Zellic

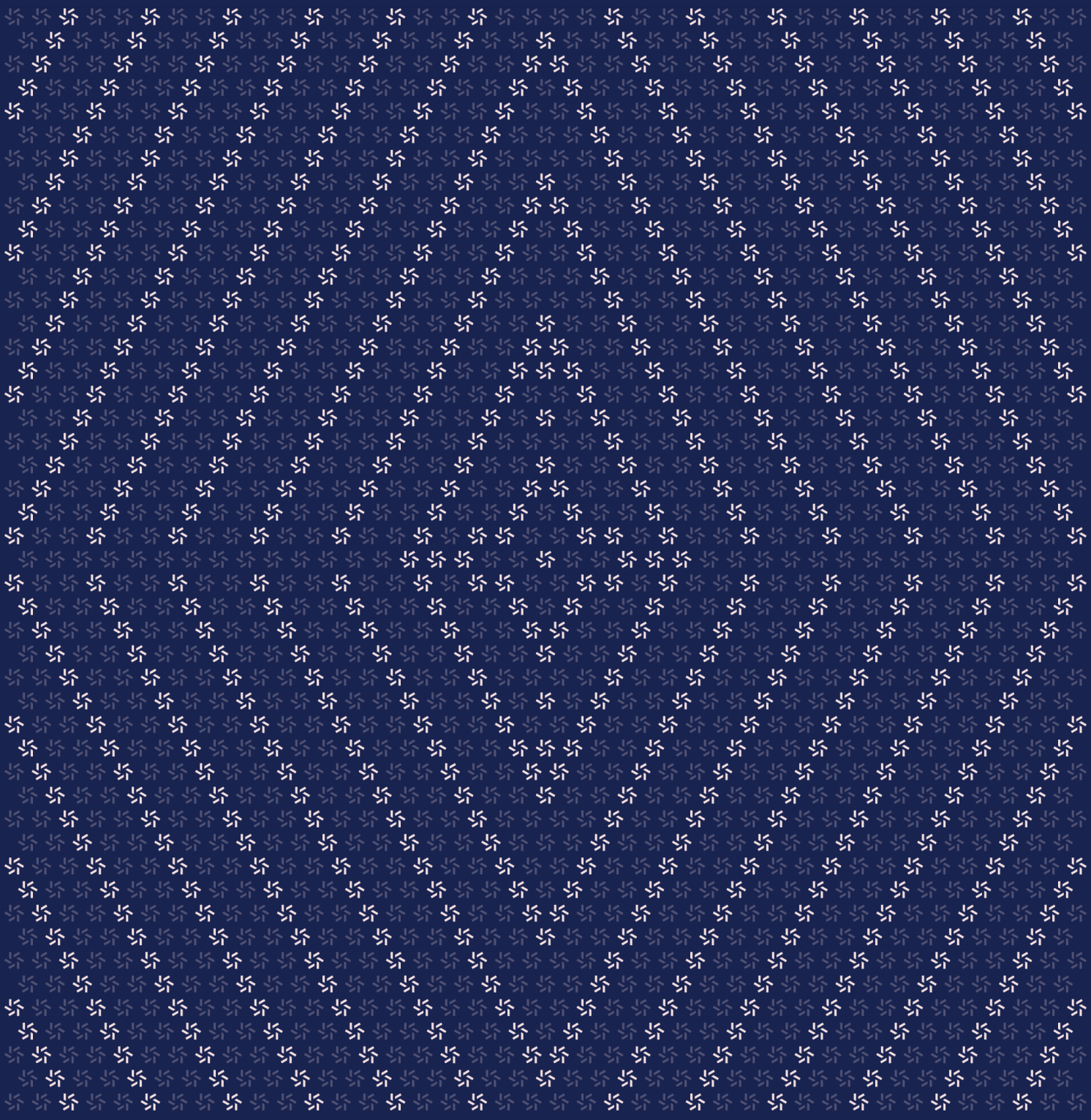
Prepared for
Tom Lehman
Facet Protocol

Prepared by
Filippo Cremonese
Kamensec
Zellic

November 4, 2024

Facet Bridge

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 Facet Protocol from October 17th to October 23rd, 2024. During this engagement, Zellic reviewed Facet Bridge'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:

- Have the deviations from Optimism introduced any security issues?
- Can the flow of funds from L1 to L2 be manipulated through alternative and unexpected pathways?

1.3. Non-goals and Limitations

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

- The migration logic that generates the L2 genesis state
- 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.

