FAILSAFE

Prepared for
dFusion

Prepared by
ret2basic.eth
jesjupyter

9 December 2025

dFusion Monad Contracts

Smart Contract

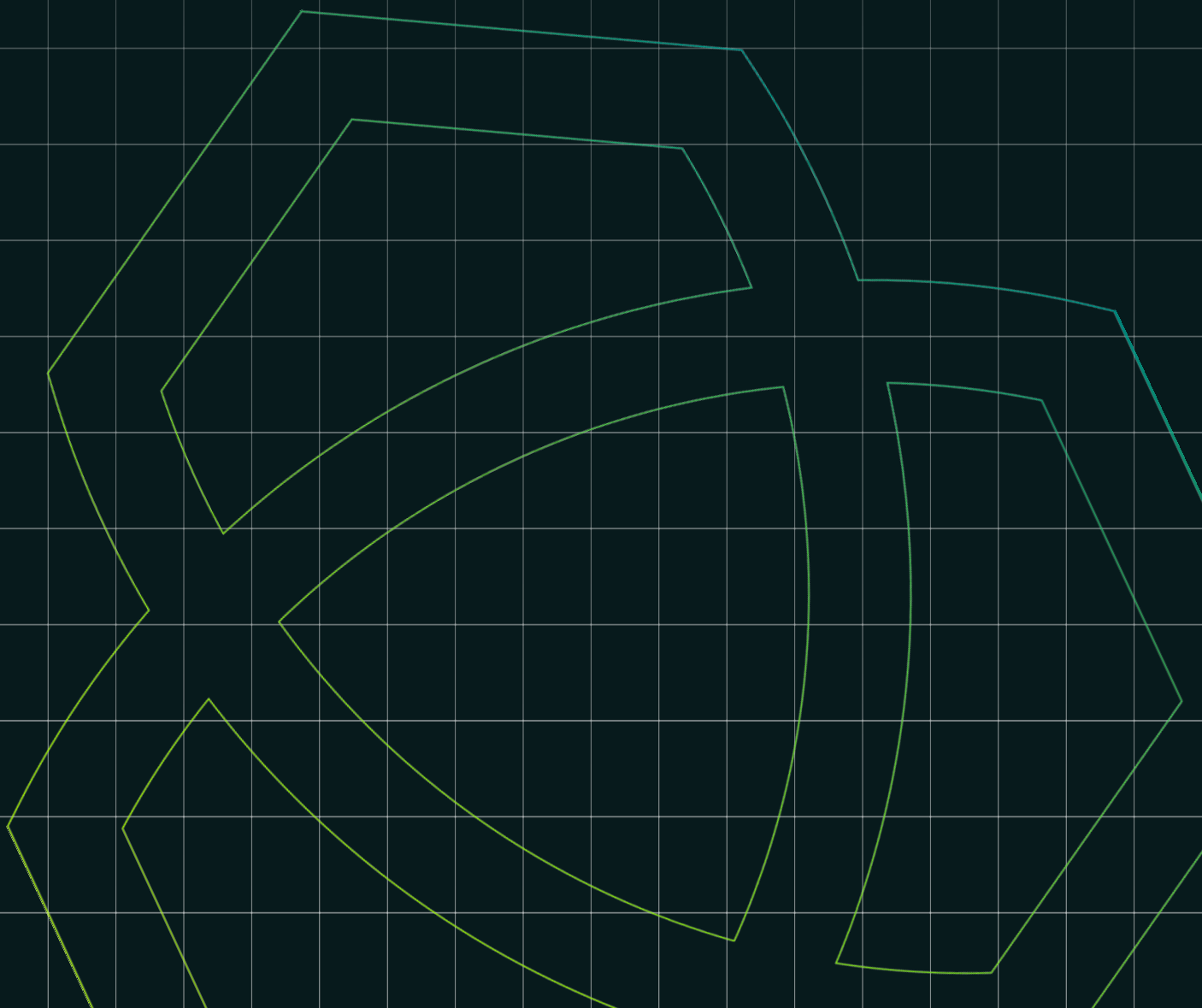

Table of Contents

Executive Summary

FailSafe was engaged to conduct an elite security review of the dFusion Monad Contracts, focusing on smart contract functionality within the Monad blockchain. Our seasoned team of security specialists executed a comprehensive analysis, applying industry-leading methodologies to identify possible vulnerabilities and areas for improvement. Through a meticulous audit process, our objective was to ensure the robustness and reliability of the contracts, ultimately fortifying the security posture of the dFusion project.

The audit unveiled several notable findings, including a high-severity vulnerability in the `SimplifiedAttestationCenter` contract that could lead to a permanent denial of service for reissued attestations following revocation. This issue highlights the critical importance of managing state transitions effectively within smart contracts. Additionally, we discovered medium-severity vulnerabilities such as the potential for unauthorized revocation by deauthorized attesters, which could enable malicious activity, and the absence of expiration for attestation signatures, posing risks in scenarios of key compromise. A low-severity finding related to potential hash collisions due to the use of `abi.encodePacked` with dynamic bytes was also identified. Moreover, an informational finding pointed out the risk of permanently disabling administrative functions through ownership renunciation, underscoring the necessity for careful administrative controls.

