DClex Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0kage

carlitox477

Assisting Auditors

Hans

Alex Roan

June 13, 2023

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

DCLEX is a trading platform that facilitates transactions using USDC and allows users to withdraw stocks as cryptographic tokens on Ethereum. Tokenized securities on DCLEX are ERC-20 compliant but require permission for use, making them tradable peer-to-peer and compatible with valid smart contracts. Non-transferable NFTs serve as digital identities, granting permissions to access tokenized securities, which can be reused in various applications. Valid digital identities are required for both senders and receivers to interact with stocks on-chain.

Key Actors:

- Users: Engage in buying, selling, and transferring tokenized stocks/USDC.
- Protocol: Approves digital IDs for wallets and smart contracts.
- KYC/AML Third-Party Service Provider: Conducts KYC (with digital ID subject to KYC) as engaged by the protocol.
- Circle: USDC issuer providing a private account for DCLEX, facilitating USDC to USD conversion and vice versa.

Components:

- DClex Frontend: User interface for interacting with the platform.
- DClex Backend: Receives frontend requests and interacts with smart contracts, Dclex Circle account, Dclex Bank Account, and DClex brokerage account.
- DClex Circle Account: Maintains a private account with Circle, used for depositing USD and minting USDC or depositing USDC and converting it back to USD.
- DClex Bank Account: Facilitates the transfer of USD between Dclex Circle and Dclex Brokerage accounts.
- DClex Brokerage Account: Directly interacts with the stock market for buying/selling stocks, backing tokenized assets with corresponding underlying stocks.

- DClex Smart Contracts: On-chain contracts controlling the creation, minting, and burning of tokenized stocks, as well as issuing digital and smart contract identities. These contracts were audited.

Smart Contract Overview:

- Digital Identity: Issues non-transferable digital identities to users who complete the KYC process. Admins can invalidate existing IDs and mint new IDs to specific addresses. Inherits ERC721 with customizations for non-transferability.
- Smart Contract Identity: Issues smart contract identities (SCIDs) to proposed smart contracts by users with existing digital identities. Admins can invalidate existing IDs and mint new IDs to specific addresses. Inherits ERC721 with customizations for non-transferability.
- Stocks: Tokenized representation of stocks based on the ERC20 standard, enabling transfer to users or smart contracts with registered IDs. Contracts can be paused, allowing the protocol to handle specific scenarios like token delisting. Minting and force transfer are controlled by the Factory contract.
- Vault: Manages USDC token withdrawals and deposits, allowing normal and emergency scenarios. Users can emergency withdraw tokens accidentally sent to the vault, even when paused. Admins hold the power to withdraw tokens to external addresses.
- Factory: Core contract holding a mapping of all stocks and their symbols. Users can mint and burn stocks and force transfer tokens with a valid signature from DCLEX. Contract is pausable with access control distinguishing user and admin roles.
- Token Builder: Deploys new tokenized stock contracts, created exclusively by the Factory contract.
- SignatureUtils: Provides transaction approval by DCLEX backend through signed parameters. Hashes are created for user actions like minting, burning, emergency transfers, etc.

Security Remarks:

- Event Logging: The audit reveals that specific on-chain actions are not properly logged. Enhancing on-chain event logging is recommended to ensure seamless operation of on/off-chain components.
- Centralization and Trust: DCLEX retains unrestricted powers to mint, burn, and transfer tokens without user consent, increasing centralization and trust. Recommendations have been made to reduce centralization and trust levels while maintaining key platform functionality.
- Unknown Off-chain risks: DCLEX is a hybrid protocol that incorporates both on-chain and off-chain components. During our audit, it was identified that significant portions of the protocol operate off-chain, which fall outside the scope of our assessment. These off-chain operations encompass crucial functions such as verifying the backing of tokenized assets by underlying stocks in the DCLEX exchange, validating digital identity issuance only after completing KYC requirements, and ensuring accurate conversion of USDC deposits into liquidity on the exchange. Evaluating these operations & their associated risks requires assessing the reliability, robustness, and error-free functioning of the off-chain infrastructure, which is beyond the purview of this audit.
- Smart contract identity risks: DCLEX demonstrates a commitment to innovation through the introduction of novel concepts such as smart contract digital IDs and leveraging these smart contracts to develop advanced financial infrastructure, including peer-to-peer exchanges and liquidity pools. Currently, the primary criterion for issuing smart contract IDs appears to be the completion of audits. However, we assert that while audits are necessary, they are insufficient in assessing the suitability of smart contracts for ID issuance. Given the programmable and composable nature of smart contracts, allowing them to manage user tokenized assets introduces unknown and higher-order risks that cannot be fully evaluated at this stage.To mitigate potential risks effectively, we recommend implementing a robust and objective set of criteria to determine which contracts are eligible for ID issuance. Such criteria should go beyond the audit process and take into account factors such as code complexity, potential vulnerabilities, and the overall security of the contracts. By establishing a comprehensive framework for assessing smart contracts, the platform can ensure that only contracts meeting stringent security standards are permitted to handle tokenized assets, thereby reducing the exposure to unforeseen risks. Adopting a thorough evaluation process for smart contract ID issuance will enhance the overall security and stability of the DCLEX platform, promoting trust among users and mitigating potential vulnerabilities associated with the management of tokenized assets.

5 Executive Summary

Over the course of 6 days, the Cyfrin team conducted an audit on the DClex smart contracts provided by DClex. In this period, a total of 21 issues were found.

Summary

Project Name	DClex
Repository	dclex-blockchain
Commit	c37490b1e432...
Audit Timeline	May 15th - May 22nd 2023
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	5
Medium Risk	3
Low Risk	1
Informational	7
Gas Optimizations	5
Total Issues	21

Summary of Findings

[H-1] If a Pool DID is invalidated by protocol, users can potentially lose access to the tokens listed in pool	Resolved
[H-2] `Factory::changeNameSymbol` should check that new symbol does not override an old symbol, otherwise multiple critical factory functionalities would be affected	Resolved
[H-3] Anyone can call and get any token in the vault if the vault is paused through Vault::emergencyWithdrawWhenPaused, including USDC	Resolved
[H-4] `Factory::forceTransfer` does not check if the `from` and `to` addresses have valid digital signature ID	Resolved
[H-5] A malicious user can procure a digital identity for a smart contract without an audit	Acknowledged
[M-1] `setValidity` can be front-run to transfer out all tokens to a user with valid DID	Acknowledged
[M-2] Any user who deposits tokens just prior to change in `stock multiplier` stands to lose some portion of his assets	Acknowledged

[M-3] Functions such as `forceMintStocks`, `forceBurnStocks` and `force-TransferAdmin` are too powerful and can cause serious damage to users	Acknowledged
[L-1] Missing emissions for key events	Resolved
[I-1] `Stocks::mintTo` seems to allow minting to users holding invalid DID	Acknowledged
[I-2] Smart Contract Identity creation process needs a rule based approach	Acknowledged
[I-3] Signed transactions cannot be canceled	Resolved
[I-4] Emergency withdrawals does not consider stuck ETH	Resolved
[I-5] Modularize DID and SCID validation for `Stocks.transfer` and `Stocks.transferFrom`	Resolved
[I-6] Change `Stocks` name to `Stock`	Resolved
[I-7] Old symbols are not deleted from `Stocks.stocks` array when they are changed	Acknowledged
[G-1] Unnecessary modifiers can be omitted	Resolved
[G-2] State variables only set in the constructor should be declared immutable	Resolved
[G-3] State variables should be cached in stack variables rather than re-reading them from storage	Resolved
[G-4] Turn DID and SCID into immutable variables in `Stocks` contract	Acknowledged
[G-5] Multiple address/ID mappings can be combined into a single mapping of an address/ID to a struct, where appropriate	Resolved

6 Findings

6.1 High Risk

6.1.1 If a Pool DID is invalidated by protocol, users can potentially lose access to the tokens listed in pool

Description: The protocol envisages that pools with their smart contract ID (SCID) can provide liquidity to users who want to exchange different tokenized assets in a `trustless third-party model`. Since all assets are backed by underlying stocks held in the brokerage account, such a model can allow for trading tokenized stocks between market makers (liquidity providers with valid Digital Identity, DID) and takers (users who also have a valid DID).

However, just as in the case of DID, protocol admins have the power to invalidate SCID. Based on our discussions with the protocol team during the audit, it is understood that the exercise of such power will be contingent upon legal compliance or a court order. In a scenario where a pool ID is invalidated, assets of genuine users with valid DIDs will be trapped in the pool without a way to exit.

