Zellic

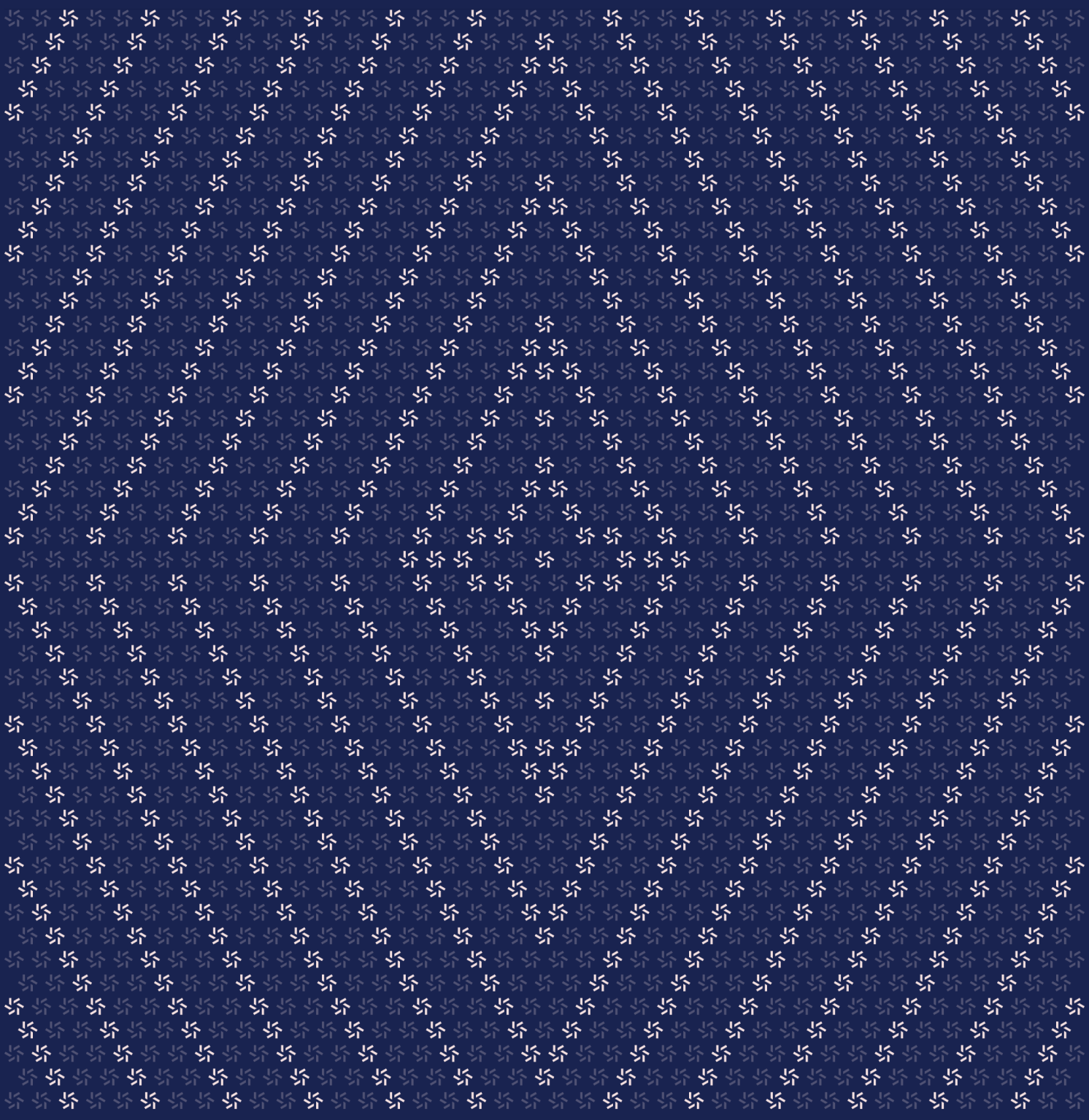
**Prepared for**
Inder Singh
D3

**Prepared by**
Syed Faraz Abrar
Hojung Han
Rainier Wu
Zellic

**June 19, 2025**

# d3-doma

## Smart Contract Security Assessment



# Contents

## About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

# 1. Overview

## 1.1. Executive Summary

Zellic conducted a security assessment for D3 from June 2nd to June 11th, 2025. During this engagement, Zellic reviewed d3-doma's code for security vulnerabilities, design issues, and general weaknesses in security posture.

## 1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could message delivery failures and race conditions during cross-chain domain operations lead to loss or theft of domain name token ownership?
- Could users be charged excessive amounts or have their domain name tokens stolen during secondary sales through the Marketplace?
- Are storage layouts properly managed to prevent storage-layout collisions?

## 1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. In this engagement, we had adequate time to review the in-scope code; however, we did not have enough time to write our own integrated tests to ensure that all cross-chain functionality works as intended.

Although we found a decent amount of issues related to cross-chain functionality, we recommend writing a more comprehensive set of integration tests that test all cross-chain functionality exhaustively. See the finding in section 3.7. ↗ for more information.

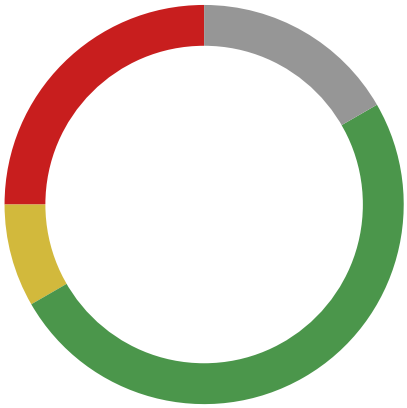
## 1.4. Results

During our assessment on the scoped d3-doma contracts, we discovered 12 findings. Three critical issues were found. One was of medium impact, six were of low impact, and the remaining findings

were informational in nature.

### Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| Critical | 3 |
| High | 0 |
| Medium | 1 |
| Low | 6 |
| Informational | 2 |



# 2. Introduction

## 2.1. About d3-doma

D3 contributed the following description of d3-doma:

> Doma Protocol provides a suite of APIs and smart contracts that enable ICANN Registrars and Registries to tokenize and detokenize domains on the blockchain. When a domain is tokenized, a Domain Ownership Token is minted on the blockchain selected by the Registrant, while Doma blockchain maintains the authoritative record of all tokenized domains. Through signature verification, Doma ensures that domains can only be tokenized only by the hosting Registrar, preventing unauthorized tokenizations.

## 2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

**Basic coding mistakes.** Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

**Business logic errors.** Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

**Integration risks.** Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

**Code maturity.** We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

## 2.3. Scope

The engagement involved a review of the following targets:

### d3-doma Contracts

| | |
|---|---|
| **Type** | Solidity |
| **Platform** | EVM-compatible |
| **Target** | doma-contracts |
| **Repository** | https://github.com/d3-inc/doma-contracts ↗ |
| **Version** | `4ad9a812e818f33112e24a9eb8db836d430e8112` |
| **Programs** | `contracts/*.sol` |

## 2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2 person-weeks. The assessment was conducted by three consultants over the course of 7 calendar days.

### Contact Information

The following project managers were associated with the engagement:

**Jacob Goreski**
Engagement Manager
jacob@zellic.io

**Chad McDonald**
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

**Syed Faraz Abrar**
Engineer
faith@zellic.io

**Hojung Han**
Engineer
hojung@zellic.io

**Rainier Wu**
Engineer
rainier@zellic.io

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **June 2, 2025** | Start of primary review period |
| **June 3, 2025** | Kick-off call |
| **June 11, 2025** | End of primary review period |

# 3. Detailed Findings

## 3.1. Bridging can be used to deliberately block detokenization

| Target | ProxyDomaRecord, DomaRecordProxyFacet, DomaRecordRegistrarFacet | | |
|---|---|---|---|
| **Category** | Business Logic | **Severity** | Critical |
| **Likelihood** | High | **Impact** | Critical |

### Description

When a registrar initiates detokenization via `DomaRecordRegistrarFacet::complianceDetokenize`, `DomaRecordRegistrarFacet::registrarDelete`, and `DomaRecordRegistrarFacet::registrarDetokenize`, the token holder may bridge the token to a remote chain before the detokenization message from the Doma chain arrives at the ProxyDomaRecord contract.

Since the call to `ProxyDomaRecord::bridge()` by the token holder will burn the token, the cross-chain call to `ProxyDomaRecord::detokenize()` from the `DomaRecordRegistrarFacet` functions listed above will revert. This will mean that the detokenization flow cannot complete, thus completely preventing the token from being detokenized.

### Impact

The user can continue to block detokenization requests by constantly bridging the token between chains as soon as they notice a cross-chain detokenization occurring. This prevents the domain from ever being tokenized, even after ownership is transferred off-chain.

This issue also affects expired domains, where detokenization may be blocked in the same way.

### Proof of Concept

Add the following test to `test/ProxyDomaRecord.test.ts` and run it using `npx hardhat test --audit grep`:

```
describe('audit', () => {
  const TARGET_CHAIN_ID = 'eip155:1';

  beforeEach(async () => {
    // Add the target chain ID to the supported list
    await
    proxyDomaRecord.connect(owner).addSupportedTargetChain(TARGET_CHAIN_ID);
  });
```

```
  it("audit - detokenization is blocked after burn from bridge", async () => {
    const { tokenId } = await mintOwnershipTokenForUser();

    // Assume that there is a detokenization request inbound from Doma chain to
    this chain.
    // The user notices this and decides to bridge their token away before the
    request arrives.
    await proxyDomaRecord.connect(user).bridge(tokenId, false,
    TARGET_CHAIN_ID, user.address);

    // Now after the bridging transaction, the detokenization request comes
    through
    const tx = proxyDomaRecord
      .connect(crossChainReceiver)
      .detokenize(BigInt(tokenId).toString(), false, user.address,
    TEST_CORRELATION_ID);

    // This detokenize request should ideally still succeed, and cross-chain
    logic must be
    // implemented to handle detokenizing the token after it arrives on the
    remote chain
    // that the user bridged to.
    // However, this transaction will revert, forcing detokenization to not
    complete.
    await expect(tx).to.not.be.reverted;
  });
});
```

It expects the detokenization to still succeed (see our recommendation below for an example of how to implement that logic here), but it will revert instead, preventing detokenization.

