Sushi Swap

Auction Maker and Furo

Smart Contract Security Review

Version: 2.0

June, 2022

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Sushi smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Sushi smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Sushi smart contracts.

Overview

Sushi Auction Maker is a smart contract which implements permissionless auctions for arbitrary tokens. It sells the swap fees generated by the SushiSwap protocol (0.05%) for a predefined token to the highest bidder. Anyone can start/bid on auctions and withdraw the proceeds to the receiver.

Furo is a system built on top of SushiSwap's `BentoBox` which allows streaming and vesting of arbitrary tokens. `FuroStream` offers the possibility to create a continuous stream of tokens from sender to receiver which accrues rewards in every block.
`FuroVesting` implements a mechanism to transfer tokens in a more discrete manner. The first tokens can only be claimed after a certain cliff period has passed. The remainder can be withdrawn in equal parts at regular intervals.

Security Assessment Summary

This review was conducted on the files hosted on the Sushi repository and were assessed at commit a5d8d50 for Auction Maker and 76e9ac1 for Furo.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts: specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 23 issues during this assessment. Categorized by their severity:

- Critical: 5 issues.
- High: 4 issues.
- Medium: 3 issues.
- Low: 3 issues.
- Informational: 8 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Sushi smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
SSAF-01	Miscalculation of Amount Returnable to Sender	Critical	Resolved
SSAF-02	Stream Updates Are Paid Directly to the Recipient	Critical	Resolved
SSAF-03	Token Amounts are Treated as Share Amounts	Critical	Resolved
SSAF-04	Miscalculation of the Returnable Amount of Tokens	Critical	Resolved
SSAF-05	Reentrancy Vulnerabilities May Drain Tokens	Critical	Resolved
SSAF-06	Incorrect Accounting When Updating Streams	High	Resolved
SSAF-07	Interaction of `ETH` and `WETH` tokens in `_depositToken()`	High	Resolved
SSAF-08	Non-standard ERC20 Tokens Are Not Supported And Locked in the Contract	High	Resolved
SSAF-09	Tokens Implementing `token0()` or a Fallback Are Not Supported	High	Resolved
SSAF-10	`updateStream()` Is Not Payable	Medium	Resolved
SSAF-11	Unchecked Return Values of ERC20 `transfer()`	Medium	Resolved
SSAF-12	Token Transfers May Fail Locking Auctions	Medium	Resolved
SSAF-13	Insufficient Input Validation in `SushiMakerAuction`	Low	Open
SSAF-14	Insufficient Validation Checks in Furo Contracts	Low	Closed
SSAF-15	Unsafe Casting From `uint256` To `uint128`	Low	Open
SSAF-16	`batch()` Incorrectly Interprets Error Messages	Informational	Resolved
SSAF-17	Any User Can Drain Ether That Is Left In The Contract	Informational	Resolved
SSAF-18	`BoringOwnable` is Unused	Informational	Resolved
SSAF-19	Inconsistent End States for Streams and Vests	Informational	Closed
SSAF-20	NFTs Persist After Use	Informational	Closed
SSAF-21	Denial-of-Service Condition by Maliciously Extending Auctions	Informational	Open
SSAF-22	Insufficient Comments, Documentation & Tests	Informational	Open
SSAF-23	Other `Auction Maker` and `Furo` Comments	Informational	Closed

SSAF-01	Miscalculation of Amount Returnable to Sender		
Asset	`FuroStream.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

There is an accounting error in `_balanceOf()` which overstates `senderBalance`. As a result, the sender will be over compensated in `cancelStream()`.

When the block timestamp is after the `startTime` and before the `endTime`, the following section of the contract is executed:

```
211 uint256 timeDelta = block.timestamp - stream.startTime;
    recipientBalance =
213   ((stream.depositedShares * timeDelta) /
        (stream.endTime - stream.startTime)) -
215   uint256(stream.withdrawnShares);
    senderBalance = uint256(stream.depositedShares) - recipientBalance;
```

On line [216], the variable `recipientBalance` is subtracted from `stream.depositedShares`, the total shares in the stream, to produce the shares still belonging to the sender. However, `recipientBalance` is not actually the total shares allocated to the recipient so far: it is the amount still payable. `recipientBalance` has had `stream.withdrawnShares` subtracted from it to account for already withdrawn shares. Therefore, `senderBalance` is being calculated in such a way that it actually includes `stream.withdrawnShares`.

$$recipientBalance = \frac{depositedShares * \Delta time}{endTime - startTime} - withdrawnShares$$

$$senderBalance = depositedShares - recipientBalance$$

Thus we have $senderBalance$ as,

$$senderBalance = depositedShares - \frac{depositedShares * \Delta time}{endTime - startTime} + withdrawnShares$$

Recommendations

The issue can be mitigated by updating `senderBalance` to represent the following.

$$senderBalance = depositedShares - \frac{depositedShares * \Delta time}{endTime - startTime}$$

Resolution

The development team has fixed the issue by calculating the shares streamed so far and then using that as the basis for calculating both `recipientBalance` and `senderBalance`.

