Zellic

Wasabi

Smart Contract Security Assessment

April 18, 2023

Prepared for:

Wasabi

Prepared by:

Varun Verma and Daniel Lu

Zellic Inc.

Contents

About Zellic 3

1 Executive Summary 4

1.1 Goals of the Assessment 4

1.2 Non-goals and Limitations 4

1.3 Results 4

2 Introduction 6

2.1 About Wasabi 6

2.2 Methodology 6

2.3 Scope 7

2.4 Project Overview 8

2.5 Project Timeline 8

3 Detailed Findings 9

3.1 Iteration over options can prevent withdraws 9

3.2 Solidity versioning permits underflow behavior 11

3.3 Usage of transfer to send ETH can prevent receiving 12

3.4 Protocol does not check return value of ERC20 swaps 14

3.5 Centralization risks 15

3.6 Factory update logic of option NFT enables owner to steal funds 16

3.7 Pool toggling functionality may allow factory owner to lock exercising of options 19

3.8 Fee manager upgrades allow factory owner to change fees and prevent option exercise 21

4 Discussion 23

4.1 AbstractWasabiPool should use upgradable variants of OpenZeppelin contracts . . . 23

4.2 Put options operate on entire collections . . . 23

4.3 Pools require well-behaved NFTs . . . 23

5 Threat Model **24**

5.1 Module: AbstractWasabiPool.sol . . . 24

5.2 Module: WasabiOption.sol . . . 33

5.3 Module: WasabiPoolFactory.sol . . . 34

6 Audit Results **40**

6.1 Disclaimer . . . 40

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Wasabi from March 27th to March 30th. During this engagement, Zellic reviewed Wasabi's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 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:

- Are there any functional bugs that could cause option positions to be formed incorrectly, assets to not be locked properly, or positions to not close correctly?
- Are there any security vulnerabilities that could be exploited by malicious individuals, such as accepting asks/bids without paying, tricking a pool to issue incorrect options, or exercising an option with invalid parameters?

1.2 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.

Due to time constraints, we were unable to explore all possible edge cases during the assessment.

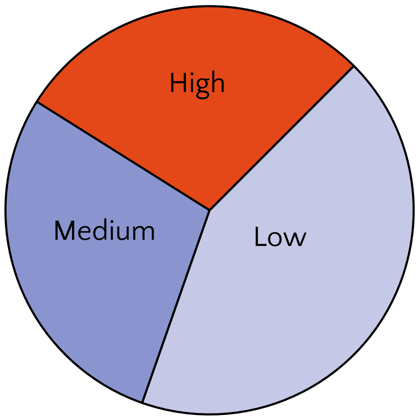
1.3 Results

During our assessment on the scoped Wasabi contracts, we discovered seven findings. No critical issues were found. Of the seven findings, two were of high impact, two were of medium impact, and three were of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Wasabi's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	2
Low	3
Informational	0



2 Introduction

2.1 About Wasabi

Wasabi is a decentralized NFT options protocol. It utilizes option liquidity pools that hold NFTs and ETH (or token) pairs to issue options. Each option is American-style and is fully collateralized. The option positions are represented as NFTs and can be traded on the open market. The options can be exercised, which allows the owner of the option to buy or sell a specific NFT at a predetermined price within a certain time frame.

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 the contracts' 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 code base 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, upgradeability 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:

Wasabi Contracts

Repository	https://github.com/dev-wasabi/wasabi
Version	wasabi: `db5b2fc2cb2e3f64c45a3fc19f2d342451d077a1`
Programs	• AbstractWasabiPool.sol • WasabiOption.sol • WasabiPoolFactory.sol • ERC20WasabiPool.sol • ETHWasabiPool.sol
Type	Solidity
Platform	Ethereum

2.4 Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Varun Verma, Engineer
varun@zellic.io

Daniel Lu, Engineer
daniel@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

March 27, 2023	Kick-off call
March 27, 2023	Start of primary review period
March 30, 2023	End of primary review period
April 18, 2023`	Closing call

3 Detailed Findings

3.1 Iteration over options can prevent withdraws

- **Target**: ETHWasabiPool, ERC20WasabiPool
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

In order for a pool owner to withdraw their assets, the contract must loop through all issued options, even expired ones. However, as the number of options written increases, the gas cost of the loop also increases. Eventually, the cost can become too high, preventing the owner from withdrawing their assets.