## Recommendations

We recommend implementing logic that prevents bridging of tokens if a detokenization request is in progress. This is quite difficult to implement since it needs to account for the fact that any such logic could be circumvented if the user can front-run such preventions and bridge the token away first anyway.

One way to solve this would be to ensure that the `ProxyDomaRecord::detokenize()` function succeeds even if the token has been burned through the `bridge()` function. Then, on the Doma chain, the `DomaRecordProxyFacet::bridge()` function can check to see if the token ID being bridged is currently being detokenized. If so, it should not complete the bridging process.

Finally, once the `DomaRecordProxyFacet::completeDetokenization()` function is called, the token details will be deleted from the `DomaRecord` contract's state, and everything will be working

as intended.

### Remediation

This issue has been acknowledged by D3, and a fix was implemented in commit 6c680667 ↗.

## 3.2. Token-transfer permanent loss

| | | | |
|---|---|---|---|
| **Target** | ProxyDomaRecord, DomaRecordProxyFacet | | |
| **Category** | Coding Mistakes | **Severity** | Critical |
| **Likelihood** | High | **Impact** | Critical |

### Description

Domain tokens can be permanently lost due to interactions between token transfer and the bridge mechanisms.

The specific scenario occurs through the following sequence:

1. Alice owns a non-EOI ownership token for `alice.xyz` on chain A.
2. Alice transfers the token to Bob, triggering the `_beforeTokenTransfer` hook in `NameToken`. This removes ownership of the token, setting the `name.claimedBy` address to the Doma Proxy address.
3. The hook makes a cross-chain call to `DomaRecordProxyFacet::tokenTransfer`, which sets `name.claimedBy` to the Doma Proxy address.
4. Bob attempts to bridge the token to chain B by calling `ProxyDomaRecord::bridge`. Particularly, he does this before claiming ownership of the token.
5. The token is burned on chain A as part of the bridge process.
6. Then, `DomaRecordProxyFacet::bridge` is called on the Doma chain.
7. The function checks `LibDoma.isClaimedByDomaProxy(name.claimedBy)` and returns early since it is true, as the name has not been claimed by anyone yet.
8. No token is minted on chain B, leaving Bob with no token while the domain state remains in the DomaRecord contract.

The result is that the token completely disappears while the system still believes the domain is tokenized, making it permanently impossible to create new tokens for that domain.

Note that this can also occur if the bridging process is interrupted due to any reason (such as, for example, a cross-chain function call reverting), as the token will always be burned on the source chain first, and any interruptions will lead to the token never being minted on any other chains.

### Impact

This issue can cause permanent loss of tokenization functionality for affected domains.

### Proof of Concept

To demonstrate this issue, we chose to modify the early `return` statement in `DomaRecordProxyFacet::bridge()` to a `revert()`, like so:

```
function bridge(
    /* ... */
) external onlyCrossChainSender {
    // [ ... ]

    // If it's an ownership token, we might need to update claimedBy status to
    a new owner
    if (nameToken.ownership) {
        // If it's in DOMA proxy, skip update
        if (LibDoma.isClaimedByDomaProxy(name.claimedBy)) {
            revert(); // <==== Zellic - modify this line
        }

        name.claimedBy = CAIP.format(targetChainId, targetOwnerAddress);
    }

    // [ ... ]
}
```

Then, we added the following test to `test/doma-record/DomaRecordProxyFacet.test.ts` and ran it with `npx hardhat test --grep audit`:

```
describe('audit', () => {
  it('audit - bridging does not complete when the token is unowned', async ()
    => {
    await utils.addRegistrar();
    await utils.createName();
    await domaRecordFacet.connect(owner).setProxyContract(DOMA_CHAIN_ID,
    PROXY_CONTRACT_ADDRESS);
    await utils.transferOwnershipTokenToAnotherUser();

    const ownershipTokenId = generateTestOwnershipTokenId();

    // This token is does not have it's ownership claimed. Bridge it
    const tx =  domaRecordProxyFacet
      .connect(crossChainSender)
      .bridge(ownershipTokenId.toString(), DOMA_CHAIN_ID, user.address,
    TEST_CORRELATION_ID);

    // The revert simulates the early return. The code expects no revert (i.e
    the bridging
```

```
    // completes all the way). However, this will revert and prevent the
    bridging process from
    // completing.
    await expect(tx).to.not.be.reverted;
  });
})
```

Running the test shows that the bridging process ends early, and thus leads to no tokens being minted on the remote chain.

### Recommendations

We recommend not allowing users to perform any critical actions using their name tokens without claiming ownership of the token first. This can be done using a modifier on most external functions on the NameToken and OwnershipToken contracts.

Additionally, we also recommend being very vigilant about ensuring that the bridging process will either always succeed completely or not succeed at all. Partial bridging will end up with unintended consequences, such as the loss of name tokens.

### Remediation

This issue has been acknowledged by D3, and a fix was implemented in commit 3dfb0670 ↗.

## 3.3. Overridden ownable functionality can lead to admin lockout

| | | | |
|---|---|---|---|
| **Target** | NameToken | | |
| **Category** | Coding Mistakes | **Severity** | Critical |
| **Likelihood** | High | **Impact** | Critical |

### Description

The NameToken contract overrides the Ownable contract's internal ownership check (`_requireCallerIsContractOwner()`) to use the AccessControl contract's `DEFAULT_ADMIN_ROLE` check instead.

However, when ownership is transferred via `Ownable::transferOwnership`, the new owner does not automatically receive the `DEFAULT_ADMIN_ROLE`.

### Impact

This will result in the new owner not being able to satisfy the role check, making owner-only functions inaccessible.

In the worst-case scenario, if the original admin renounces the `DEFAULT_ADMIN_ROLE` immediately on transferring ownership, no account will hold the `DEFAULT_ADMIN_ROLE`, and thus all protected functionality will be permanently locked forever.

We think the likelihood of this issue occurring is high, and thus the impact is Critical.

### Proof of Concept

Add the following testcase to `test/OwnershipToken.test.ts` and run it using `npx hardhat test --grep audit`:

```
describe('audit test', () => {
  it('audit - transfer ownership lockout', async () => {
    const newOwner = user;
    const newProxyDomaRecord = "0x0000000000000000000000000000000000000AA";

    // ownership is transfered to newOwner
    await ownershipTokenContract.connect(owner).transferOwnership(newOwner);

    // newOwner should be able to call onlyOwner functions
```

```
    const tx =
    ownershipTokenContract.connect(newOwner).setProxyDomaRecord(newProxyDomaRecord);
    await expect(tx).to.not.be.reverted;

    // previous owner should not be able to call onlyOwner functions
    await
    ownershipTokenContract.connect(owner).setProxyDomaRecord(newProxyDomaRecord);
    const tx2 =
    ownershipTokenContract.connect(owner).setProxyDomaRecord(newProxyDomaRecord);
    await expect(tx2).to.be.reverted;
  });
});
```

The new owner's transaction to the ownership token contract will revert, even though the ownership has already been transferred.

The test won't get past that transaction, but we've added in another transaction to ensure that after the issue is fixed, the previous owner is not still able to call a protected function on the ownership contract.

### Recommendations

Override the `Ownable::transferOwnership()` function, and ensure that the `DEFAULT_ADMIN_ROLE` is passed onto the new owner prior to the old owner renouncing their ownership. Also ensure that the `DEFAULT_ADMIN_ROLE` is removed from the old owner.

### Remediation

This issue has been acknowledged by D3, and a fix was implemented in commit `341c38ae` ↗.

## 3.4. Transfer block bypass

| | | | |
|---|---|---|---|
| **Target** | NameToken | | |
| **Category** | Business Logic | **Severity** | Medium |
| **Likelihood** | High | **Impact** | Medium |

### Description

When `blockAllTransfers` is set to true in NameToken, the intention is for all transfers to be blocked. However, even with `blockAllTransfers` being set, users can still use the bridging functionality to transfer tokens to other chains to arbitrary addresses.

This allows circumvention of transfer restrictions, as a user can bridge their token to another chain and then bridge it back to a different address, effectively performing a transfer.

The `bridge()` function does not check the transfer-restriction state, creating an inconsistency in policy enforcement.

### Impact

This issue neutralizes intended transfer-restriction policies and allows circumvention of restrictions set for regulatory or security reasons.

### Recommendations

It is recommended to check the `blockAllTransfers` flag and revert if necessary in internal cases such as `mint` or `burn` functions where the `from` or `to` address is `address(0)`.

### Remediation

This issue has been acknowledged by D3, and a fix was implemented in commit `a18a5e21` ↗.

## 3.5. Native tokens can remain stuck in some contracts forever

| | | | |
|---|---|---|---|
| **Target** | ERC7786GatewayReceiver, ERC7786GatewaySource, ProxyDomaRecord | | |
| **Category** | Coding Mistakes | **Severity** | Medium |
| **Likelihood** | Low | **Impact** | Low |

### Description

The ERC7786GatewayReceiver has an `executeMessage()` function intended to be called by trusted cross-chain senders to execute cross-chain smart contract calls. Although this function is marked `payable`, the function does not do anything with `msg.value`.

The ERC7786GatewaySource contract has a `sendMessage()` function that is intended to be called when messages are to be relayed to another chain. Although this function is marked `payable`, it does not use the `msg.value` nor does it have any method to retrieve native tokens from the contract.

The ProxyDomaRecord contract has three `payable` functions: `bridge()`, `claimOwnership()`, and `requestTokenization()`. The intention is for users to pass in a fee through `msg.value` and for this fee to be transferred to the treasury. However, it does not prevent users from attaching more than the fee in `msg.value`.

### Impact

In the ERC7786GatewayReceiver case, if the cross-chain sender accidentally attaches any `msg.value` to their transaction, the tokens will get stuck in this contract. There is no way to retrieve these tokens currently. This has a low impact.

In the ERC7786GatewaySource case, this is currently not an issue as none of the callers pass in any `msg.value`. This has an informational impact.

In the ProxyDomaRecord case, if a user accidentally passes in more fees than required by the functions, any excess fees will stay stuck in the contract. There is no way to retrieve these tokens currently. This has a low impact.

### Recommendations

We recommend adding a function to these contracts that allows withdrawing any native Ether that gets stuck.

Alternatively, either remove the `payable` modifier from the functions that do not need it, or add stricter checks on the `msg.value` so that excess tokens have no way of getting stuck.

