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

    file_out = open(name+".bin", "wb+")
    file_out.write(encrypted_key)
```

```
    file_out.close()

    pubkey = key.publickey().export_key()

    file_out = open(name+".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):
    key = RSA.import_key(open(keystate+".bin", "rb").read(),
passphrase=password)

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

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

This function verifies the signature:

```
def RSACheckSig(external_onion, i, message):
    key1 =
RSA.import_key(open(KEY_FOLDER.format(external_onion)+"realpub.pub").re
ad())
    key2 =
RSA.import_key(open(KEY_FOLDER.format(external_onion)+"fakepub.pub").re
ad())

    signature =
open(KEY_FOLDER.format(external_onion)+"messages/"+i+"signature.bin",
"rb").read()

    h = SHA512.new(message.encode("utf-8"))

    try:
        pkcs1_15.new(key1).verify(h, signature)
        return "REALKEY: Valid Signature"
    except (ValueError, TypeError):
        pass
    try:
        pkcs1_15.new(key2).verify(h, signature)
        return "FAKEKEY: Valid Signature"
    except (ValueError, TypeError):
        return "No valid signatures..."
```

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 has now been implemented into Silence v0.2.0 and will only be revised if necessary.

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]

In the settings file, two “key” fields exist where the end user can change the key names if they wish. “key1” and “key2” are simpler and sufficient. When the keys are being

generated, they will be stored in the “self” folder as “(name).pub” and “(name).bin”, .pub referring to the public key and the .bin referring to the encrypted private key. This file format is also applicable to external keys.

When the user is initiating a contact with another user, they will encode their keys with base64, specify which key is the “real” key and send them to the user they wish to initiate contact with. The user will store these as “realpub” and “fakepub”.

When a message is being sent, the user will have to input a password to unlock a private key. After this, a “signature request” will be sent, which asks for the “signedsession” signature. Using the pkcs1.15 signature algorithm and both keypairs, the sender will be able to check which key is being used at the current time. If the “real” key is being used, the regular message prompt will be shown, and the message will be encrypted with AES256, the session key being encrypted with the recipients “real” private key. The message will also be hashed and signed with the senders currently unlocked key. These will be encoded in base64 and sent to the recipient. However, if the signature request shows a “fake” key is being used the sender will be warned and prompted to either proceed with caution or not send a message right now. This allows the sender to know the recipient is potentially compromised and to not include any sensitive information in this conversation.

When reading a message, the recipient will need to unlock a private key once again. If the key unlocked does not match the session key, they cannot read any messages and will be prompted to try again. When the private key and session key match, the content is decrypted and the signature is checked. If the “real” key is used, a “VALIDSIG-REALKEY” message will be appended. If the “fake” key is used, a “VALIDSIG-FAKEKEY THE SENDER MAY BE COMPROMISED!” message will be appended. This will let the recipient know that the sender is most likely compromised and to be careful proceeding.

Assuming you choose the same key to be “real” across all your contacts, mitigating rubber hose attacks will be easy. Demanding and using both keys would be pointless as the user would be alerted of this.

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