Impact

In order to withdraw, the `availableBalance` is checked:

```
function availableBalance() view public override returns(uint256) {
    uint256 balance = token.balanceOf(address(this));
    uint256[] memory optionIds = getOptionIds(); // grows in size per
    option written
    for (uint256 i = 0; i < optionIds.length; i++) {
        WasabiStructs.OptionData memory optionData
    = getOptionData(optionIds[i]);
        if (optionData.optionType == WasabiStructs.OptionType.PUT
    && isValid(optionIds[i])) {
            balance -= optionData.strikePrice;
        }
    }
    return balance;
}
```

The code currently iterates through every option issued, but it does not set a limit on the number of options that may need to be looped through. This risks a denial of service, leaving pool owners unable to access their funds. As it currently stands, there is no way to remove expired options from the enumerable set of options.

Recommendations

We recommend including a public `clearExpiredOptions` function that iterates through options of a specified index range and deletes them from the enumerable set if they are expired. Perhaps even creating some sort of incentive mechanism to encourage users to use this function may be beneficial.

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit eea7c164.

3.2 Solidity versioning permits underflow behavior

- **Target**: All
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: High

Description

The contract specifies its version to be `pragma solidity ≥0.4.25 <0.9.0`. This means the contract can be compiled with a version of Solidity that does not perform checked math.

Impact

It is worth noting that while previous versions of Solidity (up to and including 0.7.x) did not automatically check for overflow and underflow, it was still possible to manually check for and handle such scenarios. However, in the ETH and ERC20 Wasabi pools, balance subtractions such as `balance -= optionData.strikePrice` were not properly guarded against underflow scenarios, which could result in a user's available balance being artificially inflated.

Starting with Solidity version 0.8.x, the compiler performs automatic overflow and underflow checks, helping to prevent these kinds of issues. Therefore, it is recommended to use the latest version of Solidity and follow best practices for safe arithmetic operations to avoid potential issues with underflow and overflow.

Recommendations

We recommend version locking to 0.8.x version.

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit 63ab20b9.

3.3 Usage of transfer to send ETH can prevent receiving

- **Target**: ETHWasabiPool
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The protocol employs Solidity's `.transfer` method to send Ethereum (ETH) to recipients. However, `.transfer` is limited to a hardcoded gas amount of 2,300, which may not be sufficient for contracts with logic in their fallback function. Consequently, these contracts may revert during the transaction. Additionally, the use of a hardcoded gas stipend may not be compatible with future changes to Ethereum gas costs, posing a potential risk to the protocol's long-term viability.

Impact

```
function withdrawETH(uint256 _amount) external payable onlyOwner {
    if (availableBalance() < _amount) {
        revert InsufficientAvailableLiquidity();
    }
    address payable to = payable(_msgSender());
    to.transfer(_amount);
    emit ETHWithdrawn(_amount);
}
```

The `withdrawETH` function sends ETH to the designated recipient (`msg.sender`) using the `to.transfer(_amount)` method. However, if the recipient is a contract that incurs computational costs exceeding 2,300 gas upon receiving ETH, it will be unable to receive the funds. This poses a risk of failed transactions for contracts that have high gas costs, potentially leaving the designated recipient without access to their funds.

Recommendations

We suggest using the `.call` method to send ETH and verifying the return value to confirm a successful transfer. Solidity by Example offers a helpful guide on choosing the appropriate method for sending ETH, which can be found here: https://solidity-by-example.org/sending-ether/.

Furthermore, since the `withdrawETH` function does not intend to receive ETH, the `payable` keyword can be removed.

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit 01ee7727.

3.4 Protocol does not check return value of ERC20 swaps

- **Target**: WasabiPoolFactory, ERC20WasabiPool
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The ERC20 standard requires that transfer operations return a boolean success value indicating whether the operation was successful or not. Therefore, it is important to check the return value of the transfer function before assuming that the transfer was successful. This helps ensure that the transfer was executed correctly and helps avoid potential issues with lost or mishandled funds.

Impact

If the underlying ERC20 token does not revert on failure, the protocol's internal accounting will record failed transfer operations as successful.

Recommendations

We recommend implementing one of the following solutions to ensure that ERC20 transfers are handled securely:

1. Utilize OpenZeppelin's `SafeERC20` transfer methods, which provide additional checks and safeguards to ensure the safe handling of ERC20 transfers.
2. Strictly whitelist ERC20 coins that do not return false on failure and revert. This will ensure that only safe and reliable ERC20 tokens are used within the protocol.

In general, it is important to exercise caution when integrating third-party tokens into the protocol. Tokens with hooks and atypical behaviors of the ERC20 standard can present security vulnerabilities that may be exploited by attackers. We recommend thoroughly researching and reviewing any tokens that are considered for integration and performing a comprehensive security review of the entire system to identify and mitigate any potential vulnerabilities.

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit 0b7bffe6.