We commend the dFusion development team for their proactive approach to security and their dedication to enhancing the contract's resilience. The audit findings provide valuable insights that, once addressed, will significantly improve the security and functionality of the contracts. The team's commitment to incorporating best practices and addressing vulnerabilities demonstrates a strong overall security posture and a dedication to safeguarding their users and assets. We are confident that with the recommended changes, the dFusion Monad Contracts will continue to uphold high standards of security and performance.

Project Details

Project	dFusion Monad Contracts
Website	https://www.dfusion.ai/
Repository	https://github.com/dfusionai/dFusion-Monad-contracts
Blockchain	Monad
Audit Type	Smart Contract
Initial Commit	c1e0b88d6c125bb585da9bac3977f2e26b07f4a2
Final Commit	TBD
Timeline	3 December 2025 - 9 December 2025
	Final Report: TBD

Structure & Organization of The Security Report

Issues are tagged as “Open”, “Acknowledged”, “Partially Resolved”, “Resolved” or “Closed” depending on whether they have been fixed or addressed.

- Open: The issue has been reported and is awaiting remediation from developer team.
- Acknowledged: The developer team has reviewed and accepted the issue but has decided not to fix it.
- Partially Resolved: Mitigations have been applied, yet some risks or gaps still remain.
- Resolved: The issue has been fully addressed and no further work is necessary.
- Closed: The issue is deemed no longer pertinent or actionable.

Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical	The issue affects the platform in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
High	The issue affects the ability of the platform to compile or operate in a significant way.
Medium	The issue affects the ability of the platform to operate in a way that doesn’t significantly hinder its behavior.
Low	The issue has minimal impact on the platform’s ability to operate.
Info	The issue is informational in nature and does not pose any direct risk to the platform’s operation.

Methodology

Threat Modelling

We will employ a threat modelling approach to identify potential attack vectors and risks associated with the smart contract(s). This involves:

1. Asset Identification: Enumerating the critical assets within the smart contract(s), such as tokens, sensitive data, access controls, and more.
2. Threat Enumeration: Identifying potential threats such as reentrancy, integer overflow/underflow, denial of service, and more.
3. Vulnerability Assessment: Assessing vulnerabilities in the context of the smart contract(s) and its interaction with external components.
4. Risk Prioritization: Prioritizing identified threats based on their severity and potential impact.

Manual Code Review

Our manual analysis involves an in-depth review of the smart contract(s) source code, focusing on:

1. Code Review Line-by-line examination to detect vulnerabilities and ensure compliance with best practices.
2. Logic Analysis: Analyzing the smart contract(s) Business logic for vulnerabilities and inconsistencies.
3. Gas Optimization: Identifying areas for gas optimization and efficiency improvements.
4. Access Control Review: Ensuring proper access controls and permission management.
5. External Dependencies: Assessing the security implications of external dependencies or oracles.

Functional Testing in Hardhat/Foundry

We will perform functional testing using Hardhat/Foundry to ensure the correctness and reliability of the smart contract(s). This includes:

1. Functional Testing: Writing comprehensive tests to cover various functionalities and edge cases.
2. Integration Testing: Verifying the interaction of smart contract(s) with other components.
3. Deployment Verification: Ensuring the correctness of smart contract(s) deployment.

Fuzzing and Invariant Testing

If deemed necessary based on the complexity and criticality of the smart contract(s), we will perform fuzzing and invariant testing to identify vulnerabilities that might not be caught through conventional methods. This includes:

1. Fuzz Testing: Employing fuzzing techniques to generate invalid, unexpected, or random inputs to trigger potential vulnerabilities.
2. Invariant Testing: Verifying invariants and properties to ensure the correctness and consistency of the smart contract(s) across various scenarios.

Edge Cases Scenarios Coverage

Our audit will thoroughly cover a wide spectrum of edge cases, including but not limited to:

1. Extreme Inputs: Testing with extreme and boundary inputs to assess resilience.
2. Exception Handling: Evaluating how the contract(s) handle exceptional scenarios.
3. Concurrency: Assessing the contract(s) behaviour in concurrent or simultaneous interactions.
4. Non-Standard Scenarios: Analyzing non-standard use cases that might impact contract(s) behaviour.

Reporting and Recommendations

A thorough description of the issue, highlighting the potential impact on the system.

1. The location within the codebase where the issue is found.
2. A clear explanation of the vulnerability, its root cause, and its potential exploitation.
3. Code snippets or detailed instructions on how to address the vulnerability.
4. Best practices and coding guidelines to prevent similar issues in the future.
5. We will suggest improvements in the overall system architecture or design, if relevant.
6. Wherever applicable, we'll include a PoC to demonstrate issue severity, aiding effective mitigation.

Report Generation

1. Document all findings, including identified vulnerabilities, their severity, and potential impact.
2. Provide clear and actionable recommendations for addressing security issues.

Remediation Support

1. Collaborate with the project's development team to address and remediate identified vulnerabilities.
2. Review and validate code changes and security fixes.

Final Assessment

Re-evaluate the project's security posture after remediation efforts to ensure vulnerabilities have been adequately addressed.