### Remediation

This issue has been acknowledged by D3, and a fix was implemented in commit 49d8c644 ↗.

## 3.6. Registrars can overwrite domain-name details of other registrars

| | | | |
|---|---|---|---|
| **Target** | DomaRecordRegistrarFacet | | |
| **Category** | Coding Mistakes | **Severity** | Low |
| **Likelihood** | Low | **Impact** | Low |

### Description

There is missing validation logic in the `nameTokenize()` function that creates severe security implications beyond simple duplicate tokenization.

This function does not verify whether a token already exists for the domain, allowing registrars to accidentally tokenize the same name multiple times.

This issue also enables malicious cross-registrar attacks. However, the client has stated that registrars are to be considered as trusted entities, and thus we will assume that they will not deliberately perform such attacks.

When a registrar calls `nameTokenize()`, it overwrites the existing `_domaState.nameTokens` even if it already exists, meaning a registrar could accidentally or deliberately overwrite name information controlled by another registrar.

The `approveTokenization` function has the proper validation logic, implemented below:

```
LibDoma.Name storage name = _domaState.names[nameId];
if (name.registrarIanaId != 0) {
    revert NameAlreadyTokenized(sld, tld);
}
```

However, this validation is completely absent in `nameTokenize()`, creating an attack vector for malicious registrars to overwrite details of domain names from other registrars.

Note that this issue also affects the `eoiImport()` function, where it does not check that the EOI name being modified has a matching `registrarIanaId` to the calling registrar.

### Impact

In the worst case, a registrar being able to overwrite domain name details of another registrar is critical in severity. Adversarial registrars can claim each other's onchain ownership on-chain.

However, the client has stated that registrars are to be trusted to not act maliciously. Because of this, we think the likelihood of such an action occurring is extremely low, even accidentally.

### Recommendations

It is recommended to add the aforementioned validation code to the `nameTokenize()` and `eoiImport()` functions.

### Remediation

This issue has been acknowledged by D3, and a fix was implemented in commit `2e4a5b3d` ↗.

## 3.7. Add an integrated test suite for all cross-chain interactions

| | | | |
|---|---|---|---|
| **Target** | ProxyDomaRecord, DomaRecord | | |
| **Category** | Business Logic | **Severity** | Low |
| **Likelihood** | Low | **Impact** | Low |

### Description

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project includes a comprehensive set of positive and negative tests. However, coverage could be greatly improved with fully integrated tests that check all cross-chain functionality from start to finish. Adding such tests would allow issues such as Finding 3.1. ↗, Finding 3.2. ↗, and Finding 3.6. ↗ to be caught more easily, as it would allow checking for consistent states across chains, as well as ensuring that crucial cross-chain transactions do not revert.

We understand that it is difficult to write such test cases, as it requires locally running two or more separate node instances (one for the Doma chain and one or more for any remote chains, such as Polygon). However, we highly recommend doing this, as it makes it easier for developers to test any new functionality that is added to the chain and ensure that cross-chain states stay consistent across both chains.

### Impact

The reason for this finding is because we believe that the non-existence of an integrated test suite is the sole reason behind the existence of some of the issues we've found in this code base. We think it's reasonable to treat this as a finding, as it needs to be fixed so that such issues can be caught in the future by the test suite.

### Recommendations

Add an integrated test suite that tests all cross-chain interactions in an end-to-end manner.

### Remediation

This issue has been acknowledged by D3.

The D3 Doma Team's position is as follows:

All flows are covered by Doma Backend end-to-end tests (which include actual contracts on testnets). While having an end-to-end test suite at the contract level would be beneficial, the actual implementation has been postponed as it requires a significant amount of effort.

### 3.8. Registrars can pass in expiry dates from the past when renewing domains

| | | | |
|---|---|---|---|
| **Target** | DomainRecordRegistrarFacet | | |
| **Category** | Coding Mistakes | **Severity** | Medium |
| **Likelihood** | Low | **Impact** | Low |

#### Description

In the DomaRecordRegistrarFacet contract, the `renew()` function is called by registrars to renew a domain name. It requires the registrar to pass in a new `expiresAt` timestamp for the domain.

This function is missing a check to ensure that the `expiresAt` timestamp is not from the past.

#### Impact

Although the cross-chain call to `ProxyDomaRecord::renew()` will fail (as that function does validate the `expiresAt` timestamp), the `name.expiresAt` field is already updated in the `DomaRecordRegistrarFacet::renew()` function. This causes an inconsistent cross-chain state between the two contracts.

Since registrars are trusted entities, this issue could only occur by accident, which is why we set the final impact as Low.

#### Recommendations

Add validation logic for the `expiresAt` timestamp.

#### Remediation

This issue has been acknowledged by D3, and a fix was implemented in commit `65a5b48b` ↗.

## 3.9. Stale oracle price risk

| | | | |
|---|---|---|---|
| **Target** | ProxyDomaRecord | | |
| **Category** | Business Logic | **Severity** | Low |
| **Likelihood** | Low | **Impact** | Low |

### Description

This issue is related to price-data reliability. When the `ProxyDomaRecord::getNativePrice` function queries the USD price of native currency using the Chainlink oracle, validation logic to check whether the returned price data is stale is missing.

The current implementation simply retrieves the price without verifying timestamps or round IDs, creating substantial risk of using outdated price data.

### Impact

This issue may result in inaccurate fees being charged.

### Recommendations

It is recommended to add logic that verifies the timestamps of price data to resolve this issue.

### Remediation

This issue has been acknowledged by D3, and a fix was implemented in commit `039b512d`↗.

### 3.10. IANA ID modification missing

| | | | |
|---|---|---|---|
| **Target** | DomaRecordRegistrarFacetBase | | |
| **Category** | Business Logic | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

#### Description

No mechanism exists to modify the `registrarIanaId` in the `LibDoma.Name` structure once registered.

#### Impact

If a registrar's IANA ID changes or is deleted, registrar-level functions may fail to execute. This creates operational risks when IANA ID changes occur in real-world scenarios.

This issue could make existing domain management impossible when IANA IDs change.

#### Recommendations

It is recommended to implement an administrator function to update IANA IDs.

#### Remediation

This issue has been acknowledged by D3.

The D3 Doma Team's position is as follows:

> Won't implement as of now.
>
> Moving tokenized domains between registrars requires coordination between 2 registrars, and is a complex procedure, which we don't plan to support for upcoming release.
>
> Registrar IANA ID itself can only be removed if registrar goes out of business (and it cannot be changed). This is extremely unlikely, and can be handled by detokenizing + retokenizing the domains to another registrar.

### 3.11. Secondary sales in the Marketplace will revert if the fee is not set correctly

| | | | |
|---|---|---|---|
| **Target** | Marketplace | | |
| **Category** | Coding Mistakes | **Severity** | Low |
| **Likelihood** | Low | **Impact** | Low |

#### Description

In the Marketplace contract, the `_secondaryFee` state variable is intended to be within the range `[0, 10000]`.

However, the `_setSecondarySaleFee()` function does not have a check for this, which allows the fee to be set higher than the range.

This will cause the `secondarySale()` function to revert, as the following code will underflow:

```
uint256 fee = (name.price * _secondaryFee) / 10000;
totalFee += fee;
totalAmount += name.price;

uint256 sellerProfit = name.price - fee; // Zellic: underflow here
```

#### Impact

The impact is low, as this issue will only occur if the owner accidentally sets the secondary sale fee to a value higher than 10,000. Additionally, even if this does occur, it is easy to reset the fee back to a sane value.

#### Recommendations

We recommend adding checks to `_setSecondarySaleFee()` that ensure that the fee does not exceed 10,000.

A separate recommendation is to also consider a sane upper limit for the fee. For example, setting the fee to 10,000 does not make sense, because then the profit from the sale for the seller will be zero.

### Remediation

This issue has been acknowledged by D3.

The D3 Doma Team's position is as follows:

> Won't implement. `Marketplace` contract is deprecated, and is being replaced with `Seaport`. Since issue requires misconfiguration from admin side, and only results in reverts (not in funds loss), fix is not planned.

## 3.12. Fee-on-transfer tokens will behave incorrectly in the Marketplace

| | | | |
|---|---|---|---|
| **Target** | Marketplace | | |
| **Category** | Coding Mistakes | **Severity** | Medium |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

In the Marketplace contract, the `pay()` function is used to pay for a domain name that has been put on sale off chain.

This contract allows for payment either through the chain-native token or through certain whitelisted ERC-20 tokens.

If one of the whitelisted tokens happens to take out a fee on transfer, then the amount of tokens sent to the treasury will be less than expected.

### Impact

The client has stated that none of the tokens currently whitelisted contain any fee-on-transfer functionality. Thus, the impact is currently Informational.

However, we wanted to include this finding in order to ensure that care is taken when whitelisting any new tokens in the future.

### Recommendations

Before adding new tokens to the whitelist, ensure that the token does not take a fee on transfer.

If a new token does require a fee on transfer, then some new logic must be added to the `pay()` function to account for this fee.

### Remediation

This issue has been acknowledged by D3.

The D3 Doma Team's position is as follows:

> All tokens are whitelisted on a backend side. Same approach is used for new Seaport-based implementation.

# 4. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

## 4.1. Module: DomaForwarder.sol

### Function: `execute(ForwardRequestData calldata request)`

This function executes a metatransaction on behalf of a signer using the ERC-2771 protocol. This function validates the forward request, ensures proper value matching, and delegates execution to the internal `_execute` function with strict validation requirements.

#### Inputs

- `request.from`
  - **Control**: Full.
  - **Constraints**: Must match the recovered signer from the signature verification.
  - **Impact**: Prevents signature-replay attacks and ensures only the legitimate signer can execute transactions on their behalf.
- `request.to`
  - **Control**: Full.
  - **Constraints**: Target address must trust this forwarder (checked via `_isTrustedByTarget()`).
  - **Impact**: Ensures only contracts that explicitly trust this forwarder can receive forwarded calls, preventing unauthorized contract interactions.
- `request.value`
  - **Control**: Full.
  - **Constraints**: Must exactly match `msg.value` (checked via `msg.value != request.value`).
  - **Impact**: Ensures the ETH amount sent with the transaction matches the signed request, preventing value-manipulation attacks.
- `request.gas`
  - **Control**: Full.
  - **Constraints**: Validated in `_checkForwardedGas()` to ensure sufficient gas was forwarded (minimum 1/63 of requested gas).
  - **Impact**: Prevents gas-griefing attacks where relayers provide insufficient gas

to cause subcall failures.

