Foundation contest

2022-05-19

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Foundation smart contract system written in Solidity. The audit contest took place between February 24—March 2 2022.

Wardens

34 Wardens contributed reports to the Foundation contest:

1. leastwood
2. IllIllI
3. cmichel
4. WatchPug (jtp and ming)
5. 0xliumin
6. cccz
7. gzeon
8. 0x1f8b
9. hyh
10. Dravee
11. shenwilly
12. wuwe1
13. thankthedark (d4rk and thank_you)
14. CertoraInc (danb, egjlmn1, OriDabush, ItayG, and shakedwinder)
15. defsec
16. Afanasyevich
17. pedroais

18. TerrierLover
19. csanuragjain
20. hubble (ksk2345 and shri4net)
21. kenta
22. rfa
23. robee
24. iain
25. 0xwags
26. kirk-baird
27. Ruhum
28. jayjonah8

This contest was judged by Alberto Cuesta Cañada.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 21 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 18 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 20 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 11 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Foundation contest repository, and is composed of 16 smart contracts written in the Solidity programming language and includes 1,689 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges

- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (3)

[H-01] NFT owner can create multiple auctions

Submitted by 0xliumin, also found by leastwood

NFTMarketReserveAuction.sol#L325-L349 NFTMarketReserveAuction.sol#L596-L599

NFT owner can permanently lock funds of bidders.

Proof of Concept

Alice (the attacker) calls `createReserveAuction`, and creates one like normal. let this be auction id 1.

Alice calls `createReserveAuction` again, before any user has placed a bid (this is easy to guarantee with a deployed attacker contract). We'd expect that Alice wouldn't be able to create another auction, but she can, because `_transferToEscrow` doesn't revert if there's an existing auction. let this be Auction id 2.

Since `nftContractToTokenIdToAuctionId[nftContract][tokenId]` will contain auction id 2, all bidders will see that auction as the one to bid on (unless they inspect contract events or data manually).

Alice can now cancel auction id 1, then cancel auction id 2, locking up the funds of the last bidder on auction id 2 forever.

Recommended Mitigation Steps

Prevent NFT owners from creating multiple auctions.

NickCuso (Foundation) confirmed and commented: > This is a great find! > > The impact of this bug is: > - Bidder's funds are stuck in escrow in an unrecoverable way without an upgrade, and even with an upgrade it would have been non-trivial to offer a migration path to recover the funds (but it would have been possible to recover correctly). > - It allows sellers to stop the clock and/or back out of an auction. Normally once a bid is received we do not allow the seller to cancel the auction. With this bug, they could have created a new auction and then cancel that in order to back out of the deal entirely. This violates trust with collectors. > > We have fixed this problem by adding the following code

to createReserveAuction: > > solidity > // This check must be after _transferToEscrow in case auto-settle was required > if (nftContractToTokenIdToAuctionId[nftContract][tokenId] != 0) { > revert NFTMarketReserveAuction_Already_Listed(nftContractToTokenIdToAuctionId[nftContract][t > } >

[H-02] Creators can steal sale revenue from owners' sales

Submitted by IllIllI

NFTMarketCreators.sol#L158-L160 NFTMarketCreators.sol#L196-L198 NFTMarketCreators.sol#L97-L99

According to the `README.md`:

> All sales in the Foundation market will pay the creator 10% royalties on secondary sales. This is not specific to NFTs minted on Foundation, it should work for any NFT. If royalty information was not defined when the NFT was originally deployed, it may be added using the Royalty Registry which will be respected by our market contract.

Using the Royalty Registry an owner can decide to change the royalty information right before the sale is complete, affecting who gets what.

Impact

By updating the registry to include the seller as one of the royalty recipients, the creator can steal the sale price minus fees. This is because if code finds that the seller is a royalty recipient the royalties are all passed to the creator regardless of whether the owner is the seller or not.

Proof of Concept

```
// 4th priority: getRoyalties override
if (recipients.length == 0 && nftContract.supportsERC165Interface(type(IGetRoyalti
  try IGetRoyalties(nftContract).getRoyalties{ gas: READ_ONLY_GAS_LIMIT }(tokenId)
    address payable[] memory _recipients,
    uint256[] memory recipientBasisPoints
  ) {
    if (_recipients.length > 0 && _recipients.length == recipientBasisPoints.lengt
      bool hasRecipient;
      for (uint256 i = 0; i < _recipients.length; ++i) {
        if (_recipients[i] != address(0)) {
          hasRecipient = true;
          if (_recipients[i] == seller) {
            return (_recipients, recipientBasisPoints, true);
```

https://github.com/code-423n4/2022-02-concur/blob/72b5216bfeaa7c529830
60ebfc56e72e0aa8e3b0/contracts/MasterChef.sol#L127-L154

When `true` is returned as the final return value above, the following code leaves `ownerRev` as zero because `isCreator` is `true`.

```
      uint256 ownerRev
    )
  {
    bool isCreator;
    (creatorRecipients, creatorShares, isCreator) = _getCreatorPaymentInfo(nftContract, toke

    // Calculate the Foundation fee
    uint256 fee;
    if (isCreator && !_nftContractToTokenIdToFirstSaleCompleted[nftContract][tokenId]) {
      fee = PRIMARY_FOUNDATION_FEE_BASIS_POINTS;
    } else {
      fee = SECONDARY_FOUNDATION_FEE_BASIS_POINTS;
    }

    foundationFee = (price * fee) / BASIS_POINTS;

    if (creatorRecipients.length > 0) {
      if (isCreator) {
        // When sold by the creator, all revenue is split if applicable.
        creatorRev = price - foundationFee;
      } else {
        // Rounding favors the owner first, then creator, and foundation last.
        creatorRev = (price * CREATOR_ROYALTY_BASIS_POINTS) / BASIS_POINTS;
        ownerRevTo = seller;
        ownerRev = price - foundationFee - creatorRev;
      }
    } else {
      // No royalty recipients found.
      ownerRevTo = seller;
      ownerRev = price - foundationFee;
    }
  }
```

In addition, if the index of the seller in `_recipients` is greater than `MAX_ROYALTY_RECIPIENTS_INDEX`, then the seller is omitted from the calculation and gets zero (`_sendValueWithFallbackWithdraw()` doesn't complain when it sends zero).

```
        uint256 maxCreatorIndex = creatorRecipients.length - 1;
        if (maxCreatorIndex > MAX_ROYALTY_RECIPIENTS_INDEX) {
          maxCreatorIndex = MAX_ROYALTY_RECIPIENTS_INDEX;
        }
```

https://github.com/code-423n4/2022-02-foundation/blob/4d8c8931baffae31c7506872bf1100e1598f2754/contracts/mixins/NFTMarketFees.sol#L76-L79

This issue does a lot of damage because the creator can choose whether and when to apply it on a sale-by-sale basis. Two other similar, but separate, exploits are available for the other blocks in `_getCreatorPaymentInfo()` that return arrays but they either require a malicious NFT implementation or can only specify a static seller for which this will affect things. In all cases, not only may the seller get zero dollars for the sale, but they'll potentially owe a lot of taxes based on the 'sale' price. The attacker may or may not be the creator - creators can be bribed with kickbacks.

Recommended Mitigation Steps

Always calculate owner/seller revenue separately from royalty revenue.

NickCuso (Foundation) confirmed and commented: > This is a great discovery and a creative way for creators to abuse the system, stealing funds from a secondary sale. Thank you for reporting this. > > It's a difficult one for us to address. We want to ensure that NFTs minted on our platform as a split continue to split revenue from the initial sale. We were using `isCreator` from `_getCreatorPaymentInfo` as our way of determining if all the revenue from a sale should go to the royalty recipients, which is a split contract for the use case we are concerned about here. > > The royalty override makes it easy for a creator to choose to abuse this feature at any time. So that was our primary focus for this fix. > > This is the change we have made in `_getFees`: > > `solidity > bool isCreator = false; > // lookup for tokenCreator > try ITokenCreator(nftContract).tokenCreator{ gas: READ_ONLY_GAS_LIMIT }(tokenId) returns (> address payable _creator >) { > isCreator = _creator == seller; > } catch // solhint-disable-next-line no-empty-blocks > { > // Fall through > } > > (creatorRecipients, creatorShares) = _getCreatorPaymentInfo(nftContract, tokenId); > > >` Since the royalty override is only considered in `_getCreatorPaymentInfo` we are no longer vulnerable to someone adding logic after the NFT has been released to try and rug pull the current owner(s). > > It is still possible for someone to try and abuse this logic, but to do so they must have built into the NFT contract itself a way to lie about who the `tokenCreator` is before the time of a sale. If we were to detect this happening, we would moderate that collection from the Foundation website. Additionally we will think about a longer term solution here so that this type of attack is strictly not possible with our market contract.

[H-03] An offer made after auction end can be stolen by an auction winner

Submitted by hyh, also found by leastwood, shenwilly, and WatchPug

An Offer which is made for an NFT when auction has ended, but its winner hasn't received the NFT yet, can be stolen by this winner as `_transferFromEscrow` being called by `_acceptOffer` will transfer the NFT to the winner, finalising the auction, while no transfer to the user who made the offer will happen.