In-scope

- SimplifiedAttestationCenter.sol

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved
Critical	-	-	-	-	-
High	-	-	-	-	-
Medium	3	-	-	-	3
Low	1	-	-	-	1
Info	1	-	-	-	1
Total	5	0	0	0	5

#	Findings	Severity	Status
1	Deauthorized Attesters Can Still Revoke Their Attestations	Medium	Resolved
2	Permanent DoS on Reissued Attestations After Revocation	Medium	Resolved
3	Signature Has No Expiry (Indefinite Replay Window)	Medium	Resolved
4	Hash Collision via abi.encodePacked with Dynamic bytes	Low	Resolved
5	Ownership Renunciation Permanently Bricks Contract Admin Functions	Info	Resolved

Finding 1: Deauthorized Attesters Can Still Revoke Their Attestations

Severity: Medium

Status: Resolved

Description:

The `revokeAttestation()` function in `SimplifiedAttestationCenter.sol` does not verify whether the calling attester is still authorized before allowing them to revoke their attestations. While the `createAttestation()` function properly enforces the `onlyAuthorizedAttester` modifier, the `revokeAttestation()` function only checks if the caller is the original attester or the owner, without validating the attester's current authorization status.

This inconsistency creates a security gap where a deauthorized attester (potentially removed due to misbehavior or compromised credentials) retains the ability to revoke all of their previously created attestations.

Impact:

1. **Malicious Revocation:** A deauthorized attester (removed for misbehavior or security concerns) could maliciously revoke all their historical attestations as retaliation, causing disruption to dependent systems.
2. **System Integrity:** Applications and services relying on attestations remaining valid (until explicitly revoked by authorized parties) may experience unexpected failures or trust issues.
3. **Access Control Bypass:** The authorization mechanism is effectively bypassed for the revocation operation, contradicting the intended access control model where deauthorized attesters should lose all attestation-related privileges.
4. **Mass Revocation Attack:** An attacker who gains temporary access to an attester's private key could create multiple attestations, then after being detected and deauthorized, still perform a mass revocation attack.

Source:

- **File:** `contracts/attestation-contract/src/SimplifiedAttestationCenter.sol`
- **Lines:** 116-134

Code:

```
function revokeAttestation(uint256 attestationId) external nonReentrant {
    Attestation storage attestation = attestations[attestationId];

    if (attestation.attestationId == 0) {
        revert AttestationNotFound();
    }

```

```
        if (!attestation.isValid) {
            revert AttestationAlreadyRevoked();
        }

        // Only the original attester or owner can revoke
        // @audit Missing check: authorizedAttesters[msg.sender] is not verified
        if (msg.sender != attestation.attester && msg.sender != owner()) {
            revert OnlyAttesterCanRevoke();
        }

        attestation.isValid = false;
        emit AttestationRevoked(attestationId, msg.sender);
    }
```

Proof of Concept:

Add the following test case to test/SimplifiedAttestationCenter.t.sol

```
    /**

     * @dev POC: Deauthorized attesters can still revoke their attestations
     *

     * This test demonstrates that a deauthorized attester retains the ability
     * to revoke their past attestations, which could be exploited maliciously.
     *

     * Attack scenario:
     * 1. Attester is authorized and creates an attestation
     * 2. Owner deauthorizes the attester (perhaps due to misbehavior)
     * 3. Deauthorized attester can still revoke their past attestations
     * 4. This could disrupt systems relying on those attestations
     */
    function test_POC_DeauthorizedAttesterCanRevokeAttestation() public {
        // Step 1: Attester creates an attestation while authorized
        SimplifiedAttestationCenter.AttestationRequest memory request = SimplifiedAttestationCenter.
    AttestationRequest({
            subject: subject1,
            data: data1,
            signature: validSignature1
        });

        vm.prank(attester1);
        uint256 attestationId = attestationCenter.createAttestation(request);

        // Verify attestation was created and is valid
        assertTrue(attestationCenter.isValidAttestation(attestationId), "Attestation should be valid
    after creation");
        assertTrue(attestationCenter.authorizedAttesters(attester1), "Attester1 should be authorized")
    ;

        // Step 2: Owner deauthorizes the attester
        vm.prank(owner);
        attestationCenter.setAttesterAuthorization(attester1, false);

        // Verify attester is no longer authorized
        assertFalse(attestationCenter.authorizedAttesters(attester1), "Attester1 should be
    deauthorized");

        // Step 3: Verify deauthorized attester cannot create new attestations
        bytes32 newSubject = keccak256("new subject");
        bytes memory newData = "new data";
        bytes memory newSignature = _generateSignature(attester1PrivateKey, attester1, newSubject,
    newData);

        SimplifiedAttestationCenter.AttestationRequest memory newRequest = SimplifiedAttestationCenter
    .AttestationRequest({
            subject: newSubject,
```

```
            data: newData,
            signature: newSignature
        });

        vm.prank(attester1);
        vm.expectRevert(SimplifiedAttestationCenter.UnauthorizedAttester.selector);
        attestationCenter.createAttestation(newRequest);

        // Step 4: VULNERABILITY - Deauthorized attester can still revoke their past attestations!
        vm.prank(attester1);
        attestationCenter.revokeAttestation(attestationId); // This should fail but doesn't

        // Verify the attestation was revoked by the deauthorized attester
        assertFalse(attestationCenter.isValidAttestation(attestationId), "Attestation was revoked by deauthorized attester");
    }
```

