USDV Security Review

Pashov Audit Group

Conducted by: shaflow, Bauchibred, Kurosaki, ZanyBonzy

March 6th 2025 - March 10th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **0xtakii/usd-fun** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About USDV

USDV is a stablecoin backed by a combination of assets and utilizes rebasing mechanisms to maintain stability. The protocol features wrapped tokens, price oracles, and a rewards distribution system, enabling users to mint and burn tokens through a managed whitelist interface.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 0d171ad3f776e2e7ab2f7ceb81866057a2e92e69

fixes review commit hash - 802ff9217d800675c48aa713918ebaad5d7728e0

Scope

The following smart contracts were in scope of the audit:

- `AddressesWhitelist`
- `ERC20RebasingPermitUpgradeable`
- `ERC20RebasingUpgradeable`
- `ExternalRequestsManager`
- `FlpPriceStorage`
- `RewardDistributor`
- `SimpleToken`
- `StUSF`
- `UsfPriceStorage`
- `WstUSF`
- `deploy`

7. Executive Summary

Over the course of the security review, shaflow, Bauchibred, Kurosaki, ZanyBonzy engaged with USDV to review USDV. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	USDV
Repository	https://github.com/0xtakii/usd-fun
Date	March 6th 2025 - March 10th 2025
Protocol Type	Stablecoin

Findings Count

Severity	**Amount**
Low	12
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[L-01]	Prohibit users from making small deposits and wraps to avoid losses	Low	Resolved
[L-02]	The owner cannot directly transfer tokens using transferFrom	Low	Resolved
[L-03]	setReserves and setPrice can be sandwiched	Low	Acknowledged
[L-04]	Token permit signatures cannot be cancelled before deadline	Low	Resolved
[L-05]	Unfair loss socialization or sanctions bypass	Low	Acknowledged
[L-06]	Removing allowedToken after minting leads to stuck funds and DOS	Low	Acknowledged
[L-07]	Cancellations can be blocked for some providers	Low	Resolved
[L-08]	Current treasury implementation causes security deficiencies	Low	Resolved
[L-09]	Lack of timelock enables reward sniping	Low	Acknowledged
[L-10]	Shared whitelist usage prevents token-specific address whitelisting	Low	Acknowledged
[L-11]	Incorrect parameter order in mintWithPermit	Low	Acknowledged
[L-12]	Missing signature verification limits permit functionality for smart accounts	Low	Acknowledged

8. Findings

8.1. Low Findings

[L-01] Prohibit users from making small deposits and wraps to avoid losses

In the `deposit` and `wrap` function, there should be a check to ensure that `wstUSFAmount` is not zero to avoid rounding issues that could result in users depositing some `stUSF` but minting zero shares, causing losses.

```
function deposit(uint256 _usfAmount, address _receiver) public returns
      (uint256 wstUSFAmount) {
        wstUSFAmount = previewDeposit(_usfAmount);
+       _assertNonZero(wstUSFAmount);
        _deposit(msg.sender, _receiver, _usfAmount, wstUSFAmount);
        return wstUSFAmount;
    }

    function wrap(uint256 _stUSFAmount, address _receiver) public returns
      (uint256 wstUSFAmount) {
        _assertNonZero(_stUSFAmount);

        wstUSFAmount = convertToShares(_stUSFAmount);
        IERC20(stUSFAddress).safeTransferFrom(msg.sender, address
          (this), _stUSFAmount);
+       _assertNonZero(wstUSFAmount);
        _mint(_receiver, wstUSFAmount);
        emit Wrap(msg.sender, _receiver, _stUSFAmount, wstUSFAmount);

        return wstUSFAmount;
    }
```

[L-02] The owner cannot directly transfer tokens using `transferFrom`

In the `ERC20RebasingUpgradeable` contract, when the owner transfers tokens using `transferFrom` or `transferSharesFrom`, they still need to spend an allowance. This means the owner must first call the `approve` function to authorize themselves.

