CANTINA

Predicate Contracts

Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Ladboy233, Security Researcher

May 25, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Predicate is a network for simplifying transaction prerequisites.

From Apr 30th to May 3rd the Cantina team conducted a review of predicate-contracts on commit hash 828e42c8. The team identified a total of **21** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	3	2	1
Medium Risk	5	2	3
Low Risk	3	1	2
Gas Optimizations	1	0	1
Informational	9	3	6
Total	**21**	**8**	**13**

3 Findings

3.1 High Risk

3.1.1 Missing domain separator for signatures

Severity: High Risk

Context: ServiceManager.sol#L311

Description: The current implementation of the signature digest omits a domain separator (as per EIP-712). It also does not incorporate the Service Manager contract's own address, which leads to two key problems:

- Cross-chain replays: Because there is no chain-specific domain separator, signatures can be replayed across different blockchains.
- Cross-contract replays: A signature intended for `ServiceManager` can be replayed on `SimpleService-Manager`, or vice versa, as the digest lacks the contract address in its hashing process.

This design risks allowing a valid signature for one context to be accepted in another context where it should not be valid.

Recommendation:: Transition to an EIP-712-compliant domain separator that includes both chain-specific information and the verifying contract's address. One approach is to adopt `hashTypedDataV4` from the OpenZeppelin EIP712 library. For example:

```
bytes32 domainSeparator = keccak256(
    abi.encode(
        EIP712_DOMAIN_TYPEHASH,
        keccak256(bytes("ServiceOrManagerName")),
        keccak256(bytes("1")),  // version
        block.chainid,
        address(this)
    )
);
```

Then, prepend this `domainSeparator` to the data being signed. This ensures that signatures are valid only for the specific contract on a specific chain, preventing replay attacks in other contracts or blockchains.

Predicate: Acknowledged.

Cantina Managed: Acknowledged.

3.1.2 Operator's signing keys can be overwritten

Severity: High Risk

Context: ServiceManager.sol#L138

Description: In both `rotatePredicateSigningKey` and `registerOperatorToAVS` function, the code does:

```
signingKeyToOperator[_newSigningKey] = msg.sender;
```

However, there is no check to ensure `_newSigningKey` is not already mapped to a different operator. A malicious operator can supply an already-used signing key and overwrite a legitimate operator's key, hijacking their ECDSA signature identity. This can also happen inadvertently if two operators choose the same signing key by mistake.

Recommendation: Before assigning the new key, add a requirement that the key is not currently used in the `rotatePredicateSigningKey` and `registerOperatorToAVS` functions:

```
require(
    signingKeyToOperator[_newSigningKey] == address(0),
    "Can't overwrite other operator's signing key"
);
```

Predicate: Fixed in commit 1bbeb94.

Cantina Managed: Fix verified.

3.1.3 Incomplete mapping updates in syncOperators

Severity: High Risk

Context: SimpleServiceManager.sol#L95

Description: In the SimpleServiceManager.syncOperators function, when updating an existing operator's signing key, the function sets:

```
operatorAddressToSigningKey[registrationKey] = signingKey;
```

but it does not perform the corresponding update in:

```
signingKeyToOperatorAddress[signingKey] = registrationKey;
```

for the existing operator and therefore, he will not be able to provide any valid signature as this require check would always revert:

```
require(
    signingKeyToOperatorAddress[recoveredSigner] != address(0),
    "Predicate.validateSignatures: Signer is not a registered operator"
);
```

Additionally, the old signing key remains mapped in signingKeyToOperatorAddress, meaning it is never cleared.

Recommendation: When you detect an operator is already registered and you wish to change its signing key, ensure both mappings are correctly updated:

```
// clear the old signingKeyToOperatorAddress if changing from old key
delete signingKeyToOperatorAddress[oldSigningKey];

// map the new signing key to the operator address
signingKeyToOperatorAddress[newSigningKey] = registrationKey;
```

That way, both the operatorAddressToSigningKey and signingKeyToOperatorAddress mappings stay in sync, preventing stale references to the operator's old key.

Predicate: Fixed in PR 23.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 Global spentTaskIds mapping allows malicious "Task ID" griefing

Severity: Medium Risk

Context: ServiceManager.sol#L303

Description: In the ServiceManager contract, the validateSignatures function checks:

```
require(!spentTaskIDs[_task.taskId], "Predicate.validateSignatures: task ID already spent");
```

This places the same spentTaskId key in a global scope for all the clients using this ServiceManager.