Run the PoC with:

```
forge test --match-test test_POC_DeauthorizedAttesterCanRevokeAttestation -vvv
```

Remediation:

Add an authorization check in the `revokeAttestation()` function to ensure that only currently authorized attesters (or the owner) can revoke attestations:

```
function revokeAttestation(uint256 attestationId) external nonReentrant {
    Attestation storage attestation = attestations[attestationId];

    if (attestation.attestationId == 0) {
        revert AttestationNotFound();
    }

    if (!attestation.isValid) {
        revert AttestationAlreadyRevoked();
    }

    // Only the original attester (if still authorized) or owner can revoke
    bool isAuthorizedAttester = msg.sender == attestation.attester && authorizedAttesters[msg.sender];
    if (!isAuthorizedAttester && msg.sender != owner()) {
        revert OnlyAttesterCanRevoke();
    }

    attestation.isValid = false;
    emit AttestationRevoked(attestationId, msg.sender);
}
```

Note: The owner should retain the ability to revoke attestations regardless of the attester's authorization status for administrative control.

Finding 2: Permanent DoS on Reissued Attestations After Revocation

Severity: Medium

Status: Resolved

Description:

The `SimplifiedAttestationCenter` contract has a critical availability vulnerability where the duplicate prevention mechanism (`attestationExists[messageHash]`) is set to `true` when an attestation is created but is **never cleared** when the attestation is revoked. This means that once a specific `(attester, subject, data)` tuple has been created and revoked, it can **never be recreated** - even by the legitimate attester.

In `createAttestation()` :

```
// Check for duplicate attestation
if (attestationExists[messageHash]) {
    revert AttestationAlreadyExists();
}
// ... later ...
attestationExists[messageHash] = true;  // Set but NEVER cleared
```

In `revokeAttestation()` :

```
attestation.isValid = false;  // Only flips validity flag
emit AttestationRevoked(attestationId, msg.sender);
// NOTE: attestationExists[messageHash] is NOT cleared!
```

This creates a permanent denial-of-service for any attestation tuple that has ever been created, regardless of whether it was later revoked.

Impact:

1. **No Recovery After Revocation:** Once an attestation is revoked (for any reason - mistake, policy change, or other legitimate reason), the same attestation can never be reissued. There is no way to restore the correct state.

2. **Permanent Bricking of Attestation Tuples:** For deterministic payloads such as:

    - KYC proof hashes
    - Certificate fingerprints
    - Document hashes
    - Identity verification proofs

    There is no way to “tweak” the data without changing its meaning, so the attestation is permanently bricked.

3. **Operational Risk:** Simple operator mistakes (wrong data, premature attestation) cannot be corrected - the attestation tuple is permanently unusable.

Source:

- **File:** `contracts/attestation-contract/src/SimplifiedAttestationCenter.sol`
- **Functions:**
  - `createAttestation()` (lines 81-113) - sets `attestationExists[messageHash] = true`
  - `revokeAttestation()` (lines 118-134) - does NOT clear `attestationExists`

Code:

```
// Current vulnerable pattern
function revokeAttestation(uint256 attestationId) external nonReentrant {
    Attestation storage attestation = attestations[attestationId];

    if (attestation.attestationId == 0) {
        revert AttestationNotFound();
    }

    if (!attestation.isValid) {
        revert AttestationAlreadyRevoked();
    }

    if (msg.sender != attestation.attester && msg.sender != owner()) {
        revert OnlyAttesterCanRevoke();
    }

    attestation.isValid = false;
    // BUG: attestationExists[hash] is NOT cleared!
    emit AttestationRevoked(attestationId, msg.sender);
}
```

Proof of Concept:

Add the following test case to test/SimplifiedAttestationCenter.t.sol:

```
    /**

     * @dev POC: Permanent DoS on reissued attestations after revocation
     *

     * This test demonstrates that once an attestation is created and then revoked,
     * the same (attester, subject, data) tuple can NEVER be reissued.
     *

     * The `attestationExists[messageHash]` mapping is set to true during creation
     * but is NEVER cleared during revocation. This creates a permanent denial of
     * service for that specific attestation tuple.
     *

     * Impact: For deterministic payloads (KYC proofs, certificate hashes, document
     * fingerprints), a single mistake or temporary compromise permanently bricks
     * that attestation definition with no recovery path.
     */
    function test_POC_PermanentDoSOnRevokedAttestations() public {
        // Step 1: Attester creates an attestation for a KYC proof
        bytes32 kycSubject = keccak256("user_kyc_verification");
```

```
        bytes memory kycData = "KYC_PROOF_HASH_12345";  // Deterministic proof data
        bytes memory signature = _generateSignature(attester1PrivateKey, attester1, kycSubject, kycData);

        SimplifiedAttestationCenter.AttestationRequest memory request = SimplifiedAttestationCenter.AttestationRequest({
            subject: kycSubject,
            data: kycData,
            signature: signature
        });

        vm.prank(attester1);
        uint256 attestationId = attestationCenter.createAttestation(request);

        assertTrue(attestationCenter.isValidAttestation(attestationId), "Attestation should be valid");
        console.log("Step 1: Created attestation ID:", attestationId);

        // Step 2: The attestation is revoked (due to error, key compromise, or legitimate reason)
        vm.prank(attester1);
        attestationCenter.revokeAttestation(attestationId);

        assertFalse(attestationCenter.isValidAttestation(attestationId), "Attestation should be revoked");
        console.log("Step 2: Attestation revoked");

        // Step 3: Attester tries to re-issue the SAME attestation (same subject, same data)
        // This is a legitimate use case - maybe the original was revoked by mistake,
        // or after a key rotation the attester wants to re-establish the attestation

        // Generate a fresh signature for the same data
        bytes memory newSignature = _generateSignature(attester1PrivateKey, attester1, kycSubject, kycData);

        SimplifiedAttestationCenter.AttestationRequest memory reissueRequest = SimplifiedAttestationCenter.AttestationRequest({
            subject: kycSubject,
            data: kycData,
            signature: newSignature
        });

        // Step 4: VULNERABILITY - The reissue FAILS permanently!
        vm.prank(attester1);
        vm.expectRevert(SimplifiedAttestationCenter.AttestationAlreadyExists.selector);
        attestationCenter.createAttestation(reissueRequest);

        console.log("Step 3: VULNERABILITY - Cannot reissue attestation after revocation!");
        console.log("The attestation tuple is permanently bricked with no recovery path.");
        console.log("For deterministic data like KYC proofs, this is a critical availability issue.");
    }
```

Run the PoC with:

```
forge test --match-test test_POC_PermanentDoS -vvv
```

Remediation:

Option 1: Clear `attestationExists` on revocation

Track the message hash per attestation ID and clear it during revocation:

```
// Add new state variable
mapping(uint256 => bytes32) public attestationHashById;

```

```
// In createAttestation(), after creating the attestation:
attestationHashById[attestationId] = messageHash;
attestationExists[messageHash] = true;

// In revokeAttestation(), clear the duplicate guard:
bytes32 hashToRemove = attestationHashById[attestationId];
attestationExists[hashToRemove] = false;
attestation.isValid = false;
```

Option 2: Check validity in duplicate detection

Modify the duplicate check to only block if the existing attestation is still valid:

```
// Add mapping to track which attestation ID owns a hash
mapping(bytes32 => uint256) public hashToAttestationId;

// In createAttestation():
if (attestationExists[messageHash]) {
    uint256 existingId = hashToAttestationId[messageHash];
    if (attestations[existingId].isValid) {
        revert AttestationAlreadyExists();
    }
    // If revoked, allow recreation
}

// Store the mapping
hashToAttestationId[messageHash] = attestationId;
attestationExists[messageHash] = true;
```

Option 3: Use attestation ID tracking (recommended)

```
// Replace attestationExists with a mapping that tracks the attestation ID
mapping(bytes32 => uint256) public attestationIdByHash;

// In createAttestation():
uint256 existingId = attestationIdByHash[messageHash];
if (existingId != 0 && attestations[existingId].isValid) {
    revert AttestationAlreadyExists();
}
// ... create attestation ...
attestationIdByHash[messageHash] = attestationId;
```

This approach:

- Allows recreation after revocation
- Still prevents duplicate active attestations
- Maintains gas efficiency

Finding 3: Signature Has No Expiry (Indefinite Replay Window)

Severity: Medium

Status: Resolved

Description:

The `_getAttestationHash()` function generates a signature hash that does not include any time-bound parameters such as a deadline, timestamp, or nonce. This means that once an attester signs an attestation request, that signature remains valid indefinitely as long as:

1. The attester remains authorized
2. The exact same attestation hasn't been submitted before (due to `attestationExists` check)

The signature hash only includes: `attester`, `subject`, `data`, `block.chainid`, and `address(this)`.

```
function _getAttestationHash(
    address attester,
    bytes32 subject,
    bytes memory data
) internal view returns (bytes32) {
    return keccak256(abi.encodePacked(attester, subject, data, block.chainid, address(this)));
}
```

This creates a security gap where old signatures can be used at any point in the future, creating an indefinite replay window.

Impact:

1. **Key Compromise Recovery is Difficult:** If an attester's private key is compromised, all previously signed (but not yet submitted) attestation requests can still be used by the attacker. Even if the attester is deauthorized after discovering the compromise, any signatures created while they were authorized remain valid if they get re-authorized or if the attacker acts quickly.
2. **Attester Cannot Invalidate Pending Signatures:** If an attester signs an attestation request but later changes their mind before submitting it, they have no way to invalidate that signature. The only options are:
    - Submit it anyway (wastes gas, creates unwanted attestation)
    - Hope no one else has access to the signature
    - Request deauthorization (extreme measure)
3. **Stale Signatures:** Signatures created for time-sensitive attestations remain valid forever, even when the context or circumstances that warranted the attestation have changed.

Note: The `attestationExists` mapping prevents replay of the exact same attestation, which provides some protection. However, this doesn't help when a signature was created but never submitted.

