NFTX
Findings & Analysis Report

2021-06-21

Table of contents

Overview

About C4

Code 432n4 (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code 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 code contest outlined in this document, C4 conducted an analysis of the NFTX smart contract system written in Solidity. The code contest took place between May 5th and May 11th, 2021.

Wardens

11 Wardens contributed reports to the NFTX code contest:

- cmichel
- pauliax (Thunder)
- janbro
- shw
- gpersoon
- rajeev
- a_delamo
- 0xsomeone
- heiho1 (Thisguy)
- mukesh jaiswal
- jvaqa

This contest was judged by cemozer.

Final report assembled by moneylegobatman.

Summary

The C4 analysis yielded an aggregated total of 24 unique vulnerabilities. All of the issues presented here are linked back to their original finding

Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity, 10 received a risk rating in the category of MEDIUM severity, and 10 received a risk rating in the category of LOW severity.

C4 analysis also identified 15 non-critical recommendations.

Scope

The code under review can be found within the C4 NFTX code contest repository and comprises 68 smart contracts written in the Solidity programming language.

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.

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 (4)

[H-01] Missing overflow check in `flashLoan`

`ERC20FlashMintUpgradeable.flashLoan` does not check for an overflow when adding the fees to the `flashloan` amount. The functionality might have been copied from https://eips.ethereum.org/EIPS/eip-3156 but this one already has overflow checks as it uses solidity 0.8.0. This leads to an issue where the attacker does not need to pay back the `flashloan` as they will burn 0 tokens:

```
    _burn(address(receiver), amount + fee);
```

They end up with a huge profit. (Luckily, this is currently not exploitable as the fee is set to 0 so there's no possibility to overflow. However, if governance decides to change the flashloan fee, flashloans can be taken without having to repay them). Recommend using `SafeMath`.

0xKiwi (NFTX) confirmed:

> Upgraded to 0.8.x.

[H-02] `distribute` DoS on missing `receiveRewards` implementation

`NFTXEligiblityManager._sendForReceiver` should check `returnData.length == 1` before decoding. Otherwise, if it returns no return data, the `abi.decode` call will revert and with it the whole `distribute` function .

A single poorly implemented `feeReceiver` can break the whole `distribute` function and allow a denial of service by reverting the transaction.

Recommend changing to: `bool tokensReceived = returnData.length == 1 && abi.decode(returnData, (bool));` .

[0xKiwi (NFTX) confirmed]:

[cemozer (Judge) commented]:

> Marking this as high risk because one nefarious feeReceiver can in fact deny other users to receive their fees

[H-03] `getRandomTokenIdFromFund` yields wrong probabilities for ERC1155

`NFTXVaultUpgradeable.getRandomTokenIdFromFund` does not work with ERC1155 as it does not take the deposited `quantity1155` into account.

Assume `tokenId0` has a count of 100, and `tokenId1` has a count of 1. Then `getRandomId` would have a pseudo-random 1:1 chance for token 0 and 1 when in reality it should be 100:1.

This might make it easier for an attacker to redeem more valuable NFTs as the probabilities are off.

Recommend taking the quantities of each token into account (`quantity1155`) which probably requires a design change as it is currently hard to do without iterating over all tokens.

0xKiwi (NFTX) acknowledged

cemozer (Judge) commented:

> Marking this as high risk as an attacker can weed out high-value NFTs from a vault putting other users funds at risk

[H-04] `NFTXLPStaking` Is Subject To A Flash Loan Attack That Can Steal Nearly All Rewards/Fees That Have Accrued For A Particular Vault

The LPStaking contract does not require that a stake be locked for any period of time. The LPStaking contract also does not track how long your stake has been locked. So an attacker Alice can stake, claim rewards, and unstake, all in one transaction. If Alice utilizes a flash loan, then she can claim nearly all of the rewards for herself, leaving very little left for the legitimate stakers.

The fact that the `NFTXVaultUpgradeable` contract contains a native `flashLoan` function makes this attack that much easier, although it would still be possible even without that due to flashloans on Uniswap, or wherever else the nftX token is found.

Since a flash loan will easily dwarf all of the legitimate stakers' size of stake, the contract will erroneously award nearly all of the rewards to Alice.