Due to this implementation, a malicious operator, can front-run an _authorizeTransaction call from any client and submit the same taskId setting spentTaskIDs[thatId] to true. The legitimate _authorizeTransaction call is then blocked reverting with a "task ID already spent" error. This effectively griefs the user's intended operation and can be easily abused by using a policy with just 1 signature as quorum.

Recommendation: Scope the spentTaskIds to each client or contract address so only the same user can reuse the same taskId:

```
mapping(address => mapping(string => bool)) spentTaskIDs;
spentTaskIDs[clientAddress][taskId] = true;
```

By including the client address or some unique identifier in the key, a malicious party cannot burn another client's signature.

Predicate: Acknowledged. While we do agree this is a valid concern, the current risk is mitigated by our permissioned operator network, which consists of reputable, vetted participants. Given the scope of changes required—both onchain and offchain—to fully isolate task IDs per client, we believe this is best addressed in a subsequent iteration.

Cantina Managed: Acknowledged.

3.2.2 Stale operator registrations allow under-staked operators to remain in `ServiceManager`

Severity: Medium Risk

Context: ServiceManager.sol#L173-L187

Description: A staker can sequentially delegate to multiple operators (in the `DelegationManager`) and register each as an operator with the `ServiceManager` (via `registerOperatorToAVS`). Over time, they undelegate from the first operator after waiting the minimum withdrawal delay (e.g., ~2 weeks) and delegate to a second operator, registering that second operator as well. Because the `ServiceManager` does not automatically check if previously registered operators still hold enough stake, multiple operators can remain registered even though only one is actively staked. This leads to multiple "valid" operators in `ServiceManager` with insufficient or zero stake in `DelegationManager`.

Recommendation: Implement a housekeeping or periodic re-check that verifies each registered operator still meets the staking criteria. For example, a scheduled job or on-chain function calling `updateOperatorsForQuorum` or a custom function that checks each operator's stake and deregisters those who have fallen below the threshold.

This keeps the operator list accurate and prevents operators who are no longer staked from remaining in the system.

Predicate: Acknowledged. This is valuable feedback yet given the current permissioned state of Predicate I feel we can hold off on this change until we decide to enable some form of economic security using restaking / Eigenlayer.

Cantina Managed: Acknowledged.

3.2.3 `SimpleServiceManager.sol` block.number for expiration checks is unreliable for accurately determining the L2 block timing

Severity: Medium Risk

Context: SimpleServiceManager.sol#L235-L245

Description: The protocol uses hub and spoke structure and the `SimpleServiceManager.sol` is deployed in l2 blockchain such plume network.

The code use `block.number` to check signature expiration.

However, according to the Plume documentation:

> Plume assigns its own block numbers, distinct from Ethereum's, with multiple Plume blocks potentially fitting within a single Ethereum block. However, each Plume block is always associated with exactly one Ethereum block. In Plume smart contracts, querying block.number returns a value close to the L1 Ethereum block number when the sequencer received the transaction, though it may not be exact. Timing assumptions based on block numbers are reliable over several hours but not within minutes, similar to Ethereum.

This means that in the Plume network, block.number refers to the L1 Ethereum block number, not the L2 Plume block number. As a result, using block.number for expiration checks is unreliable for accurately determining the L2 block timing.

Recommendation: Use `block.timestamp` to validate signature expiration instead of block.number.

Predicate: Fixed in PR 20.

Cantina Managed: Fix verified.

3.2.4 An operator with insufficient staking share can generate the signature to bypass the signature validation

Severity: Medium Risk

Context: ServiceManager.sol#L319

Description: When the `ServiceManager` contract performs the signature verification, the code loops over the threshold number to check if the operator status is registered. To register an operator, the operator has to have sufficient stake share.

```
uint256 totalStake;
for (uint256 i; i != strategies.length;) {
    totalStake += IDelegationManager(delegationManager).operatorShares(msg.sender, IStrategy(strategies[i]));
    unchecked {
        ++i;
    }
}

if (totalStake >= thresholdStake) {
    operators[msg.sender] = OperatorInfo(totalStake, OperatorStatus.REGISTERED);
    signingKeyToOperator[_operatorSigningKey] = msg.sender;
    ISignatureUtils.SignatureWithSaltAndExpiry memory _operatorSig = ISignatureUtils.SignatureWithSaltAndExpiry(
        _operatorSignature.signature, _operatorSignature.salt, _operatorSignature.expiry
    );
    IAVSDirectory(avsDirectory).registerOperatorToAVS(msg.sender, _operatorSig);
    emit OperatorRegistered(msg.sender);
}
```

However, when validating the signature, the code does not check if the operator has sufficient staking shares.

