Zellic

Prepared for
Tom Lehman
0xFacet

Prepared by
Filippo Cremonese
Seunghyeon Kim
Zellic

October 21, 2024

Facet Migrations

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 0xFacet from October 14th to October 15th, 2024. During this engagement, Zellic reviewed Facet node'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:

- Do Facet's contracts align with UniswapV2 functionality, except for specific, intentional changes?
- Are upgradeability mechanisms secure? Do they introduce vulnerabilities?
- Is the newly introduced router-fee mechanism implemented securely?
- Do the new router-fee and variable-fee-rate mechanisms function properly? Are they secure?
- Is token handling in swaps secure against potential draining attacks?
- Do access control mechanisms restrict privileged functions to authorized users?

Expanded Scope: Final Migration Stage

The assessment scope was expanded to include the final migration stage for Facet v0 to v1, which involves initializing token mappings and FacetSwap pairs. This portion of the audit is documented in Section 5.1. ↗. Additional questions for this phase include:

- Is the MigrationManager contract invoked exactly once at the appropriate time to prevent unintended re-execution?
- Do events emitted during migration provide an accurate view of token balances and FacetSwap pair data?
- Do all privileged functions enforce strict access controls to prevent unauthorized actions?
- Is the final migration stage guaranteed to either succeed or halt the chain, preventing state inconsistencies?

1.3. Non-goals and Limitations

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

- The contracts already deployed from the 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.

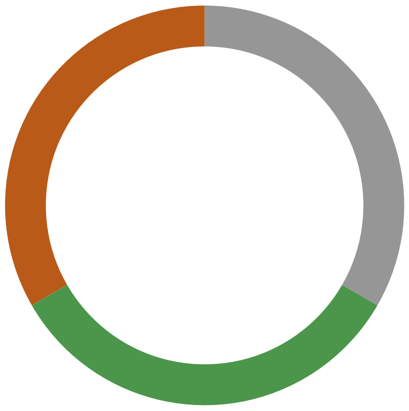
1.4. Results

During our assessment on the scoped Facet node contracts, we discovered three findings. No critical issues were found. One finding was of high impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of 0xFacet in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	1
🟨 Medium	0
🟩 Low	1
⬜ Informational	1



2. Introduction

2.1. About Facet Migrations

0xFacet contributed the following description of Facet node:

> 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 Migration - This audit focuses on the technical approach for state migration from legacy Facet (launched 2023) to the latest, permissionless EVM blockchain. Many EVM tools like the graph rely not just on “state” but also on event emissions, which cannot be migrated via a genesis block. Thus, a "MigrationManager" contract was developed to emit appropriate events (exactly once) needed to accurately reflect final, migrated Facet state.

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 Migrations Contracts

Type	Solidity
Platform	EVM-compatible

Target	facet-node
Repository	https://github.com/0xFacet/facet-node ↗
Version	`b6bf4f07c1d63a1d6cde22db2e13e8b8096e1ba8`

Programs	`FacetSwapPairVdfd.sol`
	`FacetSwapFactoryVac5.sol`
	`FacetSwapRouterV56d.sol`

2.4. Project Overview

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

After the initial audit, the scope of this engagement was extended to include a review the implementation of the final stage of the Facet v0->v1 migration, which involves the MigrationManager contract. This part of the audit is documented in Section 5.1. ↗.

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
fcremo@zellic.io ↗

Seunghyeon Kim
Engineer
seunghyeon@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 14, 2024	Start of primary review period
October 15, 2024	End of primary review period
November 4, 2024	Start of primary review period for final migration stage
November 7, 2024	End of primary review period for final migration stage

3. Detailed Findings

3.1. Arithmetic overflow leading to DOS

Target	FacetSwapPairVdfd		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The Uniswap code explicitly allows arithmetic overflows in the `_update` function. Arithmetic overflows are allowed when calculating the new value of the `price0CumulativeLast` and `price1CumulativeLast` accumulator variables.

Since Facet Swap code targets a modern version of Solidity, which uses checked arithmetic operations by default, the `_update` function does not allow arithmetic overflows and behaves differently from the UniswapV2 code.

Impact

Without allowing overflows, all pair contracts will eventually reach a state of permanent denial of service, since the `price0CumulativeLast` and `price1CumulativeLast` variables only ever increase.

When their value grows to a point where any attempt to update them causes an overflow, all functions of the contract that directly or indirectly invoke `_update` will revert.

The DOS can be recovered by upgrading the contracts.