Notice that `Stocks::transferFrom` checks that both the sender and receiver have valid SCID/DID to execute a transaction successfully.

```
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public override(IERC20, ERC20Named) whenNotPaused returns (bool) {
        if (!DID().isValid(DID().getId(from)) && !SCID().isValid(SCID().getId(from))) revert
        ↪  InvalidDID();
        //@audit checks valid ID for both from and to accounts
        if (to.code.length == 0 && !DID().isValid(DID().getId(to))) revert InvalidDID();
        if (SCID().balanceOf(to) > 0) {
            if (!SCID().isValid(SCID().getId(to))) revert InvalidSmartcontract();
        }


        ...
    }
```

The same is true for the `Stocks::transfer` function:

```
    function transfer(address to, uint256 amount) public override(IERC20, ERC20Named) whenNotPaused
    ↪  returns (bool) {
        if (!DID().isValid(DID().getId(msg.sender)) && !SCID().isValid(SCID().getId(msg.sender)))
        ↪  revert InvalidDID();
        if (to.code.length == 0 && !DID().isValid(DID().getId(to))) revert InvalidDID();
        if (SCID().balanceOf(to) > 0) {
            if (!SCID().isValid(SCID().getId(to))) revert InvalidSmartcontract();
        }

   //@audit checks valid SCID/DID for sender and receiver
  ....
    }
```

Impact: Tokens trapped in the liquidity pools have no way to be transferred out. As we've seen with Tornado Cash, a potential sanction of specific smart contracts is not an unrealistic possibility. In that case, users were anonymous and involved only in holding digital assets. In the DCLEX scenario, a loss of tokens could mean a permanent loss of underlying stocks.

Recommended Mitigation: The protocol must allow every affected user to call `Factory::forceTransfer` to pull out trapped tokens. An alternate approach is to use the protocol's overarching powers to call

`Factory::forceBurnStocks` to burn pool stocks and `Factory::forceMintStocks` to compensate every affected user.

Both options seem impractical at a large scale & we believe that this recovery process invariably can lead to losses for some users. We recommend that protocol design appropriate measures to protect users if smart contracts are sanctioned.

DCLEX: `Factory::forceTransfer` is modified to allow for transfers even if `from` address is sanctioned , ie. DID for that address is invalidated.

Cyfrin: Resolved. Verified changes made to the `Factory::forceTransfer` function to enable transfers from addresses with invalid DIDs. Changes made are deemed satisfactory.

6.1.2 `Factory::changeNameSymbol` should check that new symbol does not override an old symbol, otherwise multiple critical factory functionalities would be affected

Description: The existing implementation of `Factory::changeNameSymbol` fails to verify whether the newly provided symbol is already in use. It is also noted that the mapping corresponding to `stocks[oldSymbol]` is not deleted when the old symbol gets replaced with a new one.

```
    function changeNameSymbol(
        string calldata oldSymbol,
        string calldata name,
        string calldata symbol // @audit This is the new symbol
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        IStocks(stocks[oldSymbol]).changeNameSymbol(name, symbol); // The old symbol is changed
        stocks[symbol] = stocks[oldSymbol]; // @audit here we can use any new symbol value, if this
        ↪  previous symbol was already in use it will be overriden
        symbols.push(symbol);
        // @audit stocks[oldSymbol] was never set to address(0)
    }
```

Impact: Almost every functionality related to the `Factory` that uses the new symbol will be compromised. Specifically:

- forceMintStocks: The factory will be unable to forcefully mint stocks for the stock whose symbol has been overridden. Consequently, it may mistakenly mint stocks for a different stock.
- forceBurnStocks: The factory will be unable to forcefully burn stocks for the stock whose symbol has been overridden. As a result, it might erroneously burn stocks for a different stock.
- mintStocks: The factory will be unable to mint stocks for the stock whose symbol has been overridden. This could lead to the unintended minting of stocks for a different stock.
- forceBurnStocks: The factory will be unable to burn stocks for the stock whose symbol has been overridden. Consequently, it may erroneously burn stocks for a different stock.
- forceTransfer: The factory will be unable to forcibly transfer the stock whose symbol has been overridden. This could result in an erroneous transfer of a different stock.
- forceTransferAdmin
- pauseStocks: The factory will be unable to pause the stock whose symbol has been overridden.
- unpauseStocks: The factory will be unable to unpause the stock whose symbol has been overridden.
- setStockMultiplier: The factory will be unable to modify the multiplier values of the stock whose symbol has been overridden. Moreover, it may unintentionally override the values of the new/old stock, depending on the original intentions.

Furthermore, suppose there are transactions in the mempool that were initiated before the override of the old symbol. In that case, those transactions will still be executed, but with the contract currently being referenced. This situation poses an extremely hazardous scenario for the protocol, especially if a liquidity pool designed to interact with stocks is implemented. In such a case, incorrect tokenized stocks could be minted and traded for USDC. Consequently, users would be able to sell stocks they were not intended to possess initially.

Considering the provided information, the severity of this issue has been determined based on the following facts:

- The only individuals capable of exploiting this bug possess the `DEFAULT_ADMIN_ROLE`. Potential motivations for individuals with this role to execute this bug include:
  1. Simple human error
  2. Compromise of their private keys
  3. Social engineering tactics to manipulate the responsible party associated with an address holding the `DEFAULT_ADMIN_ROLE` to change the name to an already used symbol. This attack can be driven by the desire to profit from the bug.
  4. Intention to inflict harm upon the DClex protocol.
- Likelihood: LOW, as long as the private keys of addresses holding the DEFAULT_ADMIN_ROLE remain secure and undisclosed.
- Impact: CRITICAL. Although the current incorrect minting, burning, and transfer of stocks can be reverted by an admin, future protocols built on top of DClex may experience severe disruptions in functionality, as demonstrated by the liquidity pool example. Such events could lead to legal repercussions for DClex.

Considering the combination of low likelihood and critical impact, the severity of this issue is deemed **HIGH**.

Proof of Concept: Coded PoC

Recommended Mitigation: Simply check that the new symbol is not mapped to an existing stock contract.

```
function changeNameSymbol(string calldata oldSymbol, string calldata name, string calldata symbol)
↪  external onlyRole(DEFAULT_ADMIN_ROLE) {
+      require(stocks[symbol] == address(0));
        IStocks(stocks[oldSymbol]).changeNameSymbol(name, symbol);
        stocks[symbol] = stocks[oldSymbol];
        symbols.push(symbol);
    }
```

DCLEX: Added a `if` condition to verify if a symbol already exists.

Cyfrin: Resolved. Verified commit `7eb980c`. The changes made have been reviewed and deemed satisfactory.

6.1.3 Anyone can call and get any token in the vault if the vault is paused through Vault::emergencyWithdrawWhenPaused, including USDC

Description: The current implementation of `Vault::emergencyWithdrawWhenPaused` presents a critical vulnerability if the vault is paused. It permits any user to invoke the function when this state is reached, enabling unrestricted withdrawal of all tokens held within the vault, including USDC.

Unlike the`Vault::emergencyWithdrawal` function, `Vault::emergencyWithdrawWhenPaused` does not require a backend signature. This aspect further confirms that the function is intended solely for retrieving non-USDC tokens that may have been inadvertently trapped. However, the existing implementation of `Vault::emergencyWithdrawWhenPaused` fails to incorporate proper validation to address this concern.

Impact: If the Vault is paused, this function allows the caller to transfer USDC tokens to their own address, thereby facilitating the depletion of the Vault's funds.

Despite DClex's assertion that this function is intended exclusively for use in a hard fork scenario, there remains the possibility that the function may be paused for other reasons, either voluntarily or due to external enforcement by a legal authority. In such cases, any individual who invokes this function could withdraw all USDC tokens held within the contract.

It also must be mentioned that the current implementation is unfair and causes a loss of non-USDC tokens for DClex users, given that the first user who executes this function after pause can drain all tokens accidentally sent by other users to the vault.

Proof of Concept: Given the current `Vault::emergencyWithdrawWhenPaused` implementation:

```
    function emergencyWithdrawWhenPaused(
        address token, // @audit Nothing forbid token = USDC
        uint256 amount
    ) external whenPaused nonReentrant {
        IERC20(token).transfer(msg.sender, amount);
    }
```

If a law officer orders to freeze the whole protocol when the vault is frozen, anyone can withdraw all the USDC held by the contract.

Considering the provided information, the severity of the situation is assessed based on the following factors:

- Likelihood: MEDIUM. The regulation of cryptocurrency and blockchain products is known to be unpredictable. In a scenario where a judge issues an order to freeze the DClex protocol, resulting in the Vault being frozen, the likelihood of this event occurring is not unreasonable.
- Impact: HIGH. Under normal circumstances, this function can drain all the USDC held within the Vault. Therefore, if the vulnerability is exploited, the impact would be severe.
- Motivations: Monetary incentive. The potential for financial gain serves as a motivating factor for malicious actors to exploit this vulnerability.

Considering the combination of medium likelihood, monetary motivations, and high impact, the overall severity of the issue is determined to be HIGH.

Recommended Mitigation: *Option 1*: Restrict this function to non-USDC, as USDC withdrawals are already handled. This mean:

```
    function emergencyWithdrawWhenPaused(
        address token,
        uint256 amount
    ) external whenPaused nonReentrant {
+       if( token == USDC) revert ERROR_CANNOT_WITHDRAW_USDC_THROUGH_emergencyWithdrawWhenPaused ;
        IERC20(token).transfer(msg.sender, amount);
    }
```

This option does not contemplate the unfair withdrawal of non-USDC ERC20 tokens accidentally sent to the contract.

