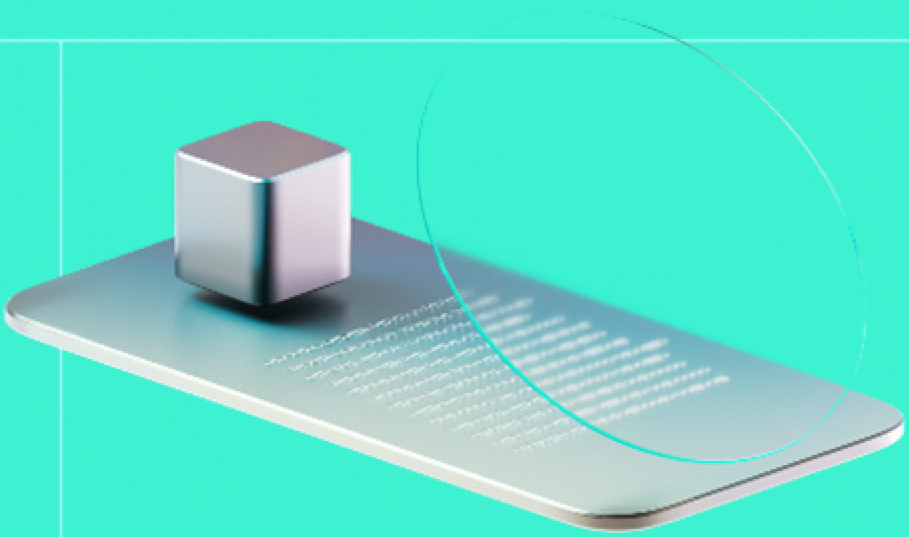
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: USSD

Date: 07/03/2024

We express our gratitude to the USSD team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

USSD is an autonomous stablecoin with crypto collateralization.

Platform: EVM (Ethereum, Arbitrum, BNB Smart Chain, Optimism)

Language: Solidity

Tags: Oracle, Staking, Stablecoin

Timeline: 15/01/2024 - 07/03/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/DavidLeeChaum/USSDv2/
Commit	bf57fe60453768e41fdfb27b683abbffe9a700d6

Audit Summary

9/10	10/10	N/A	10/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9.3/10

The system users should acknowledge all the risks summed up in the risks section of the report. Mitigated issues are excluded from the Total Findings count.

7	6	1	1
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	1
High	2
Medium	3
Low	1

Vulnerability	Status
F-2024-0540 - Arbitrage Opportunities Can Affect The Price Stability	Accepted
F-2024-0497 - Missing Checks for Zero Address	Fixed
F-2024-0515 - Funds Lock On the Insurance Contract	Fixed
F-2024-0518 - The Staking Rewards Are Not Minted	Fixed
F-2024-0519 - Staking Rewards are Not Separated From Users Balances	Fixed
F-2024-0520 - Insurance Tokens are Transferable Independent From Collateralization Level	Fixed
F-2024-0521 - Flashloan Attack to Increase The Staking Rewards	Fixed

Document

Name	Smart Contract Code Review and Security Analysis Report for USSD
Audited By	Maksym Fedorenko, Roman Tiutiun
Approved By	Przemyslaw Swiatowiec
Website	https://www.ussd.ai/
Changelog	22/01/2024 - Preliminary Report
	22/02/2024 - Second Review
	07/03/2024 - Third Review

Table of Contents

System Overview

USSD is an autonomous stablecoin with crypto collateralization with the following contracts:

- USSD — an autonomous on-chain stablecoin system. The contract uses the ERC20 standard for token implementation and includes additional functionality for minting, redeeming, and managing the stablecoin. It also interacts with external contracts and oracles to determine asset prices and manage collateral.
- USSDRewards — is an extension of the ERC20 token standard. This contract is designed to distribute rewards through USSDToken tokens to users who hold the USSDRewards token. The rewards are calculated based on the average percentage yield (APY) over time and the total supply of staked tokens.
- stUSSD — is a staking token, where users can deposit a token (presumably called USSD) and receive stUSSD tokens in return, which may earn rewards over time. The contract includes functionality for depositing, withdrawing, and redeeming tokens, as well as accounting logic to manage the relationship between assets and shares.
- ICT — is a contract that deals with insurance and rewards, using two types of tokens: WETH (Wrapped Ether) and WBGL (presumably another token). The contract allows users to mint new tokens in exchange for WETH or WBGL and to claim insurance under certain conditions.

