SUMMARY OF CODE BASE

This contract creates a stablecoin - a cryptocurrency that's designed to maintain a stable value of $1 USD. Here's what makes it special:

THE BASICS

- It's called "DecentralizedStableCoin" with the symbol "DSC"
- It's backed by real cryptocurrencies (ETH and BTC) rather than just algorithms
- It's meant to always be worth exactly $1
- It's completely controlled by another contract called DSCEngine, with no human governance

HOW IT WORKS The contract has two main functions:

1. Creating (minting) new stablecoins
2. Destroying (burning) existing stablecoins

Regular users can't directly create or destroy these coins. Only the DSCEngine contract has this power. This is important because it ensures that new coins are only created when proper collateral is provided, and coins are only destroyed when appropriate.

HIGH RISK FINDINGS DETAILED ANALYSIS

H-01: Theft of collateral tokens with fewer than 18 decimals

Core Issue

The protocol incorrectly assumes all collateral tokens have 18 decimals in price calculations, leading to severe undervaluation of tokens with different decimal places (like WBTC with 8 decimals).

Vulnerable Code

```
// DSCEngine.sol
function getTokenAmountFromUsd(address token, uint256 usdAmountInWei)
public view returns (uint256) {
    AggregatorV3Interface priceFeed =
AggregatorV3Interface(s_priceFeeds[token]);
    (, int256 price,,,) = priceFeed.staleCheckLatestRoundData();
    // No decimal adjustment!
    return (usdAmountInWei * PRECISION) / (uint256(price) *
ADDITIONAL_FEED_PRECISION);
}

function getUsdValue(address token, uint256 amount) public view returns
(uint256) {
    AggregatorV3Interface priceFeed =
AggregatorV3Interface(s_priceFeeds[token]);
    (, int256 price,,,) = priceFeed.staleCheckLatestRoundData();
```

```
    // No decimal adjustment!
    return ((uint256(price) * ADDITIONAL_FEED_PRECISION) * amount) /
PRECISION;
}
```

Impact Example

For WBTC (8 decimals) worth $30,000:

- Actual value: $30,000
- Protocol calculation: $0.000003 (severely undervalued)
- Allows attackers to liquidate positions for fraction of true value

Recommended Fix

```
function getTokenAmountFromUsd(address token, uint256 usdAmountInWei)
public view returns (uint256) {
    AggregatorV3Interface priceFeed =
AggregatorV3Interface(s_priceFeeds[token]);
    (, int256 price,,,) = priceFeed.staleCheckLatestRoundData();
    uint8 decimals = priceFeed.decimals();
    uint256 priceWithDecimals = (uint256(price) * 1e18) / (10 ** decimals);
    return (usdAmountInWei * PRECISION) / priceWithDecimals;
}
```

H-02: Liquidation Prevention Due to Strict Bonus Implementation

Core Issue

The fixed 10% liquidation bonus makes it mathematically impossible to liquidate positions with 100-110% collateralization.

Mathematical Proof

For collateral (x) and debt (y):

```
(x - 1.1 × z) × 2 ≤ y - z
// For full liquidation (z = x/1.1):
y = x/1.1
```

Vulnerable Code

```
function liquidate(address collateral, address user, uint256 debtToCover)
    external
    moreThanZero(debtToCover)
    nonReentrant
```

```
{
    // ... health factor checks ...
    uint256 tokenAmountFromDebtCovered = getTokenAmountFromUsd(collateral,
debtToCover);
    // Fixed 10% bonus - This is the problem
    uint256 bonusCollateral = (tokenAmountFromDebtCovered *
LIQUIDATION_BONUS) / LIQUIDATION_PRECISION;
    uint256 totalCollateralToRedeem = tokenAmountFromDebtCovered +
bonusCollateral;
}
```

Recommended Fix

```
// Add flexibility to bonus calculation
if (tokenAmountFromDebtCovered < totalDepositedCollateral &&
totalCollateralToRedeem > totalDepositedCollateral) {
    totalCollateralToRedeem = totalDepositedCollateral;
}
```

H-03: No Incentive to Liquidate Small Positions

Core Issue

Gas costs make it unprofitable to liquidate small positions, leading to accumulation of bad debt.