Recommendations

Allow the specific arithmetic operations intended by the UniswapV2 to overflow silently by using `unchecked` blocks.

Remediation

This issue has been acknowledged by 0xFacet, and a fix was implemented in commit `47318a0b` ↗.

3.2. Router fee is bypassable

Target	FacetSwapRouterV56d		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

Since liquidity-pair contacts can be invoked directly, the fees charged by the router contract can be avoided by manually performing multi-leg swaps.

Impact

The router cannot strictly enforce fee collection.

Recommendations

This can be deemed an acceptable compromise. If router fees must be enforced, the liquidity-pair contract could be modified to enforce all calls to come from the address of the router.

Remediation

This issue has been acknowledged by 0xFacet.

> The current behavior is an acceptable compromise between doing nothing and doing something a lot more complicated. We're going to leave it as-is.

3.3. Router pause state does not affect pair contracts

Target	FacetSwapRouterV56d		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The Facet Swap contracts introduce a pause feature to the router contract, allowing the contract owner to suspend and resume the router contract functionality.

We note, however, that the pause state of the router does not apply to the liquidity-pair contracts, which can be called directly to perform swaps even if the router is paused.

Impact

The original UniswapV2 pair contracts do not have a pause feature. It is unclear under which conditions this feature is intended to be used.

This finding is reported with Informational severity as a potential design oversight. We do not consider this a freestanding security issue, as it cannot directly cause economic damage.

Recommendations

Consider querying the router pause state in the pool contracts to apply it universally across all Facet Swap contracts.

Remediation

This issue has been acknowledged by 0xFacet.

> The pause functionality is intended to affect the Facet Swap interface and our router, it's not intended to disable the entire DEX (which is what pausing the pairs would do), so the current behavior is correct.

4. Discussion

The purpose of this section is to document all functions contained in the contracts subject of this review, comparing them with the UniswapV2 source they were derived from.

4.1. General

The Facet Swap contracts modified the Uniswap contracts to make the following notable changes:

- Introducing contract ownership and upgradability
- Supporting a variable swap fee rate instead of the fixed 30-base-points fee rate applied by Uniswap
  - The fee rate is obtained from the router and configurable by the router contract owner
  - The fee rate is expressed in tenths of base points
- Charging an additional swap fee, at the router level, on the first or last leg of a swap
  - This fee is retained by the router and can be withdrawn by the router contract owner
  - The fee rate is the same as the variable swap fee rate
  - The router requires the first or last leg of a swap to be WETH — the fee is charged on the WETH leg

We note therefore that Facet Swap charges three types of fees:

- A variable-fee rate is charged on each swap and raises the value of LP tokens.
- The same variable-fee rate is applied by the router to the first or last leg of a swap.
- A five-base-points protocol fee can be charged to LP providers when they add liquidity. This fee mechanism was already present in UniswapV2, and it was not altered.

Several trivial changes are repeated throughout the codebase, including the following:

- Supporting storage access in a manner compatible with the chosen upgradability pattern
- Replacing references to Uniswap with references to Facet Swap (e.g., in error messages)
- Replacing SafeMath usages with Solidity built-in checked arithmetic
- Replacing `uint` with the more explicit `uint256`
- Replacing `assert` with `require`
- Replacing implicit return values with explicit return statements

These changes may not be discussed for some functions when they do not have an impact on the contracts' functionality or security.

4.2. Factory

The factory is a smart contract mainly for creating and managing the pairs.

Function `createPair`

This function can be used to deploy a new pool contract for an asset pair. The function was modified from the original Uniswap version to allow Facet Swap pool contracts to be deployed behind a proxy to allow upgradability.

The function behaves as intended, and we did not find any security issues.

We note that the implementation address for the initial version of the pool contract code is derived from the hash of the pool contract creation code; this differs from how contract addresses are normally computed on EVM chains, and it only works because the pool contract is manually deployed at the address derived from the hash of the pool creation code from the genesis state of Facet.

Function `initialize`

This function allows to initialize the factory contract. It can only be called once. It was introduced in the Facet fork of Uniswap. No security issues were found in the function, provided that the function is called in the same transaction that deploys the factory, or that the factory configuration is verified, to ensure no malicious call to `initialize` was made before the legitimate call to `initialize` is made.

Function `allPairsLength`

This function allows to retrieve the number of pool contracts deployed by this factory. It is functionally equivalent to the function from the Uniswap codebase, with only changes needed to support accessing storage in a manner compatible with the chosen upgradability pattern.