Privileged roles

- The USSD contract owner has the flexibility to add a staking contract, insurance contract, and configure oracles. Additionally, they can enable WETH and DAI tokens as collateral, but it is important to note that these actions above in this paragraph can be performed only once ever by the admin.
- The admin role may be transferred from one user to another multiple times.

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

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

- Functional requirements are provided
- Technical description is provided.

Code quality

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

Test coverage

Code coverage of the project is **N/A**

- Code coverage cannot be calculated due to the lack of Truffle framework support by the code coverage tools.
- While the project includes tests, they do not cover all possible branches. Test coverage is assumed to be 100% due to the inability to measure coverage.

Security score

Upon auditing, the code was found to contain **1** critical, **2** high, **3** medium, and **1** low severity issues. The **1** critical, **2** high, and **1** medium severity issues were promptly addressed, fixed, mitigated or accepted as part of the remediation efforts. After the remediation check of the audit process, **1** medium severity issues were accepted by the client, leading to a security score of **9** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **9.3**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- The repository includes dependencies beyond the audit scope, introducing potential security risks associated with third-party code such as the solmate library and third-party protocols like Oracles and DEXs. Due to the composability of contracts, these external dependencies pose inherent security risks that are not covered in the audit.
- The **stable coin collaterization ratio might fall lower 100% any time due to the collateral tokens price fluctuations**.
- If the collateral ratio is below 95%, then the conversion of USSD within the protocol is only possible with a **5% discount.** For example with the 0.94 collateral ratio the protocol has only 0.94 cents of collateral to cover one USSD token, but with the discount it would be possible to convert one USSD to only 0.893 USD of collateral (`Collateral ratio * (1 - discount) =  0.94 * 0.95 = 0.893`)
- Before using the system it is recommended to make sure that the proper Oracles addresses are specified within the next variables: `STABLE_ORACLE`, `STABLEDAI_ORACLE`, `WBGL_ORACLE`, `WBTC_ORACLE`, and `WETH_ORACLE`. The reliability of these Price Oracles themselves poses a risk independent of the USSD admin's behavior.
- Excessive minting of USSD may occur, the risk arises not from the minting process itself, but from staking reward minting. This risk could materialize if collateral value updates are delayed. Collateral value updates occur during USSD token minting against collateral or can be triggered by anyone at any time. The admin must ensure timely updates, or trigger them if necessary.
- Insurance contract may not have enough funds to increase the collaterization ratio sufficiently.

Findings

Vulnerability Details

F-2024-0518 - The Staking Rewards Are Not Minted - Critical

Description:	The USSD contract contains a `mintRewards` function intended to mint USSD tokens as interest for staking via the USSD staking or insurance contracts. The function is only callable by `stakingContract` and `insuranceContract` contracts. However, this function is not used within the project, resulting in no interest being generated for staking either ICT or USSD tokens. This contradicts the stated requirements, which specify that ICT and USSD tokens should be capable of being staked to earn interest in the form of USSD tokens.
Assets:	• ussdv2/contracts/USSD.sol []
Status:	Fixed

Classification

Severity:	Critical
Impact:	5/5
Likelihood:	5/5

Recommendations

Recommendation:	Revise the existing logic to ensure the `mintRewards` function is utilized according to the requirements. This change will align the system with its initial requirements, enabling the minting of USSD tokens as a reward for staking ICT and USSD tokens. **Remediation (revised commit: c14801b):** The `mintRewards` function is now called within the `USSDRewards.sol` staking core implementation.