3.5 Centralization risks

In the following three findings, the audit has identified centralization risks that users of the protocol should be aware of. Although the impact of these risks is currently mitigated by Wasabi's role as the deployer and owner of the contracts, if the owner's keys were to be compromised or the owner becomes malicious, the impact on the protocol could be significant.

To address this risk and increase user confidence and security, we recommend implementing measures to remove trust from the owner. Our recommendations are aimed at reducing centralization and increasing the resilience of the protocol. It's important to note that custody of private keys is crucial for maintaining control over the protocol. We recommend using a multisig wallet with multiple signers to enhance security.

3.6 Factory update logic of option NFT enables owner to steal funds

- **Target**: WasabiOption
- **Category**: Business Logic
- **Severity**: High
- **Likelihood**: Low
- **Impact**: Low

Description

Each existing option corresponds to a WasabiOption NFT. For access control purposes, the contract stores the address of the corresponding WasabiPoolFactory. The factory provides an interface for pools to mint new option NFTs. However, it is important to note that the factory address can be upgraded in a way that allows the owner to potentially harm both individual option holders and all holders.

Specifically, to remove existing positions, the owner can first call `setFactory` on the associated option NFT. This gives them access to the burn function:

```
function burn(uint256 _optionId) external {
    require(msg.sender == factory, "Only the factory can burn tokens");
    _burn(_optionId);
}
```

After they burn a given option NFT, the owner can use `setFactory` to replace the correct factory address and resume pool mechanics.

Impact

When the owner burns option NFTs, it effectively denies their holders the right to exercise the option they purchased. Since ownership of these NFTs is checked during execution, it is crucial to ensure that the holder's rights are respected and they can exercise their options as intended.

```
function validateOptionForExecution(uint256 _optionId, uint256 _tokenId)
    private {
    require(optionIds.contains(_optionId), "WasabiPool: Option NFT
    doesn't belong to this pool");
    require(_msgSender() == optionNFT.ownerOf(_optionId), "WasabiPool:
    Only the token owner can execute the option");

    WasabiStructs.OptionData memory optionData = options[_optionId];
    require(optionData.expiry ≥ block.timestamp, "WasabiPool: Option has
    expired");

    if (optionData.optionType == WasabiStructs.OptionType.CALL) {
        validateAndWithdrawPayment(optionData.strikePrice, "WasabiPool:
    Strike price needs to be supplied to execute a CALL option");
    } else if (optionData.optionType == WasabiStructs.OptionType.PUT) {
        require(_msgSender() == nft.ownerOf(_tokenId), "WasabiPool: Need
    to own the token to sell in order to execute a PUT option");
    }
}
```

Further, the owner can prevent all holders from exercising options simply by fixing the factory address at a different value. Executing an option requires it to be successfully burned, and the factory loses the right to do so:

```
function clearOption(uint256 _optionId, uint256 _tokenId, bool _executed)
    internal {
    WasabiStructs.OptionData memory optionData = options[_optionId];
    if (optionData.optionType == WasabiStructs.OptionType.CALL) {
        if (_executed) {
            // Sell to executor, the validateOptionForExecution already
    checked if strike is paid
            nft.safeTransferFrom(address(this), _msgSender(),
    optionData.tokenId);
            tokenIds.remove(optionData.tokenId);
        }
        if (tokenIdToOptionId[optionData.tokenId] == _optionId) {
            delete tokenIdToOptionId[optionData.tokenId];
        }
    } else if (optionData.optionType == WasabiStructs.OptionType.PUT) {
        if (_executed) {
            // Buy from executor
            nft.safeTransferFrom(_msgSender(), address(this), _tokenId);
            payAddress(_msgSender(), optionData.strikePrice);
        }
    }
    options[_optionId].active = false;
    factory.burnOption(_optionId);
}
```

Recommendations

We recommend that Wasabi implement one of the following solutions:

- Prevent factory upgrades in the WasabiOption NFT, or
- Support multiple factories, allowing them to be added but not removed. This would also require more granular access control (specifically, storing which NFTs a given factory is permitted to burn).

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit 1aca0ac1.

3.7 Pool toggling functionality may allow factory owner to lock exercising of options

- **Target**: WasabiFactory
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Low

Description

The WasabiFactory contract allows its `owner` to toggle pools.

```
function togglePool(address _poolAddress, bool _enabled)
    external onlyOwner {
    require(poolAddresses[_poolAddress] ≠ _enabled, 'Pool already in same
    state');
    poolAddresses[_poolAddress] = _enabled;
}
```

This prevents them from burning options:

```
function burnOption(uint256 _optionId) external {
    require(poolAddresses[msg.sender], "Only enabled pools can burn
    options");
    options.burn(_optionId);
}
```

Impact

When pools are disabled, the existing options associated with those pools become unexercisable. This effectively allows the `owner` to prevent option holders from utilizing the options they have purchased.