This is shown in PR 16.

SSAF-02	Stream Updates Are Paid Directly to the Recipient		
Asset	`FuroStream.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

On line [252] in the `FuroStream` contract, in the function `updateStream()`, the tokens being sent from the sender to top up the stream are being transferred directly into ownership of the `recipient` address rather than the current smart contract.

The impact is that the `recipient` will receive the BentoBox deposit of `topUpAmount` in addition to any future revenue earned from the stream.

The additional future revenue will include `topUpAmount` in addition to the previous `depositedShares`, thereby double paying the `recipient`.

```
249 depositedShares = _depositToken(may
      stream.token,
251   stream.sender,
      recipient, // @audit should be address(this)
253   topUpAmount,
      fromBentoBox
255 );
```

Recommendations

Change the address on line [252] to `address(this)`, thereby transferring the token to the `FuroStreaming` smart contract rather than the `recipient`.

Resolution

The development team has fixed the issue by changing the address as recommended.

This is shown in PR 16.

SSAF-03	Token Amounts are Treated as Share Amounts		
Asset	`FuroVesting.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Inconsistencies in accounting occur from treating certain variables in units of the *underlying ERC20 tokens* rather than *BentoBox shares*, causing recipients to be overpaid.

In the function `createVesting()` on line [**65**], `depositedShares` is returned from `_depositToken`, but is not made use of. Instead, the original token parameters of `cliffAmount` and `stepAmount` are stored in `vests[vestId]`.

The function `_transferToken()` expects `amount` to be a value in shares rather than underlying tokens. As a result, `withdraw()` and `stopVesting()` will pay the `recipient` shares which are denominated in underlying token units.

As shares in BentoBox are worth more than the underlying token amounts, the recipient will be overpaid at the expense of the protocol.

Recommendations

In the struct `Vest`, rename `cliffAmount` and `stepAmount` to `cliffShares`, and `stepShares` and within `createVesting()`, calculate the appropriate share values to use in these variables from `depositedShares`.

Resolution

The development team has fixed the issue by changing the input parameters to include a total token amount and a percentage amount for the step size. The exact share values for cliff and steps are then calculated from these inputs using `depositedShares`, the return value from `_depositToken`.

Where appropriate, variable names have also been changed to accurately reflect which variables are amounts and which are shares.

This is shown in multiple commits and pull requests, with the latest being 5f88ce2.

SSAF-04	Miscalculation of the Returnable Amount of Tokens		
Asset	`FuroVesting.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

An accounting error in `stopVesting()` leads to the overstatement of `returnAmount`, thereby transferring the owner more shares than they are owed.

The `returnAmount` represents the amount of tokens in the vesting schedule that have not vested. This value is overstated as it includes the amount already claimed in addition to the amount left in the schedule. The following snippet is from `stopVesting()`:

```
uint256 canClaim = _balanceOf(vest) - vest.claimed;
uint256 returnAmount = (vest.cliffAmount +
    (vest.steps * vest.stepAmount)) - canClaim;
```

Consider the edge case where the entire duration has passed and the `recipient` has claimed all of the tokens:

$$balanceOf(vest) = vest.claimed = vest.cliffAmount + vest.steps * vest.stepAmount$$

From the code above we can calculate,

$$canClaim = balanceOf(vest) - vest.claimed = 0$$

$$returnAmount = vest.cliffAmount + vest.steps * vest.stepAmount - canClaim$$

Since $canClaim = 0$ we have,

$$returnAmount = vest.cliffAmount + vest.steps * vest.stepAmount$$

The `owner` is therefore overpaid by the amount that has already been claimed, that is `vest.claimed`.

Recommendations

Consider updating `returnAmount` such that it does not include the `vest.claimed` value, as seen below:

```
uint256 amountVested = _balanceOf(vest);
uint256 canClaim = amountVested - vest.claimed;
uint256 returnAmount = (vest.cliffAmount +
    (vest.steps * vest.stepAmount)) - amountVested;
```

Resolution

The development team has fixed the issue by implementing the recommended logic.

This is shown in PR 17.

SSAF-05	Reentrancy Vulnerabilities May Drain Tokens		
Asset	`FuroVesting.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

There is a potential reentrancy bug in `FuroVesting.stopVesting()` that allows draining the token balance of contract for any ERC20 token that relinquishes control during `transfer()`. In addition to other ERC20 tokens, it's also possible to withdraw WETH as native ETH in `BentoBox` which will relinquish control to the attacker.

In the function `stopVesting()`, `_transferToken()` is called twice before the state variable `vests` is updated on line [**156**]. When `toBentoBox = false`, the underlying asset is transferred out of the BentoBox to `recipient` and `owner` respectively.

