



MORTEN LINDERUD

F/OSS Developer, Arch Linux Developer and security team.

[BLOG](#) [ABOUT](#) [TAGS](#)



GOLANG CRYPTO/ECDH AND THE TPM

7 minutes read · Tags: [tpm](#) [golang](#) [linux](#) [english](#)

I have lately been trying to learn more about the Trusted Platform Module (TPM) as they are capable of key creation and sealing secrets in a secure manner. They are common hardware these days and make for a reasonable ways to store secrets.

[age](#) is a file encryption/decryption tool from Filippo Valsorda which a lot of people have been using to replace GnuPG for things like `password-store`. It has a few plugins doing things like storing keys on Yubikey, Trezor hardware wallets or the Apple Secure Enclave, however it doesn't have a TPM plugin. I saw the opportunity to write something that is capable of utilizing the TPM.

So after a couple of weekends hacking on something I had an initial version of such a plugin, [age-plugin-tpm](#). Which is using capable of creating and storing RSA 2048 bit keys for age to use.

However a big downside is that it is using RSA 2048, and the keys are already super long. Like 443 bytes long.

```
age1tpm1syqqppqqqqqqqqqqqqqq6uxewjcd9c30s027a4wmfud2qewzanzp6x7jljd6jf6gn45sv9lssx
39pdlgl2khwwf8u4vzevavapte5j8n5ntrhvk58wvsp3e49ghmhp36m4u7m958tuh0u7zqwq0sk6fxff
lr9rm8fwazvss4u86wjsygu0py4nl26g0np2ce5ehshfgkm2k0sgls8389e25fxta2yqt8k4gh4uuh9t
5n84uaend273yfst8ykap6unhn5dphdwa6562e80jlnme5j28yvmvdafwfa3v8rnyerxmt6zat6kez7
yvw7acvzxu3knh7m56pxyntmpsgwp3kevrh85ue8kpakgnqtsrfede2tfz98s2drd2nhj4dq8t5uu82
ex54l3g9jeh4rkhdz6slt33zgf5wcff42fgdky7gd42
```

Thus we need to try and figure out Elliptic Curve cryptography which is what `age` itself is doing.

Elliptic Curves

RSA is an asymmetric public-private key scheme where we distribute a public key, and we keep a private key. Only the private key can decrypt the material encrypted with the public key.

Elliptic Curves on the other hand uses symmetric encryption. This means that the key encrypting something, and the key decrypting something uses the same key for both these tasks. It's effectively a shared secret. A common way of doing this is through the Diffie-Hellman key exchange. This is called Elliptic-curve Diffie-Hellman (ECDH).

With the release of Go 1.20 we got a new `crypto/ecdh` library to perform operations like these.

Since the TPM only supports a subset of curves we will be using NIST P-256 for all the examples below.

We will be creating two keys. Once for Alice, and one for Bob. Alice and Bob needs to agree on a shared secret they can encrypt secrets with.

```
import (
    "crypto/ecdh"
    "crypto/rand"
    "crypto/sha256"
    "fmt"
)

alice, _ := ecdh.P256().GenerateKey(rand.Reader)
bob, _ := ecdh.P256().GenerateKey(rand.Reader)
```

Alice gives Bob their public key and Bob can perform an ECDH key exchange with the public key. We hash the shared key for the sake of representation in the string.

```
alicePubkey := alice.PublicKey()

shared, _ := bob.ECDH(alicePubkey)
bobShared := sha256.Sum256(shared)
fmt.Printf("Shared key (Bob) %x\n", shared)
```

With the following as output.

```
Shared key (Bob) 1cb538e525096ef7c3d8bbf9b3868eba183ebc153d548c934975a2107696b215
```

Cool, we have something that can probably work as a key. Now Bob gives their public key to Alice and they do the same.

```
bobPubkey := bob.PublicKey()  
  
shared, _ := alice.ECDH(bobPubkey)  
aliceShared := sha256.Sum256(shared)  
fmt.Printf("Shared key (Alice) %x\n", shared)
```

And we get the same output.

```
Shared key (Alice) 1cb538e525096ef7c3d8bbf9b3868eba183ebc153d548c934975a2107696b215
```

Cool, this was actually pretty easy. I'm not very sturdy in explaining the math behind this, but we are effectively agreeing on the same position on the curve which will work as our symmetric key.

Lets do the same but now lets store the key from Alice in the TPM.

Trusted Platform Modules

TPMs are simple devices, but the API is fairly complicated. I am not going to explain everything going on, others have written at length about this before. However I will try and give a complete example on how to implement this.

We will be using the [go-tpm](#) from Google to deal with the TPM.

First off we need a couple of variables. TPM use key hierarchies which allow us to generate deterministic create keys. We are going to create a key under the "Storage Root Key" (SRK) which will work as our application key.

```

import (
    "github.com/google/go-tpm/tpm2"
    "github.com/google/go-tpm/tpmutil"
)

var (
    // Default SRK handle
    srkHandle tpmutil.Handle = 0x81000001

    // Default SRK key template
    srkTemplate = tpm2.Public{
        Type:          tpm2.AlgRSA,
        NameAlg:       tpm2.AlgSHA256,
        Attributes:    tpm2.FlagFixedTPM | tpm2.FlagFixedParent
                  | tpm2.FlagSensitiveDataOrigin | tpm2.FlagUserWithAuth
                  | tpm2.FlagRestricted | tpm2.FlagDecrypt | tpm2.FlagNoDA,
        RSAParameters: &tpm2.RSAParams{
            Symmetric: &tpm2.SymScheme{
                Alg:      tpm2.AlgAES,
                KeyBits: 128,
                Mode:     tpm2.AlgCFB,
            },
            KeyBits: 2048,
            ModulusRaw: make([]byte, 256),
        },
    },

    // Our Key Handle
    keyHandle tpmutil.Handle = 0x81010004

    // ECC Encrypt/Decrypt key template
    eccKeyTemplate = tpm2.Public{
        Type:          tpm2.AlgECC,
        NameAlg:       tpm2.AlgSHA256,
        Attributes:    tpm2.FlagStorageDefault & ^tpm2.FlagRestricted,
        ECCParameters: &tpm2.ECCParams{
            CurveID: tpm2.CurveNISTP256,
            Point: tpm2.ECPoint{
                XRaw: make([]byte, 32),
                YRaw: make([]byte, 32),
            },
        },
    },
}
)

