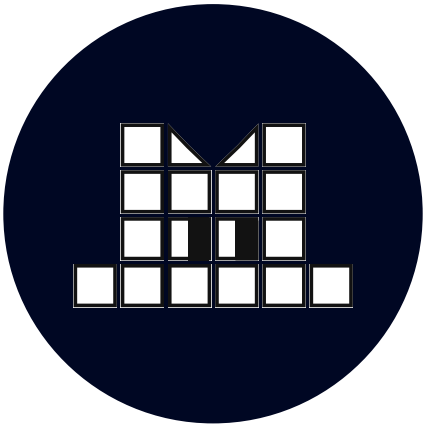

Audit report for Blast Futures Exchange

February 27, 2024

Introduction to Hats.finance

Hats.finance builds autonomous security infrastructure for integration with major DeFi protocols to secure users' assets. It aims to be the decentralized choice for Web3 security, offering proactive security mechanisms like decentralized audit competitions and bug bounties. The protocol facilitates audit competitions to quickly secure smart contracts by having auditors compete, thereby reducing auditing costs and accelerating submissions. This aligns with their mission of fostering a robust, secure, and scalable Web3 ecosystem through decentralized security solutions.

Blast Futures Exchange overview

Blast Futures Exchange (BFX) is a hybrid decentralised orderbook perpetual futures exchange with native yield. Blast Futures Exchange introduces native yield, allowing users' balances to automatically compound, and provide a high-quality trading experience with easy onboarding, advanced charting, and efficient order execution. The platform also features a Fusion Dynamic Automated Market Maker (AMM) to efficiently and optimize liquidity provision by using up-to-date market data, making it more seamless and profitable for users to provide liquidity to the exchange.

Competition details

- Type: A public audit competition hosted by Hats.finance.
- Duration: 10 days - From February 5 to 15.
- Maximum Reward: $8,000
- Total Payout: $6,000

Scope of the audit

Only the smart contracts of BFX were in the scope for this audit competition. The off-chain components of the exchange are not included in the scope of this competition. Here is the list of all the folders and files that are inside the scope:

```
|-- Blast-Futures-Exchange/
     |-- foundry
          |-- src
               |-- Bfx.sol
               |-- BfxVault.sol
               |-- EIP712Verifier.sol
               |-- IPoolDeposit.sol
               |-- IVault.sol
               |-- PoolDeposit.sol
```

These can be found in the audit competition repo: Blast-Futures-Exchange-0x97895c329b950755566ddcdad3395caaea395074.

Findings

A total of 10 valid Low severity findings were identified during the competition.

All the submissions for the competition are found here.

High severity findings:

No high severity findings were identified.

Medium severity findings:

No medium severity findings were identified.

Low severity findings:

[L01] - **There is no way to withdraw tokens from Bfx when deposited from `BfxVault` via Treasurer**

The protocol seems to work mainly off-chain. Deposits are done on-chain and after deposits, events are emitted with `(depositdId, msg.sender, amount)` , this is same for staking, only `depositId` became `StakeId` . So the withdrawals are using those events and are done off-chain. Assumingly to do that users need to provide their depositId/stakeId in UX and after the off-chain system checked that the event's (which emitted during deposit) msg.sender is the same as the requester of withdrawal (otherwise it would be possible to steal anyone's funds so this check is not optional), it will proceed with starting withdrawing process. In `BfxVault.sol` , the function `makeDeposit()` basically deposits tokens that are available in the contract to `Bfx.sol` via Bfx's `deposit()` function. The contract can not possibly interact with the UX to start the withdraw process, so it won't be able to withdraw tokens and funds will be stuck.

As a side note, this is also the case for all contracts (that are not EOA). It is important to document this thoroughly so that users won't lose funds.

[L02] - **SPDX license used in contracts deviates from readme**

The github repo of `Blast-Futures-Exchange` readme specifically mentions,

> Blast Futures Exchange is released under the MIT License.

However, this is not correct. See below contracts using different licenses.