F-2024-0519 - Staking Rewards are Not Separated From Users Balances - High

Description: The `USSDRewards.sol` contract, integral to the staking mechanism, enables users to stake USSD tokens through the `stUSSD.sol` implementation. This contract enables the rewards based on a collateralization ratio exceeding 1.05 over a designated staking period. However, the current implementation permits users to claim the rewards even when the contract does not hold a sufficient amount of USSD tokens to cover all the previously staked tokens and accrued rewards. Such reward withdrawals can deplete the contract's total balance, adversely impacting the funds deposited earlier.

This flaw could potentially lead to a scenario where users are unable to withdraw their tokens due to the contract's insufficient funds, especially if multiple users consequently initiate the withdrawals.

Assets:

- ussdv2/contracts/USSDRewards.sol []

Status: Fixed

Classification

Severity: High

Impact: 4/5

Likelihood: 5/5

Recommendations

Recommendation:

To address the issue, implement a separate variable to account the total staked tokens, distinct from the reward balances. This separation ensures that rewards are not paid out from the tokens staked by users. In the functions that handle user reward claims, by calling the `_claim` function, ensure that these rewards are sourced exclusively from the designated rewards balance, not from the users' staked tokens.

Additionally, introduce a mechanism that allows users to emergently withdraw their staked tokens in scenarios where there is an insufficient rewards balance. This feature provides an extra layer of security and flexibility for users in the event of a shortfall in the rewards fund.

Remediation (revised commit: **c14801b):** The tokens for the rewards are now minted.

F-2024-0521 - Flashloan Attack to Increase The Staking Rewards - High

Description:

The project's contract, `stUSSD.sol`, manages the staking of USSD tokens. Users can stake USSD tokens and claim rewards. Staking is active when the USSD collateralization ratio exceeds `1.05`. The reward depends on the staked token amount and the total USSD token supply, as determined by the `_calculateRewardsPerToken` function, which computes the reward per staked token:

```
collateralValuation = ((IUSSD(address(USSDToken)).collateralFactor()
) *
USSDToken.totalSupply() // total supply of USS token
) / 1e6;

rewardsPerTokenOut.accumulated = (rewardsPerTokenIn.accumulated +
(1e18 * ((collateralValuation * elapsed * targetAPY) / 1e18)) /
totalSupply_)
```

However, before claiming the reward and initiating reward calculation, it is feasible to mint additional USSD tokens (significantly, using a flash loan), thereby inflating the calculated reward. This allows claiming an artificially increased reward and then redeeming the tokens used for USSD minting to repay the flash loan.

This issue could lead to an unexpected distribution of staking rewards, contradicting the technical requirements.

Status: Fixed

Classification

Severity: High

Impact: 4/5

Likelihood: 4/5

Recommendations

Recommendation:

Implement a few blocks “cooldown” mechanism for the consequent USSD token minting and redeeming operations in order to prevent minting and redeeming the tokens with the help of flashloans within the one block to avoid staking manipulations.

Remediation (revised commit: 3aad9e7): Implemented the mechanism which stores the collateral factor and uses the collateral factor at most from the previous block which prevent the ability to execute flashloan attack. The collateral factor is updated on every USSD mint for tokens (`mintForToken`). However it is still possible to deposit significant amount of tokens in one block, mint the rewards in the next block and redeem the initially deposited tokens. Such behaviour is risky and might be arbitraged.

F-2024-0515 - Funds Lock On the Insurance Contract - Medium

Description: The insurance contract in the system enables users to deposit either wETH or wBGL tokens in exchange for ICT (Insurance Capital Treasury) tokens. ICT tokens can be staked to earn USSD interest. In certain conditions, users can use the `insuranceClaim` function to transfer wETH or wBGL tokens to the USSD contract. This function moves 1% of all deposited wETH or wBGL. However, wETH transfers only occur if the insurance balance exceeds 100 ETH (`1e20` wei) or 10,000 wBGL tokens. Below these thresholds, the tokens become permanently locked in the insurance contract, making them unusable.