Recommendations

Disabling pools is a reasonable functionality; however, it should not have an impact on the options that have already been issued. One possible solution would be to allow disabled pools to burn options but not mint new ones.

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit 28e1245c.

3.8 Fee manager upgrades allow factory owner to change fees and prevent option exercise

- **Target**: WasabiPoolFactory
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Low

Description

The WasabiPoolFactory contract allows its owner to upgrade the `feeManager`. This fee manager is retrieved by both ERC20 and native ETH pools in option creation and execution. For instance, in ERC20WasabiPool, we have

```
function validateAndWithdrawPayment(uint256 _premium,
    string memory _message) internal override {
    require(token.allowance(_msgSender(), address(this)) ≥ _premium
    && _premium > 0, _message);

    IWasabiFeeManager feeManager
    = IWasabiFeeManager(factory.getFeeManager());
    (address feeReceiver, uint256 feeAmount)
    = feeManager.getFeeData(address(this), _premium);

    token.transferFrom(_msgSender(), address(this), _premium);
    if (feeAmount > 0) {
        token.transferFrom(_msgSender(), feeReceiver, feeAmount);
    }
}
```

Impact

The current implementation of the fee manager upgrade allows the factory owner to change the fee even after options have been minted. This lack of consistency and transparency can result in significant losses for option holders. Moreover, if the fee manager is set to a contract that reverts on `feeManager.getFeeData`, the owner can entirely prevent options from being exercised.

Recommendations

The ability for the owner to change fees and potentially prevent option exercise creates an unfair situation for existing option holders. To address this, a possible solution is to lock in the fee parameters at the time of option minting and store them in the `OptionData` of each option. This would eliminate the need for external calls to `feeManager` when exercising options, thereby ensuring that holders are not impacted by any future changes made by the owner.

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit f4a1b00c.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 AbstractWasabiPool should use upgradable variants of OpenZeppelin contracts

The use of OpenZeppelin's Ownable and ReentrancyGuard in AbstractWasabiPool ensures access control and prevents reentrancy. However, since pools are deployed with OpenZeppelin's Clones library, the AbstractWasabiPool contract (and the concrete ERC20 and native ETH implementations) effectively function as logic contracts, making it more appropriate to use the upgradable variants in OpenZeppelin's `contracts-upgradable`: OwnableUpgradable and ReentrancyGuardUpgradeable.

4.2 Put options operate on entire collections

The Wasabi options have some asymmetry between call and put options. Call options are associated with specific NFTs and users can purchase them at the supplied strike price. On the other hand, put options are valid for an entire collection of NFTs and holders can exercise them to sell any NFT from that collection. It is worth noting that the prices of NFTs within a collection can vary significantly, potentially impacting the protocol's economics.

4.3 Pools require well-behaved NFTs

It is important to be aware that NFTs that behave in unusual ways can have adverse effects on the associated pools. To mitigate potential issues, liquidity providers and users should thoroughly review and verify that any special NFT mechanics do not have unexpected interactions with the pools.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart 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.

5.1 Module: AbstractWasabiPool.sol

Function: `acceptBidWithTokenId(WasabiStructs.Bid _bid, bytes calldata _signature, uint256 _tokenId)`

Accepts the bid for LPs with _tokenId.

Inputs

- `_bid`
  - **Control**: User has full control over the input.
  - **Constraints**: Assuming, the `_bid` must be checked with the corresponding signature, however that check occurs out of scope.
  - **Impact**: Only valid bids can be accepted.
- `_signature`
  - **Control**: User has full control over the input.
  - **Constraints**: The conduit may check the signature, but that check occurs out of scope.
  - **Impact**: `_signature` check is hopefully valid.
- `_tokenId`
  - **Control**: User has full control over the input.
  - **Constraints**: TokenID must not be locked up in another option.
  - **Impact**: Non-lockup tokenIDs can be used.

Branches and code coverage (including function calls)

Intended branches

☑ Valid bid is accepted.

Negative behavior

- ☑ Locked up tokenIDs cannot be used.
- ☑ Insufficient liquidity revert occurs when strike price is greater than available balance.