This way the auction winner will obtain both the NFT and the offer amount after the fees at no additional cost, at the expense of the user who made the offer.

Proof of Concept

When an auction has ended, there is a possibility to make the offers for an auctioned NFT as:

`makeOffer` checks `_isInActiveAuction`:

https://github.com/code-423n4/2022-02-foundation/blob/main/contracts/mixins/NFTMarketOffer.sol#L200

`_isInActiveAuction` returns false when `auctionIdToAuction[auctionId].endTime < block.timestamp`, so `makeOffer` above can proceed:

https://github.com/code-423n4/2022-02-foundation/blob/main/contracts/mixins/NFTMarketReserveAuction.sol#L666-L669

Then, the auction winner can call `acceptOffer -> _acceptOffer` (or `setBuyPrice -> _autoAcceptOffer -> _acceptOffer`).

`_acceptOffer` will try to transfer directly, and then calls `_transferFromEscrow`:

https://github.com/code-423n4/2022-02-foundation/blob/main/contracts/mixins/NFTMarketOffer.sol#L262-L271

If the auction has ended, but a winner hasn't picked up the NFT yet, the direct transfer will fail, proceeding with `_transferFromEscrow` in the FNDNFTMarket defined order:

```
function _transferFromEscrow(
address nftContract,
uint256 tokenId,
address recipient,
address seller
) internal override(NFTMarketCore, NFTMarketReserveAuction, NFTMarketBuyPrice, NFTMarketOffe
super._transferFromEscrow(nftContract, tokenId, recipient, seller);
}
```

NFTMarketOffer._transferFromEscrow will call super as `nftContractToIdToOffer` was already deleted:

https://github.com/code-423n4/2022-02-foundation/blob/main/contracts/mixins/NFTMarketOffer.sol#L296-L302

NFTMarketBuyPrice._transferFromEscrow will call super as there is no buy price set:

https://github.com/code-423n4/2022-02-foundation/blob/main/contracts/mixins/NFTMarketBuyPrice.sol#L283-L293

Finally, NFTMarketReserveAuction._transferFromEscrow will send the NFT to the winner via `_finalizeReserveAuction`, not to the user who made the offer:

https://github.com/code-423n4/2022-02-foundation/blob/main/contracts/mixins/NFTMarketReserveAuction.sol#L556-L560

The `recipient` user who made the offer is not present in this logic, the NFT is being transferred to the `auction.bidder`, and the original `acceptOffer` will go through successfully.

Recommended Mitigation Steps

An attempt to set a buy price from auction winner will lead to auction finalisation, so `_buy` cannot be called with a not yet finalised auction, this way the NFTMarketReserveAuction._transferFromEscrow L550-L560 logic is called from the NFTMarketOffer._acceptOffer only:

https://github.com/code-423n4/2022-02-foundation/blob/main/contracts/mixins/NFTMarketOffer.sol#L270

is the only user of

https://github.com/code-423n4/2022-02-foundation/blob/main/contracts/mixins/NFTMarketReserveAuction.sol#L550-L560

This way the fix is to update L556-L560 for the described case as:

Now:

```
// Finalization will revert if the auction has not yet ended.
_finalizeReserveAuction(auctionId, false);

// Finalize includes the transfer, so we are done here.
return;
```

To be, we leave the NFT in the escrow and let L564 super call to transfer it to the recipient:

```
// Finalization will revert if the auction has not yet ended.
_finalizeReserveAuction(auctionId, true);
```

NickCuso (Foundation) confirmed and commented: > Yes! This was a great find and a major issue with our implementation. I'm very happy that it was flagged by a few different people, it helps raise our confidence that several wardens really dove into the code. > > It was a big miss on our part that this was not thoroughly tested. Our tests for this scenario confirmed the events and payouts, but did not validate the ownership in the end! > > The proposed fix is perfect and exactly what we have implemented. This follows the patterns we established well, and actually simplifies the logic here so that things are easier to reason about.

Medium Risk Findings (18)

[M-01] EIP-712 signatures can be re-used in private sales

Submitted by thankthedark, also found by Afanasyevich and cmichel

NFTMarketPrivateSale.sol#L123-L174

Within a NFTMarketPrivateSale contract, buyers are allowed to purchase a seller's NFT. This is done through a seller providing a buyer a EIP-712 signature. The buyer can then call `#buyFromPrivateSaleFor` providing the v, r, and s values of the signature as well as any additional details to generate the message hash. If the signature is valid, then the NFT is transferred to the buyer.

The problem with the code is that EIP-712 signatures can be re-used within a small range of time assuming that the original seller takes back ownership of the NFT. This is because the NFTMarketPrivateSale#buyFromPrivateSaleFor method has no checks to determine if the EIP-712 signature has been used before.

Proof of Concept

Consider the following example:

1. Joe the NFT owner sells a NFT to the malicious buyer Rachel via a private sale.
2. Rachel through this private sale obtains the EIP-712 signature and uses it to purchase a NFT.
3. Joe the NFT owner purchases back the NFT within two days of the original sale to Rachel.
4. Joe the NFT owner puts the NFT back on sale.
5. Rachel, who has the original EIP-712 signature, can re-purchase the NFT by calling `#buyFromPrivateSaleFor` again with the same parameters they provided in the original private sale purchase in step 1.

The `#buyFromPrivateSaleFor` function runs several validation checks before transferring the NFT over to the buyer. The validations are as follows:

1. L#132 - The signature has expired.
2. L#135 - The deadline is beyond 48 hours from now.
3. L#143 - The amount argument is greater than msg.value.
4. L#149 - The msg.value is greater than the amount set.
5. L#171 - This checks that the EIP-712 signature comes from the NFT seller.

As you can see, there are no checks that the EIP-712 signature has been used before. If the original NFT seller purchases back the NFT, then they are susceptible to having the original buyer taking back the NFT. This can be problematic if the NFT has risen in value, as the original buyer can utilize the same purchase amount from the first transaction in this malicious transaction.

Recommended Mitigation Steps

Most contracts utilize nonces when generating EIP-712 signatures to ensure that the contract hasn't been used for. When a nonce is injected into a signature, it makes it impossible for re-use, assuming of course the nonce feature is done correctly.

NickCuso (Foundation) confirmed and commented: > Yes, this is a good point to raise and something like this could happen because it's not intuitive to think that the private sale remains valid after it's been used. > > This is mitigated by the short time window we use for Private Sales. Our frontend uses 24-hour expirations and in the contract we ensure the window is <= 48 hours. > > In order to remain backwards compatible with our existing integration and any signatures which are outstanding at the time of the upgrade, we decided not to use `nonce` as recommended. Instead we simply store a mapping tracking if the exact private sale terms have already been used. > > Here's the primary addition: > > `solidity > // Ensure that the offer can only be accepted once. > if (privateSaleInvalidated[nftContract][tokenId][msg.sender][seller][amount][deadline]) { > revert NFTMarketPrivateSale_Signature_Canceled_Or_Already_Claimed(); > } > privateSaleInvalidated[nftContract][tokenId][msg.sender][seller][amount][deadl = true; >`

[M-02] `SendValueWithFallbackWithdraw: withdrawFor` function may fail to withdraw ether recorded in `pendingWithdrawals`

Submitted by cccz

The NFTMarketFees contract and the NFTMarketReserveAuction contract use the _sendValueWithFallbackWithdraw function to send ether to Foundation-Treasury, CreatorRecipients, Seller, Bidder. When the receiver fails to receive due to some reasons (exceeding the gas limit or the receiver contract cannot

receive ether), it will record the ether to be sent in the pendingWithdrawals variable.

```
function _sendValueWithFallbackWithdraw(
  address payable user,
  uint256 amount,
  uint256 gasLimit
) internal {
  if (amount == 0) {
    return;
  }
  // Cap the gas to prevent consuming all available gas to block a tx from completing succ
  // solhint-disable-next-line avoid-low-level-calls
  (bool success, ) = user.call{ value: amount, gas: gasLimit }("");
  if (!success) {
    // Record failed sends for a withdrawal later
    // Transfers could fail if sent to a multisig with non-trivial receiver logic
    unchecked {
      pendingWithdrawals[user] += amount;
    }
    emit WithdrawPending(user, amount);
  }
}
```

The user can then withdraw ether via the withdraw or withdrawFor functions.

```
function withdraw() external {
  withdrawFor(payable(msg.sender));
}
function withdrawFor(address payable user) public nonReentrant {
  uint256 amount = pendingWithdrawals[user];
  if (amount == 0) {
    revert SendValueWithFallbackWithdraw_No_Funds_Available();
  }
  pendingWithdrawals[user] = 0;
  user.sendValue(amount);
  emit Withdrawal(user, amount);
}
```

However, the withdrawFor function can only send ether to the address recorded in pendingWithdrawals. When the recipient is a contract that cannot receive ether, these ethers will be locked in the contract and cannot be withdrawn.