This design flaw results in the permanent locking of funds, rendering the deposited assets for ICT token minting inaccessible.

Assets:

- ussdv2/contracts/ICT.sol []

Status: Fixed

Classification

Severity: Medium

Impact: 2/5

Likelihood: 5/5

Recommendations

Recommendation: It is recommended to rework the logic to add the ability to utilize the locked funds deposited for minting the ICT tokens, for example, by allowing users to burn the ICT tokens in exchange for initially deposited funds or share of it.

Remediation (revised commit: bf57fe6): All the WETH or WBGL tokens might be utilized as an insurance during the call to the insurance contract.

F-2024-0520 - Insurance Tokens are Transferable Independent From Collateralization Level - Medium

Description: The `ICT.sol` the contract is designed to store insurance tokens for the case when the collateralization level of the stablecoin drops. Per the requirements, when the collateralization rate falls below 90%, the insurance capital should start transferring funds to the main collateral. Specifically, 1% of the insurance capital's coins (wETH or wBGL) will be moved to the main USSD contract every 24 hours until the collateralization level returns to 90%. This process is intended to be executed through the `insuranceClaim` function. However, the current implementation allows this function to transfer funds regardless of the collateralization level, even above the specified threshold.

This behavior is a deviation from the specified requirement that the insurance fund transfer should only occur at a specific collateralization level.

Assets:

- ussdv2/contracts/ICT.sol []

Status: Fixed

Classification

Severity: Medium

Impact: 2/5

Likelihood: 5/5

Recommendations

Recommendation: It is recommended to add the conditional statement, which allows the transferring of the insurance funds only if the collateralization level is 90% or lower.

Remediation (revised commit: **c14801b):** The suggested validations have been implemented, the insurance tokens are now being transfered when the collaterization ratio falls below 90%.

F-2024-0540 - Arbitrage Opportunities Can Affect The Price Stability - Medium

Description: The USSD token's minting process is anchored to the Chainlink price feed. This reliance could potentially cause a depegging scenario due to arbitrage opportunities arising from the discrepancies between Chainlink and Decentralized Exchange (DEX) prices. Such discrepancies occur because DEXs can update prices every block, providing more current pricing data. In contrast, Chainlink's price updates are either delayed by over an hour or triggered only after the price crosses a certain threshold, such as a 0.25% change in the USDT/USD feed.

This delay in Chainlink's price feed updates could lead to price differences with real-time DEX prices, thereby creating arbitrage opportunities that might impact the stability of USSD.

Status: Accepted

Classification

Severity: Medium

Impact: 3/5

Likelihood: 3/5

Recommendations

Recommendation: Implement the price deviation validation across multiple sources (DEXs, Chainlink oracles) in order to prevent the arbitrage opportunities exploiting the Chainlink update price intervals.

Remediation (revised commit: c14801b): According to the client's response: "Introducing more complexity to the mechanism of price comparison would involve more risks of price information being misinterpreted. We acknowledge your concern but this problem does not have an optimal solution yet. Many options would require management, maintenance, or having backend infrastructure (not fully on-chain, not autonomous)."

F-2024-0497 - Missing Checks for Zero Address - Low

Description:	In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the "zero address". This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address. "The Missing zero address control" issue in `changeOwner()` arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior. For instance, a contract might allow tokens to be sent to the zero address without any checks, which essentially burns those tokens as they become irretrievable. While sometimes this is intentional, without proper control or checks, accidental transfers could occur.
Assets:	• ussdv2/contracts/ICT.sol []
Status:	Fixed

Classification

Severity:	Low
Impact:	4/5
Likelihood:	1/5

Recommendations

Recommendation:	It is strongly recommended to implement checks to prevent the zero address from being set during the `changeOwner()` of the contract. This can be achieved by adding require statements that ensure address parameters are not the zero address. **Remediation (revised commit: bf57fe6):** Recommended validation was implemented.