Function call analysis

- `acceptBidWithTokenId → factory.issueOption(_bid.buyer)`
- **What is controllable?** The buyer who will be issued an option.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is the ID of the option issued, which is used to correspond with the token ID that will be locked up. Can only go wrong if an existing option has the same ID; however, that does not appear the case.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Bid not accepted under revert condition, reentrancy, or unusual control flow is not an issue.
- `acceptBidWithTokenId → isAvailableTokenId(_tokenId)`
- **What is controllable?** The token ID to check if it is available.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is a boolean that determines if the token ID is available. Can only go wrong if this information is relayed incorrectly.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Bid not accepted under revert condition, reentrancy, or unusual control flow is not an issue.
- `acceptBidWithTokenId → availableBalance()`
  - **What is controllable?** No function params.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is the available balance of the pool and is used to check if there exists enough liquidity. Can only go wrong if this information is relayed incorrectly.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Function could revert if there are too many bids as the loop will run out of gas. Reentrancy or unusual control flow is not an issue.

Function: `acceptAsk(WasabiStructs.Ask _ask, byte[] _signature)`

Lets LPs accept an ask.

Inputs

- `_ask`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Token address and price must fit constraints of the pool.
  - **Impact**: Decides how acceptance is handled.
- `_signature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Used by the conduit to validate the ask. Functionality is out of scope.

Branches and code coverage (including function calls)

Intended branches

- Asks without an associated ERC20 can be accepted with native token.
  - [x] Test coverage
- Asks with an associated ERC20 can be accepted.
  - [x] Test coverage

Negative behavior

- Asks without a valid signature cannot be accepted.
  - [ ] Negative test
- Reverts when not called by the owner.
  - [x] Negative test
- Asks cannot be accepted if balance is insufficient.
  - [x] Negative test

Function call analysis

- `acceptAsk -> WasabiConduit(factory.getConduitAddress()).acceptAsk`

- **What is controllable?** The factory controls the conduit address. The caller controls the ask and signature.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The function reverts if the call does not succeed.
- `acceptAsk -> erc20.approve`
  - **What is controllable?** The factory controls the conduit address while the

caller controls the price.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The function reverts if the call does not succeed.

Function: `executeOption(uint256 _optionId)`

Given the ID of any option NFT associated with this pool, this function executes it.

Inputs

- `_optionId`
  - **Control**: Fully controlled by the user.
  - **Constraints**: Membership in `optionIds` and ownership by sender in `option NFT`.
  - **Impact**: Determines which option is executed; validation confirms that the option indeed belongs to the pool and has not expired.

Branches and code coverage (including function calls)

Intended branches

- Options that belong to the pool and are not expired can be executed.
  ☑ Test coverage
- Valid put options will transfer the strike price to the option holder and the NFT to the pool.
  ☑ Test coverage
- Valid call options will transfer the NFT to the option holder and the strike price to the pool.
  ☑ Test coverage
- Option NFTs are burned after execution.
  ☑ Test coverage

Negative behavior

- Existing options that do not belong to the pool cannot be executed.
  ☐ Negative test
- Options that have expired cannot be executed.
  ☑ Negative test
- Options that have already been executed cannot be executed.
  ☐ Negative test
- Only the owner of the option can execute it.

    - ☑ Negative test
- Put options cannot be executed with invalid NFT IDs.
    - ☑ Negative test
- Call options cannot be executed with insufficient payment.
    - ☑ Negative test

Function call analysis

- `validateOptionForExecution` → `optionNFT.ownerOf`
- **What is controllable?** The input option ID.
    - **If return value controllable, how is it used and how can it go wrong?** The return value is used to determine whether the sender can execute the given option. It depends on the implementation of the `optionNFT` provided during construction.
    - **What happens if it reverts, reenters, or does other unusual control flow?** The caller (`executeOption`) uses the OpenZeppelin reentrancy guard. A revert would cause option execution to fail, which could be a centralization risk depending on how the option NFT is implemented.
- `validateOptionForExecution -> nft.ownerOf`
- **What is controllable?** The input token ID.
    - **If return value controllable, how is it used and how can it go wrong?** The value is used to check that the put option is really controlled by the holder. This ownership is implicitly checked during the transfer as well.
    - **What happens if it reverts, reenters, or does other unusual control flow?** A revert would cause option execution to fail.
- `clearOption -> nft.safeTransferFrom`
- **What is controllable?** The sender and the token ID.
    - **If return value controllable, how is it used and how can it go wrong?** The return value is not used.
    - **What happens if it reverts, reenters, or does other unusual control flow?** A revert would cause option execution to fail.
- `clearOption -> factory.burnOption`
- **What is controllable?** The option ID.
    - **If return value controllable, how is it used and how can it go wrong?** The return value is not used.
    - **What happens if it reverts, reenters, or does other unusual control flow?** A revert would cause option execution to fail. This could again be a centralization risk depending on how the option NFT is implemented.
- `validateAndWithdrawPayment -> token.allowance`