It is recommended that the owner should be able to transfer their own tokens without spending an allowance.

```
function transferFrom(
      address_from,
      address_to,
      uint256_underlyingTokenAmount
    ) public returns (bool isSuccess
        address spender = _msgSender();
        uint256 shares = convertToShares(_underlyingTokenAmount);

        // instead of using `_underlyingTokenAmount` as is.
        // This prevents discrepancies between the reported 'value' and the
        // actual token amount transferred
        // due to floor rounding during the conversion to shares.
        uint256 underlyingTokenAmount = convertToUnderlyingToken(shares);
+       if (_from != msg.sender) {
+           _spendAllowance(_from, spender, underlyingTokenAmount);
+       }
-       _spendAllowance(_from, spender, underlyingTokenAmount);
        convertToUnderlyingToken(shares)
        _transfer(_from, _to, convertToUnderlyingToken(shares), shares);
        return true;
    }

    /**
     * @dev Moves a `_shares` amount from `_from` to `_to` using the

           * allowance mechanism. `_underlyingTokenAmount` is then deducted from the c
     * allowance.
     */
    function transferSharesFrom(
      address_from,
      address_to,
      uint256_shares
    ) public returns (bool isSuccess
        address spender = _msgSender();
        uint256 underlyingTokenAmount = convertToUnderlyingToken(_shares);
+       if (_from != msg.sender) {
+           _spendAllowance(_from, spender, underlyingTokenAmount);
+       }
-       _spendAllowance(_from, spender, underlyingTokenAmount);
        _transfer(_from, _to, underlyingTokenAmount, _shares);
        return true;
    }
```

[L-03] `setReserves` and `setPrice` can be sandwiched

Calls to `setReserves` and `setPrice` creates opportunity for MEV, since users monitoring mempool can conduct transactions around calls to the functions either by frontrunning or backrunning to gain quick profit. The issue is further exarcebated in `setReserves` since there is no strictly defined upper bound through which price can move e.g price can move from somewhere as low as 1e6 to 1e18 which is the max as `PRICE_SCALING_FACTOR`.

From UsfPriceStorage.sol:

```
function setReserves(
      bytes32_key,
      uint256usfSupply,
      uint256reserves
    ) external onlyRole(SERVICE_ROLE
        if (_key == bytes32(0)) revert InvalidKey();
        if (usfSupply == 0) revert InvalidUsfSupply();
        if (reserves == 0) revert InvalidReserves();
        if (prices[_key].timestamp != 0) revert PriceAlreadySet(_key);

        uint256 computedPrice = _calculatePrice(usfSupply, reserves);
        uint256 lastPriceValue = lastPrice.price;
        if (lastPriceValue != 0) {
            // assumes only possible at initialization
            uint256 lowerBound = lastPriceValue -
              (lastPriceValue * lowerBoundPercentage / BOUND_PERCENTAGE_DENOMINATOR);
            if (computedPrice < lowerBound) {
                revert InvalidPrice(computedPrice, lowerBound);
            }
        }

        uint256 currentTime = block.timestamp;
@>      Price memory priceData =
            Price(
              {price:computedPrice,
              usfSupply:usfSupply,
              reserves:reserves,
              timestamp:currentTime}
            );

        prices[_key] = priceData;
        lastPrice = priceData;

        emit PriceSet(_key, computedPrice, usfSupply, reserves, currentTime);
    }
```

From FlpPriceStorage.sol:

```
function setPrice(bytes32 _key, uint256 _price) external onlyRole
      (SERVICE_ROLE) {
        if (_key == bytes32(0)) revert InvalidKey();
        if (_price == 0) revert InvalidPrice();
        if (prices[_key].timestamp != 0) revert PriceAlreadySet(_key);

        uint256 lastPriceValue = lastPrice.price;
        if (lastPriceValue != 0) {
            uint256 upperBound = lastPriceValue +
              (lastPriceValue * upperBoundPercentage / BOUND_PERCENTAGE_DENOMINATOR);
            uint256 lowerBound = lastPriceValue -
              (lastPriceValue * lowerBoundPercentage / BOUND_PERCENTAGE_DENOMINATOR);
            if (_price > upperBound || _price < lowerBound) {
                revert InvalidPriceRange(_price, lowerBound, upperBound);
            }
        }

        uint256 currentTime = block.timestamp;
@>      Price memory price = Price({price: _price, timestamp: currentTime});
        prices[_key] = price;
        lastPrice = price;

        emit PriceSet(_key, _price, currentTime);
    }
```

Recommendation is to use private RPC's eliminating front-running and also introduce upper bound percentages in `setReserves` like is done in `setPrice`. The bound can then be capped to `PRICE_SCALING_FACTOR` if it exceeds it.