Proof of Concept

https://github.com/code-423n4/2022-02-foundation/blob/main/contracts/mixins/SendValueWithFallbackWithdraw.sol#L37-L77

Recommended Mitigation Steps

Add the withdrawTo function as follows:

```
function withdrawTo(address payable to) public nonReentrant {
  uint256 amount = pendingWithdrawals[msg.sneder];
  if (amount == 0) {
    revert SendValueWithFallbackWithdraw_No_Funds_Available();
  }
  pendingWithdrawals[msg.sneder] = 0;
  to.sendValue(amount);
  emit Withdrawal(msg.sneder, amount);
}
```

NickCuso (Foundation) confirmed, but disagreed with High severity and commented: > We believe this is better classified as a `2 (Med Risk)`. > - Worst case is funds for a user are trapped in escrow, however they are correctly attributed to that user and cannot be accessed by any other account. If we did not address this by launch, the issue could be corrected post launch via a contract upgrade and that user would then be made whole. So the funds are just temporarily unavailable to them. > > This is a great report and we will be making a change to address the problem! > > When reviewing the recommended mitigation we identified a new potential risk that could introduce. So we are going a slightly different way... > > The `_sendValueWithFallbackWithdraw` function will send `FETH` tokens when it fails to transfer ETH. For any account that currently has a non-zero `pendingWithdrawals` we will include a migration function allowing us to move the escrowed ETH out of the market contract and into that user's FETH account. > > Once that migration has been completed, we will delete the migration function and withdraw* functions – simplifying the market contract and freeing up much needed contract size to make room for new features in the future.

Alberto Cuesta Cañada (judge) decreased severity to Medium and commented: > The combination of upgradability with the limited scope of the vulnerability makes it reasonable to downgrade this to a medium severity.

[M-03] Approve race condition in FETH

Submitted by 0x1f8b

FETH.sol#L212

Front running attack in approve.

Proof of Concept

The contract of the `FETH` does not have any protection against the well-known "Multiple Withdrawal Attack" attack on the Approve/TransferFrom methods of

the ERC20 standard.

Although this attack poses a limited risk in specific situations, it is worth mentioning to consider it for possible future operations.

There are solutions to mitigate this front running such as, to first reduce the spender's allowance to 0 and set the desired value afterwards; another solution could the one that Open Zeppelin offers, where the non-standard decreaseAllowance and increaseAllowance functions have been added to mitigate the well-known issues involving setting allowances.

Recommended Mitigation Steps

Add increase and decrease allowance.

NickCuso (Foundation) acknowledged and commented: > Yes, this is a good best practice to be following. We will add these functions in the near future. In the meantime we've added an inline comment about the potential risk. > > At first we don't expect there will be much interest in using FETH outside of Foundation itself. Thanks to the trusted relationship with the market contract, approvals are not necessary. So this is not an issue that should come up in the near term. > > We are not fixing this immediately because it introduces a risk associated with another contract our users have deployed, that contract was not part of the contest repo. We are looking to patch that vulnerability and then will add `increaseAllowance` to FETH.

[M-04] `adminAccountMigration()` Does Not Update `buyPrice.seller`

Submitted by leastwood, also found by cccz

NFTMarketReserveAuction.sol#L263-L292 NFTMarketBuyPrice.sol#L125-L141

The `adminAccountMigration()` function is called by the operator role to update all sellers' auctions. The `auction.seller` account is updated to the new address, however, the protocol fails to update `buyPrice.seller`. As a result, the protocol is put in a deadlock situation where the new address cannot cancel the auction and withdraw their NFT without the compromised account first cancelling the buy price and vice-versa. This is only recoverable if the new account is migrated back to the compromised account and then `cancelBuyPrice()` is called before migrating back.

Recommended Mitigation Steps

Consider invalidating the buy offer before account migration.

NickCuso (Foundation) confirmed and commented: > Correct - if we were to use `adminAccountMigration` while the NFT had both an auction reserve price and had a buy price set this would have created a deadlock type situation where the NFT is in a bad state and we'd likely need to migrate the NFT back to the original owner in order to correct it. > > There were two possible solutions to this: > - Update `adminAccountMigration` to update both auction and buy price at the same time. > - Cut `adminAccountMigration` completely. > > We went with the latter solution. This is not a feature we have used in some time and as we continue to grow, it's not scalable since it required Foundation to get involved directly and included manual verification steps. > > Removing this feature has saved over 2KB in contract space as well, which we really needed in order to make room for new features and changes.

[M-05] Exchange does not split royalty revenue correctly

Submitted by IllIllI

According to the `README.md`:

> If royalty information was not defined when the NFT was originally deployed, it may be added using the Royalty Registry which will be respected by our market contract.

The actual exchange code only respects the Royalty Registry or other royalty information if the number of recipients is less than or equal to four.

Impact

If the `creatorRecipients.length` is more than four then the array is essentially truncated and the royalties are only split among the first four entries in the array. If the array happens to be sorted from low to high then the people who were supposed to get the largest portions of the royalties are given nothing.

Proof of Concept

```
uint256 maxCreatorIndex = creatorRecipients.length - 1;
if (maxCreatorIndex > MAX_ROYALTY_RECIPIENTS_INDEX) {
  maxCreatorIndex = MAX_ROYALTY_RECIPIENTS_INDEX;
}
```

https://github.com/code-423n4/2022-02-foundation/blob/4d8c8931baffae31c7506872bf1100e1598f2754/contracts/mixins/NFTMarketFees.sol#L76-L79

```
// Send payouts to each additional recipient if more than 1 was defined
uint256 totalDistributed;
for (uint256 i = 1; i <= maxCreatorIndex; ++i) {
  uint256 share = (creatorFee * creatorShares[i]) / totalShares;
```

```
            totalDistributed += share;
            _sendValueWithFallbackWithdraw(creatorRecipients[i], share, SEND_VALUE_GAS_LIMIT_M
        }
```

https://github.com/code-423n4/2022-02-foundation/blob/4d8c8931baffae31c7506872bf1100e1598f2754/contracts/mixins/NFTMarketFees.sol#L99-L105

Creators shouldn't have to settle the correct amounts amongst themselves afterwards and doing so may trigger unwanted tax consequences for the creators who got the larger shares of funds.

Recommended Mitigation Steps

Fetch the royalty information during offer creation, cache it for the final transfer, and reject any NFT for which the array size is more than `MAX_ROYALTY_RECIPIENTS_INDEX`.

NickCuso (Foundation) acknowledged and commented: > Yes, this is correct. This behavior is intended – however we should be more clear about that in the readme and inline comments. I'll add some comments inline to clarify. > > The reason we have a cap at all is because any number of addresses could be returned by these APIs. That would be okay if the creator themselves always paid the gas costs to process them, but since the costs are often pushed to the collectors or secondary sellers - we wanted to ensure the worst case scenario never got unreasonably expensive. > > The actual limit we put in place is arbitrary. However some limit is necessary to ensure a reasonable experience for our users. > > If the creator does want payouts to go to many different addresses, the recommended approach would be to send the royalties to a contract with then handles the split internally in a gas efficient manner, e.g. with https://www.0xsplits.xyz/

[M-06] `buyFromPrivateSaleFor()` Will Fail if The Buyer Has Insufficient Balance Due to an Open Offer on The Same NFT

Submitted by leastwood

NFTMarketPrivateSale.sol#L143-L150

The `buyFromPrivateSaleFor()` function allows sellers to make private sales to users. If insufficient `ETH` is provided to the function call, the protocol will attempt to withdraw the amount difference from the user's unlocked balance. However, if the same user has an open offer on the same NFT, then these funds will remain locked until expiration. As a result, the user cannot make use of these locked funds even though they may be needed for a successful sale.

Recommended Mitigation Steps

Consider adding a `_cancelBuyersOffer()` call to the `buyFromPrivateSaleFor()` function. This should be added only to the case where insufficient `ETH` was provided to the trade. By cancelling the buyer's offer on the same NFT, we can guarantee that the user has access to the correct amount of funds.

NickCuso (Foundation) confirmed and commented: > Yes - completely agree. This was an oversight on our end - and as a result it created an inconsistent experience for users. Since we leveraged an outstanding offer balance for a `buy` purchase, the same behavior should occur when using Private Sales so that user's are not in a state where they cannot make the purchase due to incorrectly having their funds locked up. > > As you point out without this change, it's possible that the buyer using private sales continues to have their funds locked up for an Offer that can now only be accepted by themselves. It's an awkward state that should be avoided. > > We have made the recommended change.

[M-07] `_getCreatorPaymentInfo()` is Not Equipped to Handle Reverts on an Unbounded `_recipients` Array

Submitted by leastwood

NFTMarketCreators.sol#L49-L251

The `_getCreatorPaymentInfo()` function is utilised by `_distributeFunds()` whenever an NFT sale is made. The function uses `try` and `catch` statements to handle bad API endpoints. As such, a revert in this function would lead to NFTs that are locked in the contract. Some API endpoints receive an array of recipient addresses which are iterated over. If for whatever reason the function reverts inside of a `try` statement, the revert is actually not handled and it will not fall through to the empty `catch` statement.

Proof of Concept

The end result is that valid and honest NFT contracts may revert if the call runs out of gas due to an unbounded `_recipients` array. `try` statements are only able to handle external calls.

Recommended Mitigation Steps

Consider bounding the number of iterations to `MAX_ROYALTY_RECIPIENTS_INDEX` as this is already enforced by `_distributeFunds()`. It may be useful to identify other areas where the `try` statement will not handle reverts on internal calls.

