Safe Harbor Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Kage

Farouk

January 13, 2026

Contents

1 About Cyfrin **2**

2 Disclaimer **2**

3 Risk Classification **2**

4 Protocol Summary **2**

5 Audit Scope **2**

6 Executive Summary **2**

7 Findings **4**

7.1 Medium Risk . . . 4

7.1.1 Free memory pointer corruption in inline assembly causes memory allocation panic and denial of service . . . 4

7.2 Low Risk . . . 8

7.2.1 Missing bounty percentage and cap validation . . . 8

7.2.2 Missing validation allows empty account addresses in agreement scope, creating ambiguous whitehat coverage . . . 8

7.2.3 `Agreement::removeAccounts` can leave chain with zero accounts . . . 9

7.2.4 Optional `signature` field specified in legal agreement but not implemented in `Chain` struct . . . 10

7.3 Informational . . . 11

7.3.1 Missing contact details validation . . . 11

7.3.2 In Solidity don't initialize to default values . . . 11

7.3.3 Remove obsolete final `return` statement when already using named returns . . . 12

7.3.4 Inconsistencies in names and data types between legal agreement and smart contracts . . . 12

7.4 Gas Optimization . . . 13

7.4.1 Prefer `calldata` instead of `memory` for read-only external inputs . . . 13

7.4.2 Cache storage to prevent identical storage reads . . . 13

7.4.3 Don't cache `calldata` array length . . . 13

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

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

4 Protocol Summary

The Whitehat Safe Harbor initiative is a framework in which protocols can offer legal protection to whitehats who aid in the recovery of assets during an active exploit.

5 Audit Scope

The audit scope was limited to:

```
registry-contracts/script/Deploy.s.sol
registry-contracts/script/HelperConfig.s.sol
registry-contracts/src/utils/Utils.sol
registry-contracts/src/Agreement.sol
registry-contracts/src/AgreementFactory.sol
registry-contracts/src/ChainValidator.sol
registry-contracts/src/SafeHarborRegistry.sol
```

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the Safe Harbor smart contracts provided by Security Alliance. In this period, a total of 12 issues were found.

The findings consist of 1 Medium and 4 Low severity issues with the remainder being informational and gas optimizations.

Summary

Project Name	Safe Harbor
Repository	safe-harbor
Commit	91e2165eba87...
Fix Commit	0b0abb8b627e...
Audit Timeline	Jan 5th - Jan 9th, 2026
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	4
Informational	4
Gas Optimizations	3
Total Issues	12

Summary of Findings

[M-1] Free memory pointer corruption in inline assembly causes memory allocation panic and denial of service	Resolved
[L-1] Missing bounty percentage and cap validation	Resolved
[L-2] Missing validation allows empty account addresses in agreement scope, creating ambiguous whitehat coverage	Resolved
[L-3] `Agreement::removeAccounts` can leave chain with zero accounts	Resolved
[L-4] Optional `signature` field specified in legal agreement but not implemented in `Chain` struct	Resolved
[I-1] Missing contact details validation	Resolved
[I-2] In Solidity don't initialize to default values	Resolved
[I-3] Remove obsolete final `return` statement when already using named returns	Resolved
[I-4] Inconsistencies in names and data types between legal agreement and smart contracts	Resolved
[G-1] Prefer `calldata` instead of `memory` for read-only external inputs	Resolved
[G-2] Cache storage to prevent identical storage reads	Resolved
[G-3] Don't cache `calldata` array length	Resolved

7 Findings

7.1 Medium Risk

7.1.1 Free memory pointer corruption in inline assembly causes memory allocation panic and denial of service

Description: The `AgreementFactory` contract uses inline assembly to derive a CREATE2 salt by writing intermediate values directly into memory. In both `AgreementFactory::create` and `AgreementFactory::computeAddress`, the assembly block writes to memory slot `0x40` using `mstore(0x40, salt)`.

```
// Include chainid in salt to prevent cross-chain address collisions
bytes32 finalSalt;
assembly {
    mstore(0x00, chainid())
    mstore(0x20, caller())
    mstore(0x40, salt)
    finalSalt := keccak256(0x00, 0x60)
}
```

In Solidity, memory slot `0x40` is reserved for the free-memory pointer, which tracks the next available memory location for dynamic allocations. Overwriting this slot corrupts the allocator state.