1. Wait until an NFTX vault has accrued any significant amount of fees/rewards
2. `FlashLoanBorrow` a lot of ETH using any generic flash loan provider
3. `FlashLoanBorrow` a lot of nftx-vault-token using `NFTXVaultUpgradeable.flashLoan()`
4. Deposit the ETH and nftx-vault-token's into Uniswap for Uniswap LP tokens by calling `Uniswap.addLiquidity()`
5. Stake the Uniswap LP tokens in `NFTXLPStaking` by calling `NFTXLPStaking.deposit()`
6. Claim nearly all of the rewards that have accrued for this vault due to how large the flashLoaned deposit is relative to all of the legitimate stakes by calling

`NFTXLPStaking.claimRewards()`

7. Remove LP tokens from `NFTXLPStaking` by calling `NFTXLPStaking.exit()`;
8. Withdraw ETH and nftx-vault-token's by calling `Uniswap.removeLiquidity()`;
9. Pay back nftx-vault-token flash loan
10. Pay back ETH flash loan

See GitHub issue page for an in-depth example.

Recommend requiring that staked LP tokens be staked for a particular period of time before they can be removed. Although a very short time frame (a few blocks) would avoid flash loan attacks, this attack could still be performed over the course of a few blocks less efficiently. Ideally, you would want the rewards to reflect the product of the amount staked and the duration that they've been staked, as well as having a minimum time staked.

Alternatively, if you really want to allow people to have the ability to remove their stake immediately, then only allow rewards to be claimed for stakes that have been staked for a certain period of time. Users would still be able to remove their LP tokens, but they could no longer siphon off rewards immediately.

0xKiwi (NFTX) disputed:

> After looking at the code, this is not possible. The dividend token code takes into consideration the current unclaimed rewards and when a deposit is made that value is deducted.

cemozer (Judge) commented:

> @0xKiwi do you mind showing where in code that occurs?

Medium Risk Findings (8)

[M-01] Randomization of NFTs returned in redeem/swap operations can be brute-forced

If we assume that certain NFTs in a vault over time will have different market demand/price, then the users will try to redeem those specific NFTs. Even if direct redeems are disabled to prevent such a scenario to default to returning randomized NFTs, a user can brute-forced this on-chain randomization (using nonce + blockhash) by repeatedly trying to redeem/swap from a contract, checking the NFT IDs returned from the function and reverting the transaction if those are not the NFT IDs of specific interest.

The impact will be a subversion of the randomization goal to return random NFTs which cannot be specified by the user.

A similar exploit happened recently with Meebit NFTs.

Recommend considering only EOA (external only account) for redeem/swap operations to prevent brute-forcing via contracts. Alternatively, make the user commit to pseudo-random IDs before revealing them.

**- 0xKiwi acknowledged **

[M-02] Use `safeTransfer` / `safeTransferFrom` consistently instead of `transfer` / `transferFrom`

It is good to add a `require()` statement that checks the return value of token transfers, or to use something like OpenZeppelin's `safeTransfer` / `safeTransferFrom` unless one is sure the given token reverts in case of a failure. Failure to do so will cause silent failures of transfers and affect token accounting in contract.

While most places use a `require` or `safeTransfer` / `safeTransferFrom` , there are three missing cases in the withdrawal of staking token and rescue of arbitrary tokens sent to the `FeeDistributor` contract.

Reference this similar medium-severity finding from Consensys Diligence Audit of Fei Protocol.

Recommend using `safeTransfer` / `safeTransferFrom` or `require()` consistently.

- 0xKiwi (NFTX) confirmed

[M-03] Fee Distribution Re-Entrancy

The `distribute` function of `NFTXFeeDistributor` has no access control and will invoke a fallback on the fee receivers, meaning that a fee receiver can re-enter via this function to acquire their allocation repeatedly potentially draining the full balance and sending zero amounts to the rest of the recipients.

A smart contract with a malicious `receiveRewards` function can re-enter the `distribute` function with the same vault ID, thereby causing the exploit.

Recommend that re-entrancy protection should be incorporated into the `distribute` function. I should note that a seemingly innocuous contract can cause this re-entrancy by simply asking the owners of the project to include an upgrade-able contract that is then replaced for a malicious implementation.