```

Handles are effectively locations where we store our keys. The `0x81000001` is special and for the SRK, however we can select a random handle for our key storage. It will be `0x81010004` in our examples.

```
raw, err := tpm2.OpenTPM("/dev/tpm0")
if err != nil {
    log.Fatal(err)
}
defer raw.Close()

bob, _ := ecdh.P256().GenerateKey(rand.Reader)
bobPubKey := externalKey.PublicKey()
```

The next portion opens up the TPM device, and then we generate a key pair for Bob.

If you don't want to do this directly towards your TPM, I have written a small library to initialize a instance of `swtpm` which can be used for testing.

https://github.com/Foxboron/swtpm_test

```
handle, _, err := tpm2.CreatePrimary(raw, tpm2.HandleOwner, tpm2.PCRSelection{},
                                     "", "", srkTemplate)
if err != nil {
    log.Fatalf("failed CreatedPrimary: %v", err)
}
if err = tpm2.EvictControl(raw, "", tpm2.HandleOwner, handle, srkHandle); err != nil {
    log.Fatalf("failed EvictControl: %v", err)
}
```

This section creates our SRK key under the Owner hierarchy and we use `EvictControl` to delegate the temporary key to the persistent handle.

```
priv, pub, _, _, err := tpm2.CreateKey(raw, handle, tpm2.PCRSelection{},
                                       "", "", eccKeyTemplate)
if err != nil {
    t.Fatalf("CreateKey: %v", err)
}

handle, _, err := tpm2.Load(raw, srkHandle, "", pub, priv)
if err != nil {
    t.Fatalf("Load: %v", err)
}
```

```
defer tpm2.FlushContext(rwc, handle)
if err = tpm2.EvictControl(rwc, "", tpm2.HandleOwner, handle, keyHandle); err != nil {
    t.Fatalf("EvictControl: %v", err)
}
```

In this section we are creating our key for Alice. We pass our values to `tpm2.CreateKey` which most importantly is using our `eccKeyTemplate`. We then load the private and public parts into a handle, which we then make persistent.

This allow us to reference the created key under the handle `0x81010004` for future sessions.

Elliptic-curve Diffie-Hellman

Now we can do the key exchange!

```
x, y := elliptic.Unmarshal(elliptic.P256(), bobPubKey.Bytes())
```

First we need to parse the Bobs key to retrieve the `X` and `Y` portions of the keys so we can pass them to the TPM. The `crypt/ecdh` library does not expose these values from keys, so we need to pass it through `elliptic.Unmarshal`.

```
bobKey := tpm2.ECPoint{XRaw: x.Bytes(), YRaw: y.Bytes()}

z, err := tpm2.ECDHGen(rwc, keyHandle, "", bobKey)
if err != nil {
    log.Fatalf("failed ECDHGen: %v", err)
}

shared := sha256.Sum256(z.X().Bytes())

fmt.Printf("Shared key (Alice) %x\n", shared)
// Shared key (Alice) 2912759ae2641a4a18ae08abadbf1e413339333ea266f02bbaf580e5f58b6d4
```

At this point we have done the key exchange over the TPM. The return value of `z.X().Bytes()` is the equivalent value as returned by `alice.ECDH(bobPubkey)` from the earlier example.

Returning to Bob

However, Bob still wants to produce the same secret, but now the key material is stored in the TPM. What we need to do is to fetch the Public Key material from the TPM and produce a `ecdh.PublicKey` struct for Bob.

```
pub, _, _, err := tpm2.ReadPublic(rwc, keyHandle)
if err != nil {
    t.Fatalf("ReadPublic: %v", err)
}
alicePubKey, err := pub.Key()
if err != nil {
    t.Fatalf("can't read public key: %v", err)
}

ecdsaPubKey := alicePubKey.(*ecdsa.PublicKey)
alicePubKey, err := ecdsaPubKey.ECDH()
if err != nil {
    t.Fatalf("pubkey.ECDH: %v", err)
}
```

The code above reads of the public key material. We then need to do a little bit of dancing as `google/go-tpm` only really deals with `ecdsa.PublicKey`. We need to cast the return value from `.Key()` (which is `crypto.PublicKey`), to `ecdh.PublicKey`.

Then utilize the new Go 1.20 method `.ECDH()` to produce the correct key type for our `crypto/ecdh` library.

```
shared, _ := bob.ECDH(alicePubKey)
shared = sha256.Sum256(shared)

fmt.Printf("Shared key (Bob) %x\n", shared)
// Shared key (Bob) 2912759ae2641a4a18ae08abadbf1e413339333ea266f02bbaf580e5f58b6d44
```

Now we can give our `alicePubKey` to Bob and reproduce the shared secret.

The complete code example this article is based off on can be found here:

https://github.com/Foxboron/tpm-stuff/blob/master/ecc_keys/keys_test.go

[Back to posts](#)