Option 2: Allow the function execution only via signatures. This way, the backend can verify who made the transfer and only allow a user to withdraw a specific token amount.

DCLEX: Added restriction for non-usdc tokens

Cyfrin: Resolved. Verified in commit `7eb980c`. We acknowledge that the current change restricts transfers to only non USDC tokens. However, current implementation does not address a scenario where the first user executing this function after a pause can potentially drain all tokens sent by other users to the vault, posing a risk to the entire token balance.

6.1.4 `Factory::forceTransfer` does not check if the `from` and `to` addresses have valid digital signature ID

Description: `Factory::forceTransfer` allows users to transfer tokenized stock from one address (`transfer.account`) to another (`to`). According to protocol docs, if a user updates his Ethereum address, this feature allows that user to transfer tokenized stocks from the old to the new address.

However, the function does not check if the `to` address has a valid Digital ID.

```
function forceTransfer(Transfer calldata transfer, bytes calldata signature) external {
    if (stocks[transfer.symbol] == address(0)) revert StockNotExists();
    if (nonces[transfer.nonce]) revert InvalidNonce();


    address creator = utils.recoverTransfer(transfer, signature);
    if (!hasRole(DEFAULT_ADMIN_ROLE, creator)) revert WrongSignature();
    nonces[transfer.nonce] = true;

    IStocks(stocks[transfer.symbol]).forceTransfer(transfer.account, transfer.to, transfer.amount);
    // @audit missing check on whether transfer.account and transfer.to have valid DIDs
    // a user can send a to account that is not a valid DID

    emit Events.ForceTransfer(transfer.symbol, transfer.account, transfer.to, transfer.amount);
    ↪   //-n emits force transfer event
}
```

`Stocks::forceTransfer` does not have a DID check either:

```
function forceTransfer(address from, address to, uint256 amount) external
↪   onlyRole(DEFAULT_ADMIN_ROLE) {
    _transfer(from, to, amount);  //@audit lacks check if from and to have valid DIDs

}
```

This check is missing even at the time of creating the transfer hash in `SignatureUtils::hashTransfer`

```
function hashTransfer(Transfer calldata transfer) private view returns (bytes32) {
     //@audit transfer.account and transfer.to lack DID check
     return
     _hashTypedDataV4(
         keccak256(
             abi.encode(
                 keccak256(
                     "Transfer(string symbol,address account,address to,uint256 amount,uint256
                     ↪   nonce)"
                 ),
                 keccak256(abi.encodePacked(transfer.symbol)),
                 transfer.account,
                 transfer.to,
                 transfer.amount,
                 transfer.nonce
             )
         )
     );
}
```

Impact: Sending tokens to an account without Digital ID can lock up tokens. Although documentation does provide a recovery option if a user directly transfers tokens to a non-DID address, this scenario is different because core protocol functions enable this transfer.

Recommended Mitigation: Multiple options to mitigate this issue:

1. Check digital ID when creating hashes for operations such as `mint`, `withdraw`, and `transfer`.
2. Check digital ID inside `Stocks::forceTransfer`

We believe Option 1 is more robust but leave it as a design choice to protocol devs.

DCLEX: `Factory::forceTransfer` is modified to include verification of DID for `to` address. No valid DID check for `from` address was introduced to handle scenarios where `from` address could be invalidated.

Cyfrin: Resolved. Verified commit 7eb980c . Changes made to the `Factory::forceTransfer` have been reviewed and deemed satisfactory.

6.1.5 A malicious user can procure a digital identity for a smart contract without an audit

Description: Protocol allows existing users with a valid Digital Identity (DID) to propose a smart contract to get its own smart contract digital identity (SID). To do this, a user has to submit the smart contract code, audit report, and whitepaper to the protocol. Protocol, at its discretion, can provide a SID to the said smart contract.

However, a malicious user can bypass this process by submitting a `smart contract` address instead of EOA (Externally Owned Address) address while registering for a Digital Identity (DID). Note that this smart contract need not be deployed at the time of submitting & its address can be deterministically computed by the user using wallet address and transaction nonce higher than current transaction count.

The `DigitalIdentity:mint` function can be called by a user who has completed his KYC to mint a digital identity. To mint a DID, a user submits an account address. Note below that the only check in this function is that `msg.sender` is the same as the `mintStruct.account` that is part of the signature approved by DCLEX admins.

```
    function mint(MintDID calldata mintStruct, bytes calldata signature) external whenNotPaused {
        if (mintStruct.account != msg.sender) revert InvalidSender(); //@audit -> no check if sender is
        ↪  an EOA or contract
        if (balanceOf(mintStruct.account) > 0) revert AlreadyHasDID();
        if (nonces[mintStruct.nonce]) revert InvalidNonce();

        address creator = utils.recoverMintDID(mintStruct, signature);
        if (!hasRole(DEFAULT_ADMIN_ROLE, creator)) revert WrongSignature();

        nonces[mintStruct.nonce] = true;
        __mint(mintStruct.account, mintStruct.uri);
    }
```

The function then calls `DigitalIdentity:__mint` to mint the actual NFT. Note that the only condition to check that the `account` that gets an identity is not a smart contract is by checking `account.code.length != 0`. **However, if the user deploys a smart contract that calls `DigitalIdentity:mint` function from within its constructor, this condition will not revert**. The address does not store the code in storage when the constructor is executed allowing a smart contract to mask itself as an externally owned address.

```
    function __mint(address account, string calldata uri) private {
        if (account.code.length != 0) revert MintToContract(); //@audit if a smart contract calls
        ↪  __mint in its constructor, this will not revert
        uint256 tokenId = ids[account];
        if (tokenId == 0) {
            _counter.increment();
            tokenId = _counter.current();
            ids[account] = tokenId;
        }
        _mint(account, tokenId);
        _setTokenURI(tokenId, uri);
        valids[tokenId] = true;
        emit Events.MintDID(account, tokenId);
    }
```

Impact: Any smart contract with malicious code can procure a digital ID. This allows users to create malicious pools that can potentially drain all tokens in the proposed `peer-to-pool-to-peer` trustless model.

Recommended Mitigation: One possible mitigation is to listen to the `MintDID` event off-chain & immediately check if the freshly minted account has code in its storage. If it does, admins can immediately call `DigitalIdentity:setValids` to invalidate the ID assigned to that account.

DCLEX: We implemented a MintDID listener on backend, once address is changed from EOA to a smart contract, the token is invalidated immediately

Cyfrin: The team has shared their mitigation plan for addressing this issue within their off-chain infrastructure. While the proposed implementation appears adequate in principle, Cyfrin could not verify its actual implementation due to limitations in access privileges, expertise, and scope constraints.

6.2 Medium Risk

6.2.1 `setValidity` can be front-run to transfer out all tokens to a user with valid DID

Description: In the event of legal compliance or court orders, the protocol aims to freeze users' assets by invalidating their digital IDs. This is done by `DigitalIdentity::setValids` function that allows the protocol admin to set the `isValid` to `false`. Once a digital ID is invalidated, all key functions, including minting, burning, and transfers, are frozen. However, since invalidation happens on-chain, a malicious user can front-run the `DigitalIdentity::setValids` transaction and simply call the `Stocks::transfer` to transfer all assets to a non-sanctioned account.

Note that `Stocks::transfer` is a single-step transaction that does not need any signatures from the DCLEX admin.

```
    function transfer(address to, uint256 amount) public override(IERC20, ERC20Named) whenNotPaused
    ↪   returns (bool) {
        if (!DID().isValid(DID().getId(msg.sender)) && !SCID().isValid(SCID().getId(msg.sender)))
        ↪   revert InvalidDID();
        if (to.code.length == 0 && !DID().isValid(DID().getId(to))) revert InvalidDID();
        if (SCID().balanceOf(to) > 0) {
            if (!SCID().isValid(SCID().getId(to))) revert InvalidSmartcontract();
        }

        address owner = _msgSender();
        _transfer(owner, to, amount);
        return true;
    }
```

Impact: By front-running the `DigitalIdentity::setValids` transaction, a sanctioned user can drain all assets from his account before sanctions come into force. This makes the sanctioning process ineffective.

Recommended Mitigation: In the extreme case of sanctioning specific accounts, consider ways to restrict transfers for a limited period before invoking sanctions. One possible way is to `pause` the specific `Stock` contract before invalidating a digital ID but pausing a stock contract will stop transactions even for genuine buyers.

We do recommend appropriate design changes to mitigate front-running risks.

DCLEX: We will watch the transactions around such an event and optionally punish other bad actors account supporting the front-running.

Cyfrin: Acknowledged. Although the mitigation steps appear adequate in principle, it is essential to note that the proposed plan heavily relies on off-chain infrastructure for addressing the issue. Cyfrin has not verified the implementation due to access privileges, expertise, and scope limitations.

6.2.2 Any user who deposits tokens just prior to change in `stock multiplier` stands to lose some portion of his assets

Description: Protocol has a concept of `multiplier` to consider a conversion from tokens to assets & vice-versa in the event of stock splits and reverse splits. Protocol admins can change the multiplier via the `Stocks::setMultiplier` function. Protocol admins have informed us that the multiplier is updated on-chain & in the back-end simultaneously.