- **What is controllable?** The sender.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is used to confirm that the sender has approved sufficient funds for the transfer.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Execution fails.
- `validateAndWithdrawPayment -> factory.getFeeManager`

- **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** The return value decides which fee manager to use. This in turn decides the magnitude of fees as well as the recipient of fees. The controller of the fee manager can arbitrarily set fees, even for options that have already been created.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Execution fails.
- `validateAndWithdrawPayment -> feeManager.getFeeData`

- **What is controllable?** The user controls nothing; the factory owner can set the fee manager.
  - **If return value controllable, how is it used and how can it go wrong?** The return value decides the magnitude of fees. This means that the controller of the fee manager can set fees.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this reverts, then execution fails. That means that the owner of the factory can block option execution.
- `validateAndWithdrawPayment -> token.transferFrom`

- **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is not used (but likely should be used!).
  - **What happens if it reverts, reenters, or does other unusual control flow?** The option execution fails.
- `validateAndWithdrawPayment -> feeReceiver.transfer`

- **What is controllable?** The owner controls the fee receiver.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is not used (but likely should be used!).
  - **What happens if it reverts, reenters, or does other unusual control flow?** The option execution fails. This could be a centralization risk because the owner of the factory can change the fee receiver.
- `payAddress -> factory.getFeeManager`

- **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** The return value decides what fee manager to use. This can go wrong if the owner maliciously changes the manager.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Execution fails.
- `payAddress -> feeManager.getFeeData`

- **What is controllable?** The owner controls the fee manager.
  - **If return value controllable, how is it used and how can it go wrong?** The return value decides the magnitude of fees. This means that the controller of the fee manager can set fees arbitrarily.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Execution fails. Again, this might be a centralization risk.
- `payAddress -> token.transfer`

- **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is not used (but likely should be used).
  - **What happens if it reverts, reenters, or does other unusual control flow?** Execution fails.
- `payAddress -> token.transferFrom`

- **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is not used but should be checked.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Execution fails.
- `payAddress -> _seller.transfer`

- **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Execution fails.
- `payAddress -> feeReceiver.transfer`
  - **What is controllable?** The owner controls the fee receiver.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Execution fails. This could again be a centralization risk.

Function: `onERC721Received(address None, address None, uint256 tokenId, bytes memory None)`

Receive function magic selector for ERC721 tokens so that the pool can receive NFTs. Will be inherited by all pools.

Inputs

- `tokenId`
  - **Control**: User has full control over this input.
  - **Constraints**: If the `msg.sender` is the optionNFT, the tokenId must be in the optionIds set. If the `msg.sender` is the NFT address, it is added to tokenIds.
  - **Impact**: Only tokenIDs sent from the NFT address or the optionNFT address are added to the tokenIds or optionIds sets. If the `msg.sender` is the optionNFT, the option is cleared.

Branches and code coverage (including function calls)

Intended branches

- ☑ `msg.sender` is the optionNFT.
- ☑ `msg.sender` is the NFT address.

Negative behavior

- ☑ `msg.sender` is neither the optionNFT nor the NFT address and therefore reverts.

Function call analysis

- `onERC721Received` → `clearOption(tokenId, 0, false)`
  - **What is controllable?** Nothing because the `msg.sender` is checked to be the optionNFT.
  - **If return value controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The control flow of clearOption is complex and should be reviewed to ensure it handles all edge cases.

Function: `writeOptionTo(WasabiStructs.PoolAsk _request, bytes calldata _signature, address _receiver)`

This writes an option from the pool to the respective receiver. The option request must be within the pool's configuration. The signature must be signed by either the admin or the owner of the pool. The signature must not be replayed. The option is issued to the receiver.

Inputs

- `_request`
  - **Control**: User has full control over this input.
  - **Constraints**: The request's corresponding strike price, expiration, and option type is valid.
  - **Impact**: Only legitimate requests can be written to the pool.
- `_signature`
  - **Control**: User has full control over this input.
  - **Constraints**: The signer of the signature is either the admin or the owner. Cannot be replayed because of the idOrCancelled mapping.
  - **Impact**: Signature of the request is validated.
- `_receiver`
  - **Control**: User has full control over this input.
  - **Constraints**: No constraints. Can be any address.
  - **Impact**: The option is issued to the receiver.

Branches and code coverage (including function calls)

Intended branches

- ☑ Valid request is written to the pool, and option is issued to the receiver.