[L-04] Token permit signatures cannot be cancelled before deadline

WstUSF and SImpleToken contracts implements `ERC20Permit` functionality for gasless approvals. stUSF does this also via ERC20RebasingPermitUpgradeable.sol. The implementation however lack a mechanism for users to revoke their permit signatures before they expire. Once a user signs a permit, they must wait until the deadline passes to invalidate it, even if they want to cancel the approval for other security reasons.

From stUSF.sol

```
function initialize(
      stringmemory_name,
      stringmemory_symbol,
      address_usfAddress
    ) public initializer {
        _assertNonZero(_usfAddress);

        __ERC20Rebasing_init(_name, _symbol, _usfAddress);
@>      __ERC20RebasingPermit_init(_name);
    }
```

From ERC20RebasingPermitUpgradeable.sol

```
bytes32 structHash =
@>          keccak256(abi.encode
  (PERMIT_TYPEHASH, _owner, _spender, _value, _useNonce(_owner), _deadline));

        bytes32 hash = _hashTypedDataV4(structHash);
```

From SImpleToken.sol

```
function initialize
      (string memory _name, string memory _symbol) public initializer {
        __ERC20_init(_name, _symbol);
@>      __ERC20Permit_init(_name);

        __AccessControlDefaultAdminRules_init(1 days, msg.sender);
    }
```

From WstUSF.sol

```
function initialize(
      stringmemory_name,
      stringmemory_symbol,
      address_stUSFAddress
    ) public initializer {
        __ERC20_init(_name, _symbol);
@>      __ERC20Permit_init(_name);

        _assertNonZero(_stUSFAddress);
        stUSFAddress = _stUSFAddress;

        usfAddress = address(IERC20Rebasing(_stUSFAddress).underlyingToken());
        _assertNonZero(usfAddress);
        IERC20(usfAddress).safeIncreaseAllowance(stUSFAddress, type
          (uint256).max);
    }
```

This is because the contract is based on OpenZeppelin's `ERC20PermitUpgradeable.sol` in which the function to increase nonce does not exist and the _useNonce function within `Nonces` is marked internal. It means the current nonce system only increments when a permit is executed, not allowing users to proactively invalidate pending signatures by advancing their nonce.

Consider introducing a public function that signers can directly use to consume their nonce, thereby canceling the signatures.

```
function useNonce() external returns (uint256) {
        return _useNonce(msg.sender);
    }
```

[L-05] Unfair loss socialization or sanctions bypass

Take a look at `ExternalRequestsManager.sol#requestBurn()`

```
function requestBurn(
  uint256_issueTokenAmount,
  address_withdrawalTokenAddress,
  uint256_minWithdrawalAmount
)
    public
    onlyAllowedProviders
|>   allowedToken(_withdrawalTokenAddress)
    whenNotPaused
{
    // Implementation
}
```

Now would be key to note that the protocol allows users to mint tokens by depositing various supported assets (currently USDC and USDT, *with more potentially added in the future*). However, when burning tokens to reclaim their deposits, users can specify any supported token as their withdrawal token, regardless of what they originally deposited.

This is problematic because it fails to maintain a proper relationship between what users deposit and what they withdraw. For example:

1. User A deposits 1000 USDC to mint protocol tokens.
2. User B deposits 1000 USDT to mint protocol tokens.
3. Both deposits are transferred to the treasury.
4. If USDT experiences a black swan event (depegging, hack, etc.).
5. User A can choose to withdraw USDT instead of their original USDC deposit.