Source:

- **File:** `contracts/attestation-contract/src/SimplifiedAttestationCenter.sol`
- **Functions:** `_getAttestationHash()` (lines 147-154) and `createAttestation()` (lines 81-113)

Code:

```
// Current vulnerable code - no deadline in hash
function _getAttestationHash(
    address attester,
    bytes32 subject,
    bytes memory data
) internal view returns (bytes32) {
    return keccak256(abi.encodePacked(attester, subject, data, block.chainid, address(this)));
}
```

Proof of Concept:

Add the following test case to test/SimplifiedAttestationCenter.t.sol

```
    /**

     * @dev POC: Signatures have no expiry - indefinite replay window
     *

     * This test demonstrates that signatures do not include a deadline/expiry,
     * meaning they remain valid forever. This creates security concerns:
     *

     * 1. Key Compromise: If an attester's key is compromised, old signed (but unsubmitted)
     *    attestations can still be used by the attacker
     * 2. No Invalidation: Attesters cannot invalidate signatures they've created
     *    but haven't submitted yet
     *

     * Note: The `attestationExists` check prevents replay of the SAME attestation,
     * but doesn't help if the signature was never submitted in the first place.
     */
    function test_POC_SignatureHasNoExpiry() public {
        // Step 1: Generate a signature at time T=0
        uint256 initialTime = block.timestamp;
        bytes memory signature = _generateSignature(attester1PrivateKey, attester1, subject1, data1);

        // Step 2: Fast forward 365 days into the future
        vm.warp(initialTime + 365 days);

        // Step 3: The old signature is still valid after 1 year!
        SimplifiedAttestationCenter.AttestationRequest memory request = SimplifiedAttestationCenter.
    AttestationRequest({
            subject: subject1,
            data: data1,
            signature: signature  // Using signature created 365 days ago
        });

        vm.prank(attester1);
        uint256 attestationId = attestationCenter.createAttestation(request);

        // Verify the attestation was created successfully with the old signature
```

```
38          assertTrue(attestationCenter.isValidAttestation(attestationId), "Old signature still valid
        after 365 days");
39
40          // The attestation timestamp shows current time, but signature was created much earlier
41          SimplifiedAttestationCenter.Attestation memory attestation = attestationCenter.getAttestation(
        attestationId);
42          assertEq(attestation.timestamp, initialTime + 365 days, "Attestation created with year-old
        signature");
43      }
```

Run the PoC with:

```
1  forge test --match-test "test_POC_SignatureHasNoExpiry" -vvv
```

Remediation:

Add a deadline parameter to the `AttestationRequest` struct and include it in the signature hash:

```
1  struct AttestationRequest {
2      bytes32 subject;
3      bytes data;
4      uint256 deadline;  // Add expiry timestamp
5      bytes signature;
6  }
7
8  function _getAttestationHash(
9      address attester,
10     bytes32 subject,
11     bytes memory data,
12     uint256 deadline
13 ) internal view returns (bytes32) {
14     return keccak256(abi.encodePacked(attester, subject, data, deadline, block.chainid, address(this))
       );
15 }
16
17 function createAttestation(AttestationRequest calldata request)
18     external
19     onlyAuthorizedAttester
20     nonReentrant
21     returns (uint256 attestationId)
22 {
23     // Check signature hasn't expired
24     if (block.timestamp > request.deadline) {
25         revert SignatureExpired();
26     }
27
28     // Verify signature with deadline included
29     bytes32 messageHash = _getAttestationHash(msg.sender, request.subject, request.data, request.
       deadline);
30     if (!_verifySignature(messageHash, request.signature, msg.sender)) {
31         revert InvalidSignature();
32     }
33
34     // ... rest of the function
35 }
```

Additionally, consider adding a new error type:

```
1  error SignatureExpired();
```

This ensures that:

1. Signatures have a bounded validity period
2. Attesters can control how long their signatures remain valid
3. Old signatures automatically become unusable after the deadline

Finding 4: Hash Collision via abi.encodePacked with Dynamic bytes

Severity: Low

Status: Resolved

Description:

The `_getAttestationHash()` function uses `abi.encodePacked` to create a hash from multiple parameters, including a dynamic `bytes memory data` field. Using `abi.encodePacked` with dynamic types is a known security anti-pattern because it does not include length prefixes for dynamic data, creating ambiguous encodings that can lead to hash collisions.

```
function _getAttestationHash(
    address attester,
    bytes32 subject,
    bytes memory data  // <-- Dynamic bytes without length prefix
) internal view returns (bytes32) {
    return keccak256(abi.encodePacked(attester, subject, data, block.chainid, address(this)));
}
```

The issue is that `abi.encodePacked` concatenates values directly without any delimiters or length information. This means the boundary between the end of `data` and the beginning of `block.chainid` is ambiguous in the resulting byte sequence.

Impact:

1. **Hash Collision Potential:** Two different combinations of `(data, chainid, address)` could theoretically produce identical packed byte sequences, resulting in the same hash.
2. **Duplicate Detection Bypass:** If a collision is achieved, the `attestationExists` check could be bypassed, allowing an attacker to create attestations that appear to be duplicates but with different actual data.
3. **Cross-Chain Concerns:** While `block.chainid` and `address(this)` are included for replay protection, the ambiguous encoding could theoretically be exploited in cross-chain scenarios where an attacker crafts `data` to absorb chainid/address bytes.