- The operator can undelegate from the strategy so operator's staking share is below the `ServiceManager` threshold.
- The admin can remove a strategy so operator's staking share could be below the `ServiceManager` threshold.

```
for (uint256 i = 0; i < numSignaturesRequired;) {
    if (i > 0 && uint160(signerAddresses[i]) <= uint160(signerAddresses[i - 1])) {
        revert("Predicate.validateSignatures: Signer addresses must be unique and sorted");
    }
    address recoveredSigner = ECDSA.recover(messageHash, signatures[i]);
    require(recoveredSigner == signerAddresses[i], "Predicate.validateSignatures: Invalid signature");
    address operator = signingKeyToOperator[recoveredSigner];
    require(operators[operator].status == OperatorStatus.REGISTERED, "Signer is not a registered operator");
    unchecked {
        ++i;
    }
}
```

The code does not enfroce that the operator has sufficient staking shares when performing signature validation, and therefore, an operator with insufficient staking share could still generate valid signatures.

Recommendation: Ensure that the operator has enough staking shares when `validateSignatures` is called. Consider adding the following function:

```
function getOperatorTotalStake(address _operator) external view returns (uint256) {
    uint256 totalStake;
    for (uint256 i; i != strategies.length;) {
        totalStake += IDelegationManager(delegationManager).operatorShares(_operator, IStrategy(strategies[i]));
        unchecked {
            ++i;
        }
    }
    return  totalStake;
}
```

and implementing the following change:

```
  require(recoveredSigner == signerAddresses[i], "Predicate.validateSignatures: Invalid signature");
+ require(getOperatorTotalStake(operator) >= thresholdStake, "Operator has insufficient staking");
```

Predicate: Fixed in PR 19.

Cantina Managed: Fix verified.

3.2.5 Inaccurate aggregation of strategy shares without normalization across ETH derivatives

Severity: Medium Risk

Context: ServiceManager.sol#L155-L163

Description: When registering a new operator, the code loops over all strategy and sum the total shares.

```
for (uint256 i; i != strategies.length;) {
    totalStake += IDelegationManager(delegationManager).operatorShares(msg.sender, IStrategy(strategies[i]));
    unchecked {
        ++i;
    }
}

if (totalStake >= thresholdStake) {
```

Each strategy has a different underlying token.

Therefore, this line is simply summing the raw "share balances" from each of the registered strategies without any normalization or weighting. Different strategies have different share valuations (one share in strategy A represent a different underlying stake amount/value than one share in strategy B).

Therefore, summing share counts directly across multiple strategies is inaccurate considering that the intention here is to measure "total staked value" in a unified unit (e.g., total underlying tokens USD value).

Currently, this is the list of all the live strategies:

- 0xbeaC0eeEeeeeEEeEeEEEEeeEEeEeeeEeeEEBEaC0 ⇒ underlying token is ETH.
- 0x93c4b944D05dfe6df7645A86cd2206016c51564D ⇒ underlying token is stETH.
- 0x1BeE69b7dFFfA4E2d53C2a2Df135C388AD25dCD2 ⇒ underlying token is rocket ETH.
- 0x54945180dB7943c0ed0FEE7EdaB2Bd24620256bc ⇒ underlying token is coinbase ETH.

While these underlying token are all backed by ETH (ETH derivative), the underlying assumption is that all shares backed by different ETH derivatives are equal, which is never the case and, therefore, the `thresholdStake` check enforced to become an operator is totally inaccurate.

Recommendation: Consider introducing a oracle solution to convert the share worth from different strategies to a unified unit in the future.

Predicate: Acknowledged. This recommendation will be implemented in a future version.

Cantina Managed: Acknowledged.

3.3 Low Risk

3.3.1 Orphaned signing key references upon operator deregistration

Severity: Low Risk

Context: ServiceManager.sol#L177, ServiceManager.sol#L504

Description: In the `ServiceManager` contract, in the `deregisterOperatorFromAVS` and `updateOperatorsForQuorum` functions, operators can be set to `DEREGISTERED`. However, their signing keys remain mapped in `signingKeyToOperator[_operatorSigningKey]`. This stale mapping means that the operator's signing key can still appear valid in certain checks.