So this design creates a "first to exit" scenario during market stress events, allowing early users to externalize losses to later users. When a supported asset faces a black swan event:

1. Users who deposited the compromised asset can withdraw from other, healthier asset pools.
2. This depletes the treasury of healthy assets.
3. Later users are forced to accept the compromised asset or be unable to withdraw at all.

The problem worsens as more deposit assets are added to the protocol. Instead of each user bearing the risk of their chosen deposit asset, the risk becomes socialized across all users of the protocol (asides the malicious one who intentionally delegates the loss to the other users).

As hinted by the title also, this bug case allows for OFAC sanctioned accounts to bypass their sanctions and still have access to funds, cause, assuming we have a user who initially deposited and minted with USDT and is now blacklisted on USDT, they can withdraw assets from the treasury, and all they need to do is specify `USDC` as their withdrawal token, sidestepping the sanction.

Recommendation

Since we already track the original deposit token for each mint request, we should enforce that users can only withdraw the same type of token they initially deposited.

[L-06] Removing allowedToken after minting leads to stuck funds and DOS

The `ExternalRequestsManager` contract allows adding and removing tokens from the allowedTokens list:

```
function removeAllowedToken(address _allowedTokenAddress) external onlyRole
  (DEFAULT_ADMIN_ROLE) {
    _assertNonZero(_allowedTokenAddress);
    allowedTokens[_allowedTokenAddress] = false;
    emit AllowedTokenRemoved(_allowedTokenAddress);
}
```

When users request to mint tokens, they provide a deposit token which is transferred to the treasury:

```
function requestMint
  (address _depositTokenAddress, uint256 _amount, uint256 _minMintAmount) {
    // ...
    IERC20(_depositTokenAddress).safeTransferFrom(msg.sender, address
      (this), _amount);
    // ...
}

function completeMint
  (bytes32 _idempotencyKey, uint256 _id, uint256 _mintAmount) {
    // ...
    depositToken.safeTransfer(treasuryAddress, request.amount);
    // ...
}
```

Later, when users burn their tokens, they must specify a withdrawal token that is currently allowed:

```
function requestBurn(
  uint256_issueTokenAmount,
  address_withdrawalTokenAddress,
  uint256_minWithdrawalAmount
)
    public
    onlyAllowedProviders
|>    allowedToken(_withdrawalTokenAddress)
    whenNotPaused
{
    // ...
}
```

However this flow does not consider the path below, when an admin removes a token from the `allowedTokens` list after users have already deposited that token and minted. For instance:

1. USDC and USDT are both allowedTokens
2. Users deposit $100,000 worth of USDC and USDT, which are transferred to the treasury
3. Admin removes USDC from allowedTokens
4. Users who deposited USDC cannot specify USDC as their withdrawal token due to the `allowedToken(_withdrawalTokenAddress)` modifier
5. Users are forced to either specify USDT (depleting USDT reserves) or be unable to withdraw their funds.

Which means that even users who deposited still-allowed tokens may find those tokens depleted by other users who cannot specify their original deposit token, creating a bank run scenario where late users cannot withdraw any funds

The impact is amplified in a multi-token system, as unallowing any token creates systemic liquidity issues affecting all users. In practical terms, if the protocol supports USDC and USDT, and USDC gets unallowed, approximately 50% of users would be unable to recover their original deposits using the hypothetical case shared in the report.

Consider removing the `allowedToken` modifier from the `requestBurn` function, so that users can withdraw using the same token type they originally deposited regardless of current allowedTokens status.