Immediately after this assembly block, the contract performs dynamic memory allocations via `bytes.concat(...)` and `abi.encode(...)`. Since the free-memory pointer has been overwritten with an arbitrary value (`salt`), the allocator attempts to allocate memory at an invalid or extremely large offset, triggering a Solidity panic with error code `0x41` (memory allocation error).

This issue is deterministic and affects any execution path that reaches these memory allocations after the corrupted pointer, making the affected functions unusable.

Impact: Both `AgreementFactory::create` and `AgreementFactory::computeAddress` consistently revert with `panic: memory allocation error (0x41)` which results in a denial of service for the core factory functionality.

Proof of Concept:

- `AgreementFactoryTest::test_createWithFreeMemoryPointerIssue`:

```
function test_createWithFreeMemoryPointerIssue() public {
    // Prepare a fully populated AgreementDetails struct
    // The exact contents are not important; the bug is independent of input values
    AgreementDetails memory agreementDetails = getMockAgreementDetails("0xAABB");
    bytes32 salt = keccak256("test-salt");

    // Impersonate the protocol address (msg.sender)
    vm.prank(protocol);

    // Expect a Solidity panic due to memory allocation failure (Panic(0x41))
    // This happens because mstore(0x40, salt) corrupts the free-memory pointer,
    // and the subsequent Agreement constructor call performs dynamic memory allocation.
    vm.expectRevert(stdError.memOverflowError);

    // This call deterministically reverts due to free memory pointer corruption
    factory.create(agreementDetails, address(chainValidator), protocol, salt);
}
```

- Output:

```
Ran 1 test for test/unit/AgreementFactoryTest.t.sol:AgreementFactoryTest
[PASS] test_createWithFreeMemoryPointerIssue() (gas: 25813)
Logs:
  Getting network config for chain ID: 31337
  Getting network config for chain ID: 31337
```

```
  ChainValidator implementation deployed at: 0x42Ad6372A7676878a95Ae9993D6eB88543A7D47a

Traces:
  [25813] AgreementFactoryTest::test_createWithFreeMemoryPointerIssue()
     [0] VM::prank(0x00000000000000000000000000000000000000AB)
        ← [Return]
     [0] VM::expectRevert(custom error 0xf28dceb3:  $NH{qA)
        ← [Return]
     [6024] AgreementFactory::create(AgreementDetails({ protocolName: "testProtocolV2", contactDetails:
    ↪  [Contact({ name: "Test Name V2", contact: "test@mail.com" })], chains: [Chain({
    ↪  assetRecoveryAddress: "0x0000000000000000000000000000000000000022", accounts: [Account({
    ↪  accountAddress: "0xAABB", childContractScope: 2 })], caip2ChainId: "eip155:1" })], bountyTerms:
    ↪  BountyTerms({ bountyPercentage: 10, bountyCapUSD: 100, retainable: false, identity: 0,
    ↪  diligenceRequirements: "none", aggregateBountyCapUSD: 1000 }), agreementURI: "ipfs://testHash"
    ↪  }), ERC1967Proxy: [0x2FF6CD634dfa4B2105dc8f53ca262e64Ed089049],
    ↪  0x00000000000000000000000000000000000000AB,
    ↪  0x8bcfa1e0aed22543ed44d41a95e315383294a18f9fb6e67ee082afcd585a6ff1)
        ← [Revert] panic: memory allocation error (0x41)
     ← [Return]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.22ms (170.75µs CPU time)
```

- AgreementFactoryTest::test_computeAddressWithFreeMemoryPointerIssue:

```
function test_computeAddressWithFreeMemoryPointerIssue() public {
    // Prepare inputs for address computation
    AgreementDetails memory agreementDetails = getMockAgreementDetails("0xAABB");
    bytes32 salt = keccak256("test-salt");

    // Expect the same memory allocation panic (Panic(0x41))
    // In this case, the revert occurs during:
    // bytes.concat(type(Agreement).creationCode, abi.encode(...))
    // because abi.encode reads the corrupted free-memory pointer.
    vm.expectRevert(stdError.memOverflowError);

    // Address computation reverts before returning a value
    factory.computeAddress(
        agreementDetails,
        address(chainValidator),
        protocol,
        salt,
        protocol
    );
}
```