Function `getPair`

This function allows to retrieve the address of the pool contract for a given pair of assets. It is functionally equivalent to the function from the Uniswap codebase, with only changes needed to support accessing storage in a manner compatible with the chosen upgradability pattern.

Function `feeTo`

This function allows to retrieve the address that receives the fees charged by the pool contracts. It is functionally equivalent to the function from the Uniswap codebase, with only changes needed to support accessing storage in a manner compatible with the chosen upgradability pattern.

Function `setFeeTo`

This function allows the `feeToSetter` address to change the address that receives the fees accrued by Facet Swap pools. It is functionally equivalent to the function from the Uniswap codebase, with only changes needed to support accessing storage in a manner compatible with the chosen upgradability pattern.

Function `setFeeToSetter`

This function allows the `feeToSetter` address to transfer its own role to another address. It is functionally equivalent to the function from the Uniswap codebase, with only changes needed to support accessing storage in a manner compatible with the chosen upgradability pattern.

Function `lpFeeBPS`

This function allows to retrieve the current fee rate charged by Facet Swap pools. It was introduced in the Facet fork of Uniswap. No security issues were found in the function.

Function `setLpFeeBPS`

This function allows the `feeToSetter` address to change the fees charged by Facet Swap pools. It was introduced in the Facet fork of Uniswap. No security issues were found in the function.

Function `upgradePairs`

This function allows the upgrade admin to upgrade the implementation address of a batch of Pair contracts. It was introduced in the Facet fork of Uniswap. No security issues were found in the function.

We note that the `newSource` argument is ignored, and an empty string is used to upgrade the contract instead. However, the argument appears to be unused in the implementation of the Upgradeable base contract.

Function `upgradePair`

This function allows the upgrade admin to upgrade the implementation address of a Pair contract. It was introduced in the Facet fork of Uniswap. No security issues were found in the function.

4.3. Router

The router is a smart contract for swapping the tokens through the paths.

Function `onUpgrade`

This function was introduced in the Facet Swap contracts. It can be used to reset the contract owner and the pause state of the contract. The function does not enforce access control. We were informed by the Facet team that the router contracts will be present in the genesis state and already initialized to version 3. Since the function is decorated with the `reinitializer(3)` modifier, it will not be possible to call it and change the owner or pause state.

Since failing to deploy the contracts with the correct version would create a severe security issue, we recommend adding tests to the code that generate the genesis state to ensure all routers are initialized correctly and reject calls to `onUpgrade`.

This observation was addressed in commit 8df0c6bf ↗, which removed the `onUpgrade` function from the router contract.

Functions `_swapExactTokensForTokens` and `swapExactTokensForTokens`

The internal function `_swapExactTokensForTokens` is functionally equivalent to the external function `swapExactTokensForTokens` from the Uniswap codebase. Since the function is now internal, it cannot be directly invoked, but must be invoked through the public `swapExactTokensForTokens` function. It performs a swap of an exact quantity of input tokens, in exchange for a minimum desired quantity of output tokens.

The `swapExactTokensForTokens` function is the new entry point for this functionality and differs from the original entry point in the following ways:

- The initial input or the final output asset must be WETH.
- A fee is charged either on the input or output amount, depending on which one is WETH; if both are WETH, the fee is charged on the input.

The fee is determined by the `protocolFeeBPS` parameter, which is configurable by the contract owner.

We note that the fee logic is not clear, particularly in the case where both the input and output assets are WETH. We recommend adding an assertion that ensures the WETH balance of the contract at the end of the function execution is greater than or equal to the balance at the start, as a defense-in-depth mechanism to prevent possible exploits from stealing protocol fees. This recommendation was adopted in commit 47318a0b ↗.

Functions `_swapTokensForExactTokens` and `swapTokensForExactTokens`

The internal function `_swapTokensForExactTokens` is functionally equivalent to the external function `swapTokensForExactTokens` from the Uniswap codebase. Since the function is now internal, it cannot be directly invoked, but must be invoked through the public `swapTokensForExactTokens` function. It allows to perform a swap of a maximum given quantity of input tokens, in exchange for a specific quantity of output tokens.

The `swapTokensForExactTokens` function is the new entry point for this functionality and differs from the original entry point in the following ways:

- The initial input or the final output asset must be WETH.
- A fee is charged either on the input or output amount, depending on which one is WETH; if both are WETH, the fee is charged on the input.

The fee is determined by the `protocolFeeBPS` parameter, which is configurable by the contract owner.

