Silence: An end to end encrypted,

peer to peer messaging app which does not leak IPs

by: rootfinlay

beta version 0.1.0 can be found here:
https://github.com/projectsilence/silence

Abstract:

Currently, every messaging system we have relies on trust in one way or another. Can we trust that middleman servers don't keep logs of messages? Can we trust the code we see on GitHub is being used in the application? Can we trust that hackers can't access the remote middleman server? This is where Silence comes in.

A messaging service you have complete control over is the most reliable way to ensure the recipient receives your message securely, without needing to know more than is necessary. This ensures both parties are safe, the data cannot easily be intercepted and even if it is, each individual controls the encryption keys so the data cannot be decrypted by anyone else.

A final measure of security comes with the rubber-hose protection which comes as standard with Silence. For more about rubber host attacks and the prevention Silence offers, check the rubber hose attack section.

Current state of Silence:

Currently, Silence is in an early, pre-stable release designed as a proof-of-concept public beta. It is not user friendly in the slightest and is designed to allow complete control rather than useability to make sure there are no issues with each feature. Gradually, I plan to implement new features, fix bugs and test everything rigorously until I am satisfied with them all. Once I am satisfied, I will start working on the user interface and making everything more user friendly.

The Crypto:

The crypto is an important part of Silence's security features. The crypto library can be found in the assets/silencecrypto.py file.

Until proven breakable, Silence will use AES and RSA encryption with the sent and stored messages, specifically RSA 4096bit and AES 256bit. The RSA private keys will be stored in a PKCS#8 format with a user defined password encrypted with scrypt and AES256-CBC.

This function shows the key generation:

```
def GenerateKeypairRSATrue(passphrase):
    key = RSA.generate(4098)
    encrypted_key = key.export_key(passphrase=passphrase, pkcs=8,
                                   protection="scryptAndAES256-CBC")

    file_out = open(SELF_KEY_FOLDER+"selfrsatrue.bin", "wb+")
    file_out.write(encrypted_key)
    file_out.close()

    pubkey = key.publickey().export_key()

    file_out = open(SELF_KEY_FOLDER+"selfrsatrue.pub", "wb+")
    file_out.write(pubkey)
    file_out.close()

    return "True key generation complete..."
```

Messages being transmitted will be encrypted using AES with a random 32 byte long key, which is encrypted by the receivers RSA public key.

This function shows the encryption process:

```
def RSACrypt(external_onion, message):
    recipient_key =
RSA.import_key(open(KEY_FOLDER.format(external_onion)+"realpub.pub",
"rb").read())
    session_key = get_random_bytes(32)

    cipher_rsa = PKCS1_OAEP.new(recipient_key)
    enc_session_key = cipher_rsa.encrypt(session_key)

    cipher_aes = AES.new(session_key, AES.MODE_EAX)
    ciphertext, tag =
cipher_aes.encrypt_and_digest(message.encode('utf-8'))
    file_out = open(TEMP_FOLDER+"encrypted_data.bin", "wb")
    [ file_out.write(x) for x in (enc_session_key, cipher_aes.nonce,
tag, ciphertext) ]
    file_out.close()
```

In addition to the message being encrypted, it will be hashed using SHA512 and then digitally signed using the senders RSA private key, allowing the user to verify whether the message is legitimate and from the correct sender.

This function digitally signs the message:

```
def RSASign(message, password, keystate):
    if keystate == "RealKeyUnlocked":
        key = RSA.import_key(open(SELF_KEY_FOLDER+"selfrsatrue.bin",
"rb").read(), passphrase=password)
    elif keystate == "FakeKeyUnlocked":
        key = RSA.import_key(open(SELF_KEY_FOLDER+"selfrsafalse.bin",
"rb").read(), passphrase=password)
    else:
        return "How would you get this to throw an error?"

    h = SHA512.new(bytes(message, 'utf-8'))

    signature = pkcs1_15.new(key).sign(h)
    return signature
```

As far as I am aware, these are the best practices for cryptography available now. If this changes, Silence will be updated as soon as possible.

Rubber Hose Protection:

Rubber Hose protection is not currently implemented into Silence as I am testing and adapting the algorithm before committing to implementation.

Wikipedia states this about the rubber-host attack: "In cryptography, **rubber-hose cryptanalysis** is a euphemism for the extraction of cryptographic secrets (e.g. the password to an encrypted file) from a person by coercion or torture[1]—such as beating that person with a rubber hose, hence the name—in contrast to a mathematical or technical cryptanalytic attack." [1]

Right now, two keys are generated with the names "selfrsatrue" and "selfrsafalse". This will be changed to allow users to name their keys. If they wish, "key1" and "key2" will suffice.

On initiating contact with another Silence node, the user will be prompted to choose which key they wish to be the "real" key and which key they wish to be the "fake" key. This information is stored encrypted on the recipient's device.

Sending:

When a message is being sent, the sender will be prompted to enter a password. It will go through both keys and attempt to unlock them. If it is successful in unlocking a key, it will sign and send a message using that key and look as if a normal message has been sent. When the user reads a message, it will know which key is real and which key is fake when verifying, and can alert the recipient of a potential breach if the “fake” key is used.

Due to there not being any record of which key is real and which key is fake on the sender’s device, the attacker can not confidently know if the message sent is real or fake. Demanding both passwords would also be futile because sending two identical messages, one with each key, will still alert the recipient of a potential breach.

Receiving:

When a user is receiving a message, the sender will first perform an “authentication check” which will ask for a signed message of the currently unlocked key. If the real key is being used, the message will be sent as usual, but if the fake key is being used the sender will be prompted to either change the message or not send one at all.

Coupling the rubber-hose protection with self-deleting messages will be a great line of defence.

Self-deleting messages:

In the silence settings file, the user can describe if messages automatically delete themselves after being read. It is highly recommended to keep the deleted messages feature set to True for security reasons, but it is up to the user’s discretion.

Running Silence through Tor:

To avoid leaking IPs, Silence by default uses Tor to setup a hidden service Silence node and to talk to other Silence nodes. Using this preexisting infrastructure, Silence messages are a lot more difficult to intercept by hackers, and you can chat freely to anyone, anywhere, as long as Tor and encryption is available and legal for them, without revealing your IP.

While it would be possible to not utilize the Tor tunnelling, it is not recommended as displaying your IP and running a webserver available on the clear web can be dangerous. However, Silence is meant to be quite hackable as a messenger, meaning anyone with the knowhow can customize its features to their hearts content, providing they don’t change anything too drastic which will stop them being able to interact with regular Silence nodes. If they do, the other nodes would also need to use the sender's software.

Final thoughts:

I am aware Silence will not be executed perfectly. I am a young dev and I am still learning. This project is mainly a proof of concept of what can be done to improve privacy online. If any devs would like to contribute or hard-fork Silence to change it for commercial release, please email me at finlay.business@proton.me and we can discuss the next steps. Thank you all for reading.

References:

1 – Rubber Hose information from Wikipedia - https://en.wikipedia.org/wiki/Rubber-hose_cryptanalysis