Recommendation: Whenever an operator is deregistered (either manually in `deregisterOperatorFromAVS` or via automatic stake checks in `updateOperatorsForQuorum`), also clear out their signing key references:

```
delete signingKeyToOperator[oldSigningKey];
```

This ensures that no stale key mappings remain pointing to a deregistered operator.

Predicate: Acknowledged.

Cantina Managed: Acknowledged.

3.3.2 `deployPolicy` and `syncPolicies` functions do not validate non-empty policy

Severity: Low Risk

Context: ServiceManager.sol#L195-L199, SimpleServiceManager.sol#L160

Description: In the `ServiceManager.deployPolicy` function:

```
require(bytes(idToPolicy[_policyID]).length == 0, "Predicate.deployPolicy: policy exists");
```

checks that the `_policyID` does not already exist. However, there is no check ensuring the `_policy` string itself has a nonzero length. This check is also missing in the `SimpleServiceManager.syncPolicies` function.

Recommendation: Add a check in both functions that ensure that `_policy` is not empty:

```
require(bytes(_policy).length > 0, "Predicate.deployPolicy: policy string cannot be empty");
```

so that a new policy must contain actual content.

Predicate: Fixed in PR 21.

Cantina Managed: Fix verified.

3.3.3 Validate there is no duplicate strategy in strategy array when adding new strategy

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: When `registerOperatorToAVS`, a new operator has to have sufficient stake share, the code iterates over all strategies to compute the share.

When adding strategy, the code should validate if there are duplicate strategies in the array, otherwise if the strategy array contains duplicate strategy, the operator's staking share can be double counted.

```
    function addStrategy(address _strategy, uint8 quorumNumber, uint256 index) external onlyOwner {
        IStakeRegistry.StrategyParams memory strategyParams =
            IStakeRegistry(stakeRegistry).strategyParamsByIndex(quorumNumber, index);
        if (address(strategyParams.strategy) != _strategy) {
            revert ServiceManager__InvalidStrategy();
        }
        strategies.push(_strategy);
        emit StrategyAdded(_strategy);
    }
```

Recommendation: Validate there is no duplicate strategy in strategy array.

Predicate: Acknowledged. This recommendation will be implemented in a future version.

Cantina Managed: Acknowledged.

3.4 Gas Optimization

3.4.1 Storing raw policy strings on-chain increases gas costs

Severity: Gas Optimization

Context: ServiceManager.sol#L36

Description: The `ServiceManager` contract uses:

```
mapping(string => string) public idToPolicy;
```

Both the "policy ID" and the "policy text" are unbounded strings on-chain. Large or variable-length strings can occupy multiple storage slots, leading to high gas usage when storing or updating policies.

Recommendation:

- Use an `uint256` for the `policyID`:

  ```
  mapping(uint256 => bytes32) public idToPolicyHash;
  ```

  This ensures policy IDs are stored in a single 256-bit slot, removing overhead from unbounded string operations.

- Store policy text as a hash: Instead of storing the entire policy text, store a bytes32 hash (e.g., `keccak256`) of the policy:

  ```
  bytes32 hashOfPolicy = keccak256(bytes(policyText));
  ```

This way, you reduce on-chain data to 32 bytes per policy. The full text can remain off-chain and users can verify it by hashing locally and comparing to the on-chain record. By adopting these changes, you significantly reduce storage costs and gas consumption, while preserving a unique identifier (`uint256 policyID`) and a verifiable reference to the policy content (`bytes32 policyHash`).

Predicate: Acknowledged. This recommendation will be implemented in a future version.

Cantina Managed: Acknowledged.

3.5 Informational

3.5.1 Smart contract wallet signatures are not supported

Severity: Informational

Context: ServiceManager.sol#L316

Description: In the `validateSignatures` function, operator signatures are recovered with:

```
address recoveredSigner = ECDSA.recover(messageHash, signatures[i]);
```

This only validates ECDSA signatures from externally owned accounts (EOAs). Smart contracts (such as Gnosis Safe or custom multisig wallets) cannot generate these ECDSA signatures directly, meaning a contract-based operator cannot sign tasks. This restricts potential operator implementations to EOAs only.

Recommendation: Use a library that can handle both EOAs and contract wallets, such as `OpenZeppelin's SignatureChecker.isValidSignatureNow`. This accommodates EIP-1271 for contract signatures, enabling multisig or contract-based operator accounts:

```
require(
    SignatureChecker.isValidSignatureNow(
        operatorSigner,
        messageHash,
        signatures[i]
    ),
    "Invalid signature"
);
```

If adopting contract signatures, be mindful of reentrancy or other complexities that arise from calling an external `isValidSignature` function on an untrusted contract. Employ reentrancy guards and thorough review the EIP-1271 flow to mitigate these risks.