Example Scenario

```
Position Value: $5
Liquidation Bonus (10%): $0.50
Gas Cost: > $1.00
Result: No rational liquidator would act
```

Impact

```
// These small positions can add up
totalBadDebt += smallPosition1 + smallPosition2 + smallPosition3...
// Eventually:
if (totalBadDebt > protocol_tolerance) {
    protocol_becomes_insolvent();
}
```

Recommended Fix

```
uint256 constant MIN_COLLATERAL_VALUE = 1000e18; // $1000 minimum

function depositCollateralAndMintDsc(...) {
    uint256 collateralValueInUsd = getUsdValue(token, amount);
    require(collateralValueInUsd >= MIN_COLLATERAL_VALUE, "Position too
small");
    // ... rest of function
}
```

H-04: Protocol Liquidation Arithmetic Issues

Core Issue

The combination of 200% collateralization requirement, 10% liquidation bonus, and DSC-only liquidations creates an impossible mathematical situation.

Problem Demonstration

```
// To liquidate, liquidator needs DSC
// To get DSC, must deposit collateral
// When liquidating:
collateral_received = debt_paid * (1 + bonus_rate)
// But collateral received isn't enough to mint required DSC
// Even with 10% bonus, math doesn't work due to 200% requirement
```

Recommended Solutions

1. Flash Mint Feature:

```
function flashMint(uint256 amount) external {
    require(amount <= maxFlashMint, "Exceeds flash mint limit");
    // mint DSC
    // require payback in same transaction
    // no fees
}
```

2. Allow alternative stablecoins for liquidation
3. Adjust collateralization and bonus rates to make math viable

//////

MEDIUM RISK FINDINGS DETAILED ANALYSIS

M-01: Missing Arbitrum Sequencer Checks

Core Issue

Protocol fails to verify Arbitrum sequencer status before using Chainlink price feeds, risking use of stale prices during sequencer downtime.

Vulnerable Code

```
// OracleLib.sol
function staleCheckLatestRoundData(AggregatorV3Interface priceFeed) {
    (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt,
uint80 answeredInRound) =
        priceFeed.latestRoundData();
    // Only checks time - no sequencer status check
    uint256 secondsSince = block.timestamp - updatedAt;
    if (secondsSince > TIMEOUT) revert OracleLib__StalePrice();
}
```

Recommended Fix

```
function isSequencerAlive() internal view returns (bool) {
    (, int256 answer, uint256 startedAt,,) = sequencer.latestRoundData();
    if (block.timestamp - startedAt <= GRACE_PERIOD_TIME || answer == 1)
        return false;
    return true;
}

function staleCheckLatestRoundData(AggregatorV3Interface priceFeed) {
    require(isSequencerAlive(), "Sequencer is down");
    // ... rest of function
}
```

M-02: Price Staleness Handling Issues

Core Issue

Fixed 3-hour timeout is incompatible with different chains' price feed update frequencies:

- Polygon: 25s updates → 3h too long
- Arbitrum: 24h updates → 3h too short

Vulnerable Code

```
uint256 private constant TIMEOUT = 3 hours; // Fixed timeout for all chains

function staleCheckLatestRoundData(AggregatorV3Interface priceFeed) {
    // ... price fetch ...
    uint256 secondsSince = block.timestamp - updatedAt;
```

```
    if (secondsSince > TIMEOUT) revert OracleLib__StalePrice();
}
```

Recommended Fix

```
// Allow different timeouts per token/chain
mapping(address => uint256) private s_timeouts;

function setTimeout(address token, uint256 timeout) external onlyOwner {
    s_timeouts[token] = timeout;
}
```

M-03: No Minimum Price Check on Chainlink Feeds

Core Issue

Protocol doesn't check for Chainlink's circuit breaker minimum price, risking use of incorrect prices during extreme market events.