Since the stock split is an off-chain event that is manually updated on-chain by submitting `setMultiplier` on-chain, there is a possibility that on-chain contracts and off-chain back-end become out of sync. This can occur if users call `Factory::mintStocks` or `Factory::burnStocks` in a block whose timestamp is **before** the block timestamp in which multiplier gets updated & **after** stock split comes into effect off-chain.

In such cases, two scenarios are possible:

1. User tokenizing stocks might get lesser tokens based on pre-split multiplier
2. User de-tokenizing stocks might get lesser stocks in brokerage accounts for tokens burnt

Mint/burn of tokens might lead to token/asset losses as conversions happen on the pre-split multiplier.

Impact: Tracking of stock splits/reverse-splits and other such events is done off-chain. Timing risks involved in manually updating the multiplier could mean that users could lose a significant portion of their stock only because their transaction was submitted before the transaction in which multiplier gets updated.

Recommended Mitigation: There is no simple solution to this because of business logic's off-chain/on-chain split. One possible mitigation could be that, just like in the case of stock delisting, protocol could keep track of impending stock split events and pause the specific `Stocks` contract on days when the split will happen.

We realise that pausing/unpausing may create a bad user experience & hence leave the specific mitigation methods to the protocol team. Our only recommendation is to mitigate the risks of token holders losing part of their assets in the scenario described above.

DCLEX: We will stop signing mint transaction and increase the number of blocks required to confirm deposits of stocks around the event of changing a multiplier.

Cyfrin: Acknowledged. Although the mitigation steps appear adequate in principle, it is essential to note that the proposed plan heavily relies on off-chain infrastructure for addressing the issue. Cyfrin has not verified the implementation due to access privileges, expertise, and scope limitations.

6.2.3 Functions such as `forceMintStocks`, `forceBurnStocks` and `forceTransferAdmin` are too powerful and can cause serious damage to users

Description: Three protocol functions that can directly impact user assets are:

1. `forceMintStocks` -> protocol admin can mint any tokenized stocks. A malicious admin can instantly dilute each token's value by minting many tokens into existence.
2. `forceBurnStocks` -> protocol admin can burn any tokenized stocks from any account without the owner's consent. Randomly burning tokens can make a user permanently lose access to underlying stock held in a brokerage account.
3. `forceTransferAdmin` -> protocol admin can transfer Alice's entire balance to Bob without Alice's consent.

Web3 protocols follow a design principle that even if private keys of the admin wallet are compromised, a rogue admin should not be able to directly steal funds from protocol users. A rogue admin can, of course, change vital protocol settings, such as fees, etc., to cause indirect damage to users. Still, from a security standpoint, protocol design should not allow even admins to touch user assets without consent.

Current design also has no on-chain governance where such actions are voted upon before execution.

Impact: The current implementation has dangerous functions that can instantly cause loss of assets to users. There have been instances in the past where admin wallets were compromised - if such a scenario were to happen with DCLEX, the potential loss to user assets would be significant.

Recommended Mitigation: Several mitigation steps are recommended here:

- Consider having an admin that is a multi-sig with a large enough quorum.
- Consider splitting these critical functions into a 2-step logic - admin1 will initiate the action, and admin2 will complete the action
- Consider adding a reasonable time delay between actions proposed above. Users who don't agree with the logic for using such functions will have adequate time to detokenise and exit the platform
- Documentation should clearly state that protocol has powers to mint, burn and transfer tokens. Users should be fully aware of possible protocol admin actions.

We recommend a review of existing design that introduces necessary checks to protocol admin powers when making unilateral changes to token balances.

DCLEX: Partially intended and duly acknowledged. `ForceTransferAdmin` is removed. `ForceMint` and `ForceBurn` functionalities are now limited to the MASTER_ADMIN role, which is assigned to a Gnosis safe wallet.

Cyfrin: Acknowledged. Verified commit `7eb980c`. The changes made to `ForceTransferAdmin` have been verified and are deemed satisfactory. While acknowledging the heightened protection achieved by restricting access to key functions through the MASTER_ADMIN role assigned to a Gnosis safe wallet, we recommend that the protocol proactively considers the implementation of additional security measures in the future. These measures could include the introduction of timelocks, 2-step execution processes, and governance mechanisms to further enhance the overall security posture.

6.3 Low Risk

6.3.1 Missing emissions for key events

Description: Protocol has several key actions that are currently not logging on-chain events. Some of these are:

Action	Contract	Importance
pauseStocks	Stocks	When a token for a specific stock is paused
unpauseStocks	Stocks	When a token for a specific stock is unpaused
setValid	DigitalIdentity	When an account is invalidated/re-validated
setValid	SmartcontractIdentity	When a smart contract is invalidated/re-validated
forceMintStocks	Factory	A user mint differs from admin-forced minting. Same event for both
forceBurnStocks	Factory	A user burn differs from admin-forced burning. Same event for both
changeNameSymbol	Factory	When a token symbol is changed
setStockMultiplier	Factory	When a stock multiplier is updated owing to a stock split
emergencyTokenWithdrawal	Factory	When tokens in stock/factory are withdrawn by users

Since DCLEX relies significantly on off-chain infrastructure, listening to on-chain events and initiating relevant actions is important. Events for actions such as `setValid` need to be captured to ensure legal compliance, as on-chain actions come with a clear timestamp/block number.

Impact: Off-chain infrastructure needs to listen to on-chain events to ensure the seamless functioning of DCLEX backend infrastructure. Lack of event capture may lead to compliance issues and out-of-sync off/on-chain states.

Recommended Mitigation: Consider reviewing all key actions in the protocol and emitting proper events that capture key action parameters.

DCLEX: Added and renamed events and indexed event parameters.

Cyfrin: Resolved. Verified commit `7eb980c`. The changes made have been reviewed and deemed satisfactory.

6.4 Informational

6.4.1 `Stocks::mintTo` seems to allow minting to users holding invalid DID

A Digital Identity ID can be set to invalid. The current implementation allows any holder of a DigitalIdentity NFT to receive tokenized shares minted by the administrator.

It would be advisable also to check its validity before the transfer.

`Stocks.sol` lines 32-35.

DCLEX: Accepted.

Cyfrin: Resolved. Account validity is duly verified before minting DID to that account.

6.4.2 Smart Contract Identity creation process needs a rule based approach

Description: Protocol allows existing users with a valid Digital Identity (DID) to propose a smart contract to get their own smart contract digital identity (SID). To do this, a user has to submit the smart contract code, audit report, whitepaper to the protocol and answer specific questions.

During the audit, we did not get a clear understanding on:

- What aspects would be covered under this Audit
- What kind of smart contracts would be considered for SCID & what contracts would be excluded?
- Can upgradeable smart contracts be given SCID?

Since smart contracts allow users to create secondary exchanges, liquidity pools, etc., issuing SCIDs can bring in a lot of second-order risks that cannot be evaluated without a clear set of rules. While necessary, having an audited smart contract code is not sufficient to issue a SCID. For example, a fully audited proxy smart contract can always bring in a malicious implementation at a later date that can cause harm to existing users who interact with that smart contract.

Impact: While DCLEX can leverage powerful composability by issuing SCIDs, such power also significantly expands the attack vectors for the platform. A malicious SCID contract that can attract many individual users can cause damage at scale.

Recommended Mitigation: Consider having rule-based, specific criteria for accepting smart contracts. The current process is very broad-based and doesn't have the rigour necessary to address the unknown risks associated with smart contract composability.

DCLEX: Proper code of conduct/rules for approving smart contracts will be established. Proxy contracts would not be allowed.

Cyfrin: Acknowledged.

6.4.3 Signed transactions cannot be canceled

The current state dictates that a signed transaction can only be invalidated through execution. Even though it is currently possible to prevent the execution of a transaction if the corresponding DID is invalidated, it would be prudent to contemplate the inclusion of a mechanism that enables the cancellation of a nonce that has not yet been utilized.

Here are two scenarios that demonstrate the potential utility of implementing a mechanism to cancel nonces:

1. User regrets tokenizing a stock: If a user expresses regret to DClex regarding the tokenization of a stock, and the company decides to return the stocks to the user off-chain, the existing signature remains valid. A nonce cancellation mechanism would allow for the invalidation of the corresponding nonce, providing an additional layer of assurance.
2. Mitigating the risk of malicious user actions following a human error by DClex based on user honesty: In the event of a human error by DClex, an unscrupulous user could potentially exploit the situation to harm the protocol. The steps involved in such a scenario could be as follows:

   1. The user requests a signed transaction to mint tokenized stocks.
   2. The user subsequently informs the company about an error and expresses the desire for the stocks to remain off-chain.
   3. DClex, assuming the user's honesty, accepts the request.
   4. The malicious user executes the signed transaction and proceeds to sell the stocks through a liquidity pool (LP) with its corresponding SCID. The funds obtained from the sale are then sent to another address.
   5. The malicious user then claims their private keys have been compromised.