[L-07] Cancellations can be blocked for some providers

```
function cancelMint(uint256 _id) external mintRequestExist(_id) {
// ..snip
        request.state = State.CANCELLED;
        IERC20 depositedToken = IERC20(request.token);
|>          depositedToken.safeTransfer(request.provider, request.amount);
        emit MintRequestCancelled(_id);
    }
        function cancelBurn(uint256 _id) external burnRequestExist(_id) {
// ..snip
        request.state = State.CANCELLED;
        IERC20 issueToken = IERC20(ISSUE_TOKEN_ADDRESS);
|>          issueToken.safeTransfer(request.provider, request.amount);

        emit BurnRequestCancelled(_id);
    }
```

As seen, the cancellation functions `ExternalRequestsManager::cancelMint()` and `cancelBurn()` directly transfer funds to `msg.sender` using `safeTransfer`:

Now, currently, usd-fun only support USDC/USDT (which are both blacklistable stablecoins), knowing that transfers to blacklisted addresses will revert. This creates an irrecoverable state where:

1. Blacklisted provider, who had initially placed a mint/burn request, attempts to call `cancelMint()/cancelBurn()`
2. `safeTransfer` fails due to blacklist check in token contract
3. Transaction reverts, keeping request in `CREATED` state
4. Protocol cannot process cancellation the cancellation for users

Allow the providers to pass in a recipient, this can be done by attaching a new parameter to the cancellation functions:

```
function cancelMint(uint256 _id, address _recipient) external {
    // ...
    depositedToken.safeTransfer(_recipient, request.amount);
}

function cancelBurn(uint256 _id, address _recipient) external {
    // ...
    issueToken.safeTransfer(_recipient, request.amount);
}
```

[L-08] Current treasury implementation causes security deficiencies

The treasury is implemented as a simple address, could be a contract or not, *(but for this audit we assume it as a non-contract address)*

```
// ExternalRequestsManager.sol
address public treasuryAddress;

constructor(
    address _issueTokenAddress,
    address _treasuryAddress,
    address _providersWhitelistAddress,
    address[] memory _allowedTokens
) {
    treasuryAddress = _assertNonZero(_treasuryAddress);
    // ...
}
```

From the scripts this has only been relayed to be the "fund management address"

From `deploy.s.sol#L79`

```
address public treasury; // fund management address
```

Would be key to note that tokens are directly transferred to and from this address:

```
// When completing mint
depositToken.safeTransfer(treasuryAddress, request.amount);

// When completing burn
IERC20(request.token).safeTransferFrom
  (treasuryAddress, request.provider, _withdrawalAmount);
```

In contrast, the resolv-contracts which inspires usd-fun implements a robust `Treasury` contract with:

1. Emergency pause functionality through PausableUpgradeable
2. Granular recipient/spender whitelisting
3. Operation limits for different transaction types
4. Idempotency tracking to prevent duplicate operations
5. Protocol-specific integrations with safety mechanisms

Since this is not the case in the current implementation, this then means that all the below cases are open:

- No ability to pause token movements during compromises, asides pausing the external contracts, however all the assets in the treasury could be exposed.
- No caps on transaction sizes to prevent significant asset drainage
- The owner of the specified treasury address can transfer all funds
- Most importantly, tech savvy users would scrutinize the protocol heavily as there is no verifiable contract to which their funds are transferred to during mints which they are sure tokens can only be removed from in validated instances, confirming security for when they decide to burn.

Implement a dedicated Treasury contract similar to the resolv-contracts approach.

[L-09] Lack of timelock enables reward sniping

First see:

```
function deposit(uint256 _usfAmount, address _receiver) public returns
  (uint256 wstUSFAmount) {
    wstUSFAmount = previewDeposit(_usfAmount);
    _deposit(msg.sender, _receiver, _usfAmount, wstUSFAmount);
    return wstUSFAmount;
}
```

```
function withdraw(
  uint256_usfAmount,
  address_receiver,
  address_owner
) public returns (uint256 wstUSFAmount
    uint256 maxusfAmount = maxWithdraw(_owner);
    if (_usfAmount > maxusfAmount) revert ExceededMaxWithdraw
      (_owner, _usfAmount, maxusfAmount);
    wstUSFAmount = previewWithdraw(_usfAmount);
    _withdraw(msg.sender, _receiver, _owner, _usfAmount, wstUSFAmount);
    return wstUSFAmount;
}
```

As seen, yhe WstUSF contract allows users to deposit and withdraw within the same block without any timelock mechanism. This atomic interaction capability creates an attack vector where sophisticated users can frontrun reward distributions fromt the `RewardDistributor`.

