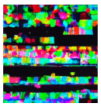

ZKSECURITY

Audit of Darkfi contracts

February 19th, 2024

Introduction

In the week from February 12th to February 16th 2024, zkSecurity performed a security audit of Darkfi's built-in contracts and circuits. The audit was performed on the public darkfi repository https://github.com/darkrenaissance/darkfi at commit `ea50f9ac`. The engagement lasted 1 week and was conducted by 2 consultants.

Scope

The scope of the audit includes the three built-in contracts of the Darkfi protocol:

- **Money contract**. Responsible for the creation and transfer of fungible tokens (native DRK token and custom tokens). Provides the core protocol functions of transaction fees and proof-of-work rewards.
- **DAO contract**. Implements an anonymous DAO, with an onchain proposal and voting system accessible to holders of a governance token.
- **Deployooor contract**. Responsible for deployment and locking of additional user-defined contracts.

Each of these contracts comes in three parts, all of which are in scope of the audit:

- **Onchain contract**. Rust code which is compiled to a WASM binary and run by network participants during transaction application, with read/write access to contract-owned state and read access to other contracts and various network state.
- **zk-SNARK circuits.** Contract code which operates on private data and is executed on the client, along with a zk proof to be verified onchain. Circuits are written in zkas, a custom DSL, and compiled to a halo2 backend. The audit covers the circuit logic, but not the implementation of zkas and halo2 gadgets it uses.
- **Client-side wrapper code**. Rust code which builds transactions, including zk proofs, and submits them to the onchain contract.

The audit explicitly covers the design and implementation of cryptographic schemes used in the contracts and client SDK, including:

- Schnorr signatures over the Pallas curve
- Homomorphic Pedersen commitments
- AEAD encryption
- Elgamal encryption
- ECVRF
- Various commitment and derivation schemes based on the Poseidon hash function
- Poseidon-based sparse Merkle tree

Summary and recommendations

Darkfi's codebase was found to be well-organized, with clarity provided by a large number of inline code comments. The protocol is elegantly designed, powered by a lightweight, unopinionated contract VM, on top of which most of the core protocol is implemented as userland contracts. Great care is taken to preserve user privacy and ownership at all levels of the software stack.

Apart from the findings described in the following section, we offer a general recommendation:

Complete protocol specification. The Darkfi book features a **detailed spec** of the DAO contract, with a section describing each individual contract method: its purpose, inputs, and contract statement. However, the other two contracts are not covered at the same level of detail; and important functionality like transaction fees, proof-of-work rewards, deployment and locking is essentially undocumented. We recommend extending the protocol specification to provide full details about all core contracts.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
00	dao	DAO Governance Tokens Can Be Reused To Meet The Proposal Threshold	High
01	crypto	Elgamal Encryption Is Not Authenticated	Medium
02	crypto	Weak-Fiat Shamir Implementation of Schnorr Signatures	Medium
03	crypto	ECVRF and Schnorr Signatures Don't Use Deterministic Nonces	Low
04	deployooor	Unnecessary ZK Proof Of Contract Id Derivation	Informational

00 - DAO Governance Tokens Can Be Reused To Meet The Proposal Threshold

dao

High

Description. The DAO contract allows the minting of new DAOs with a given threshold of governance tokens to be met for creating proposals. This threshold is checked within a "main propose" proof (`src/contract/dao/proof/propose-main.zk`):

```
circuit "ProposeMain" {
    # TRUNCATED...

    # This is the main check
    # We check that dao_proposer_limit <= total_funds
    one = witness_base(1);
    total_funds_1 = base_add(total_funds, one);
    less_than_strict(dao_proposer_limit, total_funds_1);
```

Where the `total_funds` is a public input to the circuit, that represents the sum of a number of governance tokens owned by the creator of the proposal. To prove that they own that many governance tokens, the creator also accompanies a new proposal with a number of "input propose" proofs. These proofs allow the contract to check that indeed, they are owned by the creator of the proposal, and they are of the correct type, and they sum up to the `total_funds` input.

Unfortunately, the contract does not check that all inputs passed are distinct. As one can see in `src/contract/dao/src/entrypoint/propose.rs` the logic only checks that each input was not already spent:

```
pub(crate) fn dao_propose_process_instruction(
    cid: ContractId,
    call_idx: u32,
    calls: Vec<DarkLeaf<ContractCall>>,
) -> Result<Vec<u8>, ContractError> {
    // TRUNCATED...
    for input in &params.inputs {
        // TRUNCATED...
        // Check the coins weren't already spent
        if db_contains_key(money_nullifier_db, &serialize(&input.nullifier))? {
            msg!("[Dao::Vote] Error: Coin is already spent");
            return Err(DaoError::CoinAlreadySpent.into())
        }
    }
```

This means that a user that does not have enough governance token to create a proposal, could simply reuse the same governance token multiple times to meet the threshold.