We note that, similarly to `swapExactTokensForTokens`, the fee logic is not clear, particularly in the case where both the input and output assets are WETH. We recommend adding an assertion that ensures the WETH balance of the contract at the end of the function execution is greater than or equal to the balance at the start, as a defense-in-depth mechanism to prevent possible exploits from stealing protocol fees. This recommendation was adopted in commit `47318a0b` ↗.

Function `initialize`

This function can be used to initialize the contract. It does not have a direct counterpart in the original Uniswap contracts.

It does not have security issues that can be exploited without preconditions; however, we note it must be called in the same transaction where the router contract is deployed, or, alternatively, the contract configuration must be checked to ensure no malicious call was made between the time the contract is deployed and the time it is initialized (in which case. the contract must be redeployed and reinitialized).

Functions `addLiquidity` and `_addLiquidity`

The public function `addLiquidity` corresponds to the external function `addLiquidity` in the Uniswap contracts. It is functionally equivalent and can be used to add liquidity to a pool in exchange for LP tokens.

The internal function `_addLiquidity` also matches its counterpart in the original code.

Function `removeLiquidity`

This function can be used to burn LP tokens for a given pair, getting back the corresponding assets.

Function `_swap`

This internal function can be used to execute a chain of swaps. It is functionally equivalent to the original Uniswap version, with the notable exception of the addition of a maximum swap path length. By default, the limit is initialized to three by the `initialize` function, but it is read from the router contract storage, making it potentially configurable. However, we note that at the moment, there is no functionality that allows to change this limit, besides upgrading the contract.

Function `_safeTransferFrom`

This function can be used to perform a `safeTransfer` ERC-20 call. The function does not match the version used by the original contracts and is less compatible with tokens that do not correctly implement the ERC-20 standard. This is not an issue provided that all assets used with Facet Swap correctly implement the ERC-20 standard.

Functions `getAmountsOut` and `getAmountsIn`

These functions can be used to obtain the results of, respectively, `getAmountOut` and `getAmountIn` for each swap in a chain of swaps.

The bodies of the functions were inlined from UniswapV2Library. They are almost functionally equivalent to the original functions. The Facet versions introduced a limit on the swap path length.

Function `getAmountOut`

This function can be used to get the maximum output amount to be expected when swapping a specified amount of input asset, given the values of the pool reserves.

The function was inlined from UniswapV2Library, with modifications required to support the variable swap fee rate, which is obtained from the factory contract.

Function `getAmountIn`

This function can be used to get the minimum input amount needed to perform a swap resulting in a specified amount of output asset, given the values of the pool reserves.

The function was inlined from UniswapV2Library, with modifications required to support the variable swap fee rate, which is obtained from the factory contract.

Other functions inlined from UniswapV2Library

The following functions were inlined from UniswapV2Library and remain functionally equivalent to the original code. Minor changes were made to support accessing storage in a manner compatible with the chosen upgradability pattern and to replace references to Uniswap with references to Facet Swap.

- `quote`, which is used to convert an amount of asset into the counterpart, given the current reserves — note that this function does not actually perform a swap and does not account for fees or slippage
- `getReserves`, which is used to get the value of the reserves (cached balance) for a given pair of assets
- `pairFor`, which is used to get the address of the pool for a given pair of assets from the factory contract
- `sortTokens`, which is used to sort tokens in lexicographic order

Function `calculateFeeAmount`

This function can be used to compute the fees charged by the router to a given asset amount.

Function `updateProtocolFee`

This function can be used by the contract owner to update the fee percentage charged by the router.

Function `withdrawFees`

This function can be used by the contract owner to withdraw fees accrued by the router.

Functions `pause` and `unpause`

These functions can be used to pause and unpause the contract. They were introduced by Facet Swap and work as intended work as intended with respect to the router contract, preventing swaps and adding or removing liquidity through the router.

We note, however, that the pause feature does not prevent calling the pools directly.

Function `userStats`

This function can be used to retrieve information about a given user related to a token pair:

- User balances for the two assets
- User LP token balance for the corresponding pool
- Address of the pool
- Pool reserve values
- Names of the two assets

Function `factory`

This function can be used to retrieve the address of the factory contract. The Uniswap contract did not contain this function explicitly, as it was automatically generated by the compiler due to the `factory` state variable being `public`. Since the Facet Swap contracts adopt the chosen upgradability pattern, public getters are not generated by the Solidity compiler and had to be manually implemented.