Practical Limitations:

- The attacker cannot control `block.chainid` or `address(this)` at runtime
- Creating an actual collision requires specific byte patterns that align with valid chainid values
- The signature verification provides an additional layer of protection since the attacker would need to sign the colliding message

Despite limited practical exploitability, this is a well-known Solidity security anti-pattern that should be fixed as a defense-in-depth measure.

Source:

- **File:** `contracts/attestation-contract/src/SimplifiedAttestationCenter.sol`
- **Function:** `_getAttestationHash()` (lines 147-154)

Code:

```
// Current vulnerable code
function _getAttestationHash(
    address attester,
    bytes32 subject,
    bytes memory data
) internal view returns (bytes32) {
    return keccak256(abi.encodePacked(attester, subject, data, block.chainid, address(this)));
}
```

Proof of Concept:

Add the following test case to test/SimplifiedAttestationCenter.t.sol:

```
    /**

     * @dev POC: abi.encodePacked hash collision vulnerability
     *

     * This test demonstrates that using abi.encodePacked with dynamic bytes
     * can lead to hash collisions. The issue is that abi.encodePacked does not
     * include length prefixes for dynamic types, making the encoding ambiguous.
     *

     * While a full exploit requires specific conditions (matching chainid/address bytes),
     * this demonstrates the underlying vulnerability pattern.
     */
    function test_POC_AbiEncodePackedCollisionRisk() public {
        // Demonstrate that abi.encodePacked creates ambiguous encodings
        // when dynamic bytes are involved

        address testAttester = address(0x1234567890123456789012345678901234567890);
        bytes32 testSubject = bytes32(uint256(0x1111));

        // Two different data values that, when packed with subsequent values,
        // could create identical byte sequences in specific scenarios

        // Example: data1 ends with bytes that could be confused with chainid prefix
        bytes memory dataA = hex"AABBCC";
        bytes memory dataB = hex"AABB";  // Shorter data

        // Compute hashes using the same logic as _getAttestationHash
        // Note: In a real attack, chainid and address would need to align

        bytes memory encodedA = abi.encodePacked(testAttester, testSubject, dataA, block.chainid,
    address(attestationCenter));
        bytes memory encodedB = abi.encodePacked(testAttester, testSubject, dataB, block.chainid,
    address(attestationCenter));

        // These are different lengths, so no collision here
        // But the VULNERABILITY is that there's no length delimiter for 'data'

```

```
37          // Demonstrate the encoding ambiguity with a clearer example:
38          // When data is concatenated without length, boundary is ambiguous
39
40          // Craft data that includes bytes matching start of chainid
41          uint256 currentChainId = block.chainid;
42
43          // Log the issue for demonstration
44          console.log("Chain ID:", currentChainId);
45          console.log("Encoded A length:", encodedA.length);
46          console.log("Encoded B length:", encodedB.length);
47
48          // The core issue: abi.encodePacked doesn't encode data length
49          // Compare with abi.encode which DOES include length
50          bytes memory safeEncodedA = abi.encode(testAttester, testSubject, dataA, block.chainid,
        address(attestationCenter));
51          bytes memory safeEncodedB = abi.encode(testAttester, testSubject, dataB, block.chainid,
        address(attestationCenter));
52
53          // abi.encode includes length prefix, making it unambiguous
54          console.log("Safe Encoded A length:", safeEncodedA.length);
55          console.log("Safe Encoded B length:", safeEncodedB.length);
56
57          // Demonstrate that different data with same total byte representation could collide
58          // This is a THEORETICAL vulnerability - practical exploitation requires specific chainid/
        address values
59
60          // The fix is simple: use abi.encode instead of abi.encodePacked
61          assertTrue(true, "Vulnerability pattern demonstrated - use abi.encode for safety");
62      }
```

Run the PoC with:

```
1  forge test --match-test test_POC_AbiEncodePacked -vvv
```

Remediation:

Replace `abi.encodePacked` with `abi.encode` in the `_getAttestationHash()` function:

```
1  function _getAttestationHash(
2      address attester,
3      bytes32 subject,
4      bytes memory data
5  ) internal view returns (bytes32) {
6      return keccak256(abi.encode(attester, subject, data, block.chainid, address(this)));
7  }
```

`abi.encode` properly pads all values to 32 bytes and includes a length prefix for dynamic types, eliminating any ambiguity in the encoding and preventing hash collisions.

Note: This change will affect the hash computation, so any existing off-chain signature generation code must also be updated to use `abi.encode` instead of `abi.encodePacked`.

Finding 5: Ownership Renunciation Permanently Bricks Contract Admin Functions

Severity: Info

Status: Resolved

Description:

The `SimplifiedAttestationCenter` contract inherits OpenZeppelin's `Ownable` contract but does **not override** the `renounceOwnership()` function. This allows the owner to accidentally renounce ownership, setting `owner()` to `address(0)` and **permanently disabling** all administrative functions with no recovery path.

Note: Since the admin is trusted, this is a info severity issue - it represents a potential footgun rather than an exploitable vulnerability.

From OpenZeppelin's `Ownable.sol`:

```
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}
```

Once `renounceOwnership()` is called:

1. `owner()` returns `address(0)`
2. All functions with `onlyOwner` modifier become permanently inaccessible
3. There is no way to restore ownership

This is a critical integration bug because the contract relies heavily on the owner for essential administrative operations that cannot be recovered once lost.