Negative behavior

- ☑ Strike price is out of bounds (too high or too low) of the pool configuration.
- ☑ Expiration is out of bounds (too high or too low) of the pool configuration.
- ☑ Signature does not pertain to an admin or owner.
- ☑ TokenId does not contain the `_request.tokenId`.
- ☑ Token is not free (asset is locked).
- ☑ Not enough premium is sent.
- ☑ The available balance is not enough to cover the strike price if it is a PUT option.
- ☑ The request expiration is less than the current block timestamp.

Function call analysis

- `writeOptionTo` → `validate(_request, _signature)`

- **What is controllable?** The request and the signature.
  - **If return value controllable, how is it used and how can it go wrong?** No

return value.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Option not written from the pool, and option is not issued to the receiver.
- `writeOptionTo` → `factory.issueOption(_receiver)`

- **What is controllable?** The receiver address.
    - **If return value controllable, how is it used and how can it go wrong?** Return value is used to denote the optionId and could go wrong if an existing optionId is the chosen value. However, that does not appear the case.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Option not written from the pool, and option is not issued to the receiver.
- `writeOptionTo` → `optionIds.add(optionId))`
    - **What is controllable?** Nothing, optionId is chosen by the protocol.
    - **If return value controllable, how is it used and how can it go wrong?** No return value; function simply adds optionId to an enumerable set.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Option not written from the pool, and option is not issued to the receiver.

5.2 Module: WasabiOption.sol

Function: `newMint(address to)`

Mints a new WasabiOption

Inputs

- `to`
    - **Control**: Factory has full control over this input.
    - **Constraints**: No constraints.
    - **Impact**: A factory can mint tokens to any address.

Branches and code coverage (including function calls)

Intended branches

- [x] Factory minting to a valid address.

Negative behavior

- [ ] `msg.sender` is not the factory.

Function call analysis

- `newMint` → `_safeMint(to, _currentId)`
  - **What is controllable?** `to` parameter is controllable.
  - **If return value controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reentrancy is not an issue because only the factory can call back in. Minting does not happen under revert condition. An interesting control flow aspect is that the receiver of the NFT can call into other functions if it contains logic in its `onERC721Received` function.

5.3 Module: WasabiPoolFactory.sol

Function: `burnOption(uint256 _optionId)`

A registered pool can burn an option.

Inputs

- `_optionId`
  - **Control**: User has full control over this input.
  - **Constraints**: Any optionID can be passed in.
  - **Impact**: Any option can be burned.

Branches and code coverage (including function calls)

Intended branches

- ☑ Called by a valid pool.

Negative behavior

- ☐ Called by a non-pool address.

Function call analysis

- `burnOption` → `options.burn(_optionId)`
  - **What is controllable?** The optionID given the `msg.sender` is a valid pool.
  - **If return value controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reentrancy is not an issue as this function is only permissible by a valid

pool address. Option not burned under revert scenario and no unusual control flow because the option contract does not implement the virtual function `_beforeTokenTransfer`.

Function: `createERC20Pool(address _tokenAddress, uint256 _initialDeposit, address _nftAddress, uint256[] _initialTokenIds, WasabiStructs.PoolConfiguration _poolConfiguration, WasabiStructs.OptionType[] _types, address _admin)`

This function creates a new ERC20 based pool.

Inputs

- `_tokenAddress`
  - **Control**: Fully controlled by user.
  - **Constraints**: Must have functional `transferFrom` function if `_initialDeposit` is nonzero.
  - **Impact**: Any contract can be used; those that are not ERC20 compliant will likely cause problems with pool mechanics.
- `_initialDeposit`
  - **Control**: Can be any `uint256` value.
  - **Constraints**: Must be less than or equal to the balance of the sender in the given token.
  - **Impact**: Decides the initial balance of the pool.
- `_nftAddress`
  - **Control**: User fully controls this value.
  - **Constraints**: If `_initialTokenIds` is set, the user must own all of them.
  - **Impact**: NFTs that behave in unexpected ways may cause problems with pool mechanics.
- `_initialTokenIds`
  - **Control**: User fully controls this input.
  - **Constraints**: The IDs in this array must be owned by the user under the given NFT collection.
  - **Impact**: Decides the initial NFTs in the pool.
- `_poolConfiguration`
  - **Control**: User fully controls this input.
  - **Constraints**: Checked by `WasabiValidation.validate`.
  - **Impact**: Configuration cannot have egregiously degenerate values (e.g., minimums must be less than maximums).

- `_types`
  - **Control**: User fully controls this input.
  - **Constraints**: At least one type must be supplied.
  - **Impact**: Decides the types of options that can be created in the pool.
- `_admin`
  - **Control**: User fully controls this input.
  - **Constraints**: None.
  - **Impact**: Decides the admin of the pool.