1. Bfx.sol - `// SPDX-License-Identifier: BUSL-1.1`

2. BfxVault.sol- `// SPDX-License-Identifier: BUSL-1.1`
3. EIP712Verifier.sol- `// SPDX-License-Identifier: MIT`
4. IPoolDeposit.sol- `// SPDX-License-Identifier: BUSL-1.1`
5. IVault.sol- `// SPDX-License-Identifier: BUSL-1.1`
6. PoolDeposit.sol- `// SPDX-License-Identifier: BUSL-1.1`

only one contract has MIT license and it must be noted that making source code available always touches on legal problems with regards to copyright. The Business Source License (BUSL) is not an Open Source license and MIT license gives express permission for users to reuse code for any purpose. Per readme, It can be seen that the code wants to open source, Therefore, all contracts should be MIT licensed.

Recommendation: Keep the license type same in all contracts, if readme info is correct then all contracts should have MIT license.

[L03] - **Lack of minimum amount for deposit, staking or withdrawal**

Currently, there is no minimum amount of deposit, staking or withdrawal in BFX which will raise some issues. The only check on contracts are checks if the amount for deposit, staking or withdrawal is non zero: `require(amount > 0, "WRONG_AMOUNT");` , which allows for a dust amount like 1 wei to be sent to the contract. A large number of dust accounts can contribute to network congestion and spam, increasing gas fees and disrupting the flow of dex trading operations. Also, in a bigger picture, bigger BFX perpetual future dex product, if this dust accounts are allowed to participate in certain activities like liquidations, their negligible size can create noise and distort price discovery within the DEX. On staking, dust accounts contribute very little to the staking pool, potentially diluting the rewards earned by larger stakeholders and creating an unfair distribution. Distributing rewards to a large number of small stakeholders may become logistically challenging. It could lead to increased transaction costs and potential delays in the distribution process. Moreover, if there is a case when smart contracts need to be upgraded, this massive number address issues becomes more challenging due to a large number of small stakeholders are involved.

Attack Scenario:

1. Alice knows the BFX use USDC (with 6 decimal) for deposit, staking or withdrawal
2. Alice can create script to flood the deposit and staking by creating burner wallet and deposit as low as 0.000001 USDC (assuming Blast fee is negligible), even more on `pooledDeposit()` which accept arrays of Contributions.
3. BFX flooded with this fake spam deposit events, which might disturbing the offchain accounting

Recommendation: Consider to apply minimum deposit/staking amounts

[L04] - **`makeOwnerAdmin()` is not protecting enough from malicious admin act**

BFX vault have 3 roles: Admin, Trader and Treasurer. The Admin users can add and remove roles, this include the admin role of the owner.

```
File: BfxVault.sol
189:     function addRole(address signer, uint256 role) public {
190:         require(signers[msg.sender][ADMIN_ROLE], "NOT_AN_ADMIN");
191:         signers[signer][role] = true;
192:         emit AddRole(signer, role);
193:     }
...
206:     function removeRole(address signer, uint256 role) public {
207:         require(signers[msg.sender][ADMIN_ROLE], "NOT_AN_ADMIN");
208:         signers[signer][role] = false;
209:         emit RemoveRole(signer, role);
210:     }
```

Interestingly, there is a `makeOwnerAdmin()` function to restore `owner` to become `admin` (again). Meanwhile, `owner` is immutable (means it will not be changed/transfered), and in the constructor this `owner` is already set as `admin` , so the existance of this `makeOwnerAdmin()` implies the edge case of a malicious admin removing the Admin role from the owner.

```
File: BfxVault.sol
212:     function makeOwnerAdmin() external onlyOwner {
213:         signers[owner][ADMIN_ROLE] = true;
214:     }
```

However, this (makeOwnerAdmin) backup mechanism doesn't really fix the situation. Again, assuming that `makeOwnerAdmin()` is to restore the owner to be admin again due to the admin turning malicious, then, this malicious admin need to be cleared or removed by the owner. Since admin can add make other addresses admin as well, it's possible this malicious admin can assign many address to became admin or any roles. Therefore, the owner should manually remove this malicious backup admin. It's like chasing each other between owner and malicious admin, which can involve any front-run mechanics. The only way to 'fix' this is to pause the `addRole` function, to prevent any further malicious role being added.

Attack Scenario:

1. Alice is owner (and admin), Bob, Carol are assigned by Alice as admin
2. Bob turns out act maliciously, and remove Alice, and Carol as Admin
3. Alice gain her admin via `makeOwnerAdmin()`
4. Bob sees Alice gain her admin, thus he now trying to assign other backup address as Admin
5. Alice keep up head to head with Bob, by removing role of Bob's backup address
6. Bob keep front-run Alice by adding more address as Admin, meanwhile he can act maliciously by gaining the Admin role.

