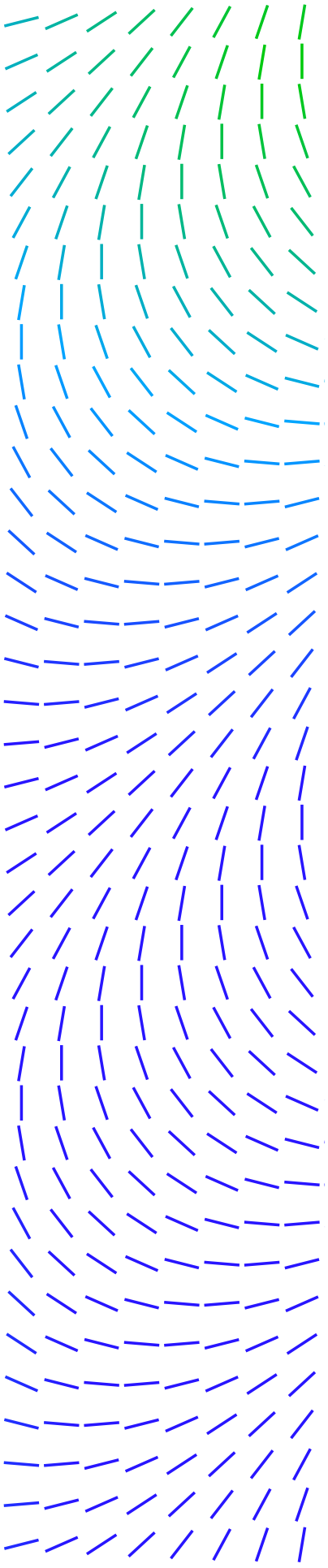

WHITE PAPER

Distributed Security: Shamir's Secret Sharing Key Shares

By: Charles McFarland

Trellix

Introduction

Privacy is both important and difficult to achieve in this day and age. To complicate matters, some data needs to be recoverable in a secure way. You may find yourself trying to solve this problem alone but fortunately others have come before you. You can use what they have discovered to build a base for your current needs. Typical requirements are that the data must be secure, recoverable in the event the owner is unavailable, and that any secret sharing should be minimized. Obviously, it would be unwise to leave the data unencrypted. While it may be recoverable, it is not secure. There is an option to encrypt the data and share a private key (PK) to whomever has need of it. Again, recoverable, but sharing the PK tends to be a bad idea. With multiple holders of the private key, you have increased the chances of the key being leaked and the data compromised. Fortunately, there is another option that can meet all three of these requirements. Shamir's Secret Sharing Scheme.

What is Shamir's Secret Sharing Scheme?

Try saying this four times fast: Shamir's Secret Sharing Scheme (SSSS). If you can get past the mouthful, Shamir's algorithm simply enables the recovery of a secret based on N *of* M shares. Shares are similar to private keys or passwords. However, they are mathematically generated from a secret and split into more than one part. A single password can be split into many shares and by combining some portion of those shares you can recover the password. Shamir's Secret Sharing Scheme takes care of the splitting of some secret into shares as well as the recombining of those shares for recovery. Here is an example of using SSSS to split the string "Hello!" into 4 shares while requiring 3 shares to recover the string.

---

example-1-c90088993281dd585978cf76b069502b9588dbe1b784e1391d7331125c7d4fe1b0957ebd6e9a3968a0ccf5d2b7b4f55f-73c7acee8ace99d566d358607430592a015c1254dba62d664e4edcb848b5761745d4ab96ac642e5b75cf5196a1dabbd-4c68269e3054032e7e4e3b83ae074eae0a00d2e886dae00c377db88249e0f8537

---

example-2-28bca8ccaf1d58fc88d8ad227f20f89f8f73a5ecd1a8765903674c8f111557ad4262ae8faa4a60e58e6a8cfb457b-fe9e8fe3701340e0b9e3c247bfff9319b951991ef219cc93966f725014d44ba42107f4ec3904db2dcfe3d4905d960cc42b-71c9529f85936df7fce1b3a5595adf5f3ba406fb6390a85fd9f66b987fec049706

---

example-3-771a4f5afa7f5526a016119dd3000bee28b737e1620dc2c471d5fbd0611a737c918b5a1df428f740f20106b0c2bd2e52e-a877b7631b5608b53f57a88e119c7092aaf5749af55beeeebae017ce8295a36b1117e6d2933073fe9b9e5f6c016587548461609e-d9554a01cb08ed6737636cacdae3b060bd8b4f953dfc151ed4875a3

---

example-4-efcfc3357ef48f5520a5504f3639e04a82e8b183643a39409182823a2359c6f03a77f57e9c42b717d1570688307079dd6a010df3ea2d9c3dc42378dfe8c6917dd74b03c89c7828d8c402a8ce341b3c68155f2ba9d17e573de89c93b14f31193169abac72fe-82decce01379a7ee980467e92990e4786e0a7a27d4af45bebc51db

---

Shares are designed to be passed to multiple people to collaborate when recovery is necessary. Collaboration simply means to join their shares together using the combine algorithm to recover the secret. In the "Hello!" example, 3 are required for recovery. This number could have been set to 2 as well. You get to choose. By turning "Hello!" into shares, you can pass four "shares" around, so that as long as three of them remain, the string can be recovered. The contributors will need to collaborate so even if one or two of them leaks their shares or are compromised, the secrets remain safe.

Shamir's Secret Sharing Scheme also happens to have widely distributed implementations across multiple operating systems. By using these implementations, you can avoid 'rolling your own' and simply build a wrapper around something that has survived years of scrutiny. Implementing your own creates risks of including vulnerabilities, both known and unknown. Cryptography is hard and this lesson continues to be relearned as detailed in this report. In it, the authors list improper implementations of Feldman's Scheme, which is in part based off Shamir's Secret Sharing Scheme. Here is the list they have discovered.