NickCuso (Foundation) confirmed and commented: > Yes - this code is trying to be very defensive so that NFTs cannot get stuck in escrow. This is a great suggestion on how we could continue to improve. > > Instead of

capping the loop lengths in `_getCreatorPaymentInfo` we have opted to remove them entirely. There was another simplification that happened recently to this function that made removal a viable option. Now the only other loop for this data is in _distributeFunds and we already cap the max length there.

[M-08] Primary seller can avoid paying the primary fee

Submitted by pedroais, also found by leastwood and WatchPug

A primary seller can circumvent the 15% fee and pay 5% as a secondary seller.

Context

The Foundation protocol charges a 15% fee if the sale is a primary sale and 5% if it's a secondary sale. https://github.com/code-423n4/2022-02-foundation/blob/4d8c8931baffae31c7506872bf1100e1598f2754/contracts/mixins/NFTMarketFees.sol#L40

There are 2 conditions that must be met for a sale to be considered primary:

1. The seller is one of the creators in the NFT metadata.
2. It's the first time this NFT is sold on the foundation protocol.

https://github.com/code-423n4/2022-02-foundation/blob/4d8c8931baffae31c7506872bf1100e1598f2754/contracts/mixins/NFTMarketFees.sol#L188

Proof of Concept

Both of these conditions can be easily circumvented by the primary seller.

1. He could transfer the NFT to a different wallet and sell it from there to break the first condition.
2. He can make a private sale to himself for 1$ (paying the 15% fee on a dust amount) and then do a public auction with the real price.

With any of these 2 methods, the primary seller can circumvent the 15% fee and pay 5% as a secondary seller which makes the primary seller fee optional to pay.

NickCuso (Foundation) acknowledged amd commented: > Yes, this **is** possible. It's a good limitation to note and we should have it called out as a known issue / potential abuse path. > > It's not clear how we could avoid this though while still keeping primary fees around - so I think all we can do is document it for now. This has been true since our launch a year ago and we are not aware of anyone abusing it yet. We think that's because NFT provenance is very important, so if it was being sold from a different account than the creator it's possible that would not get as much attention and potentially sell for less than they could have gotten accepting the primary sale fee.

[M-09] Missing receiver validation in `withdrawFrom`

Submitted by cmichel

FETH.sol#L433

The `FETH.withdrawFrom` function does not validate its `to` parameter. Funds can be lost if `to` is the zero address.

> Similar issues have been judged as medium recently, see Sandclock M-15 / Github issue.

Recommended Mitigation Steps

Check that `to != 0`.

NickCuso (Foundation) confirmed and commented: > Yes, this is a good recommendation which may prevent users losing their funds due to user error. We have made the following change as recommended: > > `solidity > } else if (to == address(0)) { > revert FETH_Cannot_Withdraw_To_Address_Zero(); > } else if (to == address(this)) { > revert FETH_Cannot_Withdraw_To_FETH(); > } >`

[M-10] `LockedBalance` library should drop parameters to 96/32 bits

Submitted by cmichel

LockedBalance.sol#L56

The `LockedBalance` contract takes 256-bit amount values but performs bit math on them as if they were 96 bit values. Bits could spill over to a different locked balance in the `else` part (`lockedBalance` stores **two** 128-bit locked balances in one 256-bit storage field):

```
function set(
  Lockups storage lockups,
  uint256 index,
  uint256 expiration,
  uint256 totalAmount
) internal {
  unchecked {
    // @audit-issue should drop totalAmount to 96, expiration to 128-96=32
    uint256 lockedBalanceBits = totalAmount | (expiration << 96);
    if (index % 2 == 0) {
```

```
      // set first 128 bits.
      index /= 2;
      // @audit-info clears upper bits, then sets them
      lockups.lockups[index] = (lockups.lockups[index] & last128BitsMask) | (lockedBalanceBi
    } else {
      // set last 128 bits.
      index /= 2;
      // @audit-info clears lower bits, then sets them
      // @audit-issue sets entire 256-bit lockedBalanceBits instead of just 128-bit
      lockups.lockups[index] = (lockups.lockups[index] & first128BitsMask) | lockedBalanceBi
    }
  }
}
```

It could then increase the other, unrelated locked balance's amount leading to stealing funds from the protocol. All callers of this function currently seem to ensure that `totalAmount` is indeed less than 96 bits but the `LockedBalance` library should be self-contained and not depend on the calling side to perform all checks.

If the code is ever extended and more calls to these functions are performed, it'll likely cause issues.

> The same issue happens in `setTotalAmount`

Recommended Mitigation Steps

Make sure that there are only 96/32 bits set in `totalAmount` and `expiration` by dropping them to their respective types.

```
function set(
  Lockups storage lockups,
  uint256 index,
  uint256 expiration,
  uint256 totalAmount
) internal {
  unchecked {
-    uint256 lockedBalanceBits = totalAmount | (expiration << 96);
+    // cast it to uint256 again for the << 96 to work on 256-bits
+    uint256 lockedBalanceBits = uint256(uint96(totalAmount)) | (uint256(uint32(expiration))

    ...
  }
}
```

NickCuso (Foundation) acknowledged and commented: > This is a good concern to flag. With this release we use a lot of slot packing strategies to try and save gas. We considered this suggestion, as well as potentially using the

safe cast helpers from the OpenZeppelin library. For the moment at least, we decided not to make any changes here. We believe the assumptions behind these values are sound, and although it's mathematically possible for the assumptions to be violated, we don't believe it'll occur in production. > > For `totalAmount` for example, that value is always bound by the amount of ETH a user has sent to the FETH contract. Here are some notes we have written up about the size assumptions in our codebase: > > - ETH: uint96 > - Circulating supply is currently 119,440,269 ETH (119440269000000000000000000 wei / 1.2 * 10^26). > - Max uint96 is 7.9 * 10^28. > - Therefore any value capped by msg.value should never overflow uint96 assuming ETH total supply remains under 70,000,000,000 ETH. > - There is currently ~2% inflation in ETH total supply (which fluctuates) and this value is expected to do down. We expect Ether will become deflationary before the supply reaches more than 500x current values. > - 96 bits packs perfectly into a single slot with an address. > - Dates: uint32 > - Current date in seconds is 1643973923 (1.6 * 10^9). > - Max uint32 is 4 * 10^9. > - Dates will not overflow uint32 until 2104. > - To ensure we don't behave unexpectedly in the future, we should require dates are <= max uint32. > - uint40 would allow enough space for any date, but it's an awkward size to pack with. > - Sequence ID indexes: uint32 > - Our max sequence ID today is 149,819 auctions. > - Max uint32 is 4 * 10^9. > - Indexes will not overflow uint32 until we see >28,000x growth. This is the equiv to ~300 per block for every block in Ethereum to date.

[M-11] `MAX_ROYALTY_RECIPIENTS_INDEX` set too low

Submitted by cmichel

NFTMarketFees.sol#L78

The creator payouts are capped at `MAX_ROYALTY_RECIPIENTS_INDEX`. It's currently set to `4` and only 5 creators are paid out. Other creators are ignored.

Recommended Mitigation Steps

I don't think cases with more than 5 creators / royalty receivers are unlikely. It can and should probably be increased, especially as the transfers are already gas restricted.

NickCuso (Foundation) acknowledged and commented: > Yes this is a fair point. The limit we put in place is arbitrary. We want some limit in order to ensure that the gas costs (which are often pushed to users other than the original creator) never get to be very expensive. > > What we can and should do is document this limitation so that it's more clear what exactly will happen when too many recipients are defined. Additionally we should comment on a possible workaround: If you have a contract that splits with too many participants you could use the Royalty Override in order to define a single contract recipient instead to handle the splits - e.g. https://www.0xsplits.xyz/

[M-12] Private sale spoofing

Submitted by cmichel

NFTMarketPrivateSale.sol#L156

Similar to spoofing in finance, users can create private sales with correct signatures but then frontrun the buy with a transfer to a different wallet they control.

No funds are lost as the NFT <> FETH exchange is atomic but it can be bad if third parties create a naive off-chain centralized NFT market based on this signature feature. It's also frustrating for the users if they try to accept the private sale but their transaction fails.

Recommended Mitigation Steps

This is made possible because private sales do not keep the NFT in escrow. Consider escrowing the NFT also for private sales.

NickCuso (Foundation) acknowledged and commented: > We are actually considering moving private sales to an escrow system. But we'll leave it as-is for now for backwards compatibility and to simplify our upcoming launch. > > We understand this could happen and will update the comments to make that more clear.

[M-13] Escrowed NFT can be stolen by anyone if no active `buyPrice` or auction exists for it

Submitted by hyh, also found by wuwe1

If a NFT happens to be in escrow with neither buyPrice, nor auction being initialised for it, there is a way to obtain it for free by any actor via `makeOffer`, `acceptOffer` combination.

I.e. a malicious user can track the FNDNFTMarket contract and obtain any NFT from it for which there are no buyPrice or auction structures initialised. For example, if a NFT is mistakenly sent to the contract, an attacker can immediately steal it.

This will happen as NFT is being guarded by buyPrice and auction structures only. The severity here is medium as normal usage of the system imply that either one of them is initialised (NFT was sent to escrow as a part of `setBuyPrice` or `createReserveAuction`, and so one of the structures is present), so this seems to leave only mistakenly sent assets exposed.