Recommendation: Consider adding a pausing mechanism on `addRole` which is accessible only by the owner.

[L05] - **Missing events for functions that change critical parameters**

The `onlyOwner` functions that change critical parameters should emit events. Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes with timelocks that allow users to evaluate them and consider if they would like to engage/exit based on how they perceive the changes as affecting the trustworthiness of the protocol or profitability of the implemented financial services. The alternative of directly querying on-chain contract state for such changes is not considered practical for most users/usages. Missing events do not promote transparency and if such changes immediately affect users' perception of fairness or trustworthiness, they could exit the protocol causing a reduction in liquidity which could negatively impact protocol TVL and reputation. In `Blast Future Exchange` contracts, below are owner functions that do not emit any events in the contracts.

1. In `Bfx.sol`

```
function setPaymentToken(address _paymentToken) external onlyOwner {
    paymentToken = IERC20(_paymentToken);
}

function changeSigner(address new_signer) external onlyOwner {
    require(new_signer != address(0), "ZERO_SIGNER");
    external_signer = new_signer;
}
```

2. In `BfxVault.sol`

```
function setPaymentToken(address _paymentToken) external onlyOwner {
    paymentToken = IERC20(_paymentToken);
}


function setBfx(address _bfx) external onlyOwner {
    bfx = IBfx(_bfx);
}
```

3. In `PoolDeposit.sol`

```
function setPaymentToken(address _paymentToken) external onlyOwner {
    paymentToken = IERC20(_paymentToken);
}


function setRabbit(address _rabbit) external onlyOwner {
```

```
        rabbit = _rabbit;
    }
```

Recommendation: Add events to all `onlyOwner` functions that change critical parameters.

[L06] - `TYPEHASH` passed in `Bfx.withdraw()` does not comply with EIP-712 and will return incorrect `digest`

`Bfx.withdraw()` allows a withdrawal operation with EIP-712 signature verification.

```
    function withdraw(
        uint256 id, address trader, uint256 amount, uint8 v, bytes32 r, bytes32 s
        ) external nonReentrant {

    . . .some code

        bytes32 digest = _hashTypedDataV4(keccak256(abi.encode(
            keccak256("withdrawal(uint256 id,address trader,uint256 amount)"),
            id,
            trader,
            amount
        )));

    . . .some code

    }
```

In order to compute `digest`, the `_hashTypedDataV4()` is taken from openzeppelin `EIP712.sol` which is compliant with EIP-712. This function takes `TYPEHASH` as one of the argument to return the bytes32 `digest`. In `Bfx.withdraw()`, it is `TYPEHASH = keccak256("withdrawal(uint256 id,address trader,uint256 amount)")`. The protocol documentation ensures the withdrawal comply 100% with EIP712, however, the above `TYPEHASH` does not comply with EIP712. EIP712 is used for `Typed structured data hashing and signing`. and the data used in `TYPEHASH` is the structured data. For example, per EIP712,

```
    struct Mail {
        address from;
        address to;
        string contents;
    }

    bytes32 constant MAIL_TYPEHASH = keccak256(
      "Mail(address from,address to,string contents)");
```

In this example a struct named `Mail` is used to get the `MAIL_TYPEHASH`. Now, in the current implementation, `withdrawal` is a struct name which does not match the correct struct definition.

The first letter of struct should be capital, like `Withdrawal` and NOT `withdrawal`. As solidity docs specifically mentions,

> Structs should be named using the CapWords style. Examples: MyCoin

A difference in first letter capitalization will give entirely different hash by keccak256.

While testing the `TYPEHASH` in `Remix IDE`, the results are different. With the current implementation given in the contract, i.e

```
bytes32 public constant WITHDRAWAL_TYPEHASH =
    keccak256("withdrawal(uint256 id,address trader,uint256 amount)");
```

The result is `0xec976281d6462ad970e7a9251148e624b8aa376c6857d4245700b1b711bb0884` and the proposed recommendation i.e incompliance with EIP712

```
bytes32 public WITHDRAWAL_TYPEHASH =
    keccak256("Withdrawal(uint256 id,address trader,uint256 amount)");
```

The result is `0x92351042d160b58aef05f219c6acec14f0d71e7be3befc892862667ae62b1a96`.