4.4. Pair

The pair is a smart contract pool for the tokens. The users can swap the tokens directly using `swap` function here.

Function `initialize`

This function does not have a direct counterpart in the UniswapV2 code, with the closest match being the contract constructor. It is invoked by the factory contract to initialize the contract ERC-20 LP token, the factory address, reentrancy lock state, and the Upgradeable contract administrator address.

It is functionally equivalent to the UniswapV2 `initialize` function, with the only changes being due to storage access, replacing Uniswap references with references to Facet, and initializing the reentrancy lock and the Upgradeable contract from which the Pair contract inherits from.

Function `init`

This function is used by the factory contract to initialize the addresses of the tokens the pool operates with.

It is functionally equivalent to the UniswapV2 `initialize` function, with trivial changes to access storage while supporting the upgradability pattern and to replace Uniswap references with references to Facet. The function was also renamed from `initialize` to `init`.

Function `getReserves`

This function is used to get the cached reserve values for the pair as well as the timestamp when they were last cached.

It is functionally equivalent to UniswapV2, with the only changes being due to storage access.

Function `_safeTransfer`

This function is not equivalent to the version used by the Uniswap contracts. The Uniswap version of the function is as follows:

```
function _safeTransfer(address token, address to, uint value) private {
    (bool success, bytes memory data)
    = token.call(abi.encodeWithSelector(SELECTOR, to, value));
    require(success && (data.length == 0 || abi.decode(data, (bool))),
    'UniswapV2: TRANSFER_FAILED');
}
```

While the Facet version is as follows:

```
function _safeTransfer(address token, address to, uint value) private {
    bool result = ERC20(token).transfer(to, value);
    require(result, "FacetSwapV1: TRANSFER_FAILED");
}
```

The Facet version is less compatible with nonstandard ERC-20 implementations where the `transfer` function does not return a boolean value, such as the ERC-20 deployed for USDT on Ethereum.

This modification is not necessarily problematic, provided that all tokens used on Facet Swap correctly implement the ERC-20 standard.

This observation was addressed in commit b13be251 ↗, which switched the `_safeTransfer` implementation to the following code, which behaves like Uniswap:

```
function _safeTransferFrom(
    address token,
    address from,
    address to,
    uint256 value
) internal {
    // bytes4(keccak256(bytes('transferFrom(address,address,uint256)')));
    (bool success, bytes memory data)
    = token.call(abi.encodeWithSelector(0x23b872dd, from, to, value));
    require(
        success && (data.length == 0 || abi.decode(data, (bool))),
        'TransferHelper::transferFrom: transferFrom failed'
    );
}
```

Function `_update`

This function is used to update the cached reserve values and cumulative price.

It is mostly functionally equivalent to UniswapV2, with the only changes being due to storage access, replacing `UQ112x112` fixed-point math usages with built-in checked arithmetic, and replacing Uniswap references with references to Facet. An additional event is also emitted, containing the cached reserve values before they are updated.

The Uniswap code explicitly states that some arithmetic overflows present in the code were intended and desired; the Facet Swap code targets a modern version of Solidity, which uses checked arithmetic operations and does not allow silent overflows. Without allowing overflows, the contract is destined to break, since the `price0CumulativeLast` and `price1CumulativeLast` variables increase for each block. When the attempt to update them causes an overflow, all functions of the contract that directly or indirectly invoke `_update` will revert.

Function `encode`

This function is used to encode a number into the encoding used by the `UQ112x112` fixed-point library (effectively multiplying the number by 2^{112}).

This function was inlined from the `UQ112x112` fixed-point library and behaves identically to the original version.

Function `uqdiv`

This function was inlined from the `UQ112x112` fixed-point library and behaves identically to the original version. However, we note that the function is unused and could be removed.

Function `_mintFee`

This function is used to mint an optional protocol fee (charged when minting or burning liquidity) in the form of LP tokens, which are transferred to a configurable fee recipient. The minted LP tokens dilute the other LP tokens, essentially granting a share of the pool liquidity to the fee recipient.

It is functionally equivalent to UniswapV2, with the only changes being due to storage access, replacing SafeMath usages with built-in checked arithmetic, and replacing Uniswap references with references to Facet.

Function `mint`

This function is used to provide liquidity to the pool by transferring assets to it in exchange for LP tokens.

It is functionally equivalent to UniswapV2, with the only changes being due to storage access, replacing SafeMath usages with built-in checked arithmetic, and replacing Uniswap references with references to Facet.