Proof of Concept

An attacker can make a tiny offer with `makeOffer`:

https://github.com/code-423n4/2022-02-foundation/blob/main/contracts/mixins/NFTMarketOffer.sol#L189

Then call `acceptOffer`, which will lead to `_acceptOffer`.

Direct NFT transfer will fail in `_acceptOffer` as the NFT is being held by the contract and `_transferFromEscrow` will be called instead:

https://github.com/code-423n4/2022-02-foundation/blob/main/contracts/mixins/NFTMarketOffer.sol#L262-L271

`_transferFromEscrow` calls will proceed according to the FNDNFTMarket defined order:

```
function _transferFromEscrow(
...
) internal override(NFTMarketCore, NFTMarketReserveAuction, NFTMarketBuyPrice, NFTMarketOffe
   super._transferFromEscrow(nftContract, tokenId, recipient, seller);
}
```

If there are no corresponding structures, the NFTMarketOffer, NFTMarketBuyPrice and NFTMarketReserveAuction versions of `_transferFromEscrow` will pass through the call to NFTMarketCore's plain transfer implementation:

https://github.com/code-423n4/2022-02-foundation/blob/main/contracts/mixins/NFTMarketCore.sol#L77-L87

This will effectively transfer the NFT to the attacker, which will pay gas costs and an arbitrary small offer price for it.

Recommended Mitigation Steps

Consider adding additional checks to control who can obtain unallocated NFTs from the contract.

Protocol controlled entity can handle such cases manually by initial sender's request.

NickCuso (Foundation) confirmed and commented: > Yes, this is an interesting scenario to raise! We do have more than one NFT in the market contract which was transferred directly (vs tracked as part of escrow). So there is some exposure already. > > We feel this is a `2 (Med Risk)`: > - It only impacts NFTs which were thought to be effectively burned as they were transferred to the Market contract with no way to get them back. > - Presumably the NFTs which get stuck in the Market contract like that were due to user error. So in theory this could have been thought of a sort of a feature - it allows those NFTs to be recovered, which is great for the original creator which will see royalties from future sales again. > > We have fixed this. We took a slightly different

approach than what was recommended, but it should accomplish the same goal. `_transferFromEscrow` changed the `seller` param to `authorizeSeller`. As the call travels through each mixin, we change `authorizeSeller` to `address(0)` once an escrow manager is able to confirm ownership. Finally when the call reaches `NFTMarketCore` it requires that `authorizeSeller` is `address(0)` before executing the transfer – this confirms that one or more escrow managers (buy now or auctions) have positively confirmed the seller. So NFTs in the Market without being escrowed now reverts on this line.

[M-14] Upgradable escrow contract

Submitted by gzeon

Upgradable escrow contract poses great risk to user who approved their NFT to the contract. Most popular token / NFT exchange do not require user to approve their asset to admin upgradable contract.

This also increases user gas usage because they would have to revoke approval when they are done with the protocol.

Proof of Concept

https://github.com/code-423n4/2022-02-foundation/blob/4d8c8931baffae31c7506872bf1100e1598f2754/contracts/FNDNFTMarket.sol

Recommended Mitigation Steps

Separate the escrow contract to make it non-upgradable with a restricted set of functionality.

NickCuso (Foundation) acknowledged and commented: > This is an interesting suggestion. I'm not sure if the recommendation really reduces the risk though. If we were to make the escrow portion immutable, the trust still depends on the upgradeable market contract to do the correct thing. If I'm understanding the suggestion correctly, the terms of the deal would still be defined in the upgradeable contract - so even with an immutable escrow manager we could theoretically upgrade the market to allow us to `buy` each of the NFTs for 0 ETH.

[M-15] Royalties can be distribution unfairly among `creatorRecipients` for NFT contracts with non-standard `getRoyalties()` returns

Submitted by WatchPug

Based on our research, `getRoyalties()` is not a standardized API for NFT contracts to indicate how the royalties should be distributed among the recipients.

However, in the current implementation, it always assumes that `getRoyalties()` return in terms of BPS.

NFTMarketCreators.sol#L85-L112

NFTMarketFees.sol#L86-L90

```
if (creatorShares[i] > BASIS_POINTS) {
    // If the numbers are >100% we ignore the fee recipients and pay just the first instead
    maxCreatorIndex = 0;
    break;
}
```

As a result, if a particular implementation is returning `get Royalties()` with higher precision (say 1e6 for 100% instead of 1e4/BPS), the distribution of royalties can be distorted.

Proof of Concept

Given:

- `NFT-Token1 Royalties`:
  - Address0 = 10,000 (1%)
  - Address1 = 10,000 (1%)
  - Address2 = 100,000 (10%)
  - Address3 = 880,000 (88%)
- Alice owns the `NFT-Token1`

1. Alice `setBuyPrice()` and listed `NFT-Token1` for `10 ETH`;
2. Bob `buy()` with `10 ETH`:

- `foundationFee` = `0.5 ETH`
- `creatorFee` = `1 ETH`
- `ownerRev` = `8.5 ETH`

Since `creatorShares[address2]` > `BASIS_POINTS` (10,000), all the `creatorFee` will be sent to the first address: `Address0`, which is expected to receive only `1%` of the royalties.

Recommended Mitigation Steps

Consider removing this and change to:

```
// Determine the total shares defined so it can be leveraged to distribute below
uint256 totalShares;
unchecked {
  // The array length cannot overflow 256 bits.
```

```
  for (uint256 i = 0; i <= maxCreatorIndex; ++i) {
    // The check above ensures totalShares wont overflow.
    totalShares += creatorShares[i];
  }
}
if (totalShares == 0) {
  maxCreatorIndex = 0;
}
```

NickCuso (Foundation) acknowledged and commented: > This is a fair concern to raise, and a reasonable solution recommendation. We will consider making a change like this in the future. > > AFAIK this API was first introduced by Manifold, which named and implemented the return value to be in basis points. It's true that since this is not a standard that others may use a different precision, however we have not yet encountered any example of that.

[M-16] Inappropriate support of EIP-2981

Submitted by WatchPug

NFTMarketCreators.sol#L65-L82

```
if (nftContract.supportsERC165Interface(type(IRoyaltyInfo).interfaceId)) {
  try IRoyaltyInfo(nftContract).royaltyInfo{ gas: READ_ONLY_GAS_LIMIT }(tokenId, BASIS_POINT
    address receiver,
    uint256 /* royaltyAmount */
  ) {
    if (receiver != address(0)) {
      recipients = new address payable[](1);
      recipients[0] = payable(receiver);
      // splitPerRecipientInBasisPoints is not relevant when only 1 recipient is defined
      if (receiver == seller) {
        return (recipients, splitPerRecipientInBasisPoints, true);
      }
    }
  } catch // solhint-disable-next-line no-empty-blocks
  {
    // Fall through
  }
}
```

The current implementation of EIP-2981 support will always pass a constant `BASIS_POINTS` as the `_salePrice`.

As a result, the recipients that are supposed to receive less than 1 BPS of the salePrice may end up not receiving any royalties.

Furthermore, for the NFTs with the total royalties rate set less than 10% for some reason, the current implementation will scale it up to 10%.

Recommended Mitigation Steps

1. Instead of passing a constant of 10,000 as the `_salePrice`, we suggest using the actual `_salePrice`, so there the royalties can be paid for recipients with less than 1 BPS of the royalties.
2. When the total royalties cut is lower than 10%, it should be honored. It's capped at 10% only when the total royalties cut is higher than 10%.

NickCuso (Foundation) acknowledged and commented: > Yes, these are valid points and something we will consider revisiting in the future. > > RE recommendations: > 1) This can only impact < 0.01% of the payment so not a concern ATM. It may be more appropriate to better honor exact amounts, but it's a non-trivial change to an important code path so we will leave it as-is for now. With payments that small, it's probably more appropriate to be using a contract to manage the payouts - e.g. https://www.0xsplits.xyz/ could handle this well. > 2) I agree. ATM we always enforce exactly 10% so that there is a consistent experience with our market and on our website. We will revisit this in the future, and the idea of capping it to 10% but accepting lower is a great one.

[M-17] There is no Support For The Trading of Cryptopunks

Submitted by leastwood

Cryptopunks are at the core of the NFT ecosystem. As one of the first NFTs, it embodies the culture of NFT marketplaces. By not supporting the trading of cryptopunks, Foundation is at a severe disadvantage when compared to other marketplaces. Cryptopunks have their own internal marketplace which allows users to trade their NFTs to other users. As such, cryptopunks does not adhere to the `ERC721` standard, it will always fail when the protocol attempts to trade them.

Proof of Concept

Here is an example implementation of what it might look like to integrate cryptopunks into the Foundation protocol.

Recommended Mitigation Steps

Consider designing a wrapper contract for cryptopunks to facilitate standard `ERC721` transfers. The logic should be abstracted away from the user such that their user experience is not impacted.

NickCuso (Foundation) acknowledged and commented: > Yes, crypto punks is an important part of the ecosystem and this is on our radar. We are

not planning on adding support at this time but we will revisiting this in the future. > > I like the idea of using a wrapper contract of sorts. We will be looking for a way to keep the complexity out of the market contract itself like you suggest if at all possible.

