything in current directory
            ask user for password again and repeat process.
        Else:
            # Decrypt encrypted key using sha256(result) as key with chacha
   algorithm
            decrypted key = chacha(encrypted key, sha256(result), iv)

            Write decrypted key to file

            set key, pepper, iv, password, and every other array stored in ram to
    zero

```

Figure 2: Key Protector