If the token being transferred is one that relinquishes execution control during `transfer()`, an attacker could reenter the `stopVesting()` function. Since `delete vests[vestId];` has not yet been processed, both the calls to `_transferToken()` would execute again.

Recommendations

To prevent reentrancy, apply the *check-effects-interactions* pattern. Specifically, `delete vests[vestId]` should occur before making any external calls.

Another mitigation is to use reentrancy guards such as OpenZeppelin's ReentrancyGuard over each of the functions which makes external calls.

Resolution

The development team has fixed the issue by moving the line `delete vests[vestId]` to before the call to `_transferToken()`.

This is shown in PR 17.

SSAF-06	Incorrect Accounting When Updating Streams		
Asset	`FuroStream.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The function `updateStream()` allows the creator of the stream to modify the parameters. There are potential accounting errors when the `topUpAmount` or `endTime` is modified.

When `toUpAmount` is non-zero, the stream will immediately attribute part of the `topUpAmount` as already vested, that is the part that is already vested or linearly proportional to the elapsed time over the total time.

For example if the duration `startTime - endTime = 100`, and we have `depositedShares = 10`, at time 50 we call `updateStream()` with `topUpAmount = 10`. Hence the new `depositedShares = 10 + 10 = 20` and `withdrawnShares = 5` since half the original `depositedShares` will be withdrawn in `updateStream()`.

After the call to `updateStream()` without passing anymore time `_balanceOf()` will calculate `recipientBalance` as,

$$recipientBalance = \frac{depositedShares*timeDelta}{endTime-startTime} - withdrawnShares$$

$$recipientBalance = \frac{20*50}{100} - 5$$

$$recipientBalance = 10 - 5 = 5$$

Thus, without any time increase, `updateStream()` has caused the `recipientBalance` to increase. The amount is proportional to the time elapsed since the start of the stream.

A similar issue exists if we have `extendTime` as non-zero. However, this will impact the denominator of the fractional part of `recipientBalance` and therefore decrease the balance. The impact here is that it is possible for `withdrawnShares` to now be greater than the fraction and therefore cause a subtraction overflow preventing the user balance from being checked.

Recommendations

One mitigation is to reset the stream as a new stream. After transferring the `recipient`, the amount owed to them adjusting the parameters such that it is the equivalent of starting a new stream with the remaining balance.

- `stream.startTime = now`
- `stream.depositedShares = depositedSharesTopUp + senderBalance`
- `stream.withdrawnShares = 0`
- `stream.endTime += extendTime`

Resolution

The development team has fixed the issue by changing the calculation to the recommended logic.

This is shown in multiple pull requests, the final version being 20.

SSAF-07	Interaction of `ETH` and `WETH` tokens in `_depositToken()`		
Asset	`FuroStream.sol` & `FuroVesting.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

If the contracts use a real `WETH` token contract address for the state variable `wETH`, there can be some undesirable interactions with `ETH` deposits.

`_depositToken()` checks that the `token` parameter is `wETH` and then only processes ETH if the contract's balance is high enough. If it is not, execution passes through to the `else` block that will process the deposit using actual `WETH` token.

Consider a situation where a user has both `WETH` tokens and `ETH`, and where the `WETH` tokens have been maximally approved to the Furo contract. If the user calls any of the `payable` functions, but sends, say, 90% of the required `ETH`, then `_depositToken()` simply ignores the `ETH` and transfers `WETH` tokens, leaving all the paid `ETH` in the contract.

Any user can then drain this `ETH` using the technique described in SSAF-17.

Futherthermore, there are two scenarios where this issue is particularly likely to occur:

1. Batched Transactions

   Because `BoringBatchable` sends all the `ETH` payments together, it is possible to trigger the conditions described above through normal use of batched transactions.

   Consider two batched calls to `FuroStream.createStream()` in the following order:

   (a) A small stream to be created using the token `wETH`.

   (b) A large stream to be created using `ETH`.

   Assume the user has lots of approved WETH and they send exactly the right amount of `ETH` for the second stream. Also, assume that these are batched and submitted using `BoringBatchable`'s option to not revert the entire transaction if a single transaction within the batch reverts.

   In this case, the tests in `_depositToken()` for the contract's own `ETH` balance will pass, and the amount of stream 1 will be taken in `ETH`. After that, the second transaction will fail.

   This will leave the difference in the contract, in `ETH`, able to be drained using the technique described in SSAF-17.

2. Increase in Share Value

   This point applies to a version of the contracts reviewed during resolution, when the input parameters were in shares not token amounts.

   `Bentobox` shares can change value whilst the user is interacting with the interface in their browser. Although the percentage increase is presumed to be quite small, there is certainly enough time for them to change slightly whilst the user is signing the transaction or whilst the transaction is in the mempool.