- `request.nonce`
  - **Control**: Full.
  - **Constraints**: Must be unique and not previously used (checked via `_nonces[nonce]` mapping in `_verifyAndStoreNonce()`).
  - **Impact**: Prevents replay attacks by ensuring each signed request can only be executed once.
- `request.deadline`
  - **Control**: Full.
  - **Constraints**: Must be greater than or equal to the current block timestamp (`request.deadline >= block.timestamp`).
  - **Impact**: Prevents execution of expired requests, ensuring time-sensitive transactions cannot be delayed indefinitely.
- `request.data`
  - **Control**: Full.
  - **Constraints**: Data is hashed and included in signature verification via `keccak256(request.data)`.
  - **Impact**: Guarantees data integrity by ensuring the executed data matches exactly what was signed by the original signer.
- `request.signature`
  - **Control**: Full.
  - **Constraints**: ECDSA signature verification via `_recoverForwardRequestSigner()` to recover signer from EIP-712 typed data hash.
  - **Impact**: Ensures the request was actually signed by the claimed signer, preventing unauthorized transaction execution.

## Branches and code coverage (including function calls)

**Intended branches**

- ☑ Valid request with matching `msg.value` and `request.value` passes validation and executes successfully.
- ☑ Successful execution with trusted forwarder, valid signature, and active deadline.
- ☑ External call to target contract via `target.call{value: value, gas: reqGas}(data)` executes successfully.

**Negative behavior**

- ☐ `msg.value != request.value` reverts with `ERC2771ForwarderMismatchedValue`.
- ☐ `_execute()` returns false, causing a revert with `Errors.FailedCall()`.
- ☑ Nontrusted sender attempting to call the function reverts due to the

`onlyTrustedSender` modifier.

- ☐ Invalid signature verification fails in `_validate()` causing `ERC2771ForwarderInvalidSigner` revert.
- ☐ Expired deadline (deadline < `block.timestamp`) causes `ERC2771ForwarderExpiredRequest` revert.
- ☐ Target (`request.to`) not trusting forwarder causes `ERC2771UntrustfulTarget` revert.
- ☑ A nonce already used causes `NonceAlreadyUsed` revert.
- ☐ Insufficient gas forwarded triggers invalid opcode in `_checkForwardedGas()`.

## 4.2. Module: DomaRecordProxyFacet.sol

### Function: `bridge()`

This function is called when an ownership token is being bridged from one remote chain to another remote chain.

### Inputs

- `tokenId`
  - **Control**: Semicontrolled.
  - **Constraints**: The user performing the bridging must have this token existing in their wallet.
  - **Impact**: The token ID of the ownership token being detokenized.
- `targetChainId`
  - **Control**: Semicontrolled.
  - **Constraints**: The target chain ID must be supported.
  - **Impact**: This is the chain ID of the target chain.
- `targetOwnerAddress`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the address on the target chain that will receive the token.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Returns early if the token currently does not have its ownership claimed.
- ☑ Successfully triggers a cross-chain call to the target chain's ProxyDomaRecord contract's `mintOwnershipTokens()` function.

**Negative behavior**

- ☑ Fails on EOI name tokens.
- ☑ Fails when not called by a cross-chain sender.

## Function: `claimOwnership()`

This function is used to claim ownership of a name token.

### Inputs

- `tokenId`
  - **Control**: Semicontrolled.
  - **Constraints**: The user claiming ownership must have this token existing in their wallet.
  - **Impact**: The token ID of the ownership token being claimed.
- `chainId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is the chain ID of the remote chain.
- `claimedBy`
  - **Control**: Not controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the `msg.sender` who is claiming ownership on the remote chain.
- `proofSource`
  - **Control**: Semicontrolled.
  - **Constraints**: N/A.
  - **Impact**: This must be set according to who signed the proof-of-contacts voucher on the remote chain.
- `registrantHandle`
  - **Control**: Semicontrolled.
  - **Constraints**: This is part of the proof-of-contacts voucher, which must be

signed by a registrar or a Doma signer.
- **Impact**: This is a handle for registrant details stored somewhere off chain.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Ownership is transferred immediately when signed by a registrar.
- ☑ Ownership transfer is delayed and awaits registrar approval when signed by a Doma signer.

**Negative behavior**

- ☐ Fails on nonexistent ownership token ID.
- ☑ Fails on mismatched chain ID.
- ☑ Fails on mismatched owner address.
- ☐ Fails on non-ownership token.
- ☑ Fails when not called by a cross-chain sender.

## Function: `completeDetokenization()`

This function is called as part of the detokenization flow from a remote chain.

### Inputs

- `tokenId`
  - **Control**: Semicontrolled.
  - **Constraints**: The user claiming ownership must have this token existing in their wallet.
  - **Impact**: The token ID of the ownership token being detokenized.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Successfully deletes all information about this token from the chain state.

**Negative behavior**

- ☑ Fails when not called by a cross-chain sender.

### Function: `initiateTokenization()`

This function is used to initiate tokenization for a domain name.

### Inputs

- `registrarIanaId`
  - **Control**: Not controlled.
  - **Constraints**: Fetched from ProxyDomaRecord storage state, set by registrars themselves.
  - **Impact**: This is the registrar through which this domain was purchased.
- `names`
  - **Control**: Semicontrolled.
  - **Constraints**: The names are fetched from a voucher provided by the user, but the voucher needs to be signed by the registrar before it can be used.
  - **Impact**: These are the names for which tokenization is to be initiated.
- `ownershipTokenChainId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is the chain ID of the remote chain.
- `ownershipTokenOwnerAddress`
  - **Control**: Not controlled.
  - **Constraints**: This is the `msg.sender` that requested tokenization on the remote chain.
  - **Impact**: This is the address of the user on the remote chain who is requesting tokenization.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Successfully creates a tokenization request.
- ☑ Skips over already registered domain names.

**Negative behavior**

- ☑ Fails on invalid TLD.
- ☑ Fails on invalid SLD.
- ☑ Fails when not called by a cross-chain sender.

### Function: `ownerDetokenize()`

This function is called by a token owner to detokenize their domain.

### Inputs

- `tokenId`
  - **Control**: Semicontrolled.
  - **Constraints**: The user claiming ownership must have this token existing in their wallet.
  - **Impact**: The token ID of the ownership token being detokenized.
- `chainId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is the chain ID of the remote chain.
- `ownerAddress`
  - **Control**: Not controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the `msg.sender` who is initiating detokenization on the remote chain.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Successfully relays a cross-chain message to the ProxyDomaRecord contract's

detokenize() function.

**Negative behavior**

- ☐ Fails on nonexistent ownership token ID.
- ☑ Fails on mismatched chain ID.
- ☑ Fails on mismatched owner address.
- ☐ Fails on non-ownership token.
- ☐ Fails if permissioned synthetic tokens exist for this ownership token.
- ☑ Fails if the token is not claimed by anyone.
- ☑ Fails when not called by a cross-chain sender.

### Function: tokenTransfer()

When a user transfers their ownership token to another wallet, the _beforeTokenTransfer() hook function on the remote chain will trigger a cross-chain message that executes this function.

### Inputs

- chainId
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is the chain ID of the remote chain.
- tokenId
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the token contract remote chain.
  - **Impact**: The token ID of the ownership token being transferred.
- oldOwnerAddress
  - **Control**: Not controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the msg.sender who initiated the token transfer on the remote chain.
- newOwnerAddress
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is address of the new owner on the remote chain.
- correlationId
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.

- **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

**Intended branches**

- ☐ Permissioned tokens return early.
- ☑ EOI domain name tokens also return early with `name.claimedBy` updated directly.
- ☑ The `name.claimedBy` is set to the the Doma Proxy address in the normal case, and all claim requests should be invalidated.

**Negative behavior**

- ☐ Fails on nonexistent ownership token ID.
- ☑ Fails on mismatched chain ID.
- ☑ Fails on mismatched owner address.
- ☑ Fails when not called by a cross-chain sender.

## 4.3. Module: DomaRecordRegistrarClaimFacet.sol

### Function: `approveClaimRequest()`

This function is called by registrars to accept ownership-claim requests.

### Inputs

- `sld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The SLD of the domain that this token is for.
- `tld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The TLD of the domain that this token is for.
- `request`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This request structure contains details about the claim request that is to be accepted.
- `correlationId`

- **Control**: Not controlled.
- **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
- **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

#### Intended branches

- ☑ The `name.claimedBy` is successfully set, and the claim request is deleted.

#### Negative behavior

- ☑ Fails on nonexistent name token.
- ☑ Fails when called by a nonregistrar.
- ☑ Fails on calling a registrar not matching the registrar that the name token is registered with.
- ☑ Fails if the claim request specified by the registrar does not match the last claim request made for this token.

## Function: `rejectClaimRequest()`

This function is called by registrars to reject ownership-claim requests.

### Inputs

- `sld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The SLD of the domain that this token is for.
- `tld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The TLD of the domain that this token is for.
- `request`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This request structure contains details about the claim request that is to be rejected.
- `correlationId`

- **Control**: Not controlled.
- **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
- **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ The claim request is successfully rejected and deleted.

**Negative behavior**

- ☑ Fails on nonexistent name token.
- ☑ Fails when called by a nonregistrar.
- ☑ Fails on calling a registrar not matching the registrar that the name token is registered with.
- ☑ Fails if the claim request specified by the registrar does not match the last claim request made for this token.

## 4.4. Module: DomaRecordRegistrarFacet.sol

### Function: `approveTokenization()`

This function is called by registrars to approve a tokenization request.

### Inputs

- `sld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The SLD of the domain that this token is for.
- `tld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The TLD of the domain that this token is for.
- `ownershipTokenOwnerAddress`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the address of the new owner of the ownership token on the remote chain.