- [0xKiwi (NFTX) confirmed]()

[M-05] Unbounded iteration in `NFTXEligiblityManager.distribute` over `_feeReceivers`

`NFTXEligiblityManager.distribute` iterates over all `_feeReceivers`. If the number of `_feeReceivers` gets too big, the transaction's gas cost could exceed the block gas limit and make it impossible to call `distribute` at all.

Recommend keeping the number of `_feeReceivers` small.

- [0xKiwi (NFTX) confirmed]()

[M-06] Manager can grief with fees

The fees in `NFTXVaultUpgradeable` can be set arbitrarily high (no restriction in `setFees`).

The manager can front-run mints and set a huge fee (for example `fee = base`) which transfers user's NFTs to the vault but doesn't mint any pool share tokens in return for the user.

Similar griefing attacks are also possible with other functions besides `mint`.

Recommend checking for a max fee as a percentage of `base` (like 10%) whenever setting fees.

[M-07] Tokens can get stuck in `NFTXMintRequestEligibility`

When dealing with ERC721 (instead of 1155) the amounts array is ignored, which leads to an issue.

User can call `NFTXMintRequestEligibility.requestMint` for an ERC721 with `amounts[i] = 0`. The `ERC721.transferFrom` is still executed but user cannot `reclaimRequestedMint` later and the NFT is stuck as it checks (`amounts[i] > 0`).

Tokens can get stuck. Also, subscribers to `Request` event could be tricked by specifying `amounts[i] > 1` in the ERC721 case, as only one token was transferred, but the amount multiple quantities get logged.

Recommend that `requestMint`: Check `amounts[i] == 1` in ERC721 case, `amounts[i] > 0` in 1155 case.

- **0xKiwi (NFTX) confirmed**

[M-08] A malicious receiver can cause another receiver to lose out on distributed fees by returning `false` for `tokensReceived` when `receiveRewards` is called on their receiver contract.

A malicious receiver can cause another receiver to lose out on distributed fees by returning `false` for `tokensReceived` when `receiveRewards` is called on their receiver contract. This causes the fee distributor to double spend the `amountToSend` because the contract incorrectly assumes the returned data is truthful.

`NFTXFeeDistributor.sol`:

```
    Line 163: (bool success, bytes memory returnData) = address(_rec
```

Recommend that you don't trust return data from externally called contracts. Only utilize whether the transaction succeeds to determine if the treasury fallback should be called.

```
    Line 165: if (!success) {
```

0xKiwi (NFTX) confirmed:

> Nice catch!

[M-09] The direct redeem fee can be circumvented

Since the random NFT is determined in the same transaction a payment or swap is being executed, a malicious actor can revert a transaction if they did not get the NFT they wanted. Combined with utilizing Flashbots miners which do not publish transactions which revert with `FlashbotsCheckAndSend`, there would be no cost to constantly attempting this every block or after the nonce is updated from `getPseudoRand()`.

`NFTXVaultUpgradeable.sol`

```
    Line 374: uint256 tokenId = i < specificIds.length
        ? specificIds[i]
        : getRandomTokenIdFromFund();
```

In this way, the `directReedemFee` can be avoided and users may lose out on potential earnings. The code below shows a transfer ownership of ERC20 tokens to attack the contract.

```
    function revertIfNotSpecifiedID(uint256 targetTokenID) public {
        NFTXVaultUpgradeable vault = NFTXVaultUpgradeable(_vault);
        uint256[] resultID = vault.redeem(1,[]);
        require(resultID[0] == targetTokenID);
    }
```

Recommend using a commit-reveal pattern for NFT swaps and redemptions.

[0xKiwi (NFTX) acknowledged]

[cemozer (Judge) commented]:

> Leaving this as medium risk as it puts user earnings into risk

Low Risk Findings (10)

[L-01] Front-running `setFees()` could avoid fees

`setVaultFeatures()` and `setFees()` are two separate privileged functions. Users could front-run `setFees()` immediately after vault is enabled in `setVaultFeatures()` to mint (and possibly redeem/directRedeem/swap) many tokens. The fees for mint/redeem/directRedeem/swap are not initialized so are 0 by default. This leads to loss of fee revenue.