   Consider a user with large amounts of `ETH` and `WETH`, having approved the maximum amount of `WETH` for use in the Furo contract. The user then interacts with a web UI and sets the value of `ETH` to send, then signs a transaction that sends precisely this amount of `ETH`. However, the share value increases by a tiny fraction of a percent while the transaction is in the mempool. As a result, the amount of `ETH` paid does not satisfy the first if test in `_depositToken()`, which checks that the Furo contract holds sufficient ETH, and so the Furo contract takes the amount in WETH instead, leaving the submitted amount of ETH in the contract, ready to be drained by the method from SSAF-17.

Recommendations

One possible approach is to have a dedicated address to signal to `_depositToken()` that the payment is being made in ETH, not the WETH token. The wETH variable would still be needed for interactions with BentoBox, and so the vests and streams would still need to be stored with wETH as their token. It would just be the calls to payable functions and to `_depositToken()` that would use the special ETH signalling address. The signalling address should certainly not be a token, and is best not being a contract at all.

Resolution

The development team implemented the recommended strategy, using the zero address to signal that a deposit was being made in ETH. This is observable in commit a01d331.

SSAF-08	Non-standard ERC20 Tokens Are Not Supported And Locked in the Contract		
Asset	`SushiMakerAuction.sol`		
Status	**Resolved:**		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

Non-ERC20 compliant tokens might not be supported by the contract. Specifically, this is true when the `transfer()` function does not correctly implement to the `IERC20` interface.

One prominent example for such tokens is the stablecoin `USDT`.

When trying to call `end()` for a related auction, the transaction reverts. This is because the call to `IERC20(token).transfer()` does not match the expected return value of `transfer()` on the target contract.

In the case of USDT, the function does not return any value which causes an execution error as it attempts to decode a `bool`. There is no other way to reclaim these tokens, so they would be locked in the contract.

Recommendations

Appropriate handling of non-standard ERC20 contracts is necessary if these tokens are to be supported. A common way to handle this is by using a vetted library such as OpenZeppelin's SafeERC20.

Resolution

The development team has addressed this issue in PR 1 by using the OpenZeppelin's SafeERC20 library.

SSAF-09	Tokens Implementing `token0()` or a Fallback Are Not Supported		
Asset	`SushiMakerAuction.sol`		
Status	**Resolved:**		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

`SushiMakerAuction` will reject ERC20 tokens that implement the function `token0()` to prevent the sale of LP tokens which must implement this function.

LP tokens are prevented from being used with the `onlyToken` modifier as seen below.

```
modifier onlyToken(IERC20 token) {
  // Any cleaner way to find if it's a LP?
  (bool success, ) = address(token).call(
    abi.encodeWithSignature("token0()")
  );
  if (success) revert LPTokenNotAllowed();
  _;
}
```

The modifier ensures that a call to `token0()` fails. However, it is possible for non-LP tokens to also fail this requirement. It is then impossible for these non-LP tokens to be used in an auction. There is no other way to reclaim these tokens which have been sent to the protocol, thus they become stuck in the contract.

If a token implements a fallback function that does not revert it will mistakenly be rejected as it is considered a LP token. One prominent example of this is wrapped Ether, `WETH9`, which has a `fallback()` function that will succeed when no other `calldata` is supplied.

Furthermore, any ERC20 that are not LP tokens and happen to implement the `token0()` function will also be rejected.

Recommendations

Unfortunately this issue cannot be easily resolved on-chain without manual intervention. To handle these tokens, the testing team suggest adding a token whitelist which works in addition to the existing checks. That is check if the token address is whitelisted or if the call to `token0()` succeeds. This will reduce the amount of manual intervention, to only the tokens which have an existing `onlyToken` check.

Resolution

The development team has resolved the issue by adjusting the `onlyToken` modifier like in the recommendation by adding the mapping `whitelistedTokens`. Added to that, they added a check for the result of the low-level `call` to deal with tokens with a `fallback` function and therefore this type of token doesn't need to be whitelisted. The fix is outlined in PR 1

SSAF-10	`updateStream()` Is Not Payable		
Asset	`FuroStream.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

A sender can update a stream using the `updateStream()` function and deposit the associated stream token into the `FuroStream` contract.

The `_depositToken()` function optionally takes native Ether and deposits into `bentoBox` when the stream token is the wrapped Ether and there is sufficient balance of Ether in the contract.

However, `updateStream()` is not `payable`, hence it is not possible to deposit tokens into a stream with the `WETH` token while paying using native Ether.

Recommendations

We recommend adding the `payable` modifier to `updateStream()` function.

Resolution

The development team has fixed the issue by adding the `payable` modifier as recommended.

This is shown in PR 16.

SSAF-11	Unchecked Return Values of ERC20 `transfer()`		
Asset	`SushiMakerAuction.sol`		
Status	**Resolved:**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The ERC20 interface definition includes return values for most functions. These values indicate whether the action was successfully executed or not, although most implementations will revert for the case where the execution fails.