- `ownershipTokenChainId`
    - **Control**: Fully controlled.
    - **Constraints**: N/A.
    - **Impact**: This is the chain ID on which the ownership token will be minted.
- `expiresAt`
    - **Control**: Fully controlled.
    - **Constraints**: N/A.
    - **Impact**: This is a Unix timestamp at which the token will expire.
- `permissions`
    - **Control**: Fully controlled.
    - **Constraints**: N/A.
    - **Impact**: This signifies any registrar-specific permissions for this ownership token.
- `nameservers`
    - **Control**: Fully controlled.
    - **Constraints**: N/A.
    - **Impact**: This is a list of name servers configured for this token.
- `dsKeys`
    - **Control**: Fully controlled.
    - **Constraints**: N/A.
    - **Impact**: This is a list of DNSSEC DS keys configured for this token.
- `correlationId`
    - **Control**: Not controlled.
    - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
    - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

#### Intended branches

- ☑ Successfully sends a cross-chain message to call the `mintOwnershipTokens()` function on the remote chain's ProxyDomaRecord contract.

#### Negative behavior

- ☑ Fails if not called by an authorized registrar.
- ☑ Fails on nonexistent tokenization request.
- ☑ Fails on calling registrar not matching the registrar that the tokenization request specified.

- ☑ Fails if the chain ID and owner address do not match the ones specified in the tokenization request.
- ☑ Fails if there already exists a token for this domain name.
- ☑ Fails if this is an EOI TLD for a different registrar.
- ☑ Fails if either the TLD or SLD is invalid.

### Function: `complianceChangeLockStatus()`

This function is used to lock and unlock domain name tokens.

### Inputs

- `sld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The SLD of the domain that this token is for.
- `tld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The TLD of the domain that this token is for.
- `isTransferLocked`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is used to determine whether to set or unset the lock.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

#### Intended branches

- ☑ Successfully sends a cross-chain message to call the ProxyDomaRecord contract's `changeLockStatus()` function.

#### Negative behavior

- ☑ Fails if not called by an authorized registrar-compliance operator.

- ☑ Fails if this domain name does not exist.
- ☑ Fails on calling a registrar-compliance operator not matching the registrar that the token was registered by.

### Function: `complianceDetokenize()`

This function is used by a registrar-compliance operator to detokenize a token for a domain name.

### Inputs

- `sld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The SLD of the domain that this token is for.
- `tld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The TLD of the domain that this token is for.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

#### Intended branches

- ☑ Successfully sends a cross-chain message to call the ProxyDomaRecord contract's `detokenizeUnchecked()` function.

#### Negative behavior

- ☑ Fails if not called by an authorized registrar-compliance operator.
- ☑ Fails if this domain name does not exist.
- ☑ Fails on calling a registrar-compliance operator not matching the registrar that the token was registered by.

### Function: `eoiImport()`

This function is used to import details for an EOI domain name without tokenizing it.

### Inputs

- `importData`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This array contains structures that contain details about the EOI domain name being imported.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

#### Intended branches

- ☑ Successfully creates a new EOI name or updates an existing EOI name.

#### Negative behavior

- ☑ Fails if not called by an authorized registrar.
- ☐ Fails if either the TLD or SLD is invalid.

### Function: `nameTokenize()`

This function is called by registrars to tokenize a domain name without going through the tokenization request process.

### Inputs

- `sld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The SLD of the domain that this token is for.
- `tld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The TLD of the domain that this token is for.
- `eoi`

- **Control**: Fully controlled.
- **Constraints**: N/A.
- **Impact**: This determines whether the token is for an EOI domain or not.

- `expiresAt`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is a Unix timestamp at which the token will expire.
- `permissions`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This signifies any registrar-specific permissions for this ownership token.
- `nameservers`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is a list of name servers configured for this token.
- `dsKeys`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is a list of DNSSEC DS keys configured for this token.
- `ownershipTokenOwnerAddress`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the address of the new owner of the ownership token on the remote chain.
- `ownershipTokenChainId`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the chain ID on which the ownership token will be minted.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

## Branches and code coverage (including function calls)

**Intended branches**

- ☑ Successfully sends a cross-chain message to call the `mintOwnershipTokens()` function on the remote chain's ProxyDomaRecord contract.

**Negative behavior**

- ☑ Fails if not called by an authorized registrar.
- ☑ Fails if this is an EOI TLD for a different registrar.
- ☑ Fails if either the TLD or SLD is invalid.
- ☑ Fails if not called by the trusted forwarder.
- ☑ Fails if the target chain is not supported.

### Function: `registrarDelete()`

This function is used by registrars to delete an expired token for a domain name.

### Inputs

- `sld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The SLD of the domain that this token is for.
- `tld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The TLD of the domain that this token is for.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Successfully sends a cross-chain message to call the ProxyDomaRecord contract's `detokenizeUnchecked()` function.

**Negative behavior**

- ☑ Fails if not called by an authorized registrar.
- ☑ Fails if this domain name does not exist.

- ☑ Fails on calling a registrar not matching the registrar that the token was registered by.
- ☑ Fails if the domain is not expired.

### Function: `registrarDetokenize()`

This function is used by registrars to detokenize a domain name.

### Inputs

- `sld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The SLD of the domain that this token is for.
- `tld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The TLD of the domain that this token is for.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

#### Intended branches

- ☑ Successfully sends a cross-chain message to call the ProxyDomaRecord contract's `detokenize()` function.

#### Negative behavior

- ☑ Fails if not called by an authorized registrar.
- ☑ Fails if this domain name does not exist.
- ☑ Fails on calling a registrar not matching the registrar that the token was registered by.
- ☑ Fails if permissioned synthetic tokens exist for this ownership token.
- ☑ Fails if this token is not claimed by an owner.

### Function: `rejectTokenization()`

This function is called by registrars to reject a tokenization request.

### Inputs

- `sld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The SLD of the domain that this token is for.
- `tld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The TLD of the domain that this token is for.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Successfully rejects and deletes the tokenization request.

**Negative behavior**

- ☑ Fails if not called by an authorized registrar.
- ☑ Fails on nonexistent tokenization request.
- ☑ Fails on calling a registrar not matching the registrar that the tokenization request specified.

### Function: `renew()`

This function is used to renew a domain.

### Inputs

- `sld`
  - **Control**: Fully controlled.

- **Constraints**: N/A.
- **Impact**: The SLD of the domain that this token is for.

- `tld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The TLD of the domain that this token is for.
- `expiresAt`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is a Unix timestamp at which the renewed token will expire.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Successfully updates the expiry timestamp for the specified domain name.

**Negative behavior**

- ☑ Fails if not called by an authorized registrar.
- ☑ Fails if a token does not exist for this domain name.
- ☑ Fails on calling a registrar not matching the registrar that the token was registered by.

## Function: `update()`

This function is used to update the DS keys and name servers for a domain name.

### Inputs

- `sld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The SLD of the domain that this token is for.
- `tld`
  - **Control**: Fully controlled.

- **Constraints**: N/A.
- **Impact**: The TLD of the domain that this token is for.

- `nameservers`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is a list of the new name servers for this domain name.
- `dsKeys`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is a list of the new DS keys for this domain name.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Successfully updates the name servers and DS keys for the specified domain name.

**Negative behavior**

- ☑ Fails if not called by an authorized registrar.
- ☑ Fails if a token does not exist for this domain name.
- ☑ Fails on calling a registrar not matching the registrar that the token was registered by.

## Function: `updateDsKeys()`

This function is used to update the DNSSEC DS keys for a domain name.

### Inputs

- `sld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The SLD of the domain that this token is for.
- `tld`
  - **Control**: Fully controlled.

- **Constraints**: N/A.
- **Impact**: The TLD of the domain that this token is for.

- `dsKeys`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is a list of the new DS keys for this domain name.
- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

#### Intended branches

- ☑ Successfully updates the DS keys for the specified domain name.

#### Negative behavior

- ☑ Fails if not called by an authorized registrar.
- ☑ Fails if a token does not exist for this domain name.
- ☑ Fails on calling a registrar not matching the registrar that the token was registered by.

### Function: `updateNameservers()`

This function is used to update the name servers for a domain name.

### Inputs

- `sld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The SLD of the domain that this token is for.
- `tld`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The TLD of the domain that this token is for.
- `nameservers`
  - **Control**: Fully controlled.

- **Constraints**: N/A.
- **Impact**: This is a list of the new name servers for this domain name.

- `correlationId`
  - **Control**: Not controlled.
  - **Constraints**: This is hardcoded by the ProxyDomaRecord contract.
  - **Impact**: This is a hash of the current block number, the chain ID, and a nonce on the remote chain.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Successfully updates the name servers for the specified domain name.

**Negative behavior**

- ☑ Fails if not called by an authorized registrar.
- ☑ Fails if a token does not exist for this domain name.
- ☑ Fails on calling a registrar not matching the registrar that the token was registered by.

## 4.5. Module: ERC7786GatewayReceiver.sol

### Function: `executeMessage(string calldata sourceChain, string calldata, bytes calldata payload, bytes[] calldata attributes)`

This function executes cross-chain messages by processing attributes to extract nonce information, validating the cross-chain sender, checking nonce uniqueness, and executing the provided payload on the current contract.

### Inputs

- `sourceChain`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Ensures nonce isolation between different source chains, preventing cross-chain replay attacks.
- `payload`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Allows arbitrary function execution on the current contract, relying on access control for security.

- `attributes`
    - **Control**: Full.
    - **Constraints**: N/A.
    - **Impact**: Provides metadata, including nonce information required for replay-attack prevention.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Valid cross-chain sender with proper nonce executes payload successfully.
- ☐ Successful payload execution returns `IERC7786Receiver.executeMessage.selector`.
- ☑ Failed payload execution with error data emits `CallFailed` event and continues.
- ☑ Payload execution via `address(this).call(payload)` executes arbitrary functions on current contract.

**Negative behavior**

- ☑ Unauthorized `msg.sender` fails `_checkCrossChainSender()` validation and reverts.
- ☑ Invalid or reused nonce fails `_useCheckedNonce()` validation and reverts.
- ☑ Payload execution failing without error data causes `CallFailedWithoutError` revert.

## 4.6. Module: ERC7786GatewaySource.sol

### Function: `sendMessage(string calldata destinationChain, string calldata receiver, bytes calldata payload, bytes[] calldata attributes)`

This function sends cross-chain messages by validating attributes, generating nonces, and emitting a `MessagePosted` event. Only permitted senders can execute this function, and it ensures proper nonce management for replay-attack prevention.

This contract will be deployed on both the Doma chain and the tokenization chain.