3. DClex human error: If DClex accidentally signs a transaction with an incorrect amount or incorrect recipient, the transaction can still be used by users.

Therefore, it is advisable to contemplate the inclusion of methods that enable the cancellation of nonces within the following contracts:

1. Factory Contract
2. SmartcontractIdentity Contract
3. Vault Contract

By introducing these cancellation mechanisms, users, and even administrators, would have the ability to invalidate nonces associated with transactions, thereby providing an additional layer of control and security within these contracts.

DCLEX: Added an admin-only function `useNonces`.

Cyfrin: Resolved.

6.4.4 Emergency withdrawals does not consider stuck ETH

It should be noted that while the contract is not designed to accept ETH, it is crucial to be aware of the possibility that any Ethereum address can receive ETH through the execution of the self-destruct method in another contract. This can lead to a situation where the ETH becomes irretrievably locked within the contracts, making it impossible to access those funds.

Consider next mitigation steps:

```
// dclex/DigitalIdentity.sol
    function emergencyTokenWithdrawal(
        address token,
        address to,
        uint256 amount
    ) external onlyRole(DEFAULT_ADMIN_ROLE) nonReentrant {
-       IERC20(token).transfer(to, amount);
+       if(token == address(0)){
+           to.call{value: amount}("");
+       } else{
+           IERC20(token).transfer(to, amount);
+       }
    }
```

```
// dclex/Factory.sol
// Factory::emergencyWithdrawal
    //...
    if (withdrawal.account == address(this)) {
-       IERC20(withdrawal.token).transfer(withdrawal.to, withdrawal.amount);
+       if(withdrawal.token == address(0)){
+           to.call{value: amount}("");
+       }else{
+           IERC20(withdrawal.token).transfer(withdrawal.to, withdrawal.amount);
+       }
    } else {
    //...
```

```
// dclex/SmartcontractIdentity.sol
// SmartcontractIdentity::emergencyWithdrawal
    function emergencyTokenWithdrawal(
        address token,
        address to,
        uint256 amount
    ) external onlyRole(DEFAULT_ADMIN_ROLE) nonReentrant {
-       IERC20(token).transfer(to, amount);
+       if(token == address(0)){
+           to.call{value: amount}("");
+       } else{
+           IERC20(token).transfer(to, amount);
+       }
    }
```

```
// dclex/Stocks.sol
// Stocks::emergencyWithdrawal
    function emergencyTokenWithdrawal(
        address token,
        address to,
        uint256 amount
    ) external onlyRole(DEFAULT_ADMIN_ROLE) nonReentrant {
-       IERC20(token).transfer(to, amount);
+       if(token == address(0)){
+           to.call{value: amount}("");
+       } else{
+           IERC20(token).transfer(to, amount);
+       }
    }
```

```
// dclex/Vault.sol
// Vault::emergencyWithdrawalAdmin
    function emergencyWithdrawalAdmin(
        address token,
        address to,
        uint256 amount
    ) external onlyRole(MASTER_ADMIN_ROLE) nonReentrant {
        if (token == address(USDC)) revert NotUSDC();
-       IERC20(token).transfer(to, amount);
+       if(token == address(0)){
+           to.call{value: amount}("");
+       } else{
+           IERC20(token).transfer(to, amount);
+       }
```

```
    }

//..
// Vault::emergencyWithdrawal
// Vault::emergencyWithdrawal
    function emergencyWithdrawal(
            Withdrawal calldata withdrawal,
            bytes memory signature
        ) external nonReentrant {
        // ...
        nonces[withdrawal.nonce] = true;

-       IERC20(withdrawal.token).transfer(withdrawal.to, withdrawal.amount);
+       if(withdrawal.token == address(0)){
+           withdrawal.to.call{value: withdrawal.amount}("");
+       } else{
+           IERC20(withdrawal.token).transfer(withdrawal.to, withdrawal.amount);
+       }
    }
//...
// Vault::emergencyWithdrawWhenPaused
// Vault::emergencyWithdrawWhenPaused
    function emergencyWithdrawWhenPaused(
        address token,
        uint256 amount
    ) external whenPaused nonReentrant {
-       IERC20(token).transfer(msg.sender, amount);
+       if(token == address(0)){
+           msg.sender.call{value: amount}("");
+       } else{
+           IERC20(token).transfer(msg.sender, amount);
+       }
    }
```

Here is an example of how `selfdestruct` works.

DCLEX: Accepted.

Cyfrin: Resolved.

6.4.5 Modularize DID and SCID validation for `Stocks.transfer` and `Stocks.transferFrom`

The corresponding check for both functions regarding the validity of the DID/SCID can be modularized in a modifier, saving gas during deployment and improving the quality of the code.

```
modifier checkTransferActors(address from, address to){
    // Cache variables.
    // Given that factory does not change, a DID and SCID cannot change, Next code can change if DID,
↪   and SCID are created as public immutable variables
    IDID _DID = DID();
    IDID _SCID = SCID();
    if (
        !_DID.isValid(_DID.getId(from)) &&
        !_SCID.isValid(_SCID.getId(from))
        ) revert InvalidDID();
    if (
        to.code.length == 0 &&
        !_DID.isValid(_DID.getId(to))
        ) revert InvalidDID();
    if (_SCID.balanceOf(to) > 0) {
        if (
            !_SCID.isValid(_SCID.getId(to))
            ) revert InvalidSmartcontract();
    }
    _;
}
```

After these changes, modify `Stocks::transfer` and `Stocks::transferFrom`

```
-   function transfer(address to, uint256 amount) public override(IERC20, ERC20Named) whenNotPaused
↪   returns (bool) {
+   function transfer(address to, uint256 amount) public override(IERC20, ERC20Named) whenNotPaused
↪   checkTransferActors(msg.sender, to) returns (bool) {
-       if (!DID().isValid(DID().getId(msg.sender)) && !SCID().isValid(SCID().getId(msg.sender)))
↪   revert InvalidDID();
-       if (to.code.length == 0 && !DID().isValid(DID().getId(to))) revert InvalidDID();
-       if (SCID().balanceOf(to) > 0) {
-           if (!SCID().isValid(SCID().getId(to))) revert InvalidSmartcontract();
-       }
-
        address owner = _msgSender();
        _transfer(owner, to, amount);
        return true;
    }
    function transferFrom(
        address from,
        address to,
        uint256 amount
-   ) public override(IERC20, ERC20Named) whenNotPaused returns (bool) {
+   ) public override(IERC20, ERC20Named) whenNotPaused checkTransferActors(from, to) returns (bool) {
-       if (!DID().isValid(DID().getId(from)) && !SCID().isValid(SCID().getId(from))) revert
↪   InvalidDID();
-       if (to.code.length == 0 && !DID().isValid(DID().getId(to))) revert InvalidDID();
-       if (SCID().balanceOf(to) > 0) {
-           if (!SCID().isValid(SCID().getId(to))) revert InvalidSmartcontract();
-       }
-
-
        address spender = _msgSender();
        _spendAllowance(from, spender, amount);
        _transfer(from, to, amount);
        return true;
    }
```

DCLEX: Accepted.

Cyfrin: Resolved. Team has addressed the issue by adding the `checkTransferActors` modifier to the Stock contract, which has been applied to both the `transfer` and `transferFrom` functions in the Stock contract.

6.4.6 Change `Stocks` name to `Stock`

The `Stocks` contract name should be changed to `Stock` to follow the standard OOP class naming. Using singular names instead of plurals to name a class in programming follows the principle of clarity and conciseness. Singular names accurately represent a class representing a single instance or entity, and it aligns with encapsulating a single responsibility within a class, promoting code readability and maintainability. Additionally, singular names facilitate consistency in naming conventions throughout the codebase, enhancing overall code comprehension.

DCLEX: Changed to “Stock”.

Cyfrin: Resolved.

6.4.7 Old symbols are not deleted from `Stocks.stocks` array when they are changed

The current `Factory::changeNameSymbol` function lets you change the symbol of a stock. However, it doesn’t remove the old symbol from the `stocks` data structure, which keeps track of all the symbol names. This can create confusing situations when dealing with events related to a stock with its symbol changed.

DCLEX: This is on purpose to account for instances when users forget about name/symbol change. No changes needed.

Cyfrin: Acknowledged.

6.5 Gas Optimization

6.5.1 Unnecessary modifiers can be omitted

Given that the following functions are getters, `whenNotPaused` modifier can be avoided, saving gas by not executing its operations

```
    // DigitalIdentity::isValid
-   function isValid(uint256 id) external view whenNotPaused returns (bool) {
+   function isValid(uint256 id) external view returns (bool) {
        return valids[id];
    }
```

```
    // SmartcontractIdentity::isValid
-   function isValid(uint256 id) external view whenNotPaused returns (bool) {
+   function isValid(uint256 id) external view returns (bool) {
        return valids[id];
    }
```

DCLEX: Acknowledged.

Cyfrin: Verified.

6.5.2 State variables only set in the constructor should be declared immutable

Avoids a Gsset (20000 gas) in the constructor, and replaces each Gwarmacces (100 gas) with a PUSH32 (3 gas).