1.4. Results

During our assessment on the scoped Facet Bridge contracts, we discovered one finding, which was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Facet Protocol in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	1

2. Introduction

2.1. About Facet Bridge

Facet Protocol contributed the following description of Facet Bridge:

> Facet Protocol is an EVM-compatible rollup that offers a novel approach to scaling Ethereum without introducing new dependencies or trust assumptions. As a fork of Optimism's OP Stack, the framework behind many of the largest Layer 2 rollups, Facet differentiates itself by eliminating all sources of centralization and privilege, resulting in the first rollup that preserves Ethereum's liveness, censorship resistance, and credible neutrality.
>
> Facet Bridge, the focus of this audit, is a trust-minimized bridge built as a fork of OP Bridge. Given the Facet Protocol itself does not force the use of any enshrined (built-in) contracts, Facet Bridge is a third-party application developed to demonstrate how trust-minimized architecture can operate on Facet.

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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Facet Bridge Contracts

Type	Solidity, Go
Platform	EVM-compatible
Target	facet-optimism
Repository	https://github.com/0xFacet/facet-optimism ↗
Version	`331bb685d91447e76b79aee34fea5d668a77e925`

Programs	`op-node/rollup/driver/state.go`
	`op-node/rollup/engine/events.go`
	`op-node/rollup/status/status.go`
	`op-node/rollup/sync/start.go`
	`op-node/service.go`
	`packages/contracts-bedrock/src/L1/L1CrossDomainMessenger.sol`
	`packages/contracts-bedrock/src/L1/L1StandardBridge.sol`
	`packages/contracts-bedrock/src/L1/OptimismPortal.sol`
	`packages/contracts-bedrock/src/L2/L2StandardBridge.sol`
	`packages/contracts-bedrock/src/L2/L2ToL1MessagePasser.sol`
	`packages/contracts-bedrock/src/libraries/Constants.sol`
	`packages/contracts-bedrock/src/libraries/GasPayingToken.sol`
	`packages/contracts-bedrock/src/libraries/LibFacet.sol`
	`packages/contracts-bedrock/src/universal/StandardBridge.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 10 person-days. The assessment was conducted by two consultants over the course of one calendar week.

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:

Filippo Cremonese
Engineer
filippo@zellic.io

Kamensec
Engineer
dimitri@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 17, 2024	Kick-off call
October 17, 2024	Start of primary review period
October 23, 2024	End of primary review period

3. Detailed Findings

3.1. Deploy script uses incorrect cast subcommand

Target	scripts/getting-started/config.sh		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Deploy scripts make use of `cast block latest` instead of `cast block finalised` to obtain relevant block and timing information. This may lead to unintended deployment parameters if a block is not finalized, parameters are stored, and a reorg happens before the code is considered deployed. Misconfigurations to timing variables may cause issues in the L2OutputOracle, a critical contract that is responsible for the integrity of transfers through the bridge.

Impact

The L2OutputOracle relies on `startingTimestamp` and `startingBlockNumber`, which may be set before the current timestamp if reorgs happen during deployment.

Recommendations

Consider using `cast block finalised` to ensure reorgs do not impact deployment.

Remediation

This issue has been acknowledged by Facet Protocol, and a fix was implemented in commit `89ebb6e6` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Changes to Optimism bridge contracts

This section documents the most notable changes made to the Optimism bridge contracts. Note that some trivial changes may not be explicitly discussed, including, for example, changes made to support access of contract storage in a way compatible with the upgradability pattern adopted by Facet Bridge.

StandardBridge

This is the base contract from which both the L1 and the L2 bridge contracts inherit from.

The high-level logic flow of ERC-20 bridging operations in both directions follows close to the previous implementation. The most significant difference between Facet and a default Optimism chain is the need to implement replay logic for L1 → L2 bridging operations, since L2 transactions may be dropped in some edge cases. Refer to section 5.1. ↗ for a more in-depth discussion.

To support the replay logic, this contract has seen the introduction of two functions:

`replayERC20Deposit`

This unpermissioned function can be invoked exclusively on the L1 bridge. It allows to trigger a replay of the L2 transaction finalizing an L1 → L2 bridging operation.

`finalizeBridgeERC20Replayable`

This function replaces the `finalizeBridgeERC20` entry point on the L2 bridge normally invoked by the L1 bridge contract. The function adds a check that ensures the bridging operation being attempted has not already been processed. If not already finalized, the function marks the operation as such and invokes the unmodified `finalizeBridgeERC20` function, which finalizes the operation by minting or releasing the assets to the recipient.