Branches and code coverage (including function calls)

Intended branches

- A pool with valid configuration can be created.
  ☑ Test coverage

Negative behavior

- Pools can only be initialized one time.
  ☐ Negative test
- A pool with invalid strike price configuration cannot be created.
  ☐ Negative test
- A pool with invalid duration configuration cannot be created.
  ☐ Negative test
- Pools without at least one option type cannot be created.
  ☐ Negative test

Function call analysis

- `createERC20Pool` → pool.initialize‘
- **What is controllable?** All parameters except for the factory address.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is unused.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the pool is not created.
- `createERC20Pool` → _poolAddress.transfer‘
- **What is controllable?** The amount of ETH to transfer.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is unused.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

  If it reverts, the pool is not created.

- `createERC20Pool` → token.transferFrom'

- **What is controllable?** The sender and amount of tokens to transfer.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is unused.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Pool creation fails.
- `createERC20Pool` → nft.safeTransferFrom'
  - **What is controllable?** The sender and token ID to transfer.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is unused.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Pool creation fails.

Function: `createPool(address _nftAddress, uint256[] _initialTokenIds, WasabiStructs.PoolConfiguration _poolConfiguration, WasabiStructs.OptionType[] _types, address _admin)`

Creates a new ETH-based pool with the given pool configuration, supported option types, and admin.

Inputs

- `_nftAddress`
  - **Control**: User has full control over this input.
  - **Constraints**: User owns the respective intialTokenIds of the NFT.
  - **Impact**: Any NFT address can be used, given the user owns the respective initialTokenIds of the collection.
- `_initialTokenIds`
  - **Control**: User has full control over this input.
  - **Constraints**: User owns the respective intialTokenIds of the NFT.
  - **Impact**: User can add any NFTs to the pool, given the user owns the respective initialTokenIds of the collection.
- `_poolConfiguration`
  - **Control**: User has full control over this input.
  - **Constraints**: Configuration passes `WasabiValidation.validate`.
  - **Impact**: Configuration is within bounds of strike price and duration.
- `_types`
  - **Control**: User has full control over this input.
  - **Constraints**: Must be at least one option type, call or put.

    - **Impact**: User can select which option types are supported by the pool.
- `_admin`
    - **Control**: User has full control over this input.
    - **Constraints**: No constraints.
    - **Impact**: Admin can be any address.

Branches and code coverage (including function calls)

Intended branches

- ☑ A pool with a valid configuration can be created.

Negative behavior

- ☑ A pool with an invalid strike price configuration cannot be created.
- ☑ A pool with an invalid duration configuration cannot be created.
- ☐ A pool cannot be initialized twice.
- ☐ A pool cannot be created with no option types.

Function call analysis

- `createPool → WasabiValidation.validate(_poolConfiguration)`
- **What is controllable?** The pool configuration.
    - **If return value controllable, how is it used and how can it go wrong?** No return value.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Pool is not created under revert condition. Reentrancy is not possible. Unusual control flow is not present.
- `createPool → ETHWasabiPool(payable(Clones.clone(address(templatePool))))`
- **What is controllable?** Nothing is controllable.
    - **If return value controllable, how is it used and how can it go wrong?** Return value corresponds to the address of the new pool. No foreseeable issues unless clone overwrites existing contract.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Pool is not created under revert condition. Reentrancy is not possible. Unusual control flow is not present.
- `createPool → pool.initialize(this, nft, options, _msgSender(), _poolConfiguration, _types, _admin)`
- **What is controllable?** The NFT, pool configuration, option types, and admin.
    - **If return value controllable, how is it used and how can it go wrong?** No return value.

  - **What happens if it reverts, reenters, or does other unusual control flow?** Pool is not created under revert condition. Reentrancy is not possible. Unusual control flow is not present.

- `createPool` → `_poolAddress.transfer(msg.value)`

- **What is controllable?** `msg.value`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Pool is not created under revert condition. Reentrancy is not possible because the receiver is the pool address. Unusual control flow is not present since the receive function is not overridden.

- `createPool` → `nft.safeTransferFrom(_msgSender(), _poolAddress, _initialTokenIds[i])`
  - **What is controllable?** The `initialTokenIds`
  - **If return value controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Pool is not created under revert condition, which can happen if the `msg.sender` does not approve or own the NFT. Reentrancy is not possible because the receiver is the pool address, which has no abnormal logic in the `onERC721Received` function.

6 Audit Results

At the time of our audit, the code was not deployed to mainnet Ethereum.

During our audit, we discovered seven findings. Of the seven findings, two were of high impact, two were of medium impact, and three were of low impact. Wasabi acknowledged all findings and implemented fixes.

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 audit version of 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.

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.