The `transfer()` function will return a `bool` which is `true` if execution is successful and `false` if execution fails.

`SushiMakerAuction` does not check these return values for `token.transfer()` and `bidToken.transfer()` calls.

For ERC20 tokens which instead return `false` rather than reverting if a transfer fails, this would result in the protocol accounting for transfers which have not happened. If this is the case for the `bidToken` then the user would not be required to transfer any tokens to place a large bid.

Recommendations

The testing team recommends checking the return values of ERC20 `transfer()` calls. A popular way to accomplish this is by using OpenZeppelins SafeERC20 library.

Auction state processing should only be continued if the calls were successful. In case of failure, the transaction should be reverted.

Resolution

The development team has addressed this issue in PR 1 by using the OpenZeppelin's SafeERC20 library.

SSAF-12	Token Transfers May Fail Locking Auctions		
Asset	`SushiMakerAuction.sol`		
Status	**Resolved:**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

It is possible for token transfers to fail for certain ERC20 tokens. For example, some ERC20 tokens prevent transfers to blacklisted accounts.

If the `token.transfer(bid.bidder, bid.rewardAmount);` fails during `end()`, then the entire transaction will fail. The impact is that it will be impossible to end the auction or start new auctions. Hence, both the amount of bid tokens and any current or future reward tokens (for this specific token) will be locked in the contract.

Recommendations

Consider adding a `clearBid()` function, which may be called within a specified period of time after a bid has ended. This function would delete the existing bid and transfer the bid tokens to the receiver.

Note that it is possible for transfers to fail for genuine reasons such as a token contract being paused.

Resolution

The development team has fixed this issue by introducing a balance mechanism inside `start()`, `placeBid()` and `end` functions. In fact, instead of transfering tokens in these functions using external calls, a balance mechanism is updated. All the token transfers are made using `deposit()` and `withdraw()` functions.

SSAF-13	Insufficient Input Validation in `SushiMakerAuction`		
Asset	`SushiMakerAuction.sol`		
Status	Open		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

- Zero or small `rewardAmount`

  It is possible to bid on tokens which have zero or smaller than desirable `rewardAmount` in `startBid()`. This may occur if a user's call to `startBid()` is not mined in the time it takes another auction to execute.

  Consider adding a `minimumRewardAmount` parameter to `startBid()` and ensuring the balance of the reward token is larger than `minimumRewardAmount`.

- Zero addresses

  Missing zero address checks on the input parameter `to` in `start()` and `placeBid()` can lead to lost funds for the bidder. By setting `bid.bidder` to the zero address the auction reaches an invalid state. This is because that value is used to determine the auction's state: When `bid.bidder` is zero, the auction is considered finished. It is assumed that transfers and correct accounting of `stakedBidToken` has been applied in `end()`. Even though this is not true in this scenario, the auction can be restarted.

Recommendations

We recommend adding the suggested input validation checks for zero address and zero amounts.

SSAF-14	Insufficient Validation Checks in Furo Contracts		
Asset	`FuroStream.sol` & `FuroVesting.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

This is a list of parameter checks that appear to be missing and may benefit the contract if added.

- `FuroStream.createStream()`
  - `amount` has no zero check.
  - `recipient` has no zero check.
  - `token` has no zero check.
- `FuroStream.withdrawFromStream()`
  - `sharesToWithdraw` has no zero check and will proceed to make external calls and create events.
- `FuroVesting.updateSender()`
  - `sender` has no zero address check.
- `FuroVesting.updateStream()`
  - `topUpAmount` and `extendTime` have no zero check: both could be zero at the same time.
- `FuroVesting.createVesting()`
  - `stepDuration` has no zero check; if it is zero `_balanceOf()` will divide by zero causing deposited funds to be locked.
  - `steps` has no zero check.
- `FuroVesting.withdraw()`
  - If `canClaim == 0` on line [109], consider returning instead of making external calls and creating events.
- `FuroVesting.updateOwner()`
  - `newOwner` has no zero address check.

Recommendations

Review the issues above and consider implementing fixes to them.

Resolution

The development team has reviewed and acknowledged the issue. Some of the recommendations have been implemented.

- `FuroVesting.createVesting()`
  - `stepDuration` and
  - `steps` have zero checks.
- `FuroVesting.withdraw()`
  - If `canClaim == 0` on line [**109**], the function now returns zero.

This is shown in PR 17.

SSAF-15	Unsafe Casting From `uint256` To `uint128`		
Asset	`FuroStream.sol`, `FuroVesting.sol` & `SushiMakerAuction.sol`		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There are a range of unsafe casts in `FuroStream`, `FuroVesting` and `SushiAuctionMaker` for `uint256` values into `uint128`. These casts will truncate any values greater than 2^{128}.

`BentoBoxV1` expects ERC20 tokens to have a maximum value less than 2^{128}. Therefore, balances in `FuroVesting` and `FuroStream` should not exceed 2^{128}.