Recommendation. Checking for duplicate inputs requires checking for duplicate nullifiers. The same logic exists in `vote.rs` and could be replicated:

```
pub(crate) fn dao_vote_process_instruction(
    cid: ContractId,
    call_idx: u32,
    calls: Vec<DarkLeaf<ContractCall>>,
) -> Result<Vec<u8>, ContractError> {
    // TRUNCATED...
    let mut vote_nullifiers = vec![];
    for input in &params.inputs {
        // TRUNCATED...
        if vote_nullifiers.contains(&input.nullifier) ||
            db_contains_key(dao_vote_nullifier_db, &null_key)?
        {
            msg!("[Dao::Vote] Error: Attempted double vote");
            return Err(DaoError::DoubleVote.into())
        }
        // TRUNCATED...
        vote_nullifiers.push(input.nullifier);
    }
```

01 - Elgamal Encryption Is Not Authenticated

crypto

Medium

Description. The Elgamal encryption implemented in the SDK does not provide authentication. This means that alterations of the ciphertexts lead to the same alterations on the plaintexts.

While we could not find a specific misuse due to the functions always being used in conjunction with proofs as an additional authentication mechanism (proving the correctness of the encryption), the presence of the functionality in the SDK leads to potential for user misuse.\

The reason that tampering of the ciphertexts is easy, is that encrypted values are simply the values added to shared blinding values:

```
let mut encrypted_values = [pallas::Base::ZERO; N];
for i in 0..N {
    encrypted_values[i] = values[i] + blinds[i];
}
```

And thus decryption is simply about removing the blinding values:

```
let mut decrypted_values = [pallas::Base::ZERO; N];
for i in 0..N {
    decrypted_values[i] = self.encrypted_values[i] - blinds[i];
}
```

So for example, adding the value 1 to a ciphertext will result in a decryption of a plaintext value added to 1 as well.

Recommendation. We recommend renaming the function to `encrypt_not_authenticated` or `encrypt_unsafe` and to document that the function must be used in conjunction with an authentication mechanism, such as a proof of correctness.

02 - Weak-Fiat Shamir Implementation of Schnorr Signatures

crypto

Medium

Description. As the public key of the signer is not included in the derivation of the challenge (during signing), the implemented signature scheme does not prove knowledge of the private key. This is called a weak Fiat-Shamir instantiation of the protocol, which is documented in more detail in papers like **How not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios**.

The issue is that one can create tuples of (public key, signature, message) that are valid without knowing the private key associated with the public key; with the caveat that the public key cannot be chosen with great accuracy.

As one can see, this is possible due to the following equation which does not include `self` in the `hash_to_scalar` function:

```
impl SchnorrPublic for PublicKey {
    fn verify(&self, message: &[u8], signature: &Signature) -> bool {
        let challenge = hash_to_scalar(DRK_SCHNORR_DOMAIN, &signature.commit.to_bytes(),
message);
        NullifierK.generator() * signature.response - self.inner() * challenge ==
signature.commit
    }
}
```

Due to this, one can set `signature.commit` and `signature.response` to random values, and then set the public key to `challenge_inv * (signature.commit - signature.response * NullifierK.generator())` (where `challenge_inv` is the inverse of the challenge). Different public keys can be obtained by modifying the `signature.response` value, but it is too hard to obtain a specific public key this way.

Recommendation. To fix this, simply add the public key to the `hash_to_scalar` function in both the signing and verification functions.

03 - ECVRF and Schnorr Signatures Don't Use Deterministic Nonces

crypto

Low

Description. Both the Schnorr signature and the ECVRF implementations use random nonces by sampling them with a random-number generator (RNG). While the code is deemed secure, this is generally considered bad practice as misimplementations and other bugs can lead to catastrophic failures. For example, a nonce reuse in Schnorr leads to the leakage of the private key.

```
let k = pallas::Scalar::random(rng);
```

For this reason, RFC 9381 specifies a function called `ECVRF_nonce_generation` that produces the nonce deterministically in the ECVRF scheme.

RFC 6979 specifies the same technique for ECDSA, and RFC 8032 specifies deterministic nonce generation for EdDSA (which is a Schnorr signature scheme).

Recommendation. Follow the 5.4.2. ECVRF Nonce Generation section of the RFC to implement the deterministic nonce generation for ECVRF.

For Schnorr, follow RFC 8032 which specifies the same techniques but for EdDSA.

04 - Unnecessary ZK Proof Of Contract Id Derivation

deployooor

Informational

Description. Both methods of the deployer contract, `DeployV1` and `LockV1`, require a zk proof of the circuit in `derive_contract_id.zk`. This small circuit consists of two parts:

- Derivation of a public key from a private `deploy_key`.
- Derivation of the contract ID from the public key.

Public key and contract ID are revealed as public inputs.

Verifying this proof in the deployer contract implies that a contract can only be (re)deployed or locked by the owner of its `deploy_key`.

However, this property is already ensured by other steps in the onchain contract logic:

```
// Public keys for the transaction signatures we have to verify
let signature_pubkeys: Vec<PublicKey> = vec![params.public_key];

// Derive the ContractID from the public key
let (sig_x, sig_y) = params.public_key.xy();
let contract_id = ContractId::derive_public(params.public_key);
```

In the code snippet above, the contract takes a public key from the call parameters and requires a signature from it on the transaction. It derives the contract ID from the same public key.

These steps alone ensure that a contract can only be changed by the owner of the `deploy_key` the contract ID is derived from. The zk proof is therefore redundant and can be removed.

Recommendation. Remove the unnecessary proof verification step from the deployer contract.