Instances:

1. `DigitalIdentity::utils`
2. `Factory::utils`
3. `Factory::DID`
4. `Factory::SCID`
5. `SmartcontractIdentity::utils`
6. `SmartcontractIdentity::factory`
7. `Stocks::factory`
8. `TokenBuilder::factory`
9. `Vault::USDC`
10. `Vault::utils`

DClex: Mitigated.

Cyfrin: Validated. The issues regarding `Vault::USDC` and `Vault::utils` were not mitigated as a result of an error on our part. These issues were reported with the correct link reference, but they were mistakenly attributed to variables in `TokenBuilder` instead of `Vault` in the initial report. The current report has rectified this mistake.

6.5.3 State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replaces each Gwarmaccess (100 gas) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs or having local caches of state variable contracts/addresses.

```
// Factory::forceBurnStocks
+   address _stock = stocks[symbol];
-   if (stocks[symbol] == address(0)) revert StockNotExists();
-   IStocks(stocks[symbol]).burnFrom(account, amount);
+   if (_stock == address(0)) revert StockNotExists();
+   IStocks(_stock).burnFrom(account, amount);
    emit Events.Burn(symbol, account, amount);
```

```
// Factory::mintStocks
-   if (stocks[mint.symbol] == address(0)) revert StockNotExists();
+   address _stock = stocks[mint.symbol];
+   if (_stock == address(0)) revert StockNotExists();
    if (nonces[mint.nonce]) revert InvalidNonce();


    address creator = utils.recoverMint(mint, signature);

    if (!hasRole(DEFAULT_ADMIN_ROLE, creator)) revert WrongSignature();

    nonces[mint.nonce] = true;

-   IStocks(stocks[mint.symbol]).mintTo(mint.account, mint.amount);
+   IStocks(_stock).mintTo(mint.account, mint.amount);
```

```
// Factory::burnStocks
-   if (stocks[burn.symbol] == address(0)) revert StockNotExists();
+   address _stock = stocks[burn.symbol];
+   if (_stock == address(0)) revert StockNotExists();
    if (nonces[burn.nonce]) revert InvalidNonce();

    address creator = utils.recoverBurn(burn, signature);

    if (!hasRole(DEFAULT_ADMIN_ROLE, creator)) revert WrongSignature();

    nonces[burn.nonce] = true;

-   IStocks(stocks[burn.symbol]).burnFrom(burn.account, burn.amount);
+   IStocks(_stock).burnFrom(burn.account, burn.amount);
```

```
// Factory::forceTransfer
-   if (stocks[transfer.symbol] == address(0)) revert StockNotExists();
+   address _stock = stocks[transfer.symbol];
+   if (_stock == address(0)) revert StockNotExists();
    if (nonces[transfer.nonce]) revert InvalidNonce();

    address creator = utils.recoverTransfer(transfer, signature);
    if (!hasRole(DEFAULT_ADMIN_ROLE, creator)) revert WrongSignature();
    nonces[transfer.nonce] = true;

-   IStocks(stocks[transfer.symbol]).forceTransfer(transfer.account, transfer.to, transfer.amount);
+   IStocks(_stock).forceTransfer(transfer.account, transfer.to, transfer.amount);
```

```
// Factory::setStockMultiplier
    function setStockMultiplier(
        string calldata symbol,
        uint256 numerator,
        uint256 denominator
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
-       if (stocks[symbol] == address(0)) revert StockNotExists();
-       IStocks(stocks[symbol]).setMultiplier(numerator, denominator);
+       address _stock = stocks[symbol];
+       if (_stock == address(0)) revert StockNotExists();
+       IStocks(_stock).setMultiplier(numerator, denominator);
    }
```

```
// TokenBuilder::createToken
    function createToken(
        string memory name,
        string memory symbol
    ) external returns (address) {
-       require(msg.sender == factory);
-       Stocks stocks = new Stocks(name, symbol, factory);
-
-       stocks.grantRole(0x00, factory);
-       stocks.grantRole(MASTER_ADMIN_ROLE, factory);
+       address _factory = factory;
+       require(msg.sender == _factory);
+       Stocks stocks = new Stocks(name, symbol, _factory);
+
+       stocks.grantRole(0x00, _factory);
+       stocks.grantRole(MASTER_ADMIN_ROLE, _factory);
        stocks.revokeRole(0x00, address(this));
        stocks.revokeRole(MASTER_ADMIN_ROLE, address(this));
        return address(stocks);
    }
```

```
// Vault::withdraw
-   if (withdrawal.token != address(USDC)) revert NotUSDC();
+   address _USDC = address(USDC);
+   if (withdrawal.token != _USDC) revert NotUSDC
    if (withdrawal.account != address(this)) revert InvalidFromAddress();
    if (withdrawal.to != msg.sender) revert InvalidToAddress();
-   if (USDC.balanceOf(address(this)) < withdrawal.amount) revert WrongAmount();
+   if (IERC20(_USDC).balanceOf(address(this)) < withdrawal.amount) revert WrongAmount();

    address creator = utils.recoverWithdrawal(withdrawal, signature);
    if (!hasRole(DEFAULT_ADMIN_ROLE, creator)) revert WrongSignature();

    nonces[withdrawal.nonce] = true;

-   USDC.transfer(withdrawal.to, withdrawal.amount);
+   IERC20(_USDC).transfer(withdrawal.to, withdrawal.amount);
```

If any of these variables are turned immutable, the modifications are not required.

DCLEX: Accepted.

Cyfrin: Resolved. Verified in commit `7eb980c`.

6.5.4 Turn DID and SCID into immutable variables in `Stocks` contract

The current implementation of the `Stocks` contract can be optimized in terms of gas usage by replacing the public methods `Stocks::DID()` and `Stocks::SCID()` with two immutable variables, namely `DID` and `SCID`, if and only if the factory is not intended to act as a proxy. This approach eliminates the need for method invocations, reducing gas consumption. It is important to note that these variables, once set within the `Factory`, cannot be modified since their values related to `DID` and `SCID` is intended to remain constant throughout the contract's lifespan.

DCLEX: Identity token contracts may change. No code change applied

Cyfrin: Acknowledged.

6.5.5 Multiple address/ID mappings can be combined into a single mapping of an address/ID to a struct, where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, we can avoid a Gsset (20000 gas) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and fit in the same storage slot. Finally, suppose both fields are accessed in the same function. In that case, it can save ~42 gas per access due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations.

```
    // SmartcontractIdentity.sol
    /// @notice token id -> FullKYC passed bool
    mapping(uint256 => bool) private valids;

    /// @notice token id -> address EOA who proposed the token
    mapping(uint256 => address) proposers;
```

`SmartcontractIdentity.sol` lines 31-35.

DCLEX: Added `tokenDetails` mapping and ScID struct + DIDTokenDetails for DID.

Cyfrin: Resolved. Verified in commit `7eb980c`.

7 Appendix

8 4naly3er Static Analysis

Output from the 4naly3er tool.

8.1 Gas Optimizations

	Issue	Instances
GAS-1	Use assembly to check for `address(0)`	19
GAS-2	Using bools for storage incurs overhead	6
GAS-3	Cache array length outside of loop	4
GAS-4	Use calldata instead of memory for function arguments that do not get mutated	6
GAS-5	For Operations that will not overflow, you could use unchecked	4
GAS-6	Use Custom Errors	11
GAS-7	Don't initialize variables with default value	4
GAS-8	Long revert strings	8
GAS-9	Functions guaranteed to revert when called by normal users can be marked `payable`	27
GAS-10	`++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)	4
GAS-11	Splitting require() statements that use && saves gas	1
GAS-12	Use != 0 instead of > 0 for unsigned integer comparison	5

8.1.1 [GAS-1] Use assembly to check for `address(0)`

Saves 6 gas per instance

Instances (19):

```
File: dclex/DigitalIdentity.sol

37:         require(_utils != address(0));
```

```
File: dclex/ERC20Named.sol

236:         require(from != address(0), "ERC20: transfer from the zero address");

237:         require(to != address(0), "ERC20: transfer to the zero address");

265:         require(account != address(0), "ERC20: mint to the zero address");

291:         require(account != address(0), "ERC20: burn from the zero address");

326:         require(owner != address(0), "ERC20: approve from the zero address");

327:         require(spender != address(0), "ERC20: approve to the zero address");
```

```
File: dclex/Factory.sol

41:         if (stocks[symbol] != address(0)) revert StockAlreadyExists();

63:         if (stocks[symbol] == address(0)) revert StockNotExists();

73:         if (stocks[mint.symbol] == address(0)) revert StockNotExists();

94:         if (stocks[burn.symbol] == address(0)) revert StockNotExists();

112:         if (stocks[transfer.symbol] == address(0)) revert StockNotExists();

162:         if (stocks[symbol] == address(0)) revert StockNotExists();

180:             if (stocks[IStocks(withdrawal.account).symbol()] == address(0) &&

207:         require(_did != address(0));

212:         require(_scid != address(0));
```

```
File: dclex/SmartcontractIdentity.sol

40:         require(_utils != address(0) && _factory != address(0));
```

```
File: dclex/Vault.sol

22:         require(_usdc != address(0));

23:         require(_utils != address(0));
```

DCLEX: Let's not change ERC implementations

Cyfrin: Acknowledged

8.1.2 [GAS-2] Using bools for storage incurs overhead

Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past. See source.