- Output:

```
Ran 1 test for test/unit/AgreementFactoryTest.t.sol:AgreementFactoryTest
[PASS] test_computeAddressWithFreeMemoryPointerIssue() (gas: 25485)
Logs:
  Getting network config for chain ID: 31337
  Getting network config for chain ID: 31337
  ChainValidator implementation deployed at: 0x42Ad6372A7676878a95Ae9993D6eB88543A7D47a

Traces:
  [25485] AgreementFactoryTest::test_computeAddressWithFreeMemoryPointerIssue()
     [0] VM::expectRevert(custom error 0xf28dceb3:  $NH{qA)
        ← [Return]
```

```
    [6210] AgreementFactory::computeAddress(AgreementDetails({ protocolName: "testProtocolV2",
    ↪ contactDetails: [Contact({ name: "Test Name V2", contact: "test@mail.com" })], chains: [Chain({
    ↪ assetRecoveryAddress: "0x0000000000000000000000000000000000000022", accounts: [Account({
    ↪ accountAddress: "0xAABB", childContractScope: 2 })], caip2ChainId: "eip155:1" })], bountyTerms:
    ↪ BountyTerms({ bountyPercentage: 10, bountyCapUSD: 100, retainable: false, identity: 0,
    ↪ diligenceRequirements: "none", aggregateBountyCapUSD: 1000 }), agreementURI: "ipfs://testHash"
    ↪ }), ERC1967Proxy: [0x2FF6CD634dfa4B2105dc8f53ca262e64Ed089049],
    ↪ 0x000000000000000000000000000000000000AB,
    ↪ 0x8bcfa1e0aed22543ed44d41a95e315383294a18f9fb6e67ee082afcd585a6ff1,
    ↪ 0x000000000000000000000000000000000000AB) [staticcall]
        ← [Revert] panic: memory allocation error (0x41)
    ← [Return]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.63ms (410.21µs CPU time)

Ran 1 test suite in 146.15ms (3.63ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended Mitigation: Consider allocating memory using the free-memory pointer safely:

```
    function create(
        AgreementDetails memory details,
        address chainValidator,
        address owner,
        bytes32 salt
    )
        external
        returns (address agreementAddress)
    {
        // Include chainid in salt to prevent cross-chain address collisions
        bytes32 finalSalt;
        assembly {
-           mstore(0x00, chainid())
-           mstore(0x20, caller())
-           mstore(0x40, salt)
-           finalSalt := keccak256(0x00, 0x60)
+           let ptr := mload(0x40)
+           mstore(ptr, chainid())
+           mstore(add(ptr, 0x20), caller())
+           mstore(add(ptr, 0x40), salt)
+           finalSalt := keccak256(ptr, 0x60)
+           mstore(0x40, add(ptr, 0x60))
        }
```

```
    function computeAddress(
        AgreementDetails memory details,
        address chainValidator,
        address owner,
        bytes32 salt,
        address deployer
    )
        external
        view
        returns (address)
    {
        bytes32 finalSalt;
        assembly {
-           mstore(0x00, chainid())
-           mstore(0x20, deployer)
-           mstore(0x40, salt)
-           finalSalt := keccak256(0x00, 0x60)
+           let ptr := mload(0x40)
+           mstore(ptr, chainid())
```

```
+            mstore(add(ptr, 0x20), deployer)
+            mstore(add(ptr, 0x40), salt)
+            finalSalt := keccak256(ptr, 0x60)
+            mstore(0x40, add(ptr, 0x60))
        }
```

SafeHarbor: Fixed in 57434a1.

Cyfrin: Verified.

7.2 Low Risk

7.2.1 Missing bounty percentage and cap validation

Description: The `Agreement::_validateBountyTerms` lacks two important validations:

1. No upper bound is enforced on `bountyPercentage`
2. No validation that `aggregateBountyCapUSD >= bountyCapUSD`

Impact: If `bountyPercentage > 100` is set mistakenly by the owner, protocol can become legally liable to pay more than what was recovered when `retainable = false`.

`aggregateBountyCapUSD >= bountyCapUSD` also creates ambiguity that can lead to disputes. For eg., If `bountyCapUSD = $1M` but `aggregateBountyCapUSD = $500K`, it's unclear whether a single whitehat can receive $1M or is limited to $500K.

Recommended Mitigation: Consider adding additional validations to `Agreement::_validateBountyTerms`.

SafeHarbor: Fixed in 3c0b5a95.

Cyfrin: Verified.

7.2.2 Missing validation allows empty account addresses in agreement scope, creating ambiguous whitehat coverage

Description: The `Agreement::_validateChains` validates that the accounts array is non-empty but does not validate the contents of individual `Account` structs. This allows accounts with empty `accountAddress` strings to be added to agreements.

Note that `assetRecoveryAddress` is validated for non-empty strings, but the same validation is not applied to `accountAddress` within each account.

Impact: A protocol could accidentally deploy an agreement that appears valid but provides no actual scope definition, rendering the Safe Harbor adoption ineffective. Also, when querying the Safe Harbor Registry, whitehats cannot determine if an empty account address represents "all addresses" or "no addresses".

Proof of Concept: Add this test to `AgreementTest.t.sol`:

```
function test_POC_emptyAccountAddressAllowed() public {
    // Create account with empty address
    SHAccount[] memory accounts = new SHAccount[](1);
    accounts[0] = SHAccount({
        accountAddress: "",  // Empty!
        childContractScope: ChildContractScope.All
    });

    SHChain[] memory chains = new SHChain[](1);
    chains[0] = SHChain({
        assetRecoveryAddress: "0x742d35Cc6634C0532925a3b844Bc454e",
        accounts: accounts,
        caip2ChainId: "eip155:56"
    });

    vm.prank(owner);
    agreement.addChains(chains); //@note this does not revert

    // Verify empty account was added
    AgreementDetails memory details = agreement.getDetails();

    bool foundBscChain = false;
    for (uint256 i = 0; i < details.chains.length; i++) {
        if (keccak256(bytes(details.chains[i].caip2ChainId)) == keccak256(bytes("eip155:56"))) {
            foundBscChain = true;
            assertEq(details.chains[i].accounts.length, 1);
```

```
                assertEq(details.chains[i].accounts[0].accountAddress, "");
                break;
            }
        }
        assertTrue(foundBscChain, "BSC chain not found");
    }
```

Recommended Mitigation: Consider adding validation for individual account addresses within `_validateChains`, consistent with the existing validation for `assetRecoveryAddress`.

SafeHarbor: Fixed in e32dd76 and a881fdc.

Cyfrin: Verified.

7.2.3 `Agreement::removeAccounts` can leave chain with zero accounts

Description: When creating or modifying chains via `Agreement::addChains`, `Agreement::addOrSetChains`, the `Agreement::_validateChains` function enforces that each chain must have at least one account:

```
function _validateChains(Chain[] memory _chains) internal {
    for (uint256 i = 0; i < _chains.length; i++) {
        // ...
        if (_chains[i].accounts.length == 0) { //@audit does not allow chain with zero accounts
            revert Agreement__ZeroAccountsForChainId(_chains[i].caip2ChainId);
        }
        // ...
    }
}
```

However, `Agreement::removeAccounts` does not check whether removal would leave zero accounts:

```
function removeAccounts(string memory _caip2ChainId, string[] memory _accountAddresses) external
↪   onlyOwner {
    if (!_chainExists(_caip2ChainId)) {
        revert Agreement__ChainNotFoundByCaip2Id(_caip2ChainId);
    }
    // @audit No check for remaining accounts after removal
    for (uint256 i = 0; i < _accountAddresses.length; i++) {
        uint256 accountIndex = _findAccountIndex(_caip2ChainId, _accountAddresses[i]);
        emit AccountRemoved(_caip2ChainId, _accountAddresses[i]);

        uint256 lastAccountId = accounts[_caip2ChainId].length - 1;
        accounts[_caip2ChainId][accountIndex] = accounts[_caip2ChainId][lastAccountId];
        accounts[_caip2ChainId].pop();
    }
}
```

Impact: A chain can exist with zero accounts in scope, violating the check enforced during creation.

Proof of Concept: Add the following to `Agreement.t.sol`:

```
function test_POC_removeAllAccountsFromChain() public {
        AgreementDetails memory detailsBefore = agreement.getDetails();
        assertEq(detailsBefore.chains[0].accounts.length, 1);

        string memory chainId = detailsBefore.chains[0].caip2ChainId;
        string memory accountAddr = detailsBefore.chains[0].accounts[0].accountAddress;