[M-18] Fees Are Incorrectly Charged on Unfinalized NFT Sales

Submitted by leastwood

NFTMarketOffer.sol#L255-L271 NFTMarketReserveAuction.sol#L557 NFTMarketReserveAuction.sol#L510-L515 NFTMarketFees.sol#L188-L189

Once an auction has ended, the highest bidder now has sole rights to the underlying NFT. By finalizing the auction, fees are charged on the sale and the NFT is transferred to `auction.bidder`. However, if `auction.bidder` accepts an offer before finalization, fees will be charged on the `auction.bidder` sale before the original sale. As a result, it is possible to avoid paying the primary foundation fee as a creator if the NFT is sold by `auction.bidder` before finalization.

Proof of Concept

Consider the following scenario:

- Alice creates an auction and is the NFT creator.
- Bob bids on the auction and is the highest bidder.
- The auction ends but Alice leaves it in an unfinalized state.
- Carol makes an offer on the NFT which Bob accepts.
- `_acceptOffer()` will distribute funds on the sale between Bob and Carol before distributing funds on the sale between Alice and Bob.
- The first call to `_distributeFunds()` will set the `_nftContractToTokenIdToFirstSaleCompleted` to true, meaning that future sales will only be charged the secondary foundation fee.

Recommended Mitigation Steps

Ensure the `_nftContractToTokenIdToFirstSaleCompleted` is correctly tracked. It might be useful to ensure the distribution of funds are in the order of when the trades occurred. For example, an unfinalized auction should always have its fees paid before other sales.

NickCuso (Foundation) confirmed and commented: > Yes! This is a good find. > > The core of the problem is we were calling `_distributeFunds` before calling `_transferFromEscrow` where the auction finalization would be processed. We have flipped those calls so now the auction will be fully settled before we calculate fees for the offer sale.

Low Risk and Non-Critical Issues

For this contest, 20 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **leastwood** received the top score from the judge.

The following wardens also submitted reports: 0xliumin, defsec, hubble, Dravee, csanuragjain, kenta, rfa, IllIllI, gzeon, 0xwags, cmichel, robee, CertoraInc, kirkbaird, 0x1f8b, Ruhum, TerrierLover, jayjonah8, and WatchPug.

[L-01] Unhandled `marketLockupFor()` Edge Case

`marketLockupFor()` will revert if too much `ETH` is provided. This can be modified to refund any surplus `ETH` to the user's `FETH` balance.

[N-01] Signature Re-Use

The `adminAccountMigration()` function intends to allow a seller to update `auction.seller` in the event their account is compromised. However, the signature in `requireAuthorizedAccountMigration()` can be re-used because there is no use of a nonce to prevent this.

Consider adding a nonce to account for this behaviour.

[N-02] Unclear Function Naming

The `_cancelBuyersOffer()` function isn't exactly clear as it will only cancel the offer if the buyer is `msg.sender`. Therefore, it would be more clear to rename this to `_cancelSellersOffer()` or similar to avoid confusion.

[N-03] Improper `BASIS_POINTS` Check in `_distributeFunds()`

The `_distributeFunds()` function checks if `creatorShares[i] > BASIS_POINTS` when it should actually be checking if `totalShares > BASIS_POINTS`. This ensures that the sum of creator shares do not exceed an expected amount.

[N-04] Improper State Handling

If `_autoAcceptBuyPrice()` is executed by a new buyer that is not `offer.buyer`, then there will be an excess of `ETH`. The original `offer.buyer` will then have to wait until the offer expires as it isn't invalidated.

Ensure this is understood.

[N-05] No Support For `ERC1155` Tokens

Many new NFT contracts adhere to the `ERC1155` standard. Currently, it is not possible to trade these tokens on the Foundation marketplace.

Consider making the necessary integrations.

[N-06] Unclear `_recipients` Array Restriction

The `_recipients` array is restricted to just 5 receivers. Ensure this is understand by NFT creators as they may intend to use additional receivers but these receivers may be disproportionally rewarded if the majority of creator shares are not included.

[N-07] Inconsistent Use of `sendValue()`

`_buy()` should make use of `_sendValueWithFallbackWithdraw()` instead of `sendValue()`. This will ensure invalid transfers are correctly handled.

NickCuso (Foundation) commented: > Thanks for the feedback! > > - [**L-01**] **Unhandled marketLockupFor() Edge Case:** ATM there is no use case we are aware of where it would make sense for too much ETH to be provided. Because of this we are going to revert instead of refunding - it'll save a bit of gas for other users and may prevent an invalid transaction (possibly user error or a frontend bug). > - [**N-01**] **Signature re-use:** In this use case, signature reuse is actually desirable. However after a different issue was reported we opted to remove the `adminAccountMigration` function completely instead. It's not something we have used in some time. > - [**N-02**] **Unclear naming:** Yes, we agree. The function has been renamed. > - [**N-03**] **Improper BASIS_POINTS Check:** This is a fair point. It's somewhat an arbitrary decision though. We are expecting values to be in BP and this logic is simply trying to handle invalid results gracefully. Either way we bring the total back down to 10% and guard against overflowing. ATM we are going to leave it as is. > - [**N-04**] **Improper State Handling:** This is a valid concern but currently working as designed. When auto buy now applies we treat this the same as if they called buy directly. When someone purchases using `buy` we have opted to allow the highest offer to remain valid. It may not be common, but it's not 100% clear that the offer should have been canceled. e.g. the new owner may accept that offer in order to have a fast exit at a small loss due to buyers remorse. > - [**N-05**] **No Support For ERC1155 Tokens:** Agree, we will add support for ERC-1155 at some point. It's a non trivial change though and we don't want to rush into it just yet. > - **[N-06] Unclear _recipients Array Restriction:** Agree. We need some cap in order to limit amount of gas that non-creators would be required to pay. 5 is arbitrary and we should at least provide clear documentation. For now, we have added some comments about this inline. Longer term we can continue to communicate this more clearly, and encourage those with more than 5 recipients to use solutions such as

https://www.0xsplits.xyz/. > - [**N-07**] **Inconsistent Use of sendValue():** Yes, this is inconsistent. However the use case here is slightly different. Generally the use of `sendValueWithFallbackWithdraw` is when we are sending ETH to an address other than the msg.sender - we want to ensure that a malicious user cannot cause other user's transactions to revert. Here the refund is going to the msg.sender - if they are non-receivable they should be able to retry using the correct value instead. So we are leaving it as-is for now.

Alberto Cuesta Cañada (judge) commented: > Unadjusted score: 100 (80 + 20 from clean formatting)

Gas Optimizations

For this contest, 11 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Dravee** received the top score from the judge.

The following wardens also submitted reports: CertoraInc, TerrierLover, thankthedark, iain, 0x1f8b, csanuragjain, gzeon, rfa, kenta, and robee.

Table of Contents

See original submission.

Foreword

- **Storage-reading optimizations**

  The code can be optimized by minimising the number of SLOADs. SLOADs are expensive (100 gas) compared to MLOADs/MSTOREs (3 gas). In the paragraphs below, please see the `@audit-issue` tags in the pieces of code's comments for more information about SLOADs that could be saved by caching the mentioned **storage** variables in **memory** variables.

- **Unchecking arithmetics operations that can't underflow/overflow**

  Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation, or the operation doesn't depend on user input), some gas can be saved by using an `unchecked` block: https://docs.soliditylang.org/en/v0.8.10/control-structures.html#checked-or-unchecked-arithmetic

- **`@audit` tags**

The code is annotated at multiple places with `//@audit` comments to pinpoint the issues. Please, pay attention to them for more details.

File: FETH.sol

function _deductAllowanceFrom()

```
File: FETH.sol
441:   function _deductAllowanceFrom(AccountInfo storage accountInfo, uint256 amount) privat
442:     if (accountInfo.allowance[msg.sender] != type(uint256).max) { //@audit accountInfo.
443:       if (accountInfo.allowance[msg.sender] < amount) { //@audit accountInfo.allowance[
444:         revert FETH_Insufficient_Allowance(accountInfo.allowance[msg.sender]); //@audit
445:       }
446:       // The check above ensures allowance cannot underflow.
447:       unchecked {
448:         accountInfo.allowance[msg.sender] -= amount; //@audit accountInfo.allowance[msg
449:       }
450:     }
451:   }
```

Cache `accountInfo.allowance[msg.sender]` Caching this in memory can save around 2 SLOADs (around 200 gas). This is due to the fact that both conditions will get evaluated before L448, which is using `-=` (therefore making a SLOAD + SSTORE) I recommend caching this value and using it as such:

```
function _deductAllowanceFrom(AccountInfo storage accountInfo, uint256 amount) private {
  uint256 _allowance = accountInfo.allowance[msg.sender]; //@audit 1 MSTORE + 1 SLOAD
  if (_allowance != type(uint256).max) { //@audit 1 MLOAD spent
    if (_allowance < amount) { //@audit 1 SLOAD saved, 1 MLOAD spent
      revert FETH_Insufficient_Allowance(_allowance); //@audit 1 SLOAD saved, 1 MLOAD spen
    }
    // The check above ensures allowance cannot underflow.
    unchecked {
      accountInfo.allowance[msg.sender] = _allowance - amount; //@audit 1 SLOAD saved, 1 M
    }
  }
}
```

function _deductBalanceFrom()

```
File: FETH.sol
456:   function _deductBalanceFrom(AccountInfo storage accountInfo, uint256 amount) private
457:     // Free from escrow in order to consider any expired escrow balance
458:     if (accountInfo.freedBalance < amount) { //@audit accountInfo.freedBalance SLOAD 1
459:       revert FETH_Insufficient_Available_Funds(accountInfo.freedBalance); //@audit acco
460:     }
461:     // The check above ensures balance cannot underflow.
```

```
462:     unchecked {
463:       accountInfo.freedBalance -= uint96(amount); //@audit accountInfo.freedBalance SLO
464:     }
465:   }
```