L1StandardBridge

The only major modification to the L1StandardBridge contract has been the addition of a `depositWethTo` external function, which allows to bridge native L1 ETH by first wrapping it into WETH and bridging the wrapped asset as any other ERC-20.

In the version of the code being initially reviewed, this function had an unintuitive quirk. Since the function receives ETH and wraps them, the bridge contract has ownership of the WETH. Therefore, the ERC-20 bridging was initiated using as a sender address the address of the bridge. This flow was secure but meant that the funds appeared as if they originated from the bridge contract, for ex-

ample in events emitted by the bridge contracts. Using the bridge address as source was necessary because the address of the sender is used by the internal function `_initiateERC20Deposit` as the source of the bridged assets (as the `from` parameter in a `transferFrom` call).

This quirk was addressed in commit `10d1a797` ↗ by adding an argument to `_initiateERC20Deposit`, which allows to skip the `transferFrom` call. This allows `depositWethTo` to call the internal function with the correct sender address without triggering a transfer of WETH from the sender. This modification does not add a security issue, since all other usages of the internal function correctly trigger the ERC-20 transfer.

L2StandardBridge

The L2StandardBridge contract logic had no significant modifications.

L1CrossDomainMessenger

This contract was modified to disable the `_sendMessage` function. This modification reflects on the CrossDomainMessenger contract, effectively disabling the external `sendMessage` function, normally used on OP-based L1s to send a message to the L2.

Since Facet uses a different mechanism to send messages to the L2, the function was disabled by reverting with an error message inviting to use the `LibFacet.sendFacetTransaction` function instead.

An additional effect of disabling this function is that the L1 bridge functions normally used to initiate native L1 ETH bridging (`bridgeETHTo`, `bridgeETH`, `depositETH`, `depositETHTo`, and the fallback receive function) will revert, as they all use `_sendMessage`. This is intended, since Facet L2 native currency is not ETH equivalent. Users are meant to bridge WETH instead.

4.2. Changes to the OptimismMintableERC20 contract

This section documents the most notable changes made to the Optimism OptimismMintableERC20 contract. Some trivial changes may not be explicitly discussed, including, for example, changes made to support access of contract storage in a way compatible with the upgradability pattern adopted by Facet.

Immutable parameters changed to storage variables

The standard Optimism contract stores the address of the remote token and of the bridge as immutable parameters — set by the constructor directly in the contract bytecode at initialization time. This change was introduced to support the storage-access pattern chosen by Facet. However, we note that the parameters are now technically not immutable anymore; a function `setBridgeAndRemoteToken` allows the owner of the contract to change the addresses of the bridge and remote

token.

New `selfBurn` function

The Facet version of the contracts added a `selfBurn` function, which allows to burn an amount of tokens; the function is safe, as the tokens are taken from `msg.sender`.

Inheritance from FacetERC20

The Facet version of OptimismMintableERC20 inherits from a base contract named FacetERC20. This base contract implements a few trivial getters (name, symbol, decimal), but most importantly it implements overrides to `transferFrom` and `allowance` to support the buddy addresses functionality during migration. This functionality existed in the legacy Facet version and allowed users to delegate permissions to a buddy address which could operate on their tokens on their behalf. Reviewing the buddy functionality was not part of the scope of this engagement. As of commit `f7c6ea66`↗, these overrides only work while the Facet v0 → v1 migration is ongoing, and are automatically disabled afterwards.

The base contract also implements a function `emitTransferEvent`, which is used in the last stage of the migration. The function can only be invoked by the special migration manager contract and emits a bogus transfer event that can be used by off-chain indexers to bootstrap the database that records the owners of the ERC-20.

4.3. Changes to the op-node

Unlike standard OP-stack L2s, Facet op-node does not have the responsibility to act as a sequencer. Facet transactions sequencing order is determined by the order in which transactions and events are submitted on Ethereum, and blocks are created by facet node (a different component from op-node, included in a separate scope). Therefore, op-node only has read access to op-geth. The most substantial changes made to op-node have been made to enable op-node to work in this read-only fashion.

5. System Design

This provides a description of the high-level components of the system and how they interact.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. L1 → L2 flow

This section summarizes the logic flow of funds bridged from L1 (Ethereum) to Facet (L2).

Note: The description ignores events emitted by the contracts that are not strictly needed for the bridge operation.