Recommend Setting defaults at initialization or combine this with `setVaultFeatures()` for atomically enabling functions and setting their fees in the same transaction.

- [0xKiwi (NFTX) confirmed]

[L-02] Missing pool existence check in `balanceOf`

In `NFTXLPStaking.sol`, `deposit()`, `exit()`, `withdraw()`, `claimRewards()` and other related functions that take a `vaultID` as parameter perform a pool existence check on the staking pool associated with that `vaultID`. However, `balanceOf` is missing a similar pool check.

This may result in returning an invalid balance of a non-existing or stale pool.

Recommend adding check `require(pool.stakingToken != address(0), "LPStaking: Nonexistent pool”);` before L170.

- [0xKiwi (NFTX) confirmed]

[L-03] Missing parameter validation

Missing parameter validation for functions:

- `NFTXEligiblityManager.addModule, updateModule`
- `NFTXFeeDistributor` all `setter` functions (`setTreasuryAddress` , ...)
- `NFTXVaultUpgradeable.setManager`

Some wallets still default to zero addresses for a missing input which can lead to breaking critical functionality like setting the manager to the zero address and being locked out.

Recommend validating the parameters.

- [0xKiwi (NFTX) confirmed]

[L-04] Missing usage of SafeMath

The following code does not use `SafeMath` and can potentially lead to overflows:

- `NFTXFeeDistributor.distribute`
- `NFTXFeeDistributor._sendForReceiver`

While looping through all `_feeReceivers` it could be that a broken vault was whitelisted that allows an attacker to perform an external call and break the invariant that always 1000 tokens are left in the contract.

Add `SafeMath` to `_sendForReceiver` even though one would expect the math to be safe.

[0xKiwi (NFTX) confirmed]:

> Confirmed and updated all code to 0.8.x.

[L-05] Inconsistent solidity pragma

The source files have different solidity compiler ranges referenced. This leads to potential security flaws between deployed contracts depending on the compiler version chosen for any particular file. It also greatly increases the cost of maintenance as different compiler versions have different semantics and behavior.

This defect has numerous surfaces at https://github.com/code-423n4/2021-05-nftx/tree/main/nftx-protocol-v2/contracts/solidity

Recommend fixing a definite compiler range that is consistent between contracts and upgrade any affected contracts to conform to the specified compiler.

[0xKiwi (NFTX) commented]():

> We have updated everything to 0.8.x.

[L-06] Unchecked external calls in `NFTXLPStaking`

The `emergencyExit` / `emergencyExitAndClaim` functions take the staking and reward tokens as parameters and trust them for the withdrawal.

This does not lead to a critical issue (like being able to withdraw all funds) as one cannot deploy a fake reward smart contract to a `_rewardDistributionTokenAddr` and a random address without a smart contract will fail because of the `dist.balanceOf(msg.sender)` call not returning any data. However, checking if the distribution token exists is still recommended.

Recommend requiring `isContract(dist)`.

- [0xKiwi (NFTX) confirmed]()

[L-07] Vault’s flash loan not implemented according to EIP-3156

The `NFTXVaultUpgradeable.flashLoan` is not correctly implemented according to EIP-3156 (but it tries to implement it as it inherits from `IERC3156FlashLenderUpgradeable`).

> "If successful, flashLoan MUST return true." - https://eips.ethereum.org/EIPS/eip-3156

It misses the return and currently always returns `false`.

Always returning `false` indicates that the flash loan was unsuccessful, when in reality it could have been successful. This breaks any contract trying to integrate with it.

Recommend adding the return statement: `return super.flashLoan(...)`

- 0xKiwi (NFTX) confirmed

- cemozer (Judge) commented:

> Keeping this as low-risk as flash loan returning the project does not pose a security threat for the NFTX project itself

[L-08] `eligibilityManager` is always 0x0

the contract `NFTXVaultFactoryUpgradeable`, variable `eligibilityManager` is never set thus it gets a default value of 0x0. So function `deployEligibilityStorage` should always fail as the eligibility manager does not exist on address 0x0.