Predicate: Acknowledged. We will consider implementing this in a future version.

Cantina Managed: Acknowledged.

3.5.2 Use an `uint256` Task ID instead of a string for `spentTaskIds` mapping

Severity: Informational

Context: ServiceManager.sol#L37

Description: Currently, the contract tracks replay protection with a mapping of the form:

```
mapping(string => bool) public spentTaskIds;
```

This stores task IDs as arbitrary strings. However, strings are more expensive to store and may be prone to collisions or unexpected issues if different encodings are used. Relying on a string for a nonce or unique identifier can also complicate off-chain logic.

Recommendation: Use a numerical task ID (an incrementing uint256) in place of a string:

```
mapping(uint256 => bool) public spentTaskIds;
```

This is cheaper to store (only one storage slot per integer) and ensures simpler, more consistent logic around incrementing and checking uniqueness. It also reduces ambiguity about possible string collisions or encoding differences.

Predicate: Acknowledged. This will be implemented in a future version.

Cantina Managed: Acknowledged.

3.5.3 Lack of a double-step transfer ownership pattern

Severity: Informational

Context: ServiceManager.sol#L16

Description: All the contracts are using the standard OpenZeppelin's Ownable library. The standard OpenZeppelin's Ownable contract allows transferring the ownership of the contract in a single step:

```
/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Can only be called by the current owner.
 */
function transferOwnership(address newOwner) public virtual onlyOwner {
    if (newOwner == address(0)) {
        revert OwnableInvalidOwner(address(0));
    }
    _transferOwnership(newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

If the nominated EOA account is not a valid account, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing the access to all functions with the `onlyOwner` modifier.

Recommendation: It is recommended to implement a two-step transfer process in all the contracts in the codebase where the owner nominates an account and the nominated account needs to call an `acceptOwnership()` function for the transfer of the ownership to fully succeed. This ensures the nominated EOA account is a valid and active account. A good code example could be OpenZeppelin's Ownable2Step contract:

```
/**
 * @dev Starts the ownership transfer of the contract to a new account. Replaces the pending transfer if there
 ↪   is one.
 * Can only be called by the current owner.
 *
 * Setting `newOwner` to the zero address is allowed; this can be used to cancel an initiated ownership
 ↪   transfer.
 */
function transferOwnership(address newOwner) public virtual override onlyOwner {
    _pendingOwner = newOwner;
    emit OwnershipTransferStarted(owner(), newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`) and deletes any pending owner.
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual override {
    delete _pendingOwner;
    super._transferOwnership(newOwner);
}

/**
 * @dev The new owner accepts the ownership transfer.
 */
function acceptOwnership() public virtual {
    address sender = _msgSender();
    if (pendingOwner() != sender) {
        revert OwnableUnauthorizedAccount(sender);
    }
    _transferOwnership(sender);
}
```

Predicate: Fixed in commit f922c15.

Cantina Managed: Fix verified.

3.5.4 Unused code

Severity: Informational

Context: ServiceManager.sol#L75-L80, ServiceManager.sol#L97-L121, ServiceManager.sol#L478

Description: Several pieces of code in `ServiceManager` are unused or do not serve a functional purpose:

```
modifier onlyPermissionedOperator() {
    if (!permissionedOperators[msg.sender]) {
        revert ServiceManager__Unauthorized();
    }
    _;
}

function addPermissionedOperators(address[] calldata _operators) external onlyOwner { ... }
function removePermissionedOperators(address[] calldata _operators) external onlyOwner { ... }
```

Similarly, in the function `updateOperatorsForQuorum`, the function never references the `quorumNumbers[i]`, only `quorumNumbers.length`.

In short:

- `onlyPermissionedOperator` is never called, and `addPermissionedOperators` / `removePermissioned-Operators` are similarly unused.
- `quorumNumbers` is read only for its length; the actual bytes inside are never parsed.

Recommendation: Remove all these unused features to maintain clarity and minimize maintenance. Specifically:

- Remove `onlyPermissionedOperator`, `addPermissionedOperators` and `removePermissionedOpera-tors` if permissioned-operator functionality is no longer intended.

- Eliminate the `quorumNumbers` parameter if it is never used for anything but length checks. If you truly require multiple quorums, parse and apply those bytes meaningfully; otherwise, drop the parameter altogether.

Predicate: Acknowledged. Some of this code will be re-used and the other will be removed in a future version.

Cantina Managed: Acknowledged.

3.5.5 Strategies array can be optimized

Severity: Informational

Context: ServiceManager.sol#L42

Description: Currently, the `ServiceManager` contract uses a simple dynamic array:

```
address[] public strategies;
```

to track recognized strategies. This approach allows duplicates, provides no straightforward way to check if a strategy is already stored and forces a manual search to remove items. It can lead to inconsistent or duplicated data if a strategy is pushed twice.

Recommendation: Use the OpenZeppelin's EnumerableSet library instead of an array:

```
EnumerableSet.AddressSet public supportedStrategies;
```

This ensures:

- You can add a strategy only if it's not already in the set.
- You can remove a strategy cleanly without searching the array.
- You avoid duplicates and can still enumerate all supported strategies when needed.

Predicate: Acknowledged. This will be implemented in a future version.

Cantina Managed: Acknowledged.

3.5.6 Incompatibility with non-standard ERC20 tokens in `TellerWithMultiAssetSupportPredicateProxy`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `deposit` and `depositAndBridge` functions, the contract calls `ERC20.transferFrom`, `ERC20.approve`, etc. directly. Some tokens do not adhere strictly to the `ERC20` standard or have non-standard return values; using raw `transferFrom` or `approve` can fail to detect such cases or revert incorrectly.

Recommendation: Adopt OpenZeppelin's SafeERC20 wrapper:

```
using SafeERC20 for IERC20;

IERC20(depositAsset).forceApprove(address(vault), depositAmount);
IERC20(depositAsset).safeTransferFrom(msg.sender, address(this), depositAmount);
```

These `SafeERC20` variant:

- Verify the token call's return value, ensuring it either returns true or does not return data at all.
- Revert if the token's call fails or returns an unexpected value.
- Improve compatibility with tokens that have non-standard implementations.

Predicate: Acknowledged.

Cantina Managed: Acknowledged.

3.5.7 Operators can not validate or limit `deposit` or `depositAndBridge` parameters

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `TellerWithMultiAssetSupportPredicateProxy` contract, the `_authorizeTransaction` call only encodes:

```
bytes memory encodedSigAndArgs = abi.encodeWithSignature("deposit()");
```

or

```
bytes memory encodedSigAndArgs = abi.encodeWithSignature("depositAndBridge()");
```

This means the operator signatures only approve the function name without referencing parameters such as `depositAmount`, `recipient` or `BridgeData`. Consequently, operators are "blind" to these arguments and cannot validate or limit them.

Recommendation: If you want operators to have visibility or control over the actual parameters (e.g., maximum deposit, permitted recipient, bridging details), include those arguments in `encodedSigAndArgs`. For example:

```
bytes memory encodedSigAndArgs = abi.encodeWithSignature(
    "deposit(address,uint256,uint256,address,address)",
    depositAsset,
    depositAmount,
    minimumMint,
    recipient,
    teller
);
```

This ensures that operator signatures explicitly cover the parameter values, preventing users from passing unexpected amounts or addresses. If parameter-level oversight is not desired, the current design is acceptable, but typically advanced operator gating requires parameter-aware authorization.

Predicate: Acknowledged.

Cantina Managed: Acknowledged.

3.5.8 Missing events in key setter functions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The following state-modifying functions do not emit events.

- `addPermissionedOperators`.
- `removePermissionedOperators`.
- `rotatePredicateSigningKey`.

Recommendation: Consider emitting events that reflect the state changes in the functions listed above.

Predicate: Fixed in commit 279b9fcb.

Cantina Managed: Fix verified.

3.5.9 `syncPolicies` function comment is inconsistent with implementation

Severity: Informational

Context: SimpleServiceManager.sol#L156-L180

Description: The `syncPolicies`'s comment outlines that the function should be able to register or update policy IDs. However, the function can only register new policy IDs. Once a policy Id is registered, the function can not update that policy Id because `policyIDToThreshold[policyIDs[i]]` is no longer 0.

```
if (policyIDToThreshold[policyIDs[i]] == 0) {
    policyIDToThreshold[policyIDs[i]] = thresholds[i];
    deployedPolicyIDs.push(policyIDs[i]);
    emit PolicySynced(policyIDs[i]);
}
```

Recommendation: Consider correcting the `syncPolicies`'s comment.

Predicate: Fixed in PR 21.

Cantina Managed: Fix verified.