Function `burn`

This function is used to burn LP tokens in exchange for a corresponding amount of assets.

It is functionally equivalent to UniswapV2. Changes were made to allow access to contract storage while being compatible with the chosen upgradability pattern used in the Facet version of the contracts. Usages of SafeMath were removed in favor of built-in Solidity 0.8.0+ safe math. References to Uniswap were replaced with references to Facet.

Function `swap`

This function is used to perform a swap. It is functionally equivalent to UniswapV2, with one notable exception; instead of supporting a fixed 30-base-point fee, Facet supports dynamic fees retrieved from the factory contract. Notably, the k-invariant assertion has not been modified.

Additionally, changes were made to allow access to contract storage while being compatible with the chosen upgradability pattern used in the Facet version of the contracts. Usages of SafeMath were removed in favor of built-in Solidity 0.8.0+ safe math. References to Uniswap were replaced with references to Facet.

Function `skim`

This function is used to take away excess balances owned by the contract, meaning the difference between the current balance of the contract and the cached reserve values.

It is functionally equivalent to UniswapV2; changes were made to access contract storage due to use of the chosen upgradability pattern in the Facet version of the contracts. Additionally, usages of SafeMath were removed in favor of built-in Solidity 0.8.0+ safe math.

Function `sync`

This function is used to update the cached reserve values stored by the contract.

It is functionally equivalent to UniswapV2; the only changes are related to storage access due to use of the chosen upgradability pattern in the Facet version of the contracts.

Function `sqrt`

This function does not exist in the original UniswapV2Pair contract; it invokes the floating-point square-root precompile introduced for backwards compatibility with the legacy Rubidity language used in the development stage of Facet.

Function `min`

This function does not exist in the original UniswapV2Pair contract; it trivially returns the minimum between the two input numbers.

5. Scope extension

5.1. Final migration stage

The scope of this engagement was extended to include the review the implementation of the final stage of the Facet v0->v1 migration, which involves the MigrationManager contract. This L2 system contract is invoked automatically on the first L2 block to complete the migration. Note that this audit extension did not cover the entire migration process – most of the migration is performed by other out of scope code which creates the initial genesis state.

This review also assumed that the MigrationManager contract storage was correctly initialized to reflect the tasks required to complete the migration, including the set of ERC20 tokens, the mapping of ERC20 token holders, the set of ERC721 contracts, the mapping of ERC721 token IDs, the set of FacetSwap factories and the mapping of factories to deployed FacetSwap pairs.

The final migration stage is needed to emit events that allow offchain indexers to build an up to date view of the balances of the Facet ERC20/ERC721/FacetSwap contracts (and of the existing pairs). The MigrationManager contract invokes some privileged functions on the respective contracts which in turn emit the events.

The engagement fixed the following objectives:

- ensuring MigrationManager is invoked once, and exactly once, at the appropriate time
- ensuring the events emitted during the migration provide an accurate view of the token balances and FacetSwap pair data
- ensuring all privileged functions enforce strict access controls to prevent unauthorized actions
- ensuring that if the migration fails for any reason the chain cannot progress, preventing state inconsistencies

The audit was performed by reviewing the changes between base commit 75877a54 ↗ and commit 844777a2 ↗, and only considering relevant changes to the following files:

- `contracts/src/predeploys/MigrationManager.sol`: system contract called automatically by the node on the first block (from the system address `0xDeaDDEaDDeAdDeAdDEAdDEaddeAddEAdDEAd0001`) to finish the migration. Causes events to be emitted, allowing token holders and their balances to be indexed.
- `FacetERC20.sol`, `FacetERC721.sol`, `MigrationLib.sol` in `contracts/src/libraries/`: invoked by the MigrationManager to emit events
- `contracts/src/predeploys/FacetSwapFactoryVac5.sol`: invoked by the MigrationManager to emit events
- `lib/geth_driver.rb` and `app/models/facet_transaction.rb`: part of the Facet node core, these files contain the code that triggers the final migration stage

Discussion of MigrationManager `executeMigration` function

The `executeMigration` function is invoked on the first block of Facet L2 to perform the final migration steps. The function can only be invoked from the system address `0xDeaDDEaDDeAdDeAdDEAdDEaddeAddEAdDEAd0001`, and only until the migration has not finished.

The migration is performed in batches, emitting 100 events per batch[1]; the `executeMigration` function is invoked repeatedly until the migration is completed. The first time the function is invoked the total number of events to emit is computed. This number is used to compute the number of batches needed and as a sanity check to ensure all needed events have been emitted.