### Inputs

- `destinationChain`
    - **Control**: Full.
    - **Constraints**: Used as input to `_useNonce()` for destination-specific nonce generation.
    - **Impact**: Ensures nonce isolation between different destination chains, preventing cross-chain replay attacks.
- `receiver`

- **Control**: Full.
- **Constraints**: N/A.
- **Impact**: Specifies the target receiver address on the destination chain — no validation performed at source.

- `payload`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Contains the actual data to be executed on the destination chain — no validation at source level.
- `attributes`
  - **Control**: Full.
  - **Constraints**: Each attribute selector is validated via `supportsAttribute()` to ensure only supported attributes are included.
  - **Impact**: Prevents inclusion of unsupported metadata that could cause processing issues on the destination chain.

### Branches and code coverage (including function calls)

#### Intended branches

- ☐ Valid permitted sender with supported attributes successfully sends message and returns `outboxId`.
- ☑ Attributes contain `NONCE_KEY_ATTRIBUTE`, which is extracted and used for the nonce key.
- ☑ A new nonce is generated and appended to the attributes array.
- ☑ Nonce generation via `_useNonce()` creates a unique nonce for the destination chain and key combination.
- ☑ Sender address formatting via `CAIP.format()` converts address to CAIP standard format.
- ☑ The `MessagePosted` event emission with `outboxId`, sender, receiver, payload, value, and attributes.

#### Negative behavior

- ☑ Unauthorized sender without `PERMITTED_SENDER_ROLE` reverts due to `onlyPermittedSender` modifier.
- ☑ Unsupported attribute selector causes `UnsupportedAttribute` revert.

## 4.7. Module: Marketplace.sol

### Function: `pay(PaymentVoucher calldata voucher, bytes calldata signature)`

This function processes payment for off-chain orders using signed vouchers. It supports both ETH and ERC-20 token payments with signature verification and payment-ID replay protection.

### Inputs

- `voucher.buyer`
  - **Control**: Full.
  - **Constraints**: Must match `msg.sender` (checked via `_verifyBuyerMatchesSender()`).
  - **Impact**: Ensures only the designated buyer can execute the payment, preventing unauthorized payment execution.
- `voucher.token`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Specifies payment token (`address(0)` for ETH, ERC-20 address for tokens).
- `voucher.amount`
  - **Control**: Full.
  - **Constraints**: Must match `msg.value` for ETH payments or be transferred from buyer for ERC-20.
  - **Impact**: Ensures the payment amount matches the voucher specification, preventing underpayment attacks.
- `voucher.voucherExpiration`
  - **Control**: Full.
  - **Constraints**: Must be greater than the current block timestamp (checked via `_verifyNotExpiredVoucher()`).
  - **Impact**: Prevents execution of expired vouchers, ensuring time-sensitive payment conditions.
- `voucher.paymentId`
  - **Control**: Full.
  - **Constraints**: Must be unique and not previously used (checked via `usedPaymentIdHashes` mapping).
  - **Impact**: Prevents replay attacks by ensuring each payment voucher can only be used once.
- `voucher.orderId`

- **Control**: Full.
- **Constraints**: Included in signature hash but no direct validation.
- **Impact**: Links payment to specific order for off-chain tracking and verification.

- `signature`
  - **Control**: Full.
  - **Constraints**: ECDSA signature verification via `_verifySignature()` to ensure voucher was signed by an authorized signer.
  - **Impact**: Ensures voucher authenticity and prevents unauthorized voucher creation.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Valid ETH payment with matching `msg.value` successfully transfers to treasury.
- ☑ Valid ERC-20 payment successfully transfers tokens from buyer to treasury.
- ☑ Successful payment emits `PaymentFulfilled` event.

**Negative behavior**

- ☑ Expired voucher (`voucherExpiration < block.timestamp`) causes `SignatureExpired` revert.
- ☑ Invalid signature verification causes `InvalidSigner` revert.
- ☐ Zero treasury address causes `ZeroAddressTreasury` revert.
- ☑ Reused `paymentId` causes `InvalidPaymentId` revert.
- ☑ Buyer mismatch (`msg.sender != voucher.buyer`) causes `InvalidSender` revert.
- ☑ ETH payment amount mismatch (`msg.value != voucher.amount`) causes `InvalidDeposit` revert.
- ☐ ERC-20 payment with nonzero `msg.value` causes `InvalidDeposit` revert.
- ☑ Failed ETH transfer to treasury causes `TransferFailed` revert.
- ☐ Failed ERC-20 token transfer causes `TransferFailed` revert.

### Function: `secondarySale(SecondarySaleVoucher calldata voucher, bytes calldata signature)`

This function fulfills secondary sale orders by transferring NFTs from sellers to buyer, distributing payments to sellers with fee deduction, and transferring fees to the treasury. It supports batch transfers of multiple NFTs in a single transaction.

### Inputs

- `voucher.buyer`

  - **Control**: Full.
  - **Constraints**: Must match `msg.sender` (checked via `_verifyBuyerMatchesSender()`).
  - **Impact**: Ensures only the designated buyer can execute the secondary sale, preventing unauthorized purchase execution.
- `voucher.amount`
  - **Control**: Full.
  - **Constraints**: Must match `msg.value` and the sum of all name prices.
  - **Impact**: Ensures total payment matches the voucher specification and prevents underpayment or overpayment attacks.
- `voucher.voucherExpiration`
  - **Control**: Full.
  - **Constraints**: Must be greater than the current block timestamp (checked via `_verifyNotExpiredVoucher()`).
  - **Impact**: Prevents execution of expired vouchers, ensuring time-sensitive sale conditions.
- `voucher.paymentId`
  - **Control**: Full.
  - **Constraints**: Must be unique and not previously used (checked via `usedPaymentIdHashes` mapping).
  - **Impact**: Prevents replay attacks by ensuring each sale voucher can only be used once.
- `voucher.orderId`
  - **Control**: Full.
  - **Constraints**: Included in the signature hash but no direct validation.
  - **Impact**: Links sale to specific order for off-chain tracking and verification.
- `voucher.names`
  - **Control**: Full.
  - **Constraints**: Total `names.price` sum must match `voucher.amount`.
  - **Impact**: Array of NFT transfer information including registry, `tokenId`, owner, and price for each NFT.
- `signature`.
  - **Control**: Full.
  - **Constraints**: ECDSA signature verification via `_verifySignature()` to ensure voucher was signed by an authorized signer.
  - **Impact**: Ensures voucher authenticity and prevents unauthorized voucher creation.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Valid secondary sale with multiple NFTs successfully transfers tokens.
- ☑ Fee calculation and distribution to the treasury works correctly.
- ☑ Seller payments after fee deduction are transferred successfully.
- ☑ ETH transfer to sellers via `name.owner.call{value: sellerProfit}("")` succeeds for each NFT.
- ☑ NFT transfer via `name.registry.safeTransferFrom()` successfully transfers ownership to buyer.
- ☑ Fee transfer to treasury via `_treasury.call{value: totalFee}("")` succeeds.

**Negative behavior**

- ☑ Expired voucher (`voucherExpiration < block.timestamp`) causes `SignatureExpired` revert.
- ☑ Invalid signature verification causes `InvalidSigner` revert.
- ☑ Reused `paymentId` causes `InvalidPaymentId` revert.
- ☐ Zero treasury address causes `ZeroAddressTreasury` revert.
- ☑ Buyer mismatch (`msg.sender != voucher.buyer`) causes `InvalidSender` revert.
- ☑ Payment amount mismatch (`msg.value != voucher.amount`) causes `InvalidDeposit` revert.
- ☑ Total names price mismatch (`totalAmount != voucher.amount`) causes `InvalidPaymentAmount` revert.
- ☑ Failed ETH transfer to seller causes `TransferFailed` revert.
- ☑ Failed NFT transfer causes revert in `safeTransferFrom`.
- ☑ Failed fee transfer to treasury causes `TransferFailed` revert.

## 4.8. Module: OwnershipToken.sol

### Function: `bulkBurn(uint256[] calldata tokenIds, string calldata correlationId)`

This function burns multiple tokens in a single transaction by iterating through token IDs, recording owners, and emitting burn events. Only minters can execute this function, and user-initiated burns are blocked.

### Inputs

- `tokenIds`
  - **Control**: Full.

- **Constraints**: Each token ID is validated via `ownerOf(tokenId)`, which reverts if the token does not exist.
- **Impact**: Array of token IDs to burn — allows batch processing for gas efficiency.

- `correlationId`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Used for off-chain tracking and correlation with burn operations.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Valid token IDs array with existing tokens successfully burns all tokens.

**Negative behavior**

- ☑ Unauthorized caller without `MINTER_ROLE` reverts due to `onlyMinter` modifier.
- ☐ Nonexistent token ID in array causes `ownerOf(tokenId)` revert.
- ☐ Already burned token ID causes `ownerOf(tokenId)` revert.

### Function: `bulkMint(OwnershipTokenMintInfo[] calldata names, string calldata correlationId)`

This function mints multiple ownership tokens in a single transaction by processing mint information for each name, creating tokens, and emitting mint events. Only minters can execute this function, and user-initiated mints are blocked.

### Inputs

- `names`
  - **Control**: Full.
  - **Constraints**: Each name's expiration date is validated via `_validateExpirationDate()` in `_mint()`.
  - **Impact**: Array of mint information including `registrarIanaId`, `sld`, `tld`, `tokenId`, `expiresAt`, and owner.
- `correlationId`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Used for off-chain tracking and correlation with mint operations.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Valid names array with proper expiration dates successfully mints all tokens.

**Negative behavior**

- ☑ Unauthorized caller without `MINTER_ROLE` reverts due to `onlyMinter` modifier.
- ☑ Invalid expiration date (past date) causes `PastExpirationDate` revert in `_validateExpirationDate()`.
- ☐ Invalid expiration date (too far in future, > 10 years) causes `TooBigExpirationDate` revert in `_validateExpirationDate()`.
- ☑ Duplicate token ID causes revert in `_safeMint()` when trying to mint existing token.

## Function: `renew(uint256 tokenId, uint256 expiresAt, string calldata correlationId)`

This function renews an existing ownership token by updating its expiration date. It validates the new expiration date and ensures the token exists before updating. Only minters can execute this function.