Recommend either adding a setter for `eligibilityManager` or refactor function `deployEligibilityStorage` to work in such case.

0xKiwi (NFTX) confirmed:

> Nice find!

[L-09] lack of zero address validation

Init function like `__FeeDistributor__init__()` is used to initialize the state variables. Since these state variables are used in many functions, it is possible that due to lack of input validation, an error in these state variables can lead to redeployment of contract.

- `In NFTXFeeDistributor.sol --> __FeeDistributor__init__()`
- `in NFTXLPStaking.sol --> __NFTXLPStaking__init()`
- `in NFTXVaultUpgradeable.sol -- > __NFTXVault_init()`
- `in StakingTokenProvider.sol --> __StakingTokenProvider_init()`

Recommend adding zero address validation.

- 0xKiwi (NFTX) confirmed

[L-10] `__Ownable_init` will be called twice in multiple Eligibility contracts

Here you have more info:
https://gist.github.com/alexon1234/43bf4a72a5b06651f04fc8052349ac5a

- 0xKiwi (NFTX) confirmed

Non-Critical Findings

[N-01] `LockIds` not according to spec

The `PausableUpgradeable.onlyOwnerIfPaused` doc comment specifies the pause states as:

```
    // 0 : createFund
    // 1 : mint
    // 2 : redeem
    // 3 : mintAndRedeem
```

But `lockId = 3` does not prevent mints and redeems in `NFTXVaultUpgradeable`. Instead, it prevents swaps. There is also an undocumented `lockId = 4` to prevent `flashLoans`.

A manager might look at the spec and try to prevent mints and redeems in an emergency by setting the highest `lockId 3`, which would prevent these according to spec. However, users can still mint and redeem as `lockId` prevents swaps only.

Recommend updating the documentation to reflect the code.

- 0xKiwi (NFTX) confirmed

cemozer (Judge) commented:

> Marking this as non-critical as the issue is only in the documentation.

[N-01] simpler way to suppress compiler warning

In the function `flashFee` of `ERC20FlashMintUpgradeable.sol`, the variable amount is referenced to suppress a compiler warning. There is a simpler way to do this, by commenting out the variable name.

```
function flashFee(address token, uint256 amount) public view
    require(token == address(this), "ERC20FlashMint: wrong t
    // silence warning about unused variable without the adc
    amount;
    return 0;
```

Recommend using the following code:

```
function flashFee(address token, uint256 /*amount*/) public \
    require(token == address(this), "ERC20FlashMint: wrong t
    return 0;
```

- 0xKiwi (NFTX) acknowldged

[N-02] Not checked if within array bounds

In the function `updateModule` and `deployEligibility` of `NFTXEligiblityManager.sol`, the array modules is used without checking if the index is within bounds. If index would be out of bounds, the function will revert, but it's more difficult to troubleshoot.

NFTXEligiblityManager.sol:

```
    function updateModule(uint256 index, address implementation) puk
        modules[index].impl = implementation;
    }

    function deployEligibility(uint256 moduleIndex, bytes calldata c
        address eligImpl = modules[moduleIndex].impl;
        ...
    }
```

Recommend adding something like: `require(index < modules.length,"out or range");`

- [0xKiwi (NFTX) confirmed]

[N-03] Missing documentation for `flashloan` paused number

The contract `PausableUpgradeable.sol` documents the paused variables 0..3. However, `onlyOwnerIfPaused` is also used with a parameter of 4. This is used for `flashloans`.

PausableUpgradeable.sol:

```
        // 0 : createFund
        // 1 : mint
        // 2 : redeem
        // 3 : mintAndRedeem
    .\NFTXVaultFactoryUpgradeable.sol:          onlyOwnerIfPaused(0);
    .\NFTXVaultUpgradeable.sol:         onlyOwnerIfPaused(1);
    .\NFTXVaultUpgradeable.sol:         onlyOwnerIfPaused(2);
    .\NFTXVaultUpgradeable.sol:        onlyOwnerIfPaused(3);
    .\NFTXVaultUpgradeable.sol:        onlyOwnerIfPaused(4);
```

Recommend adding documentation for #4. Or even better, create constants or enums.