        // Remove the account
        string[] memory toRemove = new string[](1);
        toRemove[0] = accountAddr;

        vm.prank(owner);
```

```
        agreement.removeAccounts(chainId, toRemove);

        // Verify chain now has zero accounts
        AgreementDetails memory detailsAfter = agreement.getDetails();
        assertEq(detailsAfter.chains[0].accounts.length, 0);  // @audit chain now has zero accounts

    }
```

Recommended Mitigation: Consider adding a check in `Agreement::removeAccounts` to ensure at least one account remains.

SafeHarbor: Fixed in bf411ed.

Cyfrin: Verified.

7.2.4 Optional `signature` field specified in legal agreement but not implemented in `Chain` struct

Description: The SEAL Whitehat Safe Harbor Agreement legal document specifies an optional `signature` field within the `Chain` struct that is not implemented in the smart contract code.

> Legal Agreement Reference (Section 1.1(c)(ii)): "D. optionally, the signature for such account, which may be used as additional evidence that such account has affirmatively accepted being subject to this Agreement."

This field was designed to allow protocols to provide cryptographic proof that specific accounts on a given chain have explicitly consented to being included in the Safe Harbor scope.

However the current implementation has:

```
struct Chain {
    // The address to which recovered assets will be sent.
    string assetRecoveryAddress;
    // The accounts in scope for the agreement.
    Account[] accounts;
    // The CAIP-2 chain ID.
    string caip2ChainId;
    // @audit Missing: signature field as specified in agreement Section 1.1(c)(ii)(D)
}
```

Impact: While the `signature` field is described as "optional" in the agreement, its complete omission from the implementation means protocols cannot utilize this feature even if they wish to provide additional evidence of account-level consent for a specific chain.

Recommended Mitigation: Consider adding `signature` field to the Chain struct to maintain feature parity with the legal agreement. Alternatively, update the agreement to remove this field.

SafeHarbor: Fixed in ebfcb1a.

Cyfrin: Verified.

7.3 Informational

7.3.1 Missing contact details validation

Description: `Agreement::_setDetails` copies contact details without any validation. The `Contact` struct fields can contain empty strings:

Neither the constructor nor `setContactDetails` validates that contact fields are non-empty:

```
function _setDetails(AgreementDetails memory _details) internal {
     // code...
     delete contactDetails;
     for (uint256 i = 0; i < _details.contactDetails.length; ++i) {
         contactDetails.push(_details.contactDetails[i]); //@audit no check for empty name or contact
      }
}

function setContactDetails(Contact[] memory _contactDetails) external onlyOwner {
    emit ContactDetailsSet(_contactDetails);
    delete contactDetails;
    for (uint256 i = 0; i < _contactDetails.length; i++) {
        contactDetails.push(_contactDetails[i]); //@audit no check for empty name or contact
    }
}
```

Impact: Whitehats rely on contact details for prior notification before rescue operations. Empty or invalid contact information could prevent whitehats from reaching out to protocol teams effectively.

Recommended Mitigation: Consider adding validation for contact details. Alternatively, document that empty contacts are allowed.

SafeHarbor: Fixed in 0f492fc.

Cyfrin: Verified.

7.3.2 In Solidity don't initialize to default values

Description: In Solidity don't initialize to default values:

```
Agreement.sol
95:         for (uint256 i = 0; i < _contactDetails.length; i++) {
105:         for (uint256 i = 0; i < _chains.length; i++) {
114:             for (uint256 j = 0; j < _chains[i].accounts.length; j++) {
125:         for (uint256 i = 0; i < _chains.length; i++) {
137:             for (uint256 j = 0; j < _chains[i].accounts.length; j++) {
149:         for (uint256 i = 0; i < _chains.length; i++) {
158:             for (uint256 j = 0; j < _chains[i].accounts.length; j++) {
169:         for (uint256 i = 0; i < _caip2ChainIds.length; i++) {
192:         for (uint256 i = 0; i < _accounts.length; i++) {
207:         for (uint256 i = 0; i < _accountAddresses.length; i++) {
235:         for (uint256 i = 0; i < _details.contactDetails.length; ++i) {
241:         for (uint256 i = 0; i < _details.chains.length; ++i) {
247:             for (uint256 j = 0; j < _details.chains[i].accounts.length; ++j) {
257:         for (uint256 i = 0; i < _chains.length; i++) {
288:         for (uint256 i = 0; i < _chains.length; i++) {
307:         for (uint256 i = 0; i < contactsLength; ++i) {
314:         for (uint256 i = 0; i < chainsLength; ++i) {
321:             for (uint256 j = 0; j < accts.length; ++j) {
365:         for (uint256 i = 0; i < length; ++i) {
378:         for (uint256 i = 0; i < length; i++) {
398:         for (uint256 i = 0; i < chainAccounts.length; i++) {