L1 side

To initiate a deposit, a user must first grant approval for the token to be bridged to the bridge contract. The user then calls the bridge `depositERC20` or `depositERC20To` functions:

```
function depositERC20(
    address _l1Token,
    address _l2Token,
    uint256 _amount,
    uint32 _minGasLimit,
    bytes calldata _extraData
) external virtual onlyEOA { /* ... */ }

function depositERC20To(
    address _l1Token,
    address _l2Token,
    address _to,
    uint256 _amount,
    uint32 _minGasLimit,
    bytes calldata _extraData
) external virtual { /* ... */ }
```

The two functions invoke the same underlying function `_initiateBridgeERC20`; the only difference between the two is `depositERC20` implicitly sets the address of the receiver of the funds to the address of the sender.

The `_initiateBridgeERC20` function performs the two critical operations needed to initiate the L1 → L2 bridging: 1) taking funds from the sender address and 2) emitting an event that triggers a transaction on the L2 invoking the destination bridge.

How funds are taken from the sender address varies on the nature of the source token. If the source token is a regular ERC-20 that originates on the L1, the amount to be bridged is taken from the sender

using `transferFrom`.[1] The source token could also be a representation of an ERC-20 that originated and was first bridged from the L2 — in other words, the ERC-20 could be a token minted by the bridge when an L2-native asset has bridged to the L1. The bridge assumes this is the case if the token contract declares support for the `IOptimismMintableERC20` or `ILegacyMintableERC20` interfaces. In this case, the bridge burns the tokens directly from the user balance.

The function then computes a unique deposit ID. This deposit ID is associated in a map to the hash of a struct describing the deposit (the addresses of the source and destination tokens, the addresses of the source and recipient, the asset amount, and an arbitrary bytestring that can be used freely, e.g., to facilitate off-chain processing). Therefore, the bridge maintains a mapping with one record for each successful L1 deposit.

Finally, `_initiateBridgeERC20` invokes `replayERC20Deposit`, which uses `LibFacet.sendFacetTransaction` to invoke the L2 bridge contract and finalize the bridging operation:

```
LibFacet.sendFacetTransaction({
    gasLimit: 500_000,
    to: address(otherBridge),
    data: abi.encodeWithSelector(
        this.finalizeBridgeERC20Replayable.selector,
        _depositId,
        _payload
    )
});
```

L2 side

The transaction triggered by `replayERC20Deposit` generates a call to the `finalizeBridgeERC20Replayable` function of the L2 bridge contract, which receives the deposit ID identifying the operation and the payload containing the operation parameters (tokens, source and recipient addresses, amount, and extra data).

The `finalizeBridgeERC20Replayable` function ensures that the call originates from the L1 bridge contract; this establishes the trustworthiness of the ID and payload.

It then ensures that the deposit ID is not already recorded as processed and marks it as such, preventing attempts to repeat the same deposit multiple times.

Finally, `finalizeBridgeERC20` is called; this function is responsible for minting or releasing the locked destination assets held in escrow to the recipient.

Under normal operation, if the source asset was an L1-native ERC-20, the destination asset is set to the address of an OptimismMintableERC20 contract owned by the bridge, which represents the asset being bridged from the L1. After ensuring that the L2 destination asset matches the L1 source

[1] The bridge also maintains a mapping that stores the amount deposited for each pair of L1/L2 tokens. This mapping is needed to prevent spoofed balances and is updated when a native ERC20 enters or exits the bridge custody.

asset, the bridge invokes the L2 asset `mint` function to mint tokens that represent the bridged L1 assets; the minted tokens are credited to the recipient chosen by the L1 sender.

Alternatively, the source asset could be an OptimismMintableERC20 contract representing an L2-native ERC-20 that was bridged to the L1. In this case, the destination asset would be set to the address of the L2-native ERC-20 asset, which the bridge is holding in escrow. The amount being bridged is released to the recipient via `safeTransfer`.