Instances (6):

```
File: dclex/DigitalIdentity.sol

28:     mapping(uint256 => bool) private nonces;

31:     mapping(uint256 => bool) private valids;
```

```
File: dclex/Factory.sol

19:     mapping(uint256 => bool) private nonces;
```

```
File: dclex/SmartcontractIdentity.sol

29:     mapping(uint256 => bool) private nonces;

32:     mapping(uint256 => bool) private valids;
```

```
File: dclex/Vault.sol

14:     mapping(uint256 => bool) nonces;
```

DCLEX: Replaced bools to uints. Added constants to Model.sol (TRUE/FALSE)

Cyfrin: Resolved.

8.1.3 [GAS-3] Cache array length outside of loop

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Instances (4):

```
File: dclex/DigitalIdentity.sol

77:         for (uint i = 0; i < _ids.length; i++) {
```

```
File: dclex/Factory.sol

143:         for (uint256 i = 0; i < _symbols.length; i++) {

151:         for (uint256 i = 0; i < _symbols.length; i++) {
```

```
File: dclex/SmartcontractIdentity.sol

84:         for (uint i = 0; i < _ids.length; i++) {
```

8.1.4 [GAS-4] Use calldata instead of memory for function arguments that do not get mutated

Mark data types as `calldata` instead of `memory` where possible. This makes it so that the data is not automatically loaded into memory. If the data passed into the function does not need to be changed (like updating values in an array), it can be passed in as `calldata`. The one exception to this is if the argument must later be passed into another function that takes an argument that specifies `memory` storage.

Instances (6):

```
File: dclex/DigitalIdentity.sol

35:     constructor(string memory _name, string memory _symbol, address _utils)
```

```
File: dclex/ERC20Named.sol

59:     constructor(string memory name_, string memory symbol_) {
```

```
File: dclex/SmartcontractIdentity.sol

38:     constructor(string memory _name, string memory _symbol, address _utils, address _factory)
```

```
File: dclex/Stocks.sol

22:     constructor(string memory name, string memory _symbol, address _factory)
```

```
File: dclex/TokenBuilder.sol

23:     function createToken(string memory name, string memory symbol) external returns (address) {
```

```
File: dclex/Vault.sol

68:     function emergencyWithdrawal(Withdrawal calldata withdrawal, bytes memory signature) external
↪   nonReentrant {
```

DCLEX: Let's not change ERC implementations.

Cyfrin: Acknowledged.

8.1.5 [GAS-5] For Operations that will not overflow, you could use unchecked

Instances (4):

```
File: dclex/DigitalIdentity.sol

77:         for (uint i = 0; i < _ids.length; i++) {
```

```
File: dclex/Factory.sol

143:        for (uint256 i = 0; i < _symbols.length; i++) {

151:        for (uint256 i = 0; i < _symbols.length; i++) {
```

```
File: dclex/SmartcontractIdentity.sol

84:        for (uint i = 0; i < _ids.length; i++) {
```

8.1.6 [GAS-6] Use Custom Errors

Source Instead of using error strings, to reduce deployment and runtime cost, you should use Custom Errors. This would save both deployment and runtime cost.

Instances (11):

```
File: dclex/ERC20Named.sol

209:        require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");

236:        require(from != address(0), "ERC20: transfer from the zero address");

237:        require(to != address(0), "ERC20: transfer to the zero address");

242:        require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");

265:        require(account != address(0), "ERC20: mint to the zero address");

291:        require(account != address(0), "ERC20: burn from the zero address");

296:        require(accountBalance >= amount, "ERC20: burn amount exceeds balance");

326:        require(owner != address(0), "ERC20: approve from the zero address");

327:        require(spender != address(0), "ERC20: approve to the zero address");

348:            require(currentAllowance >= amount, "ERC20: insufficient allowance");
```

```
File: dclex/SmartcontractIdentity.sol

108:        require(factory.getDID().isValid(factory.getDID().getId(msg.sender)), "DCLEX: Proposer
↪   needs valid DID");
```

8.1.7 [GAS-7] Don't initialize variables with default value

Instances (4):

```
File: dclex/DigitalIdentity.sol

77:         for (uint i = 0; i < _ids.length; i++) {
```

```
File: dclex/Factory.sol

143:         for (uint256 i = 0; i < _symbols.length; i++) {

151:         for (uint256 i = 0; i < _symbols.length; i++) {
```

```
File: dclex/SmartcontractIdentity.sol

84:         for (uint i = 0; i < _ids.length; i++) {
```

8.1.8 [GAS-8] Long revert strings

Instances (8):

```
File: dclex/ERC20Named.sol

209:         require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");

236:         require(from != address(0), "ERC20: transfer from the zero address");

237:         require(to != address(0), "ERC20: transfer to the zero address");

242:         require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");

291:         require(account != address(0), "ERC20: burn from the zero address");

296:         require(accountBalance >= amount, "ERC20: burn amount exceeds balance");

326:         require(owner != address(0), "ERC20: approve from the zero address");

327:         require(spender != address(0), "ERC20: approve to the zero address");
```

8.1.9 [GAS-9] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

Instances (27):

```
File: dclex/DigitalIdentity.sol

45:     function mintAdmin(address account, string calldata uri) external whenNotPaused
↪   onlyRole(DEFAULT_ADMIN_ROLE) {

75:     function setValids(uint256[] calldata _ids, bool[] calldata isValids) external
↪   onlyRole(DEFAULT_ADMIN_ROLE) {

153:     function emergencyTokenWithdrawal(address token, address to, uint256 amount) external
↪   onlyRole(DEFAULT_ADMIN_ROLE) nonReentrant {
```

```
File: dclex/Factory.sol

40:     function createStocks(string calldata name, string calldata symbol) external
↪   onlyRole(DEFAULT_ADMIN_ROLE) whenNotPaused {

53:     function forceMintStocks(string calldata symbol, address account, uint256 amount) external
↪   onlyRole(MASTER_ADMIN_ROLE) {

62:     function forceBurnStocks(string calldata symbol, address account, uint256 amount) external
↪   onlyRole(MASTER_ADMIN_ROLE) {

129:     function forceTransferAdmin(string calldata symbol, address from, address to, uint256 amount)
↪   external onlyRole(MASTER_ADMIN_ROLE) {

134:     function changeNameSymbol(string calldata oldSymbol, string calldata name, string calldata
↪   symbol) external onlyRole(DEFAULT_ADMIN_ROLE) {

142:     function pauseStocks(string[] calldata _symbols) external onlyRole(MASTER_ADMIN_ROLE) {

150:     function unpauseStocks(string[] calldata _symbols) external onlyRole(MASTER_ADMIN_ROLE) {

161:     function setStockMultiplier(string calldata symbol, uint256 numerator, uint256 denominator)
↪   external onlyRole(DEFAULT_ADMIN_ROLE) {

189:     function setTokenBuilder(address _builder) external onlyRole(MASTER_ADMIN_ROLE) {

206:     function setDID(address _did) external onlyRole(MASTER_ADMIN_ROLE) {

211:     function setSCID(address _scid) external onlyRole(MASTER_ADMIN_ROLE) {
```

```
File: dclex/Security.sol

22:     function pause() external onlyRole(MASTER_ADMIN_ROLE) {

26:     function unpause() external onlyRole(MASTER_ADMIN_ROLE) {
```

```
File: dclex/SmartcontractIdentity.sol

49:     function mintAdmin(address account, string calldata uri) external whenNotPaused
↪  onlyRole(DEFAULT_ADMIN_ROLE) {

82:     function setValids(uint256[] calldata _ids, bool[] calldata isValids) external
↪  onlyRole(DEFAULT_ADMIN_ROLE) {

146:     function emergencyTokenWithdrawal(address token, address to, uint256 amount) external
↪  onlyRole(DEFAULT_ADMIN_ROLE) nonReentrant {
```

```
File: dclex/Stocks.sol

32:     function mintTo(address account, uint256 amount) external onlyRole(DEFAULT_ADMIN_ROLE)
↪  whenNotPaused {

41:     function burnFrom(address account, uint256 amount) external onlyRole(DEFAULT_ADMIN_ROLE) {

88:     function forceTransfer(address from, address to, uint256 amount) external
↪  onlyRole(DEFAULT_ADMIN_ROLE) {

97:     function emergencyTokenWithdrawal(address token, address to, uint256 amount) external
↪  onlyRole(DEFAULT_ADMIN_ROLE) nonReentrant {

105:     function setMultiplier(uint256 numerator, uint256 denominator) external
↪  onlyRole(DEFAULT_ADMIN_ROLE) {

112:     function changeNameSymbol(string calldata name, string calldata symbol_) external
↪  onlyRole(DEFAULT_ADMIN_ROLE) {
```

```
File: dclex/Vault.sol

32:     function withdrawAdmin(address to, uint256 amount) external onlyRole(DEFAULT_ADMIN_ROLE) {

59:     function emergencyWithdrawalAdmin(address token, address to, uint256 amount) external
↪  onlyRole(MASTER_ADMIN_ROLE) nonReentrant {
```

DCLEX: Fixing may lead to unwanted calls for emergency withdrawals later on. No change.