Thats because when rewards are about to be distributed to the underlying StUSF contract (which WstUSF wraps), a malicious actor can:

1. Monitor the mempool for incoming reward distribution transactions
2. Frontrun these transactions with a large deposit to WstUSF
3. Receive a proportionally large share of the distributed rewards
4. Immediately withdraw their position in the same block
5. Extract value without maintaining a long-term position

The attack is particularly effective because of the ERC4626-style vault architecture that calculates share distribution based on point-in-time snapshots, though impact here is medium since not too much value can be leaked via this path, but this still reduces rewards for genuine long-term stakers and creates an unfair distribution mechanism that favors MEV-capable actors.

Recommendation

Consider implementing a timelock mechanism that enforces a minimum holding period between deposit and withdrawal operations.

[L-10] Shared whitelist usage prevents token-specific address whitelisting

First, take a look at `deploy.s.sol`

```
// deploying the whitelist contract
whitelist = IAddressesWhitelistExtended(address(new AddressesWhitelist()));

address[] memory whitelistedTokens = new address[](2);
whitelistedTokens[0] = usdcAddress;
whitelistedTokens[1] = usdtAddress;

// deploying the ExternalRequestsManager contract for FunLP tokens
externalRequestsManager = IExternalRequestsManagerExtended(
    address(new ExternalRequestsManager(address(funLpToken), treasury, address
      (whitelist), whitelistedTokens))
);

funLpToken.grantRole(SERVICE_ROLE, address(externalRequestsManager));

externalRequestsManager.grantRole(SERVICE_ROLE, service);

usfExternalRequestsManager = IExternalRequestsManagerExtended(
    address(new ExternalRequestsManager(address(funToken), treasury, address
      (whitelist), whitelistedTokens))
);

funToken.grantRole(SERVICE_ROLE, address
//(usfExternalRequestsManager)); // requires for mint and burn functions


// ..snip

whitelist.transferOwnership(admin);
```

As seen in the script, we use the same `whitelist` contract instance for both `ExternalRequestsManager` instances which are then relayed for the `funLpToken` and `funToken` tokens. Which then creates a shared whitelist situation where any provider whitelisted for one token is automatically whitelisted for the other.

Now the use of a shared whitelist contract for both ExternalRequestsManager instances eliminates the possibility of implementing token-specific provider access controls.

Note that this can be translated as not the appropriate path, since per the script there is an intention to have two seperate `ExternalRequestsManager`s, which should translate to two different instances for the whitelisting of `funLpToken` and `funToken` tokens.

However the current design decision means we now have the inability to implement different risk profiles for both tokens - providers that might be trusted to interact with one token cannot be restricted from interacting with the other.

One could also hint regulatory considerations here, since different tokens might have different regulatory requirements for provider whitelisting that cannot be accommodated, assume OFAC where one is to be compliant and block those in the OFAC list whereas the other isn't, however this is impossible.

i.e in our case, looking at `requestMint()`, there could be an argument that, some providers might have regulatory approval to interact with liquidity tokens per their complexity but not with stablecoins.

Recommendation

Since we are having two seperate `ExternalRequestsManager`s, we should also have two seperate `AddressesWhitelist`s to enable token-specific access control:

```
// Deploy separate whitelist contracts
IAddressesWhitelistExtended funLpWhitelist = IAddressesWhitelistExtended
  (address(new AddressesWhitelist()));
IAddressesWhitelistExtended usfWhitelist = IAddressesWhitelistExtended(address
  (new AddressesWhitelist()));

// Use token-specific whitelists for each manager
externalRequestsManager = IExternalRequestsManagerExtended(
    address(new ExternalRequestsManager(address(funLpToken), treasury, address
      (funLpWhitelist), whitelistedTokens))
);

usfExternalRequestsManager = IExternalRequestsManagerExtended(
    address(new ExternalRequestsManager(address(funToken), treasury, address
      (usfWhitelist), whitelistedTokens))
);
```