Sequence diagram

The following sequence diagram shows the logic flow of an L1 to L2 bridging operation.

```
sequenceDiagram
    actor U as User
    participant L1B as L1StandardBridge
    participant L2B as L2StandardBridge

    U ->> L1B: depositERC20(l1Token, l2Token, amount, ...)

    L1B ->> L1B: _initiateBridgeERC20

    alt Source token is L1 native ERC20
        note over L1B: safeTransferFrom amount to bridge from
    sender<br>Increase variable keeping tack of deposit flows
    else Source token was bridged from L2
        note over L1B: Burn amount of L1 token to bridge from sender balance
    end

    note over L1B: Generate unique deposit ID
    note over L1B: Compute deposit payload (L1/L2 tokens, sender/recipient
    addr. and amount)
    note over L1B: Associate deposit payload hash with deposit ID

    L1B ->> L1B: replayERC20Deposit(depositID, payload)

    L1B ->> L1B: LibFacet.sendFacetTransaction(<call
    finalizeBridgeERC20Replayable on L2 bridge>)

    note over L1B: Event emitted triggers a transaction on Facet L2

    L1B -->> L2B: finalizeBridgeERC20Replayable(depositId, payload)

    note over L2B: Check sender is L1 bridge<br>Check deposit not already
    processed<br>

    L2B ->> L2B: finalizeBridgeERC20(...)
```

```
    alt Destination token represents native L1 ERC20
        note over L2B: Ensure source and destination tokens match<br>Mint
    tokens to recipient
    else Destination token is native L2 ERC20
        note over L2B: Decrease variable keeping track of
    deposits<br>safeTransfer bridged amount to recipient
    end
```

Replaying failed L2 transactions

Unlike most L2s, Facet does not have a sequencer queue; all Facet transactions originating in a given Ethereum block are processed in a Facet L2 block.

However, the Facet L2 blocks have a total gas limit. At the time of this writing, the exact limit was not finalized, but the Facet development team indicated the limit would be very high compared to ordinary L1s. Nevertheless, the limit could potentially be reached, in which case L2 transactions would be rejected without being rescheduled for execution in a subsequent block.

Facet Bridge accounts for this eventuality by allowing anyone to repeat the triggering of the L2 flow of any L1 deposit. Anyone can invoke the `replayERC20Deposit` on the L1 bridge using the deposit ID and payload of a successful deposit, which will generate an L2 transaction calling `finalizeBridgeERC20Replayable` to finalize that deposit.

This mechanism is secure because the deposit ID and payload are validated against the recorded successful deposits by the L1 bridge and because the L2 bridge ensures that each deposit ID is only processed once.

Native ETH bridging

Facet Bridge exclusively supports bridging ERC-20 assets; native ETH can be bridged by wrapping it using a WETH contract and following the regular ERC-20 flow. Facet Bridge provides a convenience function, `depositWethTo`, which performs the wrapping on behalf of the user.

5.2. L2 → L1 flow

This section summarizes the logic flow of funds bridged from L2 (Facet) to L1 (Ethereum).

Note: the description ignores events emitted by the contracts which are not strictly needed for the bridge operation.

L2 side

There are three types of bridge-compatible tokens:

1. Standard ERC-20 tokens
2. `IOptimismMintable` legacy tokens
3. `IOptimismMintable` remote tokens

There are two types of bridging in L2 to L1 transactions:

1. ERC-20 native to L1. These are burned from the contract on L2 and unlocked on L1.
2. ERC-20 native to L2 (and not compatible with `IOptimismMintableERC20` or `ILegacyMintableERC20`). These tokens are escrowed in this contract and created on L2 for use by intended recipient.

Both scenarios are determined through four externally facing functions.

Legacy tokens can be withdrawn through the L2StandardBridge as follows:

```
function withdraw(
    address _l2Token,
    uint256 _amount,
    uint32 _minGasLimit,
    bytes calldata _extraData
)
    external
    payable
    virtual
    onlyEOA
{ /* .. */ }

function withdrawTo(
    address _l2Token,
    address _to,
    uint256 _amount,
    uint32 _minGasLimit,
    bytes calldata _extraData
)
    external
    payable
    virtual
{ /* ... */ }
```

The two functions invoke the same underlying function `_initiateWithdrawal()`, which in turn calls `_initiateBridgeERC20()` with `withdrawTo` controlling the `_to` input parameter instead of using `msg.sender`.

Alternatively, `bridgeERC20()` and `bridgeERC20To()` can be called directly on the StandardBridge, which is inherited from by the L2StandardBridge.

```
function bridgeERC20To(
    address _localToken,
    address _remoteToken,
    address _to,
    uint256 _amount,
    uint32 _minGasLimit,
    bytes calldata _extraData
) public virtual { /* */ }

function bridgeERC20(
    address _localToken,
    address _remoteToken,
    uint256 _amount,
    uint32 _minGasLimit,
    bytes calldata _extraData
) public virtual onlyEOA {/* ... */}
```

Both these functions also call `initiateBridgeERC20()`. This function burns or escrows tokens according to whether the token is considered native, legacy, or remote.