Impact:

If the trusted owner accidentally calls `renounceOwnership()`, the following capabilities are **permanently lost**:

1. **Cannot Authorize New Attesters:** The protocol cannot onboard any new attesters, limiting growth and adaptability.
2. **Cannot Deauthorize Compromised Attesters:** If an attester's key is compromised or they become malicious, there is **no way** to remove their authorization. They can continue creating attestations indefinitely.
3. **Cannot Batch Manage Attesters:** All batch operations for attester management are permanently disabled.

4. **Owner Cannot Revoke Malicious Attestations:** The `revokeAttestation()` function allows `owner()` to revoke any attestation:

```
if (msg.sender != attestation.attester && msg.sender != owner()) {
    revert OnlyAttesterCanRevoke();
}
```

   When `owner()` is `address(0)`, this check becomes:

```
if (msg.sender != attestation.attester && msg.sender != address(0))
```

   Since `msg.sender` can never be `address(0)`, the owner pathway is completely closed.

5. **Protocol Governance Permanently Frozen:** The contract cannot evolve or respond to security incidents.

Risk: Accidental call to `renounceOwnership()` by a trusted admin with no recovery path.

Source:

- **File:** `contracts/attestation-contract/src/SimplifiedAttestationCenter.sol`
- **Inheritance:** Line 14 - `contract SimplifiedAttestationCenter is Ownable, ReentrancyGuard`
- **Affected Functions:**
  - `setAttesterAuthorization()` (line 189) - `onlyOwner`
  - `batchSetAttesterAuthorization()` (line 199) - `onlyOwner`
  - `revokeAttestation()` (line 128) - owner pathway in authorization check

Code:

```
// Current vulnerable pattern - Ownable inherited without renounceOwnership override
contract SimplifiedAttestationCenter is Ownable, ReentrancyGuard {
    // ...
    // renounceOwnership() is available and can be called by owner
}
```

Remediation:

Option 1: Override and disable `renounceOwnership()` (Recommended)

```
/**

 * @dev Disables the ability to renounce ownership to prevent bricking the contract
 */
function renounceOwnership() public pure override {
    revert("SimplifiedAttestationCenter: ownership renunciation disabled");
}
```

Option 2: Use OpenZeppelin's `Ownable2Step`

Replace `Ownable` with `Ownable2Step` which requires the new owner to accept ownership, preventing accidental transfers to invalid addresses. Still override `renounceOwnership()` to disable it:

```
import "@openzeppelin/contracts/access/Ownable2Step.sol";

contract SimplifiedAttestationCenter is Ownable2Step, ReentrancyGuard {
    // ...

    function renounceOwnership() public pure override {
        revert("SimplifiedAttestationCenter: ownership renunciation disabled");
    }
}
```

Option 3: Add a recovery mechanism

If ownership renunciation should be allowed for specific scenarios, implement a time-locked recovery mechanism or multi-sig requirement.

Disclaimer

This security report (“Report”) is provided by FailSafe (“Tester”) for the exclusive use of the client (“Client”). The scope of this assessment is limited to the security testing services performed against the systems, applications, or environments supplied by the Client. This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer, and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This Report, provided in connection with the Services set forth in the Agreement, shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This Report may not be transmitted, disclosed, referred to, or relied upon by any person for any purpose, nor may copies be delivered to any other person other than the Company, without FailSafe’s prior written consent in each instance.

This Report is not, nor should it be considered, an “endorsement” or “disapproval” of any particular project, system, or team. This Report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts FailSafe to perform security testing. This Report does not provide any warranty or guarantee regarding the absolute security or bug-free nature of the technology analyzed, nor does it provide any indication of the technology’s proprietors, business, business model, or legal compliance.

This Report should not be used in any way to make decisions around investment or involvement with any particular project. This Report in no way provides investment advice, nor should it be leveraged as investment advice of any sort. This Report represents an extensive testing process intended to help our customers identify potential security weaknesses while reducing the risks associated with complex systems and emerging technologies.

Technology systems, applications, and cryptographic assets present a high level of ongoing risk. FailSafe’s position is that each company and individual are responsible for their own due diligence and continuous security practices. FailSafe’s goal is to help reduce attack vectors and the high level of variance associated with utilizing new and evolving technologies, and in no way claims any guarantee of security or functionality of the systems we agree to test.

The security testing services provided by FailSafe are subject to dependencies and are under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. The testing process may include false positives, false negatives, and other unpredictable results. The services may access and depend upon multiple layers of third-party technologies.

ALL SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, FAILSAFE HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RE-

SPECT TO THE SERVICES, TESTING REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, FAILSAFE SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, FAILSAFE MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE.

WITHOUT LIMITATION TO THE FOREGOING, FAILSAFE PROVIDES NO WARRANTY OR DISCLAIMER UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER FAILSAFE NOR ANY OF FAILSAFE'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. FAILSAFE WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, TESTING REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, TESTING REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO ANY OTHER PERSON WITHOUT FAILSAFE'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF FAILSAFE CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED TESTING REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.