Cache `accountInfo.freedBalance` Caching this in memory can save around 1 SLOAD (around 100 gas). I recommend caching this value and using it as such:

```
function _deductBalanceFrom(AccountInfo storage accountInfo, uint256 amount) private {
  uint256 _freedBalance = accountInfo.freedBalance; //@audit 1 MSTORE + 1 SLOAD
  // Free from escrow in order to consider any expired escrow balance
  if (_freedBalance < amount) { ///@audit 1 MLOAD spent
    revert FETH_Insufficient_Available_Funds(_freedBalance); //@audit 1 SLOAD saved, 1 MLO
  }
  // The check above ensures balance cannot underflow.
  unchecked {
    accountInfo.freedBalance = _freedBalance - uint96(amount); //@audit 1 SLOAD saved, 1 M
  }
}
```

function __marketLockupFor()

Cache `accountInfo.freedBalance` or call `_deductBalanceFrom` to save gas while saving some size Impacted code:

```
File: FETH.sol
535:       // The check above prevents underflow with delta.
536:       unchecked {
537:         uint256 delta = amount - msg.value; //@audit just use / should call private _de
538:         if (accountInfo.freedBalance < delta) { //@audit accountInfo.freedBalance SLOAD
539:           revert FETH_Insufficient_Available_Funds(accountInfo.freedBalance); //@audit
540:         }
541:         // The check above prevents underflow of freed balance.
542:         accountInfo.freedBalance -= uint96(delta); //@audit accountInfo.freedBalance SL
543:       }
```

The optimization by caching `accountInfo.freedBalance` would be exactly the same as above, which would save around 100 gas. However, here, it'd be better to actually call the optimized `_deductBalanceFrom` to benefit from the previous gas-gains and reduce the contract's size (0.061KB saved).

The code should become:

```
File: FETH.sol
534:     if (msg.value < amount) {
535:       _deductBalanceFrom(accountInfo, amount - msg.value);
536:     } else if (msg.value != amount) {
```

```
537:       // There's no reason to send msg.value more than the amount being locked up
538:       revert FETH_Too_Much_ETH_Provided();
539:     }
```

File: NFTMarketBuyPrice.sol

function cancelBuyPrice()

```
File: NFTMarketBuyPrice.sol
125:   function cancelBuyPrice(address nftContract, uint256 tokenId) external nonReentrant {
126:     BuyPrice storage buyPrice = nftContractToTokenIdToBuyPrice[nftContract][tokenId];
127:     if (buyPrice.seller == address(0)) { //@audit buyPrice.seller SLOAD 1
128:       // This check is redundant with the next one, but done in order to provide a more
129:       revert NFTMarketBuyPrice_Cannot_Cancel_Unset_Price();
130:     } else if (buyPrice.seller != msg.sender) { //@audit buyPrice.seller SLOAD 2 (evalu
131:       revert NFTMarketBuyPrice_Only_Owner_Can_Cancel_Price(buyPrice.seller); //@audit b
132:     }
...
141:   }
```

Cache `buyPrice.seller` Caching this in memory can save around 2 SLOADs (around 200 gas).

function setBuyPrice()

```
File: NFTMarketBuyPrice.sol
150:   function setBuyPrice(
...
165:     BuyPrice storage buyPrice = nftContractToTokenIdToBuyPrice[nftContract][tokenId];
...
170:     if (buyPrice.seller == address(0)) { //@audit buyPrice.seller SLOAD 1
171:       // Transfer the NFT into escrow, if it's already in escrow confirm the `msg.sende
172:       _transferToEscrow(nftContract, tokenId);
173:
174:       // The price was not previously set for this NFT, store the seller.
175:       buyPrice.seller = payable(msg.sender);
176:     } else if (buyPrice.seller != msg.sender) { //@audit buyPrice.seller SLOAD 2 (evalu
177:       // Buy price was previously set by a different user
178:       revert NFTMarketBuyPrice_Only_Owner_Can_Set_Price(buyPrice.seller); //@audit buyP
179:     }
...
182:   }
```

Cache `buyPrice.seller` Caching this in memory can save around 2 SLOADs (around 200 gas).

function _transferFromEscrow()

```
File: NFTMarketBuyPrice.sol
276:   function _transferFromEscrow(
...
282:     BuyPrice storage buyPrice = nftContractToTokenIdToBuyPrice[nftContract][tokenId];
283:     if (buyPrice.seller != address(0)) { //@audit buyPrice.seller SLOAD 1
284:       // A buy price was set for this NFT.
285:       if (buyPrice.seller != seller) { //@audit buyPrice.seller SLOAD 2
286:         // When there is a buy price set, the `buyPrice.seller` is the owner of the NFT
287:         revert NFTMarketBuyPrice_Seller_Mismatch(buyPrice.seller); //@audit buyPrice.se
288:       }
...
294:   }
```

Cache `buyPrice.seller` Caching this in memory can save around 2 SLOADs (around 200 gas).

function _transferToEscrow()

```
File: NFTMarketBuyPrice.sol
316:   function _transferToEscrow(address nftContract, uint256 tokenId) internal virtual ove
317:     BuyPrice storage buyPrice = nftContractToTokenIdToBuyPrice[nftContract][tokenId];
318:     if (buyPrice.seller == address(0)) { //@audit buyPrice.seller SLOAD 1
319:       // The NFT is not in escrow for buy now.
320:       super._transferToEscrow(nftContract, tokenId);
321:     } else if (buyPrice.seller != msg.sender) { //@audit buyPrice.seller SLOAD 2
322:       // When there is a buy price set, the `buyPrice.seller` is the owner of the NFT.
323:       revert NFTMarketBuyPrice_Seller_Mismatch(buyPrice.seller); //@audit buyPrice.sell
324:     }
325:   }
```

Cache `buyPrice.seller` Caching this in memory can save around 2 SLOADs (around 200 gas).

function getBuyPrice()

```
File: NFTMarketBuyPrice.sol
337:   function getBuyPrice(address nftContract, uint256 tokenId) external view returns (add
338:     BuyPrice storage buyPrice = nftContractToTokenIdToBuyPrice[nftContract][tokenId];
339:     if (buyPrice.seller == address(0)) { //@audit buyPrice.seller SLOAD 1
340:       return (address(0), type(uint256).max);
341:     }
342:     return (buyPrice.seller, buyPrice.price); //@audit buyPrice.seller SLOAD 2
343:   }
```

Cache `buyPrice.seller` Caching this in memory can save around 1 SLOAD (around 100 gas).

File: NFTMarketFees.sol

function _distributeFunds()

```
File: NFTMarketFees.sol
48:   function _distributeFunds(
...
74:     if (creatorFee > 0) {
75:       if (creatorRecipients.length > 1) {
76:         uint256 maxCreatorIndex = creatorRecipients.length - 1; //@audit should be unche
77:         if (maxCreatorIndex > MAX_ROYALTY_RECIPIENTS_INDEX) {
78:           maxCreatorIndex = MAX_ROYALTY_RECIPIENTS_INDEX;
79:         }
80:
81:         // Determine the total shares defined so it can be leveraged to distribute below
82:         uint256 totalShares;
83:         unchecked {
84:           // The array length cannot overflow 256 bits.
85:           for (uint256 i; i <= maxCreatorIndex; ++i) {
86:             if (creatorShares[i] > BASIS_POINTS) {
87:               // If the numbers are >100% we ignore the fee recipients and pay just the
88:               maxCreatorIndex = 0;
89:               break;
90:             }
91:             // The check above ensures totalShares wont overflow.
92:             totalShares += creatorShares[i];
93:           }
94:         }
```

Uncheck line L76 This line can't underflow due to L75. Therefore, it should be wrapped in an `unchecked` block. I'd suggest starting the `unchecked` block L83 at line 76.

File: NFTMarketOffer.sol

function makeOffer()

```
File: NFTMarketOffer.sol
189:   function makeOffer(
...
204:     Offer storage offer = nftContractToIdToOffer[nftContract][tokenId];
205:
206:     if (offer.expiration < block.timestamp) { //@audit offer.expiration SLOAD 1
...
```

```
211:     } else {
...
214:       if (amount < _getMinIncrement(offer.amount)) { //@audit offer.amount SLOAD 1
215:         // A non-trivial increase in price is required to avoid sniping
216:         revert NFTMarketOffer_Offer_Must_Be_At_Least_Min_Amount(_getMinIncrement(offer.
217:       }
...
221:       expiration = feth.marketChangeLockup{ value: msg.value }(
222:         offer.buyer,
223:         offer.expiration, //@audit offer.expiration SLOAD 2
224:         offer.amount, //@audit offer.amount SLOAD 2
225:         msg.sender,
226:         amount
227:       );
228:     }
```

Cache `offer.expiration` Caching this in memory can save around 1 SLOAD (around 100 gas).