However, balances in `SushiMakerAuction` can be for any arbitrary token which is used in SushiSwap. If the `rewardToken` has a balance greater than `uint128`, then it is possible to overflow and potentially cause significant accounting errors.

Recommendations

This issue may be partially mitigated by ensuring all casts from `uint256` to `uint128` are done by first checking for overflows. Consider using the `BoringMath.to128()` function or OpenZeppelin's SafeCast library.

Consider also updating all balances in these three contracts to use `uint256` values rather than `uint128`.

SSAF-16	`batch()` Incorrectly Interprets Error Messages
Asset	`BoringBatchable.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The function `batch()` calls multiple functions in one transaction. If any of the calls fails the transaction will revert and propagate the error message.

There is a bug in `_getRevertMsg()` related to the error message propagation which results in an empty string being propagated.

```
function _getRevertMsg(bytes memory _returnData)
  internal
  pure
  returns (string memory)
{
  // If the _res length is less than 68, then the transaction failed silently (without a revert message)
  if (_returnData.length < 68) return "Transaction reverted silently";

  assembly {
    // Slice the sighash.
    _returnData := add(_returnData, 0x04)
  }
  return abi.decode(_returnData, (string)); // All that remains is the revert string
}
```

The error occurs in the above code as it assumes the error message in `result` is of type `Error(string)`. The type `Error(string)` occurs when a call reverts via `require(false, "Some error msg")`. The layout of `bytes memory result` will be the first 4 bytes of `keccak256("Error(string)")` followed by the encoding of the `string`. Hence, in the code above it will move the `result` memory pointer forward 4 bytes and then only read the `string` error message.

However, custom errors like those used in `FuroVesting`, `FuroStream` and `SushiMakerAuction` will have a different layout. The custom errors will have a return type of `keccak256("ErrorName(type1,type2,...)")`. Since the custom errors in these contracts do not have additional data, they are all 4 bytes in length.

Hence, when applying the `_getRevertMsg()` error propagation logic, `_returnData` is always 4 bytes for the custom errors and will trigger the conditional statement `if (_returnData.length < 68) return "Transaction reverted silently";`.

For example, the error `error NotRecipient()` will have `result` equal to the first 4 bytes of `keccak256("NotRecipient()()")`.

Recommendations

Consider wrapping the inner error in a batch error which contains inner results as seen in the following code:

```
error BatchError(bytes innerError);

...
function batch(bytes[] calldata calls, bool revertOnFail) external payable {
  for (uint256 i = 0; i < calls.length; i++) {
    (bool success, bytes memory result) = address(this).delegatecall(
      calls[i]
    );
    if (!success) {
      revert BatchError(result);
    }
  }
}
```

Resolution

The development team has fixed the issue by creating a custom error:

```
error BatchError(bytes innerError);
```

The code reverts with this error if the length of the return bytes from the `delegatecall()` is less than 68.

This is shown in multiple commits, the latest being ed94c55.

SSAF-17	Any User Can Drain Ether That Is Left In The Contract
Asset	`FuroStream.sol` & `FuroVesting.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

In the functions `FuroStream.createStream()` and `FuroVesting.createVesting()`, the amount of ETH used to create the stream or vest is determined by a parameter to the function, not by reference to `msg.value`. This approach is used to provide compatibility with `BoringBatchable`. However, a side effect is that it is possible to overpay and leave ETH in the contract.

If one user overpays ETH by submitting a low value for `amount`, it would be possible for another user to then claim all that overpaid ETH by calling the same function with `msg.value` set to zero and the function parameters set to the balance of ETH in the contract. This would create a new stream or vest in the attacker's control, which could have a very short duration. Hence the attacker could quickly claim all the ETH.

Recommendations

`BoringBatchable` requires an approach of this nature, which causes this issue. One potential mitigation would be to remove `BoringBatchable`, but that removes the batching functionality. Alternatively, ensure the development team and users are aware of this behaviour in the related documentation.

Resolution

The development team acknowledged the issue and decided to keep `BoringBatchable`.

SSAF-18	`BoringOwnable` is Unused
Asset	`FuroStream.sol` & `FuroVesting.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Both contracts `FuroStream` and `FuroVesting` are inheriting from the contract `BoringOwnable`. However, neither of the contracts use any of the modifiers, functions or variables from `BoringOwnable`.

In the case of `FuroVesting`, which uses the term *"owner"* to refer to the address that creates a vest, the existence of `BoringOwnable` also creates a number of ambiguities in function, modifier and variable names.

Recommendations

We recommend removing the `BoringOwnable` inheritance from both `FuroStream` and `FuroVesting`.

Resolution

The development team has fixed the issue by removing `BoringOwnable` as recommended.

This is shown in PR 16.

SSAF-19	Inconsistent End States for Streams and Vests
Asset	`FuroStream.sol` & `FuroVesting.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Both contracts `FuroStream` and `FuroVesting` have two end states: the stream/vest can be terminated by its creator, or it can come to the end of its time period (after which it can also be terminated).