- Binance's tss-lib
- Clover Network's threshold-crypto
- Keep Network's keep-ecdsa
- Swingby's tss-lib
- ZenGo X's curv

To avoid this pitfall, I'll be referencing the B. Poettering implementation which is not susceptible to these vulnerabilities. This implementation can be found in many popular package managers. Installation is fairly easy. For Debian-based Linux distributions you can use apt-get or the Homebrew package manager to install it on a Mac.

Installation

Debian based installation:

```
$sudo apt-get update
```

```
$sudo apt-get install ssss
```

Mac OS installation:

```
$brew install ssss
```

The above commands give you access to two tools: **ssss-split** and **ssss-combine**. These implement the split and combine algorithms. More information about their usage can be found in the Man page.

The Wrapper

Since this implementation only encrypts up to a 1024 bit secret, you can use a combination of AES 256 CBC with PBKDF2 (Password based key derivation function 2) to secure larger secrets. SSSS can split the password into shares so you can keep the password private. All you need to do is generate a 1024 based secret to use as a password for your AES encrypted files. In order to streamline the process and use better cryptographic random number generators you can wrap everything up in Python 3. By using STDIN and STDOUT you'll be able to use these as a base for any further work you wish to do and perform actions without the password ever being visible to the user.

Start with encrypting the secret and splitting the password into shares.

1. Generate a random password
2. Encrypt a file with AES 256 using the generated password
3. Split the generated password into shares using Shamir's scheme.
4. Output the shares to STDOUT and distribute.

Here is the code for the split wrapper which completes these tasks.

split.py

```
import subprocess
import string
import secrets

def generate_key(bits):
    characters = string.ascii_letters + string.digits + string.punctuation
    hex_string = ''.join([secrets.choice(characters) for _ in range(int(bits/8))])
    return hex_string


def main(threshold, shares, bitsize):
    key = generate_key(bitsize)
    shares = subprocess.Popen(['ssss-split', '-w' 'projectname', '-t', str(threshold), '-n', str(shares), '-s', str(bitsize), '-q'], text=True, stdin=subprocess.PIPE, stdout=subprocess.PIPE)
    output = shares.communicate(input=key)[0]
    print(output)

if __name__ == "__main__":
    main(3, 4, 1024)
```

The **subprocess** package is used to call the install ssss implementation. This is equivalent to:

$ssss-split -w projectname -t 3 -n 4 -s 1024 -q

The parameter **projectname** is simply a prefix token in text that will be added for each share. It is not necessary, but useful to remind someone what the share is for. The **3** is the threshold of shares necessary to recover the password. The **4** is the total number of shares being distributed. Normally after execution, this command will ask the user for the password as input. The line **share.communicate(input=key)[0]** avoids user input by passing our generated key in automatically. This enables us to hide the generated password from the user.

The generated password uses the **strings** and **secrets** package to generate a random ASCII 1024-bit password. Secrets is a more appropriate method for generating random numbers than the random package, since we want our generation to be cryptographically strong. More information on the secrets package can be found here.

Next, the shares must be combined to recover the secret:

1. Take shares as input.
2. Output to STDOUT for use with further scripting

combine.py

```
import pexpect

def main(threshold, shares, bitsize):
    s = f'ssss-combine -t {threshold} -n {shares} -s {bitsize} -Q'
    combine = pexpect.spawnu(s)
    for _ in range(threshold):
        share = input("Share:")
        combine.sendline(share)
    output = combine.read().split('\r\n')[-2]
    print(output)

if __name__ == "__main__":
    main(3, 4, 1024)
```

The combine.py wrapper needs to be able to interact with ssss-combine to input each share when requested. Use the package **pexpect** with the method **sendline** to do this. Again, you supply the threshold (3), the total number of shares (4) and the bitsize (1024). Also add the -Q flag to put the tool in quiet mode. This makes it much easier for scripting as you'll know exactly where the secret is printed.

This is equivalent to executing the following:

$ssss-combine -t 3 -n 4 -s 1024 -Q

It then takes the required threshold of shares as user input and outputs the password to STDOUT. You can pipe this to your decryption tool for further scripting while avoiding printing the password.

Bringing It All Together

For encryption and decryption, you can use openssl. The following two commands will allow you to pipe in output from your tool into openssl as part of a scripting process.

Encrypt:

$openssl enc -aes-256-cbc -md sha512 -pbkdf2 -iter 100000 -salt -in secrets.txt -out secrets.enc -pass stdin

Decrypt:

$openssl enc -d -aes-256-cbc -md sha512 -pbkdf2 -iter 100000 -salt -in secrets.enc -out secrets.txt -pass stdin

Here is what the entire process looks like.

1. Create a recoverable password with split.py

$python3 ./split.py

2. Supply combine.py with the shares, then pipe the output to the encryption process (openssl)

$python3 ./combine.py | openssl enc -aes-256-cbc -md sha512 -pbkdf2 -iter 100000 -salt -in secrets.txwt -out secrets.enc -pass stdin

3. Decrypt the secret using combine.py and pipe the output to the decryption process.

$python3 ./combine.py | openssl enc -d -aes-256-cbc -md sha512 -pbkdf2 -iter 100000 -salt -in secrets.enc -out secrets.txt -pass stdin

By using a wrapper and Shamir's Secret Sharing Scheme you now have a base for scripting and can successfully hide the password from the user. If you distribute the 4 keys to other users and the encrypted container, then if any of them need the secrets they can simply combine their share with two others and run the scripts above. The files remain secure encrypted in AES 256, three of your users can collaborate and recover the password when necessary, and the PK is never distributed. This matches the original requirements you've set out for.