### Inputs

- `tokenId`
  - **Control**: Full.
  - **Constraints**: Token existence is validated via `_ownerOf(tokenId)`, which returns `address(0)` for nonexistent tokens.
  - **Impact**: Identifies the specific token to renew — ensures operation only applies to existing tokens.
- `expiresAt`
  - **Control**: Full.
  - **Constraints**: Validated via `_validateExpirationDate()` to ensure future date is within a 10-year limit.
  - **Impact**: New expiration timestamp for the token — must be a valid future date.
- `correlationId`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Used for off-chain tracking and correlation with renewal operations.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Token existence check via `_ownerOf(tokenId)` succeeds for existing tokens.
- ☑ Expiration-date update in `_expirations[tokenId]` mapping succeeds.

**Negative behavior**

- ☑ Unauthorized caller without `MINTER_ROLE` reverts due to `onlyMinter` modifier.
- ☑ Invalid expiration date (past date) causes `PastExpirationDate` revert in `_validateExpirationDate()`.
- ☑ Invalid expiration date (too far in future, > 10 years) causes `TooBigExpirationDate` revert in `_validateExpirationDate()`.
- ☑ Nonexistent token ID causes `NameTokenDoesNotExist` revert when `_ownerOf(tokenId)` returns `address(0)`.

## Function: `_beforeTokenTransfer(address from, address to, uint256 firstTokenId, uint256)`

This function validates token transfers by checking expiration status, transfer locks, and global transfer blocks. It initiates cross-chain ownership transfer from the tokenization chain to Doma chain via proxy contract and performs ERC-721-C validation for transfers between owners.

### Inputs

- `from`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Used to determine transfer type (mint/burn/transfer) and as source address for cross-chain ownership transfer.
- `to`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Used to determine transfer type (mint/burn/transfer) and as destination address for cross-chain ownership transfer.
- `firstTokenId`
  - **Control**: Full.
  - **Constraints**: Checked for expiration via `_isTokenExpired()`, transfer lock via `_transferLocks`, and global block via `blockAllTransfers`.
  - **Impact**: Ensures only valid, nonexpired, and unlocked tokens can be transferred and identifies the token for cross-chain ownership updates.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Valid transfer between owners (`from != address(0) && to != address(0)`) with a nonexpired, unlocked token succeeds.
- ☑ Cross-chain ownership-transfer initiation via `proxyDomaRecord.tokenTransfer(firstTokenId, from, to)` succeeds.
- ☑ ERC-721-C validation via `_preValidateTransfer()` succeeds.

**Negative behavior**

- ☑ Expired token transfer (`from != address(0) && to != address(0) && _isTokenExpired(firstTokenId)`) causes `TransferLocked` revert.
- ☑ Locked token transfer (`from != address(0) && to != address(0) && _transferLocks[firstTokenId]`) causes `TransferLocked` revert.
- ☑ Global transfer block (`from != address(0) && to != address(0) && blockAllTransfers`) causes `TransferLocked` revert.
- ☐ Missing proxy contract (`from != address(0) && to != address(0) && address(proxyDomaRecord) == address(0)`) causes `ProxyDomaRecordNotSet` revert.
- ☐ Cross-chain ownership-transfer failure in `proxyDomaRecord.tokenTransfer()` causes revert and blocks local transfer.

## 4.9. Module: ProxyDomaRecord.sol

### Function: `bridge(uint256 tokenId, bool isSynthetic, string calldata targetChainId, string calldata targetOwnerAddress)`

This function moves ownership tokens between different chains by verifying ownership, collecting fees, burning the token on the current chain, and initiating cross-chain transfer to the target chain via the `_relayMessage` call to Doma chain.

### Inputs

- `tokenId`
  - **Control**: Full.
  - **Constraints**: Token ownership verified via `_verifyTokenOwnership()` and lock status checked via `lockStatusOf()`.
  - **Impact**: Identifies the specific token to bridge between chains.
- `isSynthetic`
  - **Control**: Not controlled.
  - **Constraints**: Must be false (synthetic tokens not implemented).
  - **Impact**: Distinguishes between regular and permissioned tokens.

- `targetChainId`
  - **Control**: Full.
  - **Constraints**: Must be in supported chains list via `isTargetChainSupported()`.
  - **Impact**: Specifies destination chain for token bridging.
- `targetOwnerAddress`
  - **Control**: Full.
  - **Constraints**: N/A (passed to destination chain for validation).
  - **Impact**: Specifies owner address on target chain.

### Branches and code coverage (including function calls)

#### Intended branches

- ☑ Fee collection via `_collectFee()` succeeds for `BRIDGE_OPERATION`.
- ☑ Token burning via `_burnToken()` succeeds on the current chain.
- ☑ Cross-chain bridge initiation via `_relayMessage()` to Doma chain succeeds.

#### Negative behavior

- ☐ Synthetic token usage (`isSynthetic == true`) causes `NotImplemented` revert.
- ☑ Unsupported target chain causes `UnsupportedTargetChain` revert.
- ☑ Non-owner attempting to bridge causes `InvalidOwnerAddress` revert in `_verifyTokenOwnership()`.
- ☑ Locked token bridging causes `TransferLocked` revert.
- ☑ Insufficient fee payment causes revert in `_collectFee()`.
- ☑ Cross-chain message relay failure causes revert and prevents bridging.

## Function: `changeLockStatus(string calldata tokenId, bool isSynthetic, bool isTransferLocked, string calldata correlationId)`

This function changes the transfer lock status of ownership tokens. This function can only be called by authorized cross-chain senders to enforce compliance controls on token transfers.

### Inputs

- `tokenId`
  - **Control**: Full.
  - **Constraints**: Converted to `uint256` via `Strings.parseUint()`.
  - **Impact**: Identifies the specific token to change lock status for.
- `isSynthetic`

- **Control**: Not controlled.
- **Constraints**: Must be false (synthetic tokens not implemented).
- **Impact**: Distinguishes between regular and permissioned tokens.

- `isTransferLocked`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: New lock status to apply.
- `correlationId`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Used for cross-chain operation tracking and correlation.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Token lock-status update via `ownershipToken().setLockStatus()` succeeds.

**Negative behavior**

- ☑ Unauthorized caller without `CROSS_CHAIN_SENDER_ROLE` reverts due to `onlyCrossChainSender` modifier.
- ☐ Synthetic token usage (`isSynthetic == true`) causes `NotImplemented` revert.
- ☑ Invalid `tokenId` string format causes revert in `Strings.parseUint()`.

## Function: `claimOwnership(uint256 tokenId, bool isSynthetic, ProofOfContactsVoucher calldata proofOfContactsVoucher, bytes calldata signature)`

This function claims domain ownership using ownership tokens and proof-of-contacts verification. It validates voucher signatures from registrars or Doma, verifies token ownership, collects fees, and initiates cross-chain ownership claim via `_relayMessage` to Doma chain.

### Inputs

- `tokenId`
  - **Control**: Full.
  - **Constraints**: Token ownership verified via `_verifyTokenOwnership()` and registrar validation for registrar vouchers.
  - **Impact**: Identifies the ownership token being used to claim domain ownership.

- `isSynthetic`
  - **Control**: Not controlled.
  - **Constraints**: Must be false (synthetic tokens not implemented).
  - **Impact**: Distinguishes between regular and permissioned tokens.
- `proofOfContactsVoucher`
  - **Control**: Full.
  - **Constraints**: Expiration date checked via `_verifyNotExpiredVoucher()` and nonce verified via `_verifyAndUpdateNonce()`.
  - **Impact**: Contains registrant handle, proof source, nonce, and expiration for contact verification.
- `signature`
  - **Control**: Full.
  - **Constraints**: Verified via `_verifyRegistrarSignature()` or `_verifyDomaSignature()` depending on proof source.
  - **Impact**: Ensures voucher authenticity from authorized signers.

## Branches and code coverage (including function calls)

### Intended branches

- ☑ Valid token owner with valid registrar voucher successfully claims ownership.
- ☑ Valid token owner with valid Doma voucher successfully claims ownership.
- ☑ Fee collection via `_collectFee()` succeeds for `CLAIM_OWNERSHIP_OPERATION`.
- ☑ Cross-chain ownership-claim initiation via `_relayMessage()` to Doma chain succeeds.

### Negative behavior

- ☑ Synthetic token usage (`isSynthetic == true`) causes `NotImplemented` revert.
- ☑ Expired voucher causes `VoucherExpired` revert in `_verifyNotExpiredVoucher()`.
- ☑ Reused nonce causes `NonceAlreadyUsed` revert in `_verifyAndUpdateNonce()`.
- ☑ Invalid signature causes `InvalidSigner` revert in signature verification.
- ☑ Registrar mismatch for token causes `InvalidRegistrar` revert.
- ☑ Non-owner attempting to claim causes `InvalidOwnerAddress` revert in `_verifyTokenOwnership()`.
- ☑ Insufficient fee payment causes revert in `_collectFee()`.
- ☑ Cross-chain message-relay failure causes revert and prevents claim.

### Function: `detokenize(string calldata tokenId, bool isSynthetic, string calldata claimedBy, string calldata correlationId)`

This function detokenizes domain names by verifying ownership, burning tokens, and completing the detokenization process. It can only be called by authorized cross-chain senders from Doma chain. It includes ownership verification before proceeding.

#### Inputs

- `tokenId`
  - **Control**: Full.
  - **Constraints**: Converted to `uint256` via `Strings.parseUint()`.
  - **Impact**: Identifies the specific token to detokenize.
- `isSynthetic`
  - **Control**: Not controlled.
  - **Constraints**: Must be false (synthetic tokens not implemented).
  - **Impact**: Distinguishes between regular and permissioned tokens.
- `claimedBy`
  - **Control**: Full.
  - **Constraints**: Converted to address via `Strings.parseAddress()` and must match token owner.
  - **Impact**: Expected owner address that must match the current token owner.
- `correlationId`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Used for cross-chain operation tracking and correlation.

#### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Authorized cross-chain sender with valid ownership verification successfully detokenizes.
- ☑ Token burning via `_burnToken()` succeeds.
- ☑ Detokenization completion via `_completeDetokenization()` and subsequent `_relayMessage()` to Doma chain succeeds.

**Negative behavior**

- ☑ Unauthorized caller without `CROSS_CHAIN_SENDER_ROLE` reverts due to `onlyCrossChainSender` modifier.

- ☑ Synthetic token usage (`isSynthetic == true`) causes `NotImplemented` revert.
- ☑ Invalid `tokenId` string format causes revert in `Strings.parseUint()`.
- ☑ Invalid `claimedBy` address format causes revert in `Strings.parseAddress()`.
- ☑ Ownership mismatch causes `InvalidOwnerAddress` revert in `_verifyTokenOwnership()`.
- ☑ Cross-chain completion message-relay failure causes revert.