SafeHarborRegistry.sol
34:         uint256 migratedCount = 0;
```

```
36:         for (uint256 i = 0; i < length; i++) {

ChainValidator.sol
39:         for (uint256 i = 0; i < length; i++) {
64:         for (uint256 i = 0; i < length; i++) {
80:         for (uint256 i = 0; i < length; i++) {
```

SafeHarbor: Fixed in commit ed67312.

Cyfrin: Verified.

7.3.3 Remove obsolete final `return` statement when already using named returns

Description: Remove obsolete final `return` statement when already using named returns:

- `AgreementFactory::create`

SafeHarbor: Fixed in commit 220e5fb.

Cyfrin: Verified.

7.3.4 Inconsistencies in names and data types between legal agreement and smart contracts

Description: The legal agreement (Section 1.1(c)) specifies the `DAO Adoption Procedures` must call `adoptSafeHarbor` on `SafeHarborRegistryV2.sol` deployed at `0x1eaCD100B0546E433fbf4d773109cAD482c34686`, setting an `AgreementDetailsV2` struct.

The current implementation diverges in the following ways:

Component	Legal Agreement	Implementation
Registry Contract	SafeHarborRegistryV2.sol	SafeHarborRegistry.sol
Version	V2 (implied)	VERSION = "3.0.0"
Details Struct	AgreementDetailsV2	AgreementDetails

There is also a difference in naming of variables: Legal Agreement (section 1.1.c) | Implementation | Location -- | -- | -- chainID|caip2ChainId| `Chain` struct `identityRequirement`| `identity` | `BountyTerms` struct

There is a type naming difference: Legal Agreement (section 1.1.c.iv) | Implementation -- | -- `identityRequirement`| `identityRequirements`

There is a type difference: Field | Legal agreement (section 1.1.c.iv) | Implementation -- | -- | -- `bountPercentage`| `string`| `uint256`

Recommended Mitigation: Consider making the implementation consistent with the legal agreement terminology.

SafeHarbor: Fixed in 0d70af3.

Cyfrin: Verified.

7.4 Gas Optimization

7.4.1 Prefer `calldata` instead of `memory` for read-only external inputs

Description: Prefer `calldata` instead of `memory` for read-only external inputs which also don't get passed to internal functions that need them in `memory`:

- `Agreement.sol`

```
84:     function setProtocolName(string memory _protocolName) external onlyOwner {
91:     function setContactDetails(Contact[] memory _contactDetails) external onlyOwner {
// for `_accounts` only
187:     function addAccounts(string memory _caip2ChainId, Account[] memory _accounts) external
↪  onlyOwner {
```

- `ChainValidator.sol`

```
35:     function initialize(address _initialOwner, string[] memory _initialValidChains) external
↪  initializer {
```

SafeHarbor: Fixed in commits d72fd17, 754edb9.

Cyfrin: Verified.

7.4.2 Cache storage to prevent identical storage reads

Description: Cache storage to prevent identical storage reads:

- `Agreement.sol`

```
// cache `accts.length` in `getDetails`
320:             _details.chains[i].accounts = new Account[](accts.length);
321:             for (uint256 j = 0; j < accts.length; ++j) {

// cache `chainAccounts.length` in `_findAccountIndex`
398:         for (uint256 i = 0; i < chainAccounts.length; i++) {
```

SafeHarbor: Fixed in commit eb51eb0.

Cyfrin: Verified.

7.4.3 Don't cache `calldata` array length

Description: It is cheaper not to cache `calldata` array length (which is another reason why it is better to use `calldata` for input read-only arrays than `memory`):

- `ChainValidator::setvalidChains, setInvalidChains`

SafeHarbor: Fixed in commit af8cbc7.

Cyfrin: Verified.