`executeMigration` processes the pending tasks in a specific order, by invoking functions that perform migration tasks for FacetSwap pairs, ERC20 tokens, and ERC721 tokens.

The functions are always invoked; they process sets of pending factories/pairs, ERC20 tokens, and ERC721 tokens respectively. Once an element has been processed, it is removed from the set to avoid processing it again. After each action that can emit an event, the functions check whether the batch is complete via the `batchFinished` function. If the batch is complete, the functions return without doing further processing. The remaining pending elements in the sets will be processed in subsequent invocations of `executeMigration`.

`processFactories`

This function performs the migration tasks needed to emit events that give offchain indexers information about FacetSwap pairs.

The function iterates over each FacetSwap pair deployed, doing the following:

- call `migrateERC20` to perform the migration of two ERC20 tokens forming the pair
- call `emitPairCreateEventIfNecessary`
- call `migrateERC20` to perform the migration of ERC20 token representing the pair LP tokens
- call `pair.sync()` to ensure the pair liquidity reflects the pair liquidity balance
    - NOTE: this causes two events to be emitted, `PreSwapReserves` and `Sync`, which are not accounted for (or not entirely, since currentBatchEmittedEvents is only incremented once)

The `emitPairCreateEventIfNecessary` is called to emit a `PairCreated` event which allows offchain indexers to record which pairs have been deployed. The function is a no-op if the event is already emitted for a given pair. If the corresponding event has not been emitted yet, the function invokes `emitPairCreated` on the factory that created the pair. The factory ensures that the caller address is the migration manager contract and emits the `PairCreated` event.

The `migrateERC20` function is described in the following section discussing `processERC20Tokens`.

`processERC20Tokens`

This function performs the migration of ERC20 tokens, emitting events that allow offchain indexers to build a view of the balances of all ERC20 holders.

It iterates over the `allERC20Tokens` set, which contains the addresses of all the ERC20 tokens. Note that ERC20 tokens that are included in a FacetSwap pair have already been processed by `processFactories` and removed from `allERC20Tokens`, and are not processed twice.

1 This number does not strictly reflect the number of emitted events. Refer to section 5.1. ↗.

The `migrateERC20` function is called for every ERC20 address in `allERC20Tokens`. The addresses of the holders of the ERC20 token are read from the `erc20TokenToHolders` mapping. If the balance of the holder (read from the ERC20 contract) is greater than zero then the `emitTransferEvent` function is called on the ERC20 contract.

`emitTransferEvent` checks that the caller is the migration manager and emits a `Transfer` event. The event `to` and `amount` fields correspond to the holder address and their balance. The `from` field is set to the zero address. Indexers are therefore able to derive the balances of all ERC20 holders but not the source of the balances from the events emitted during the migration.

`processERC721Tokens`

This function performs the migration of ERC721 tokens, emitting events that allow offchain indexers to build a view of the holders of all ERC721 assets.

It iterates over the address of every deployed ERC721 contract by reading the `allERC721Tokens` set.

The migration for each individual ERC721 contract is handled by the `migrateERC721` function, which iterates over the existing token IDs by reading the set of existing token IDs from the `erc721TokenToTokenIds` mapping. The owner of the asset is retrieved from the ERC721 contract. If the owner returned by the ERC721 contract is not the zero address, the `emitTransferEvent` function is invoked on the ERC721 contract. `emitTransferEvent` checks that the caller is the migration manager contract, and emits a `Transfer` event. The `to` and `id` fields correspond to the owner and the asset ID being processed, while the `from` field is set to the zero address. By reading these events, indexers are therefore able to derive a mapping associating each existing ERC721 asset to its holder (and vice versa), but not the source of the ERC721 assets.

Note that the function cannot distinguish between nonexisting token IDs and ERC721 tokens that were sent to the zero address. Therefore, no events will be emitted for any ERC721 tokens that were transferred to the zero address.

Small gas inefficiency

Every call to `executeMigration` terminates with an event emission:

```
emit BatchComplete(
    currentBatchEmittedEvents,
    remainingEvents,
    calculateTotalEventsToEmit(),
    transactionsRequired()
);
```

This is inefficient, since the third field is computed by calling `calculateTotalEventsToEmit`, and the fourth field calls `transactionsRequired` which also uses `calculateTotalEventsToEmit`. Since `calculateTotalEventsToEmit` performs a number of operations proportional to the total number of ERC20, ERC721, and FacetSwapFactory contracts registered in the system, avoiding this repeated call should save a significant amount of gas and might allow a higher number of events per batch.