Observation Details

F-2024-0466 - Commented Code Parts - Info

Description: In the contract, `USSD.sol` lines 139 and `StableOracleWBGL` lines 32, 34, and `StableOracleUSDT.sol` lines 35 - 48 are commented on parts of the code. This reduces code quality.

```
USSD.sol:
//require(hasRole(MINTER_ROLE, msg.sender), "minter");

StableOracleWBGL:
//uint256 WETHPriceUSD = ethOracle.getPriceUSD();
//return (WETHPriceUSD * 1e18) / WBGLWETHPrice;

StableOracleUSDT.sol:
/*address[] memory pools = new address[](1);
pools[0] = 0x6fe9E9de56356F7eDBfcBB29FAB7cd69471a4869;
uint256 WBNBUSDTDexPrice = staticOracleUniV3.quoteSpecificPoolsWithT
imePeriod(
1000000000000000000, // 1 BNB
0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c, // WBNB (base token)
0x55d398326f99059fF775485246999027B3197955, // USDT (quote token)
pools, // USDT/WBNB pool uni v3
3600 // period - quote at the block start
);*/

//uint256 WETHUSDFeedPrice = ethOracle.getPriceUSD();

// chainlink price data is 18 decimals for DAI/ETH, so multiply by 1
0 decimals to get 18 decimal fractional
//(uint80 roundID, int256 price, uint256 startedAt, uint256 updatedA
t, uint80 answeredInRound) = priceFeedDAIETH.latestRoundData();
```

Assets:

- ussdv2/contracts/USSD.sol []
- ussdv2/contracts/oracles/StableOracleUSDT.sol []
- ussdv2/contracts/oracles/StableOracleWBGL.sol []

Status: Fixed

Recommendations

Recommendation: Remove commented parts of the code.

Remediation (revised commit: c14801b): The proposed fix has been implemented.

F-2024-0496 - Emitting Incorrect Event - Info

Description: In the `insuranceClaim` function, the event `InsuranceClaim` is emitted with `WETH` as the token address even when the claim is for `WBGL`.

Assets:

- ussdv2/contracts/ICT.sol []

Status: Fixed

Recommendations

Recommendation: It is recommended to emit the correct address in the `InsuranceClaim` event,

Remediation (revised commit: **c14801b):** The proposed fix has been implemented by adding the correct event parameter `WBGL` in `insuranceClaim()` function

F-2024-0510 - Redundant Import Statements - Info

Description:

The contract `StableOracleUSDT` and `StableOracleWBGL` are imported but never used.

This redundancy in import operations has the potential to result in unnecessary gas consumption during deployment and could potentially impact the overall code quality.

Assets:

- ussdv2/contracts/USSD.sol []
- ussdv2/contracts/oracles/StableOracleWBGL.sol []

Status: Fixed

Recommendations

Recommendation:

Remove redundant imports and ensure that the contract is imported only in the required locations, avoiding unnecessary duplications.

Remediation (revised commit: **c14801b):** The proposed fix has been implemented.

F-2024-0511 - Redundant Mathematical Operation - Info

Description: The current implementation in the `USSD.sol` contract performs mathematical operations involving multiplication with `(10 ** decimals()) / 1e36`. Since `decimals()` consistently returns 6, this specific operation is redundant and results in unnecessary gas consumption. To optimize the contract and reduce gas costs, it is advisable to simplify this calculation by directly using 1e30 instead.

Assets:

- ussdv2/contracts/USSD.sol []

Status: Fixed

Recommendations

Recommendation: It is recommended to simplify the aforementioned calculation.