Thus, the current implementation deviates from EIP712 and solidity struct naming convention. This will deviate from EIP712 which the contracts should adhere to.

Recommendation: Modifiy the code as below so that TYPEHASH should be as per EIP712.

```
+  bytes32 public WITHDRAWAL_TYPEHASH =
+        keccak256("Withdrawal(uint256 id,address trader,uint256 amount)");

    function withdraw(

   . . .some code

-        bytes32 digest = _hashTypedDataV4(keccak256(abi.encode(
-            keccak256("withdrawal(uint256 id,address trader,uint256 amount)"),
-            id,
-            trader,
-            amount
-        )));


+        bytes32 digest = _hashTypedDataV4(keccak256(abi.encode(
+            WITHDRAWAL_TYPEHASH,
+            id,
+            trader,
+            amount
+        )));
```

```
. . .some code
```

[L07] - **Vulnerability in `pooledDeposit` function enables Denial of Service (DoS) attack**

The function named `PoolDeposit:pooledDeposit()` accepts an array of contributions (`Contribution[]`). This function aggregates the total amount of contributions and then transfers the total amount to a designated address (rabbit). However, there's a vulnerability in the code due to the lack of limit on the size of the contributions array, combined with the function's external visibility.

Attack Scenario: Denial of Service (DoS) Attack. An attacker exploits the vulnerability by calling the `PoolDeposit:pooledDeposit()` function with an excessively large contributions array. Since the function's visibility is external, it can be called by anyone. The attacker sends a huge array of contributions, causing the function to iterate through the array in a loop. As the function iterates through the large array, it consumes significant gas for each iteration. With a sufficiently large array, the gas cost of processing the loop exceeds the block gas limit, leading to a denial of service condition.Impact: The excessive gas consumption prevents legitimate transactions from being processed.

Recommendation: Consider implementing gas limits on the number of contributions that can be processed in a single transaction or using batch processing techniques to handle large arrays efficiently. Additionally, consider adjusting the visibility of the function to restrict access and prevent unauthorized calls.

Attachments

1. **Proof of Concept (PoC) File**

```
function pooledDeposit(Contribution[] calldata contributions) external {
        uint256 poolId = allocatePoolId();
        uint256 totalAmount = 0;
        for (uint i = 0; i < contributions.length; i++) {
            Contribution calldata contribution = contributions[i];
            uint256 contribAmount = contribution.amount;
            totalAmount += contribAmount;
            require(contribAmount > 0, "WRONG_AMOUNT");
            require(totalAmount >= contribAmount, "INTEGRITY_OVERFLOW_ERROR");
            uint256 depositId = allocateDepositId();
            emit Deposit(depositId, contribution.contributor, contribAmount, poolId)
        }
        require(totalAmount > 0, "WRONG_AMOUNT");
        emit PooledDeposit(poolId, totalAmount);
        bool success = makeTransferFrom(msg.sender, rabbit, totalAmount);
        require(success, "TRANSFER_FAILED");
    }
```

[L08] - **Deposit is broken for tokens that transfer the max balance of user on max transfer of `uint256.max` , which can lead to loss of fund to protocol**

There are some tokens, that on transferring `uint256.max` amount of tokens, instead of reverting in case the user doesn't hold that amount of balance, transfers the whole user balance. So in case the underlying token is cUSDCv3, which is not a low market cap token so possibility of it being used is pretty good. The market cap of cUSDCv3 at the time of submission is $139,433,046 (Reference from here : https://www.coingecko.com/en/coins/compound-usd-coin).

Example: if the user has approved `uint256.max` to the contract but has 100 tokens, and the contract tries to transfer `uint256.max` tokens from the user it will receive 100 tokens, but the accounting will believe the user has transferred `uint256.max` tokens.

Look at the following submission in one of Sherlock's contests that is similar to the current case: https://github.com/sherlock-audit/2023-09-Gitcoin-judging?tab=readme-ov-file#issue-m-8-problems-with-tokens-that-transfer-less-than-amount-separate-from-fee-on-transfer-issues1.