[L-11] Incorrect parameter order in `mintWithPermit`

The current implementation of `WstUSF::mintWithPermit` uses the calculated `usfAmount` in the permit verification:

```
function mintWithPermit(
135:    uint256 _wstUSFAmount,
        address _receiver,
        uint256 _deadline,
        uint8 _v,
        bytes32 _r,
        bytes32 _s
) external returns (uint256 usfAmount) {
    IERC20Permit usfPermit = IERC20Permit(usfAddress);
    usfAmount = previewMint(_wstUSFAmount);
146:      try usfPermit.permit(msg.sender, address
  (this), usfAmount, _deadline, _v, _r, _s) {} catch {}
    _deposit(msg.sender, _receiver, usfAmount, _wstUSFAmount);
    return usfAmount;
}
```

Since the transfer we make is of the usfAmount itself post the verification (`WstUSF.sol#L250`, `WstUSF.sol#L146`), it's better we change the function signature so the user specifies the exact amount of usf they want to spend instead of the wrapped token amount (`WstUSF.sol#L135`).

```
function _deposit(
      address_caller,
      address_receiver,
      uint256_usfAmount,
      uint256_wstUSFAmount
    ) internal {
        IStUSF stUSF = IStUSF(stUSFAddress);
        IERC20 usf = IERC20(usfAddress);

250:        usf.safeTransferFrom(_caller, address(this), _usfAmount);
        stUSF.deposit(_usfAmount);
        _mint(_receiver, _wstUSFAmount);

        emit Deposit(_caller, _receiver, _usfAmount, _wstUSFAmount);
    }
```

But, the current approach relies on users calling `previewMint` first off-chain to see how much `usf` they need to approve and then permit this via a signature, but if for any reason the rates change before the `mintWithPermit` is called on-chain then the tx could revert during the `safeTransferFrom` if a bit more `usf` is needed.

This approach also generally goes against the idea of permitting. Naturally, the EIP-2612 permit system is used so users don't even come on-chain for the approval which would mean we are trying to reduce their on-chain presence as much as possible so having them preview first and then sign something that could fail if exchange rates change between signature creation and on-chain execution is just counterintuitive to this approach.

Recommendation

We should instead specify the `usfAmount` and then this is converted post permit verification, and the wrapped tokens should be minted to the receiver:

```
function mintWithPermit(
    uint256 _usfAmount,   // Changed from _wstUSFAmount
    address _receiver,
    uint256 _deadline,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) external returns (uint256 wstUSFAmount) {
    IERC20Permit usfPermit = IERC20Permit(usfAddress);

    // Use the exact amount the user specified for the permit
    try usfPermit.permit(msg.sender, address
      (this), _usfAmount, _deadline, _v, _r, _s) {} catch {}

    // Calculate how many wrapped tokens this will yield
    wstUSFAmount = convertToShares(_usfAmount);

    _deposit(msg.sender, _receiver, _usfAmount, wstUSFAmount);
    return wstUSFAmount;
}
```

[L-12] Missing signature verification limits permit functionality for smart accounts

Take a look at `ERC20RebasingPermitUpgradeable.sol#permit()`

```
function permit(address _owner, address _spender,
    uint256 _value, uint256 _deadline,
    uint8 _v, bytes32 _r, bytes32 _s
) public virtual {
// ..snip
    address signer = ECDSA.recover(hash, _v, _r, _s);
    if (signer != _owner) {
        revert ERC2612InvalidSigner(signer, _owner);
    }

    _approve(_owner, _spender, _value);
}
```

As seen, the permit implementation relies solely on ECDSA.recover for signature verification, which prevents smart contract wallets and accounts from using the permit functionality. Many modern wallet implementations (like Gnosis Safe, Argent, etc.) cannot produce ECDSA signatures in the format expected by this implementation, i.e `v, r, s`, effectively excluding a significant portion of users from gas-efficient approvals, and these users would be expected to be integrated being the fact that protocol is a stablecoin.

Implement support for EIP-1271 signature verification to allow smart contract wallets to use the permit functionality.

Additionally, you can consider implementing EIP-6492 support for counterfactual contract wallets that haven't been deployed yet.