Vulnerable Code

```
function getUsdValue(address token, uint256 amount) public view returns
(uint256) {
    AggregatorV3Interface priceFeed =
AggregatorV3Interface(s_priceFeeds[token]);
    (, int256 price,,,) = priceFeed.staleCheckLatestRoundData();
    // No min/max price checks!
    return ((uint256(price) * ADDITIONAL_FEED_PRECISION) * amount) /
PRECISION;
}
```

Recommended Fix

```
function getUsdValue(address token, uint256 amount) public view returns
(uint256) {
    (, int256 answer,,,) = priceFeed.latestRoundData();
    // Add price bounds checks
    if (answer <= minPrice) revert PriceBelowMin();
    if (answer >= maxPrice) revert PriceAboveMax();
    // ... rest of function
}
```

M-04: Price Feed Decimal Handling

Core Issue

Protocol assumes all price feeds use 8 decimals, but some (like AMPL/USD) use different decimal places.

Vulnerable Code

```
// Assumes 8 decimals for all feeds
return ((uint256(price) * ADDITIONAL_FEED_PRECISION) * amount) / PRECISION;
```

Recommended Fix

```
function getUsdValue(address token, uint256 amount) public view returns
(uint256) {
    AggregatorV3Interface priceFeed =
AggregatorV3Interface(s_priceFeeds[token]);
    (, int256 price,,,) = priceFeed.staleCheckLatestRoundData();
    uint8 decimals = priceFeed.decimals();
    uint256 priceWithDecimals = (uint256(price) * 1e18) / (10 ** decimals);
    return (priceWithDecimals * amount) / PRECISION;
}
```

M-05: Uncontrolled Token Burning via burnFrom()

Core Issue

Anyone can burn tokens using `burnFrom()` inherited from ERC20Burnable, bypassing owner restrictions.

Current Implementation

```
contract DecentralizedStableCoin is ERC20Burnable, Ownable {
    function burn(uint256 _amount) public override onlyOwner {
        // Protected
    }
    // But burnFrom() from ERC20Burnable is unprotected!
}
```

Recommended Fix

```
function burnFrom(address, uint256) public pure override {
    revert DecentralizedStableCoin__BlockFunction();
}
```

M-06 through M-12: Other Medium Risks

M-06: Double-spending vulnerability in DSC token

The DSCEngine contract allows collateral tokens to be registered multiple times in the constructor, leading to double-counting of collateral values. When a token like ETH is registered twice, a user depositing 10 ETH would be credited with 20 ETH worth of collateral value, enabling them to mint more DSC than their actual collateral warrants.

M-07: Lack of fallbacks for price feed oracle

The protocol has no fallback mechanism if Chainlink price feeds fail. This creates a single point of failure where the protocol could become completely non-functional if the primary oracle fails.

M-08: Fee-on-transfer token vulnerability

The protocol doesn't account for tokens that take fees on transfers. When such tokens are used as collateral, the protocol records the pre-fee amount rather than the actual received amount, leading to over-collateralization in the system.

M-09: Liquidator health factor restriction

The protocol unnecessarily prevents liquidators with their own health factor below 1 from performing liquidations, even though liquidations don't affect the liquidator's health factor. This reduces the pool of potential liquidators unnecessarily.

M-10: Protocol vulnerability with proxy tokens

The protocol can break when used with tokens that have proxy and implementation contracts (like TUSD). If the implementation changes, it could introduce features that break protocol assumptions about token behavior.

M-11: Liquidation front-running vulnerability

Liquidators are vulnerable to oracle price manipulations and MEV front-running attacks that could cause them to receive less collateral than expected for their liquidations.

M-12: DoS of full liquidations through front-running

Malicious actors can prevent full liquidations by front-running liquidation transactions with minimal partial liquidations, forcing liquidators to revert due to incorrect debt amounts.

These issues, while not as severe as the high-risk findings, still represent significant vulnerabilities that could impact protocol operations and user funds under specific circumstances.