In our case, the deposit function looks like the following

```
function deposit(uint256 amount) external nonReentrant {
    bool success = makeTransferFrom(msg.sender, address(this) , amount);
    require(success, "TRANSFER_FAILED");
    uint256 depositId = allocateDepositId();
    emit Deposit(depositId, msg.sender, amount);
}
```

which means that if a token like cUSDCv3 is being used that contains a special case for amount == type(uint256).max in their transfer functions it results in only the user's balance being

transferred. This means that a user can deposit as low as 1 wei but the off-chain nodes will see that the user deposits a very large amount as the function emits the user passed-in amount value that will be uin256.max. So in such cases, a user can get a signature signed for a withdraw amount that he hasn't even deposited, and can in such cases cause loss of funds to the protocol.

[L09] - **Protocol will not work with tokens that do not return bool value on approve calls**

The protocol is intended to work with any ERC20 compatible tokens and even with some incompatible i.e. tokens that do not return a `bool` value when calling some of their functions such as `transfer` or `transferFrom` directly. This is achieved by utilizing low-level calls within the transfer functions implemented in the protocol. However within the `BfxVault.sol` contract we have the `makeDeposit()` function that is callable only by users with `TREASURER_ROLE` . This function calls an internal `_doDeposit()` function which executes two steps:

1. Approves the BFX contract to spend a desired amount
2. Invokes the `deposit()` function on the BFX contract which transfers the provided amount from the BfxVault contract.

In order for the execution to happen properly the `TREASURER` has to transfer the specified amount before invoking the `makeDeposit()` function. The vulnerability here presents itself during the execution of `makeDeposit()` where a call is made to `paymentToken.approve()` , where `paymentToken` is wrapped with the IERC20 interface. However for tokens that do not return a `bool` value when calling approve such as USDT on Ethereum, the transaction will revert with an EVM error.

Attack Scenario: If `paymentToken` is set to USDT or any other token that do not return a `bool` value when `approve()` function is called, the TREASURER of the BfxVault will not be able to make a deposit from the vault to the Bfx Exchange due to failure in the `makeDeposit()` function execution.

Attachments:

1. **Proof of Concept (PoC) File** The `BfxVault.t.sol` is the PoC for the issue. Steps to run it:

- Add the file to the `test` folder
- Add an RPC url as a local env variable called `ETH_RPC_URL`
- Run the test with `forge test --match-contract BfxFail --fork-url $ETH_RPC_URL -vv`
- You should see a failed test with `EvmError: Revert` message

2. **Revised Code File (Optional)**

- Added a `_makeApprove` internal function which utilizes low-level calls similar to `_makeTransfer` and `_makeTransferFrom`
- Optional: You can also add a `_makeApprove` call to set the approval to 0 in order to avoid

future issues with tokens that require approval to 0 first

Files:

- BfxVault.t.sol
- BfxVault.sol

[L10] - `_tokenCall()` does not revert for a `paymentToken` with no deployed code on it

Three contracts process payment tokens by means of the function `_tokenCall()`. This function is meant to handle different ERC20 weirdness. However, it fails to handle a contract address with no code deployed on it, as the `_tokenCall()` will return `success=true` for such addresses. If the contract operators wrongly configure the `paymentToken` address in any of the contracts, a malicious address could call functions like `deposit()` or `stake()` which will successfully go through, without reverting. This will break the protocol accounting balances.

Attack Scenario:

- Contract owners set a wrong `paymentToken` by mistake, which has no code deployed on it
- An observant attacker performs `stake()` with a large amount, which will be processed successfully, making it look as if he truly staked a large amount.

Attachments:

1. **Proof of Concept (PoC) File: emptyPaymentTokenConfigured.sol** The PoC is a test. The stake function should revert if the vulnerability is fixed.
2. **Revised Code File: BfxVault.sol** The fix is provided for `BfxVault.sol`, but the same fix applies to `Bfx.sol` and `PoolDeposit.sol`

Files:

- emptyPaymentTokenConfigured.sol
- BfxVault.sol

Conclusion

The audit uncovered 10 Low severity issues. The overall logic of the smart contracts is minimal as accounting and order matching occurrs off-chain. This means there is a small attack surface and low complexity. The most concerning part was the signature verification, but that showed no major issues after review.

Disclaimer

This report does not assert that the audited contracts are completely secure. Continuous review

and comprehensive testing are advised before deploying critical smart contracts.

The BFX audit competition illustrates the collaborative effort in identifying and rectifying potential vulnerabilities, enhancing the overall security and functionality of the platform.