Cyfrin: Acknowledged.

8.1.10 [GAS-10] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

Saves 5 gas per loop

Instances (4):

```
File: dclex/DigitalIdentity.sol

77:         for (uint i = 0; i < _ids.length; i++) {
```

```
File: dclex/Factory.sol

143:         for (uint256 i = 0; i < _symbols.length; i++) {

151:         for (uint256 i = 0; i < _symbols.length; i++) {
```

```
File: dclex/SmartcontractIdentity.sol

84:         for (uint i = 0; i < _ids.length; i++) {
```

8.1.11 [GAS-11] Splitting require() statements that use && saves gas

Instances (1):

```
File: dclex/SmartcontractIdentity.sol

40:         require(_utils != address(0) && _factory != address(0));
```

8.1.12 [GAS-12] Use != 0 instead of > 0 for unsigned integer comparison

Instances (5):

```
File: dclex/DigitalIdentity.sol

46:         if (balanceOf(account) > 0) revert AlreadyHasDID();

55:         if (balanceOf(mintStruct.account) > 0) revert AlreadyHasDID();
```

```
File: dclex/SmartcontractIdentity.sol

61:         if (balanceOf(mintStruct.account) > 0) revert AlreadyHasDID();
```

```
File: dclex/Stocks.sol

52:         if (SCID().balanceOf(to) > 0) {

73:         if (SCID().balanceOf(to) > 0) {
```

8.2 Non Critical Issues

	Issue	Instances
NC-1	Missing checks for `address(0)` when assigning values to address state variables	1
NC-2	`require()` / `revert()` statements should have descriptive reason strings	8

	Issue	Instances
NC-3	Event is missing `indexed` fields	6
NC-4	Functions not used internally could be marked external	8

8.2.1 [NC-1] Missing checks for `address(0)` when assigning values to address state variables

Instances (1):

```
File: dclex/TokenBuilder.sol

16:        factory = _factory;
```

8.2.2 [NC-2] `require()` / `revert()` statements should have descriptive reason strings

Instances (8):

```
File: dclex/DigitalIdentity.sol

37:        require(_utils != address(0));
```

```
File: dclex/Factory.sol

190:        require(ITokenBuilder(_builder).getFactory() == address(this));

207:        require(_did != address(0));

212:        require(_scid != address(0));
```

```
File: dclex/SmartcontractIdentity.sol

40:        require(_utils != address(0) && _factory != address(0));
```

```
File: dclex/TokenBuilder.sol

24:        require(msg.sender == factory);
```

```
File: dclex/Vault.sol

22:        require(_usdc != address(0));

23:        require(_utils != address(0));
```

DCLEX: Only affects deployment gas costs; this will be done carefully

Cyfrin: Acknowledged.

8.2.3 [NC-3] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

Instances (6):

```
File: libs/Events.sol

6:     event StocksCreated(string symbol, address tokenAddress);

7:     event Mint(string symbol, address account, uint256 amount);

8:     event Burn(string symbol, address account, uint256 amount);

9:     event MintDID(address account, uint256 tokenId);

10:     event MintSCID(address proposer, address account, uint256 tokenId);

12:     event ForceTransfer(string symbol, address from, address to, uint256 amount);
```

8.2.4 [NC-4] Functions not used internally could be marked external

Instances (8):

```
File: dclex/DigitalIdentity.sol

139:     function tokenURI(uint256 tokenId) public view override(ERC721, ERC721URIStorage) returns
↪  (string memory) {

144:     function supportsInterface(bytes4 interfaceId) public view override(ERC721, AccessControl)
↪  returns (bool){
```

```
File: dclex/Security.sol

30:     function hasRole(bytes32 role, address account) public view override returns (bool) {
```

```
File: dclex/SmartcontractIdentity.sol

132:     function tokenURI(uint256 tokenId) public view override(ERC721, ERC721URIStorage) returns
↪  (string memory) {

137:     function supportsInterface(bytes4 interfaceId) public view override(ERC721, AccessControl)
↪  returns (bool){
```

```
File: dclex/Stocks.sol

49:     function transfer(address to, uint256 amount) public override(IERC20, ERC20Named) whenNotPaused
↪  returns (bool) {

66:     function transferFrom(

126:     function symbol() public view override(ERC20Named, IStocks) returns (string memory) {
```

DCLEX: Let's not change ERC implementations.

Cyfrin: Acknowledged.

8.3 Medium Issues

	Issue	Instances
M-1	Centralization Risk for trusted owners	28

8.3.1 [M-1] Centralization Risk for trusted owners

Impact: Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

Instances (28):

```
File: dclex/DigitalIdentity.sol

45:     function mintAdmin(address account, string calldata uri) external whenNotPaused
↪  onlyRole(DEFAULT_ADMIN_ROLE) {

75:     function setValids(uint256[] calldata _ids, bool[] calldata isValids) external
↪  onlyRole(DEFAULT_ADMIN_ROLE) {

153:     function emergencyTokenWithdrawal(address token, address to, uint256 amount) external
↪  onlyRole(DEFAULT_ADMIN_ROLE) nonReentrant {
```

```
File: dclex/Factory.sol

40:     function createStocks(string calldata name, string calldata symbol) external
↪   onlyRole(DEFAULT_ADMIN_ROLE) whenNotPaused {

53:     function forceMintStocks(string calldata symbol, address account, uint256 amount) external
↪   onlyRole(MASTER_ADMIN_ROLE) {

62:     function forceBurnStocks(string calldata symbol, address account, uint256 amount) external
↪   onlyRole(MASTER_ADMIN_ROLE) {

129:     function forceTransferAdmin(string calldata symbol, address from, address to, uint256 amount)
↪   external onlyRole(MASTER_ADMIN_ROLE) {

134:     function changeNameSymbol(string calldata oldSymbol, string calldata name, string calldata
↪   symbol) external onlyRole(DEFAULT_ADMIN_ROLE) {

142:     function pauseStocks(string[] calldata _symbols) external onlyRole(MASTER_ADMIN_ROLE) {

150:     function unpauseStocks(string[] calldata _symbols) external onlyRole(MASTER_ADMIN_ROLE) {

161:     function setStockMultiplier(string calldata symbol, uint256 numerator, uint256 denominator)
↪   external onlyRole(DEFAULT_ADMIN_ROLE) {

189:     function setTokenBuilder(address _builder) external onlyRole(MASTER_ADMIN_ROLE) {

206:     function setDID(address _did) external onlyRole(MASTER_ADMIN_ROLE) {

211:     function setSCID(address _scid) external onlyRole(MASTER_ADMIN_ROLE) {
```

```
File: dclex/Security.sol

13: abstract contract Security is AccessControl, Pausable, ReentrancyGuard {

22:     function pause() external onlyRole(MASTER_ADMIN_ROLE) {

26:     function unpause() external onlyRole(MASTER_ADMIN_ROLE) {
```

```
File: dclex/SmartcontractIdentity.sol

49:     function mintAdmin(address account, string calldata uri) external whenNotPaused
↪   onlyRole(DEFAULT_ADMIN_ROLE) {

82:     function setValids(uint256[] calldata _ids, bool[] calldata isValids) external
↪   onlyRole(DEFAULT_ADMIN_ROLE) {

146:     function emergencyTokenWithdrawal(address token, address to, uint256 amount) external
↪   onlyRole(DEFAULT_ADMIN_ROLE) nonReentrant {
```

```
File: dclex/Stocks.sol

32:     function mintTo(address account, uint256 amount) external onlyRole(DEFAULT_ADMIN_ROLE)
↪   whenNotPaused {

41:     function burnFrom(address account, uint256 amount) external onlyRole(DEFAULT_ADMIN_ROLE) {

88:     function forceTransfer(address from, address to, uint256 amount) external
↪   onlyRole(DEFAULT_ADMIN_ROLE) {

97:     function emergencyTokenWithdrawal(address token, address to, uint256 amount) external
↪   onlyRole(DEFAULT_ADMIN_ROLE) nonReentrant {

105:     function setMultiplier(uint256 numerator, uint256 denominator) external
↪   onlyRole(DEFAULT_ADMIN_ROLE) {

112:     function changeNameSymbol(string calldata name, string calldata symbol_) external
↪   onlyRole(DEFAULT_ADMIN_ROLE) {
```

```
File: dclex/Vault.sol

32:     function withdrawAdmin(address to, uint256 amount) external onlyRole(DEFAULT_ADMIN_ROLE) {

59:     function emergencyWithdrawalAdmin(address token, address to, uint256 amount) external
↪   onlyRole(MASTER_ADMIN_ROLE) nonReentrant {
```

DCLEX: Gnosis for master admin + working on admin account safety.

Cyfrin: Acknowledged.