Mismatch between `calculateTotalEventsToEmit` and actual number of emitted events

The `calculateTotalEventsToEmit` function computes the number of events expected to be emitted by the migration process. The total number is the sum of:

1. the total number of ERC20 token holders; that is the sum of all the lengths of the sets of holder addresses recorded in the `erc20TokenToHolders` mapping (assuming `allERC20Tokens` has an entry for each key in the mapping)
    - note that these mappings also record ERC20 tokens that represent LP positions in FacetSwap pairs
2. the total number of ERC721 assets; that is the total number of individual token IDs for all ERC721 contracts recorded in the `erc721TokenToTokenIds` mapping (assuming `allERC721Tokens` has an entry for each key in the mapping)
3. the total number of FactorySwap pairs times two

The number of events actually emitted is the sum of the events emitted by three functions.

We note that in some cases there may be a mismatch between the number of events actually emitted by the code, and the counters maintained by the code; this does not impact the successful execution of the migration, and we consider this as a naming issue rather than a security issue with a practical impact.

Events emitted by `processFactories`

For every factory and every pair, this function emits the following events that contribute to the first component of the sum returned by `calculateTotalEventsToEmit`:

- one `Transfer` event for each holder of the first token forming the FacetSwap pair with a greater than zero balance
- one `Transfer` event for each holder of the second token forming the FacetSwap pair with a greater than zero balance
- one `Transfer` event for each holder of a nonzero balance of the ERC20 token representing pair liquidity

Note 1: that these events are only emitted once for each unique (ERC20, holder) pair; they are not emitted multiple times if the token appears in multiple pairs, nor are they emitted again by `processERC20Tokens`. This means that if two pairs USDC<->USDT and USDT<->ETH exist, events recording USDT holders are emitted only once by the migration process.

Note 2: the `currentBatchEmittedEvents` counter is increased even if the holder has a nonzero balance (but the caveat about an ERC20 being processed only once still apply), therefore `currentBatchEmittedEvents` at the end of the migration `currentBatchEmittedEvents` will be increased by a total that matches the total number of holders of unique tokens associated with FacetSwap pairs (either as assets or LP tokens).

The function also emits the following events, which contribute to the third component of the total returned by `calculateTotalEventsToEmit`:

- one `PairCreated` event for each pair
- one `PreSwapReserves` and one `Sync` event for each pair, due to a call to `pair.sync()`

Note 3: the `currentBatchEmittedEvents` counter is increased by one when `PairCreated` is emitted, and by one (and not two) when `PreSwapReserves` and `Sync` are emitted. Therefore, `currentBatchEmittedEvents` undercounts the actual number of events.

Events emitted by `processERC20Tokens`

For every holder of every ERC20 token with a greater than zero balance that was not already processed by `processFactories`, this function emits one `Transfer` event. These events contribute to the first component of the sum returned by `calculateTotalEventsToEmit`.

Note: as with `processFactories`, the `currentBatchEmittedEvents` variable is increased even if the holder has a nonzero balance, therefore `currentBatchEmittedEvents` at the end of the migration `currentBatchEmittedEvents` will be increased by a total that matches the total number of holders of ERC20 tokens that are not associated with any FacetSwap pair.

The total number of times the `currentBatchEmittedEvents` counter is increased is therefore equal to the total number of holders of every unique ERC20 asset recorded in `erc20TokenToHolders` (including holders with zero balances).

Events emitted by `processERC721Tokens`

This function emits one `Transfer` event for every token ID of every ERC721 contract, excluding token IDs owned by the zero address. These events contribute to the second component of the sum returned by `calculateTotalEventsToEmit`.

The `currentBatchEmittedEvents` variable is increased even if the holder is the zero address, therefore `currentBatchEmittedEvents` will be increased by the total number of token IDs recorded in the `erc721TokenToTokenIds` mapping, which might be greater than the actual number of emitted events if any tokens are owned by the zero address.

5.2. Conclusions

No exploitable security issues emerged from this review, and the objectives defined for this scope extension were met successfully. The notes above were acknowledged by the Facet team.

6. Assessment Results

At the time of our assessment, Facet was still under development, and therefore the reviewed code was not deployed to mainnet.

During our assessment on the scoped Facet node contracts, we discovered three findings. No critical issues were found. One finding was of high impact, one was of low impact, and the remaining finding 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.