When a creator terminates, the contracts delete the state variables. When the stream or vest reaches its end point, it perists in storage.

Recommendations

Consider deleting the storage after a stream/vest has transferred all funds to the recipient.

Resolution

The development team acknowledged the issue and decided not to remediate.

SSAF-20	NFTs Persist After Use
Asset	`FuroStream.sol` & `FuroVesting.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Both contracts `FuroStream` and `FuroVesting` create NFTs for their recipients. When a stream or vest is finished or cancelled, the NFTs are left under the control of those users.

There might be scope for these NFTs to be sold under false pretences or otherwise used in a scam.

Recommendations

The NFTs can be safely burnt when streams or vests are deleted.

Note that burning the NFT when funds are exhausted but the stream is not deleted will cause an issue if future funds are added to the stream/vest.

Resolution

The development team acknowledged the issue and decided not to remediate.

SSAF-21	Denial-of-Service Condition by Maliciously Extending Auctions
Asset	`SushiMakerAuction.sol`
Status	Open
Rating	Informational

Description

It is possible to prevent genuine use of any auction for up to `bid.maxTTL` in certain scenarios.

This can be done by starting and repeatedly updating (via `placeBid()`) an auction for a token with no or very low initial balance. This auction will run for a maximum of `maxTTL` seconds.

When the token balance is increased significantly after that, a new auction can not happen until the previous auction has finished.

Since `unwindLP()` is a public function which can be called by any user, it is possible for an attacker to transfer their own tokens of negligible value, e.g. `MIN_BID`, then start the auction. After the auction is started, the attacker may call `unwindLP()` transferring the reward tokens into the auction make contract. The genuine users will have to wait for `maxTTL` before they are able to start a new auction for the unwound tokens.

Recommendations

Ensure the risks and potential attack vectors are understood.

SSAF-22	Insufficient Comments, Documentation & Tests
Asset	`FuroStream.sol`, `FuroVesting.sol` & `SushiMakerAuction.sol`
Status	**Open**
Rating	Informational

Description

There are very limited inline comments throughout the contract. It is considered best practice to document the purpose and limitations of functions and calculations.

There is also insufficient testing for these contracts. Testing helps identify and prevent bugs during the development cycle.

Recommendations

Consider adding both NatSpec and inline comments to increase readability and explain intended behaviour.

In addition to inline comments, it is also beneficial to have external documentation describing the high level overview of the protocol and how users may interact with it.

More exhaustive tests should be added, with the aim of 100% code coverage.

SSAF-23	Other `Auction Maker` and `Furo` Comments
Asset	`auction-maker/` and `furo/`
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings in `auction-maker` and `furo` repository that do not have direct security implications:

1. **`auction-maker/SushiMakerAuction.sol`**

   1a) Gas Optimisations:

   - Redundant storage reads
     The function `placeBid()` repeatedly reads the same storage variables of `bid` defined in line [**96**]. To save some gas the storage location could be changed to `memory`. The storage writes in line [**112-114**] need to be handled separately.
   - `nonReentrant` modifier is unnecessary
     The `start()`, `placeBid()` and `end()` functions use the `nonReentrant` to prevent againt reentrancy attack. This modifier is not needed anymore in the new PR 1 as these functions do not have an external call.

   1b) Users may outbid themselves:
   Users can outbid themselves because it is not checked if a new bidder is the same as the latest bidder. This could be unexpected and lead to lost funds. It could be a side effect of the feature that the bid's receiver is not equal to `msg.sender` but set by the user. However, users should at least be made aware of this.

   1c) Possible missing event emission:
   `updateReceiver()` is a critical function. Off-chain monitoring platforms could potentially be interested in this. Consider emitting an event when it's executed to indicate that the receiver has changed.

2. **`furo/FuroStream.sol`**

   2a) Possible missing calls to `onTaskReceived()`:
   `ITasker(to).onTaskReceived(taskData)` is called in `withdrawFromStream()` but not in any other functions that interact with the stream. The testing team recommends ensuring that this functionality matches the intended functionality.

   2b) `ITasker(to).onTaskReceived(taskData)` can be triggered by stream sender:
   The recipient can not trust the data passed into `onTaskReceived()` even when it is coming from `FuroStream` because the sender can trigger its execution by calling `withdrawFromStream()`, too. This is a feature, but it's critical that the recipient is aware of this and does not perform sensitive operations in the function. If necessary, exercise caution and employ appropriate access checks.

   2c) Unused custom error:
   There is an unused custom error `NotRecipient()` that can be removed.

   2d) Similar names of functions and variables.
   There are some function and variable names that are very similar to those provided by inherited contracts, and yet have different meanings.

- There is a `balanceOf()` function in OpenZeppelin's `ERC721` and we have `FuroStream._balanceOf()` performing a very different function.
- In `FuroStream._transferToken()` The `amount` parameter is actually a shares parameter. It would be clearer if this were renamed.