Remediation (revised commit: c14801b): The proposed fix has been implemented by adding `1e30`.in `calculateMint()` function.

F-2024-0514 - Redundant Event Declaration RewardsSet - Info

Description: The contract `USSDRewards` has declared an event `RewardsSet` but never used.

This redundancy in event operations has the potential to result in unnecessary gas consumption during deployment and could potentially impact the overall code quality.

Assets:

- ussdv2/contracts/USSDRewards.sol []

Status: Fixed

Recommendations

Recommendation: Remove the redundant event.

Remediation (revised commit: c14801b): The proposed fix has been implemented.

F-2024-0516 - Missing Variable Visibility Specification - Info

Description: Adding visibility modifiers to variable `lastClaimed` improves code readability and explicitly communicates the intended access level for this variable.

Assets:

- ussdv2/contracts/ICT.sol []

Status: Fixed

Recommendations

Recommendation: Specify the visibility of the `lastClaimed` variable.

Remediation (revised commit: c14801b): The proposed fix has been implemented `lastClaimed` was marked as public.

F-2024-0522 - Redundancy with Solmate and OpenZeppelin in Safe Transfers and ERC-20 Interactions - Info

Description: The smart contracts `USSDRewards.sol`, `USSD.sol`, `stUSSD.sol`, `ICT.sol`, incorporates both Solmate and OpenZeppelin libraries for handling safe transfers and ERC-20 interactions. While these libraries serve similar purposes, integrating both may introduce redundancy, increase contract size, and potentially lead to compatibility issues or conflicts between the two libraries.

Assets:

- ussdv2/contracts/ICT.sol []
- ussdv2/contracts/stUSSD.sol []
- ussdv2/contracts/USSD.sol []
- ussdv2/contracts/USSDRewards.sol []

Status: Fixed

Recommendations

Recommendation: It is recommended to choose either Solmate or OpenZeppelin for handling safe transfers and ERC-20 interactions consistently throughout the contract.

Remediation (revised commit: c14801b): The proposed fix has been changed from OpenZeppelin to Solmate.

F-2024-0539 - Redundant Code Blocks - Info

Description: In the `StableOracleWBGLV2.sol,` and `StableOracleUSDT.sol` contracts, the constructor argument `_wethoracle` is not utilized. This implies that providing the address of the oracles as an argument will not impact the resulting price returned by the contract.

Redundant parts of the code create excessive Gas costs.

Assets:

- ussdv2/contracts/oracles/StableOracleUSDT.sol []
- ussdv2/contracts/oracles/StableOracleWBGL.sol []

Status: Fixed

Recommendations

Recommendation: Remove redundant code blocks.

Remediation (revised commit: c14801b): The proposed fix has been implemented.

Disclaimers

Hacken Disclaimer

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

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

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

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

Technical Disclaimer

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

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

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

Scope Details

Repository	https://github.com/DavidLeeChaum/USSDv2
Commit	bf57fe60453768e41fdfb27b683abbffe9a700d6
Whitepaper	-
Requirements	-
Technical Requirements	Readme.md

Contracts in Scope

ussdv2/contracts/USSDRewards.sol

ussdv2/contracts/USSD.sol

ussdv2/contracts/stUSSD.sol

ussdv2/contracts/ICT.sol

ussdv2/contracts/Migrations.sol

ussdv2/contracts/interfaces/IStableOracle.sol

ussdv2/contracts/interfaces/IStaticOracle.sol

ussdv2/contracts/interfaces/IUSSD.sol

ussdv2/contracts/interfaces/IUSSDInsurance.sol

ussdv2/contracts/oracles/SimOracle.sol

ussdv2/contracts/oracles/StableOracleUSDT.sol

ussdv2/contracts/oracles/StableOracleWBGL.sol

ussdv2/contracts/oracles/StableOracleWBTC.sol

ussdv2/contracts/oracles/StableOracleWETH.sol