[N-04] no check `_rangeStart<=_rangeEnd`

In `NFTXRangeEligibility.sol` a range is defined via `__NFTXEligibility_init` and `setEligibilityPreferences`. No check is done to make sure `_rangeStart<=_rangeEnd`, so one could accidentally define it as a range that is effectively empty.

```
    function setEligibilityPreferences(uint256 _rangeStart, uint256
            rangeStart = _rangeStart;
            rangeEnd = _rangeEnd;
            emit RangeSet(_rangeStart, _rangeEnd);
        }
```

Recommend considering adding a check to make sure `_rangeStart<=_rangeEnd`

- 0xKiwi (NFTX) confirmed

[N-05] Incorrect Type Specified For Argument `_address` In `NFTXFeeDistributor.rescueTokens()`

`NFTXFeeDistributor.rescueTokens()` is not functional since it casts `_address` as a uint256 in the function declaration in L141

Recommend changing this:

```
function rescueTokens(uint256 _address) external override onlyOwner {
```

To this:

```
function rescueTokens(address _address) external override onlyOwner {
```

- [0xKiwi (NFTX) confirmed]

[N-06] Two Duplicate "rescueTokens" Functions In `NFTXFeeDistributor`

There are two duplicate functions that both intended to rescue tokens that have incorrectly been sent to the `NFTXFeeDistributor`. The only difference is a typo in one of the functions which incorrectly casts `_address` as a uint256.

Recommend deleting the `rescueTokens` function on L141 that incorrectly casts `_address` as a uint256.

- [0xKiwi (NFTX) confirmed]

[N-07] [INFO] function `publicMint` is for testing only

The function `publicMint` has a comment that says it is meant for testing only:

```
    // For testing
    function publicMint(address to, uint256 tokenId, uint256 amo
        _mint(to, tokenId, amount, "");
    }
```

Recommend not to forget to comment or delete this code before going to production.

[0xKiwi (NFTX) acknowledged]:

> Test code

[N-08] Using `calldata` when not appropriate

Here's more info:
https://gist.github.com/alexon1234/7b6799434cccda5e3f3d461b8186ec89

- [0xKiwi (NFTX) disputed]

cemozer (Judge) commented:

> @0xKiwi can you explain why you've disputed this issue?

Gas Optimizations

[G-01] Unused storage variables

Unused storage variables in contracts use up storage slots and increase contract size and gas usage at deployment and initialization. Multiple variables across different contracts including `manager` , `allVaults` , `vaultsForAsset` , and `prevContract` . Recommend removing unused variables.

- 0xKiwi (NFTX) confirmed

[G-02] Unused events

Unused events increase contract size at deployment. Multiple events in `NFTXMintRequestEligibility.sol` listed on the issue page. Recommend removing unused events or emit at appropriate places.

- 0xKiwi (NFTX) confirmed

[G-03] Change function visibility from public to external

Various functions across contracts are never called from within contracts yet are declared public. Their visibility can be made external to save gas.

As described in https://mudit.blog/solidity-gas-optimization-tips/:

> "For all the public functions, the input parameters are copied to memory automatically, and it costs gas. If your function is only called externally, then you should explicitly mark it as external. External function's parameters are not copied into memory but are read from `calldata` directly. This small optimization in your solidity code can save you a lot of gas when the function input parameters are huge."

Recommend changing function visibility from public to external.

- [0xKiwi (NFTX) confirmed]()

[G-04] Gas optimization for `StakingTokenProvider.nameForStakingToken`

`StakingTokenProvider.nameForStakingToken`: `if (keccak256(abi.encode(_pairedPrefix)) == keccak256(abi.encode(address(0))))` can be simplified to `if(bytes(_pairedPrefix).length== 0)`

- [0xKiwi (NFTX) confirmed]()

[G-05] Revert inside a loop

Here you have more info:
https://gist.github.com/alexon1234/a2275d1724ce2122d36bc555e46a25c1

- [0xKiwi (NFTX) acknowledged]()

[G-06] Unused variables

There are unused variables in contract `NFTXVaultUpgradeable` (string public description) and in contract `NFTXMintRequestEligibility` (address public manager). Recommend deleting unused variables to reduce the deployment costs.

Disclosures

of users.

Top