3. **`furo/FuroVesting.sol`**

3a) Misleading custom error usage: `NotOwner`.

In `stopVesting()` and `updateOwner()`, the custom error `NotOwner` is used to indicate that the caller is not the owner. In `withdraw()`, the same error is used to indicate that the caller is not the recipient. This inconsistency should be addressed.

3b) Similar names of functions and variables.

There are some function and variable names that are very similar to those provided by inherited contracts, and yet have different meanings.

- Throughout `FuroVesting`, the term "balance" is being used for two quantities: the currently claimable amount in `vestBalance` and the total amount qualified for claiming in `_balanceOf`.
- As above, there is a `balanceOf()` function in OpenZeppelin's `ERC721` and we also have `FuroVesting._balanceOf()` performing a very different function.
- `FuroVesting` will have `transferOwnership()` and `updateOwner()` that perform very different functions. The contract also uses "owner" for two entirely different meanings (note that contract owner is not being used).

4. **`furo/BoringBatchable.sol`**

4a) ABI v2 does not need to be activated.

line [3] activates `ABIEncoderV2` but in solidity 0.8 this is on by default. This line can be safely removed.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

These points have been acknowledged by the development team, and some related recommendations have been applied.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

Auction Maker Tests

```
test_start                              PASSED  [4%]
test_start_token_with_fallback          PASSED  [9%]
test_start_bid_token                    PASSED  [13%]
test_start_lp_token                     PASSED  [18%]
test_start_insufficient_bid_amount      PASSED  [22%]
test_start_auction_started              PASSED  [27%]
test_start_invalid_bidder               XPASS   (Nozero...)
test_start_no_reward                    XPASS   (Noinput
test_placeBid                           PASSED  [40%]
test_placeBid_auction_not_started       PASSED  [45%]
test_placeBid_auction_finished          PASSED  [50%]
test_placeBid_insufficient_bid          PASSED  [54%]
test_placeBid_invalid_bidder            XFAIL   (Noz...)
test_end_changing_rewards_balance       PASSED  [63%]
test_end                                PASSED  [68%]
test_end_non_standard_token             PASSED  [72%]
test_end_auction_not_started            PASSED  [77%]
test_end_auction_not_finished           PASSED  [81%]
test_unwindLP                           PASSED  [86%]
test_skimBidToken                       PASSED  [90%]
test_updateReceiver                     PASSED  [95%]
test_updateWhitelistToken               PASSED  [100%]
```

Furo Tests

```
test_batch                              PASSED  [2%]
test_batchETHandWETH                    PASSED  [4%]
test_constructor                        PASSED  [7%]
test_renounceownership                  PASSED  [9%]
test_renounceownershipfalse             PASSED  [11%]
test_transferownershipdirect            PASSED  [14%]
test_transferownershipindirect          PASSED  [16%]
test_transferownershipindirectbadclaim  PASSED  [19%]
test_onlyowner                          PASSED  [21%]
test_constructor                        PASSED  [23%]
test_createStream                       PASSED  [26%]
test_createStreamETH                    PASSED  [28%]
test_sweepETHWithWETH                   PASSED  [30%]
test_withdrawFromStream                 PASSED  [33%]
test_withdrawFromStreamWithTask         PASSED  [35%]
test_cancelStream                       PASSED  [38%]
test_getStream                          PASSED  [40%]
test_streamBalanceOfStart               PASSED  [42%]
test_streamBalanceOfMiddle              PASSED  [45%]
test_streamBalanceOfMiddleWithWithdrawal PASSED  [47%]
test_streamBalanceOfEnd                 PASSED  [50%]
test_updateSender                       PASSED  [52%]
test_updateStream                       PASSED  [54%]
test_updateStreamETH                    PASSED  [57%]
test_createStreamETHUnderpay            PASSED  [59%]
test_constructor                        PASSED  [61%]
test_createVesting                      PASSED  [64%]
test_createVestingInvalidStart          PASSED  [66%]
test_withdraw                           PASSED  [69%]
test_withdrawNotOwner                   PASSED  [71%]
test_stopVesting                        PASSED  [73%]
test_stopVestingCalculationError        PASSED  [76%]
```

```
test_amountSharesIssue                    PASSED  [78%]
test_vestBalanceNothing                   PASSED  [80%]
test_vestBalanceCliffOnly                 PASSED  [83%]
test_vestBalanceSomeSteps                 PASSED  [85%]
test_vestBalanceAllSteps                  PASSED  [88%]
test_updateOwner                          PASSED  [90%]
test_updateOwnerNotOwner                  PASSED  [92%]
test_oneGiantLeap                         PASSED  [95%]
test_emptyVest                            PASSED  [97%]
test_excessiveSteps                       PASSED  [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact \ Likelihood	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].

σ'