Cache `offer.amount` Caching this in memory can save around 1 SLOAD (around 100 gas).

function getOffer()

```
File: NFTMarketOffer.sol
379:   function getOffer(address nftContract, uint256 tokenId)
...
388:     Offer storage offer = nftContractToIdToOffer[nftContract][tokenId];
389:     if (offer.expiration < block.timestamp) { //@audit offer.expiration SLOAD 1
390:       // Offer not found or has expired
391:       return (address(0), 0, 0);
392:     }
393:
394:     // An offer was found and it has not yet expired.
395:     return (offer.buyer, offer.expiration, offer.amount); //@audit offer.expiration SL
396:   }
```

Cache `offer.expiration` Caching this in memory can save around 1 SLOAD (around 100 gas).

File: NFTMarketReserveAuction.sol

function adminAccountMigration()

```
File: NFTMarketReserveAuction.sol
263:   function adminAccountMigration(
```

```
264:     uint256[] calldata listedAuctionIds,
265:     address originalAddress,
266:     address payable newAddress,
267:     bytes memory signature //@audit gas should be calldata
268:   ) external onlyFoundationOperator {
269:     // Validate the owner of the original account has approved this change.
270:     originalAddress.requireAuthorizedAccountMigration(newAddress, signature);
271:
272:     unchecked {
273:       // The array length cannot overflow 256 bits.
274:       for (uint256 i; i < listedAuctionIds.length; ++i) {
275:         uint256 auctionId = listedAuctionIds[i];
276:         ReserveAuction storage auction = auctionIdToAuction[auctionId];
277:         if (auction.seller != address(0)) { //@audit auction.seller SLOAD 1
278:           // Only if the auction was found and not finalized before this transaction.
279:
280:           if (auction.seller != originalAddress) { //@audit auction.seller SLOAD 2
281:             // Confirm that the signature approval was the correct owner of this auctio
282:             revert NFTMarketReserveAuction_Cannot_Migrate_Non_Matching_Seller(auction.s
283:           }
284:
285:           // Update the auction's seller address.
286:           auction.seller = newAddress;
287:
288:           emit ReserveAuctionSellerMigrated(auctionId, originalAddress, newAddress);
289:         }
290:       }
291:     }
292:   }
```

Use `calldata` instead of `memory` for external functions where the function argument is read-only

Here, `bytes memory signature` should be `bytes calldata signature`

Cache `auction.seller` Caching this in memory can save around 2 SLOADs (around 200 gas).

function placeBidOf()

```
File: NFTMarketReserveAuction.sol
386:   function placeBidOf(uint256 auctionId, uint256 amount) public payable nonReentrant {
...
402:     ReserveAuction storage auction = auctionIdToAuction[auctionId];
403:
404:     if (auction.amount == 0) { //@audit auction.amount SLOAD 1
```

```
405:       // No auction found
406:       revert NFTMarketReserveAuction_Cannot_Bid_On_Nonexistent_Auction();
407:     }
408:
409:     if (auction.endTime == 0) { //@audit auction.endTime SLOAD 1
410:       // This is the first bid, kicking off the auction.
411:
412:       if (auction.amount > amount) { //@audit auction.amount SLOAD 2
413:         // The bid must be >= the reserve price.
414:         revert NFTMarketReserveAuction_Cannot_Bid_Lower_Than_Reserve_Price(auction.amou
415:       }
...
429:     } else {
430:       if (auction.endTime < block.timestamp) { //@audit auction.endTime SLOAD 2
431:         // The auction has already ended.
432:         revert NFTMarketReserveAuction_Cannot_Bid_On_Ended_Auction(auction.endTime); //
433:       } else if (auction.bidder == msg.sender) { //@audit auction.bidder SLOAD 1
434:         // We currently do not allow a bidder to increase their bid unless another user
435:         revert NFTMarketReserveAuction_Cannot_Rebid_Over_Outstanding_Bid();
436:       } else if (amount < _getMinIncrement(auction.amount)) { //@audit auction.amount S
437:         // If this bid outbids another, it must be at least 10% greater than the last b
438:         revert NFTMarketReserveAuction_Bid_Must_Be_At_Least_Min_Amount(_getMinIncrement
439:       }
440:
441:       // Cache and update bidder state
442:       uint256 originalAmount = auction.amount; //@audit auction.amount SLOAD 3
443:       address payable originalBidder = auction.bidder; //@audit auction.bidder SLOAD 2
444:       auction.amount = amount;
445:       auction.bidder = payable(msg.sender);
446:
447:       unchecked {
448:         // When a bid outbids another, check to see if a time extension should apply.
449:         // We confirmed that the auction has not ended, so endTime is always >= the cur
450:         if (auction.endTime - block.timestamp < auction.extensionDuration) { //@audit a
451:           // Current time plus extension duration (always 15 mins) cannot overflow.
452:           auction.endTime = block.timestamp + auction.extensionDuration; //@audit aucti
453:         }
454:       }
455:
456:       // Refund the previous bidder
457:       _sendValueWithFallbackWithdraw(originalBidder, originalAmount, SEND_VALUE_GAS_LIN
458:     }
459:
460:     emit ReserveAuctionBidPlaced(auctionId, msg.sender, amount, auction.endTime); //@au
461:   }
```

Cache `auction.endTime` Caching this in memory can save around 3 SLOADs (around 300 gas).

Cache `auction.extensionDuration` Caching this in memory can save around 1 SLOAD (around 100 gas).

Cache `auction.bidder` Caching this in memory can save around 1 SLOAD (around 100 gas).

Cache `auction.amount` Caching this in memory can save around 2 SLOADs (around 200 gas).

General recommendations

For-Loops

An array's length should be cached to save gas in for-loops Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack. Caching the array length in the stack saves around 3 gas per iteration. Therefore, it's possible to save a significant amount of gas (>= 102 after 34 iterations).

Here, I suggest storing the array's length in a variable before the for-loop, and use it instead:

```
mixins/OZ/ERC165Checker.sol:64:        for (uint256 i = 0; i < interfaceIds.length; ++i) {
mixins/OZ/ERC165Checker.sol:90:      for (uint256 i = 0; i < interfaceIds.length; ++i) {
mixins/NFTMarketCreators.sol:94:             for (uint256 i = 0; i < _recipients.length; ++i)
mixins/NFTMarketCreators.sol:155:                 for (uint256 i = 0; i < _recipients.length;
mixins/NFTMarketCreators.sol:193:                 for (uint256 i = 0; i < _recipients.length;
mixins/NFTMarketOffer.sol:161:      for (uint256 i = 0; i < nftContracts.length; ++i) {
mixins/NFTMarketReserveAuction.sol:274:      for (uint256 i = 0; i < listedAuctionIds.length
```

Increments can be unchecked In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save a significant amount of gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

Instances include:

```
mixins/OZ/ERC165Checker.sol:64:        for (uint256 i = 0; i < interfaceIds.length; ++i) {
mixins/OZ/ERC165Checker.sol:90:      for (uint256 i = 0; i < interfaceIds.length; ++i) {
mixins/NFTMarketCreators.sol:94:             for (uint256 i = 0; i < _recipients.length; ++i)
mixins/NFTMarketCreators.sol:155:                 for (uint256 i = 0; i < _recipients.length;
mixins/NFTMarketCreators.sol:193:                 for (uint256 i = 0; i < _recipients.length;
mixins/NFTMarketOffer.sol:161:      for (uint256 i = 0; i < nftContracts.length; ++i) {
```

```
mixins/NFTMarketReserveAuction.sol:274:      for (uint256 i = 0; i < listedAuctionIds.length
FETH.sol:552:      for (uint256 escrowIndex = accountInfo.lockupStartIndex; ; ++escrowIndex)
FETH.sol:679:      for (uint256 escrowIndex = accountInfo.lockupStartIndex; ; ++escrowIndex)
FETH.sol:713:      for (uint256 escrowIndex = accountInfo.lockupStartIndex; ; ++escrowIndex)
FETH.sol:733:      for (uint256 escrowIndex = accountInfo.lockupStartIndex; ; ++escrowIndex)
FETH.sol:760:      for (uint256 escrowIndex = accountInfo.lockupStartIndex; ; ++escrowIndex)
```

The code would go from:

```
for (uint256 i; i < numIterations; ++i) {
 // ...
}
```

to:

```
for (uint256 i; i < numIterations;) {
 // ...
 unchecked { ++i; }
}
```

The risk of overflow is inexistant for a `uint256` here.

NickCuso (Foundation) commented: > This was a very detailed and helpful gas report! I love that you commented the code with the @audit tags and detailed exactly how much savings could be made and why. This was very useful when evaluating the recommendations. > > We have implemented many of the suggestions here. Some we chose not to change in order to preserve readability. Generally the savings is inline with the expected values you have stated here.

Alberto Cuesta Cañada (judge) commented: > Great report. Score: 4300 (including +20% from formatting)

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.