Subject: Grin Schnorr Challenge
From: Jakob Abfalter <jakobabfalter@gmail.com>
Date: 03.07.2020, 18:19
To: "Moreno Sanchez, Pedro" <pedro.sanchez@tuwien.ac.at>

Hi Pedro,

I have now finished my investigation into the Grin Codebase to figure out how exactly the Schnorr challenge is calculated. To make it short Grin are
using the forked Blockstream C library for the partial schnorr signatures (same as for the bulletproofs) which will calculate the e including the combined nonces from Alice kA and Bob kB raised to generator (g^kA + g^kB)
and will also include the separate public keys of all participants into the hash, if the are passed. (And in the Grin wallet they are) So instead of pk_A + pk_B what I originally thought it would have pk_A || pk_B but that shouldn't make any difference.
additionally they also hash the number of participants, so the final e would look like this : e = h(m || g^kA + g^kB || 2 || pk_A || pk_B)
Also they do not share the e between each other, so each participant should calculate on its own from the public values.

Knowing exactly how this works now I will rework the signature scheme according to your feedback and the Grin implementation.
Then we can maybe have another look at it on Tuesday.

I documented my investigation here in this E-Mail for future reference and will also push it to the repository.

See you Tuesday!
Jakob

Grin Signing

Starting from the grin repository https://github.com/mimblewimble/grin (master branch)

I found out that they are using a package called aggsig for creating and verifying transactions. For example a call in transaction.rs : 1907

```
let excess = keychain
    .commit(0, &key_id, SwitchCommitmentType::Regular)
    .unwrap();
let skey = keychain
    .derive_key(0, &key_id, SwitchCommitmentType::Regular)
    .unwrap();
let pubkey = excess.to_pubkey(&keychain.secp()).unwrap();

let excess_sig =
    aggsig::sign_single(&keychain.secp(), &msg, &skey, None, Some(&pubkey)).unwrap();

kernel.excess = excess;
kernel.excess_sig = excess_sig;
```

I found some documentation of the package:

https://docs.rs/grin_core/0.5.1/grin_core/libtx/aggsig/index.html

Particularly interesting is the calculate_partial_sig function used to create the partial signatures

https://docs.rs/grin_core/0.5.1/grin_core/libtx/aggsig/fn.calculate_partial_sig.html

Arguments

- `secp` - A Secp256k1 Context initialized for Signing
- `sec_key` - The signer's secret key
- `sec_nonce` - The signer's secret nonce (the public version of which was added to the `nonce_sum` total)
- `nonce_sum` - The sum of the public nonces of all signers participating in the full signature. This value is encoded in e.
- `pubkey_sum` - (Optional) The sum of the public keys of all signers participating in the full signature. If included, this value is encoded in e.
- `msg` - The message to sign.

Here it says that the nonce sum (so k_alice + k_bob) will be encoded into e and the combined pubkey (pk_alice + pk_bob) can be passed optionally and will then also be encoded into e.

Next I have checked the Grin Wallet code (https://github.com/mimblewimble/grin-wallet) and looked at how they are calling the function

```
    sec_key: &SecretKey,
    sec_nonce: &SecretKey,
) -> Result<(), Error>
where
    K: Keychain,
{
    // TODO: Note we're unable to verify fees in this instance
    if !self.is_compact() {
        self.check_fees()?;
    }

    self.verify_part_sigs(keychain.secp())?;
    let sig_part = aggsig::calculate_partial_sig(
        keychain.secp(),
        sec_key,
        sec_nonce,
        &self.pub_nonce_sum(keychain.secp())?,
        Some(&self.pub_blind_sum(keychain.secp())?),
        &self.msg_to_sign()?,
    )?;
    let pub_excess = PublicKey::from_secret_key(keychain.secp(), &sec_key)?;
    let pub_nonce = PublicKey::from_secret_key(keychain.secp(), &sec_nonce)?;
    for i in 0..self.num_participants() as usize {
        // find my entry
        if self.participant_data[i].public_blind_excess == pub_excess
            && self.participant_data[i].public_nonce == pub_nonce
        {
            self.participant_data[i].part_sig = Some(sig_part);
            break;
        }
    }
    Ok(())
}
```

so it seems that they are passing the (pk_alice + pk_bob) so it will be included into the e.

Finally I wanted to look into the aggsig package itself, and found it here https://github.com/mimblewimble/rust-secp256k1-zkp and this library are just a bunch of bindings
to use the Blockstream secp256k1-zkp library inside Rust, which is the same C library I already investigated before for the bulletproofs :)

Here is the c code calculating the partial signature

```
secp256k1_scalar_set_b32(&sec, seckey32, &overflow);
if (overflow) {
    secp256k1_scalar_clear(&sec);
    return 0;
}
secp256k1_scalar_mul(&sec, &sec, &sighash);
secp256k1_scalar_add(&sec, &sec, &aggctx->secnonce[index]);
```

Here is the calculation for s = sk * e + k

seckey32 is the private key, sighash is e and secnonce[index] will be the nonce

and here is the calculation of the e

```
        secp256k1_ge_neg(&tmp_ge, &tmp_ge);
    }
    secp256k1_fe_normalize(&tmp_ge.x);
    secp256k1_compute_prehash(ctx, prehash, aggctx->pubkeys, aggctx->n_sigs, &tmp_ge.x, msghash32);
    if (secp256k1_compute_sighash(&sighash, prehash, index) == 0) {
        return 0;
    }
```

msghash32 is m, &tmp_ge.x is the x value of the combined nonce (g^k_alice + g^k_bob), n_sigs is the number of participants, pubkeys is a list of pubkeys, so going back to
our notation e will be calculated as

e = h(m || g^kA + g^kB || 2 || pk_A || pk_B)