If the source token is a native ERC-20 (i.e., is not compatible with `IOptimismMintableERC20` or `ILegacyMintableERC20`) that originates on the L2, the amount to be bridged is taken from the sender using `transferFrom`.[2]

If the token is an `IOptimismMintableERC20` or `ILegacyMintableERC20`, both options will end up burning tokens on L2 and then minting tokens on L1. When called on L2, the `initiateBridgeERC20()` function passes a message through the CrossDomainMessenger on L2. This results in an `initiateWithdrawal()` call on the L2ToL1MessagePasser. This, in turn, creates a withdrawal hash stored in the `sentMessages` along with an emitted event `MessagePassed`.

```
bytes32 withdrawalHash = Hashing.hashWithdrawal(
    Types.WithdrawalTransaction({
        nonce: messageNonce(),
        sender: msg.sender,
        target: _target,
        value: msg.value,
        gasLimit: _gasLimit,
        data: _data
    })
);

sentMessages[withdrawalHash] = true;
```

[2] The bridge also maintains a mapping that stores the amount deposited for each pair of L1/L2 tokens. This mapping is needed to prevent spoofed balances and is updated when a native ERC20 enters or exits the bridge custody.

```
emit MessagePassed(messageNonce(), msg.sender, _target, msg.value, _gasLimit,
    _data, withdrawalHash);
```

The `_data` is encoded as follows: `abi.encodeWithSelector(this.relayMessage.selector, messageNonce(), msg.sender, _target, msg.value, _minGasLimit, _message)`. This allows the `_message` to be relayed through the OptimismPortal and matching L1CrossDomainMessenger.

L1 side

The transaction triggered by `initiateWithdrawal()` generates a call to the `finalizeBridgeERC20()`. The `finalizeBridgeERC20()` on the L1 contract checks that the call is being made through the L2 bridge attempts to unlock, minting tokens if the token is an `IOptimismMintableERC20` or `ILegacyMintableERC20` or using `safeTransfer()` where the token is a native ERC-20 token.

Sequence diagram

The following sequence diagram shows the logic flow of an L1 to L2 bridging operation.

```
sequenceDiagram
    actor U as User
    participant L2B as L2StandardBridge
    participant L2ToL1MP as L2ToL1MessagePasser
    participant OpPort as OptimismPortal
    participant CDM as CrossDomainMessenger
    participant L1B as L1StandardBridge

    U ->> L2B: bridgeERC20(address _localToken, address _remoteToken,...)

    alt Source token is L2 native ERC20
        note over L2B: safeTransferFrom amount to bridge from
    sender<br>Increase variable keeping tack of deposit flows
    else Source token was bridged from L1
        note over L2B: Burn amount of L2 token to bridge from sender balance
    end

    L2B ->> L2ToL1MP: initiateWithdrawal(address _target, uint256 _gasLimit)
    L2B ->> L1B: finalizeBridgeERC20(...)

    note over OpPort: an L2 outputRoot is stored
    note over OpPort: If the relevant withdrawal is part of the outputRoot it
    is added to provenWithdrawals
    note over OpPort: From here we can
    finalizeWithdrawalTransaction(Types.WithdrawalTransaction memory _tx)
```

```
U ->> OpPort: finalizeWithdrawalTransaction(Types.WithdrawalTransaction
memory _tx)

note over L1B: Event emitted triggers a transaction on Facet L2

OpPort -->> CDM: relayMessage(_nonce, ....)

CDM ->> L1B: finalizeBridgeERC20(...)

alt Destination token represents native L2 ERC20
    note over L1B: Ensure source and destination tokens match<br>Mint
tokens to recipient
else Destination token is native L1 ERC20
    note over L1B: Decrease variable keeping track of
deposits<br>safeTransfer bridged amount to recipient
end
```

Replaying failed L2 transactions

Transactions on L1 will require replaying if they revert during `relayMessage()` on the L1's CrossDomainMessenger contract. Users are able to replay transactions that failed for one of two reasons:

1. The transaction did not have enough minimum gas to call the L1StandardBridge contract, or
2. The external call made to the destination contract reverted.

In both circumstances, the user is able to reissue the message again to retry finalization of the bridging operation.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to facet chain.

During our assessment on the scoped Facet Bridge contracts, we discovered one finding, which was informational in nature.

6.1. 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.