### Function: `detokenizeUnchecked(string calldata tokenId, bool isSynthetic, string calldata correlationId)`

This function detokenizes domain names without ownership verification. It can only be called by authorized cross-chain senders from Doma chain. It bypasses ownership checks for administrative or emergency detokenization scenarios.

#### Inputs

- `tokenId`
  - **Control**: Full.
  - **Constraints**: Converted to `uint256` via `Strings.parseUint()`.
  - **Impact**: Identifies the specific token to detokenize without ownership verification.
- `isSynthetic`
  - **Control**: Not controlled.
  - **Constraints**: Must be false (synthetic tokens not implemented).
  - **Impact**: Distinguishes between regular and permissioned tokens.
- `correlationId`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Used for cross-chain operation tracking and correlation.

#### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Authorized cross-chain sender successfully detokenizes without ownership checks.
- ☑ Token burning via `_burnToken()` succeeds.
- ☑ Detokenization completion via `_completeDetokenization()` and subsequent `_relayMessage()` to Doma chain succeeds.

**Negative behavior**

- ☑ Unauthorized caller without `CROSS_CHAIN_SENDER_ROLE` reverts due to `onlyCrossChainSender` modifier.
- ☐ Synthetic token usage (`isSynthetic == true`) causes `NotImplemented` revert.
- ☑ Invalid `tokenId` string format causes revert in `Strings.parseUint()`.
- ☑ Cross-chain completion message-relay failure causes revert.

### Function: `mintOwnershipTokens(uint256 registrarIanaId, OwnershipTokenInfo[] calldata tokens, string calldata ownerAddress, string calldata correlationId)`

This function mints ownership tokens in response to successful tokenization requests from Doma chain. It can only be called by authorized cross-chain senders. It processes multiple tokens in batch and handles address parsing from string format.

#### Inputs

- `registrarIanaId`
  - **Control**: Full.
  - **Constraints**: N/A (validated on Doma chain before cross-chain call).
  - **Impact**: IANA ID of the registrar authorizing the tokenization.
- `tokens`
  - **Control**: Full.
  - **Constraints**: Array elements are converted and used to create `OwnershipTokenMintInfo` structures.
  - **Impact**: Array of token information including SLD, TLD, `tokenId`, and expiration date.
- `ownerAddress`
  - **Control**: Full.
  - **Constraints**: Converted to address via `Strings.parseAddress()`.
  - **Impact**: Owner address for the minted tokens in string format from cross-chain call.
- `correlationId`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Used for cross-chain operation tracking and correlation.

#### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Authorized cross-chain sender successfully mints ownership tokens.
- ☑ Owner address parsing via `Strings.parseAddress()` succeeds.
- ☑ Token-information array processing and conversion to `OwnershipTokenMintInfo` structures succeeds.
- ☑ Bulk minting via `ownershipToken().bulkMint()` succeeds.

**Negative behavior**

- ☑ Unauthorized caller without `CROSS_CHAIN_SENDER_ROLE` reverts due to `onlyCrossChainSender` modifier.
- ☑ Invalid `ownerAddress` string format causes revert in `Strings.parseAddress()`.
- ☑ Invalid `tokenId` string format in token array causes revert in `Strings.parseUint()`.
- ☑ Token minting failure in `bulkMint()` causes revert (e.g., duplicate `tokenId`, invalid expiration).

### Function: `renew(string calldata tokenId, bool isSynthetic, uint256 expiresAt, string calldata correlationId)`

This function renews ownership tokens by updating their expiration dates. It can only be called by authorized cross-chain senders from Doma chain in response to domain renewals. It updates token expiration without ownership verification.

#### Inputs

- `tokenId`
  - **Control**: Full.
  - **Constraints**: Converted to `uint256` via `Strings.parseUint()`.
  - **Impact**: Identifies the specific token to renew.
- `isSynthetic`
  - **Control**: Not controlled.
  - **Constraints**: Must be false (synthetic tokens not implemented).
  - **Impact**: Distinguishes between regular and permissioned tokens.
- `expiresAt`
  - **Control**: Full.
  - **Constraints**: Validated in `ownershipToken().renew()` via inherited validation logic.
  - **Impact**: New expiration timestamp for the token.
- `correlationId`
  - **Control**: Full.
  - **Constraints**: N/A.

- **Impact**: Used for cross-chain operation tracking and correlation.

### Branches and code coverage (including function calls)

#### Intended branches

- ☑ Authorized cross-chain sender successfully renews token with valid expiration date.
- ☑ Token renewal via `ownershipToken().renew()` succeeds.

#### Negative behavior

- ☑ Unauthorized caller without `CROSS_CHAIN_SENDER_ROLE` reverts due to `onlyCrossChainSender` modifier.
- ☑ Synthetic token usage (`isSynthetic == true`) causes `NotImplemented` revert.
- ☑ Invalid `tokenId` string format causes revert in `Strings.parseUint()`.
- ☑ Invalid expiration date causes revert in `ownershipToken().renew()` (e.g., past date, too far in future).
- ☑ Nonexistent token causes `NameTokenDoesNotExist` revert in `ownershipToken().renew()`.

## Function: `requestDetokenization(uint256 tokenId, bool isSynthetic)`

This function initiates a detokenization request by verifying token ownership and sending a cross-chain message to Doma chain for validation and processing. Only token owners can request detokenization of their tokens.

### Inputs

- `tokenId`
  - **Control**: Full.
  - **Validation**: Token ownership verified via `_verifyTokenOwnership()`.
  - **Impact**: Identifies the specific token to request detokenization for.
- `isSynthetic`
  - **Control**: Not controlled.
  - **Validation**: Must be false (synthetic tokens not implemented).
  - **Impact**: Distinguishes between regular and permissioned tokens.

### Branches and code coverage (including function calls)

#### Intended branches

- ☑ Valid token owner successfully initiates detokenization request.

- ☑ Correlation ID generation via `_useCorrelationId()` succeeds.
- ☑ Cross-chain detokenization request via `_relayMessage()` to Doma chain succeeds.

**Negative behavior**

- ☑ Synthetic token usage (`isSynthetic == true`) causes `NotImplemented` revert.
- ☑ Non-owner attempting to request detokenization causes `InvalidOwnerAddress` revert in `_verifyTokenOwnership()`.
- ☑ Cross-chain message-relay failure causes revert and prevents request.

### Function: `requestTokenization(TokenizationVoucher calldata voucher, bytes calldata signature)`

This function initiates domain tokenization requests by validating signed vouchers from registrars, verifying domain names, collecting fees, and sending cross-chain tokenization requests to Doma chain via `_relayMessage` for registrar approval.

#### Inputs

- `voucher`
  - **Control**: Full.
  - **Constraints**: Expiration checked via `_verifyNotExpiredVoucher()`, nonce verified via `_verifyAndUpdateNonce()`, and owner must match `msg.sender`.
  - **Impact**: Contains names array, nonce, expiration, and owner address for tokenization request.
- `signature`
  - **Control**: Full.
  - **Constraints**: Verified via `_verifyRegistrarSignature()` to ensure voucher was signed by an authorized registrar.
  - **Impact**: Ensures voucher authenticity and prevents unauthorized tokenization requests.

#### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Valid voucher with authorized registrar signature successfully initiates tokenization.
- ☑ Domain name validation via `NameValidator.ensureValidLabel()` succeeds for all names.
- ☑ Fee collection via `_collectFee()` succeeds for `REQUEST_TOKENIZATION_OPERATION`.
- ☑ Cross-chain tokenization initiation via `_relayMessage()` to Doma chain succeeds.

**Negative behavior**

- ☑ Expired voucher causes `VoucherExpired` revert in `_verifyNotExpiredVoucher()`.
- ☑ Reused nonce causes `NonceAlreadyUsed` revert in `_verifyAndUpdateNonce()`.
- ☑ Owner mismatch causes `InvalidOwnerAddress` revert when `msg.sender != voucher.ownerAddress`.
- ☑ Invalid signature causes `InvalidSigner` revert in `_verifyRegistrarSignature()`.
- ☑ Already tokenized name causes `NameAlreadyTokenized` revert when token already exists.
- ☑ Invalid domain labels cause revert in `NameValidator.ensureValidLabel()`.
- ☑ Insufficient fee payment causes revert in `_collectFee()`.
- ☑ Cross-chain message-relay failure causes revert and prevents tokenization request.

### Function: `tokenTransfer(uint256 tokenId, address from, address to)`

This function handles cross-chain ownership record updates when tokens are transferred between addresses on the tokenization chain. It is called by the OwnershipToken contract during transfers to synchronize ownership changes with Doma chain via `_relayMessage`.

### Inputs

- `tokenId`
  - **Control**: Semicontrolled.
  - **Constraints**: N/A (validated by calling the OwnershipToken contract).
  - **Impact**: Identifies the specific token being transferred for the cross-chain record update.
- `from`
  - **Control**: Not controlled.
  - **Constraints**: N/A (validated by calling the OwnershipToken contract).
  - **Impact**: Source address of the token transfer for the cross-chain ownership update.
- `to`
  - **Control**: Full.
  - **Constraints**: N/A (validated by calling the OwnershipToken contract).
  - **Impact**: Destination address of the token transfer for the cross-chain ownership update.

### Branches and code coverage (including function calls)

**Intended branches**

- ☑ Valid token transfer from the OwnershipToken contract successfully initiates the cross-chain update.
- ☑ Cross-chain gateway validation succeeds (nonzero address check).
- ☑ Correlation ID generation via `_useCorrelationId()` succeeds.
- ☑ Cross-chain token-transfer notification via `_relayMessage()` to Doma chain succeeds.

**Negative behavior**

- ☑ Unauthorized caller (not OwnershipToken contract) causes `InvalidCaller` revert due to the `onlyOwnershipToken` modifier.
- ☑ Zero-address cross-chain gateway causes `ZeroAddress` revert.
- ☑ Cross-chain message-relay failure causes revert and prevents ownership record update.

# 5. Assessment Results

During our assessment on the scoped d3-doma contracts, we discovered 12 findings. Three critical issues were found. One was of medium impact, six were of low impact, and the remaining findings were informational in nature.

## 5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.