SMART CONTRACT AUDIT

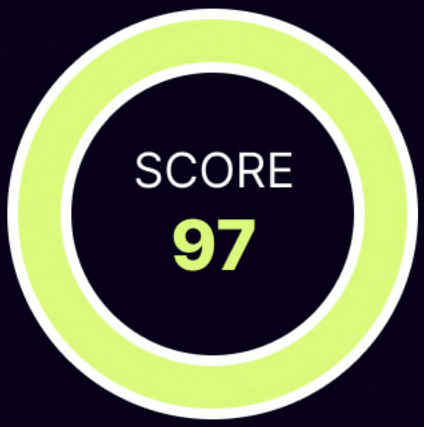
zokyo

August 8th 2023 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges.



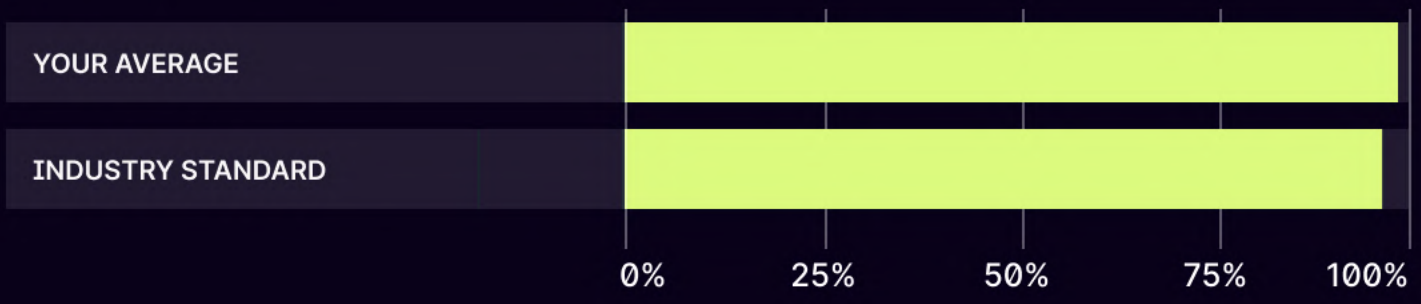
TECHNICAL SUMMARY

This document outlines the overall security of the TigrisTrade smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the TigrisTrade smart contract codebase for quality, security, and correctness.

Contract Status

Testable Code



98% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the TigrisTrade team put in place a bug bounty program to encourage further active analysis of the smart contract.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the TigrisTrade repository:

Intial commit: bb6a7f95e7300d629a1384493477d2eec28b865c
Final commit: 9dba99f6988ce9d8edf0e70a5e2415f34c0a25aa

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- TigrisLPStaking.sol
- xTIG.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most resent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of TigrisTrade smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	03	Testing contract logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contracts by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

Zokyo Security received three contracts provided by the TigrisTrade team. The LPStaking contract is responsible for staking liquidity, providing tokens, and receiving rewards with a feature of enabling and disabling autocompounding. The xTig contract, for its part, allows users to vest their xTig tokens with opportunities of claiming Tig before rewards are unlocked with a certain penalty.

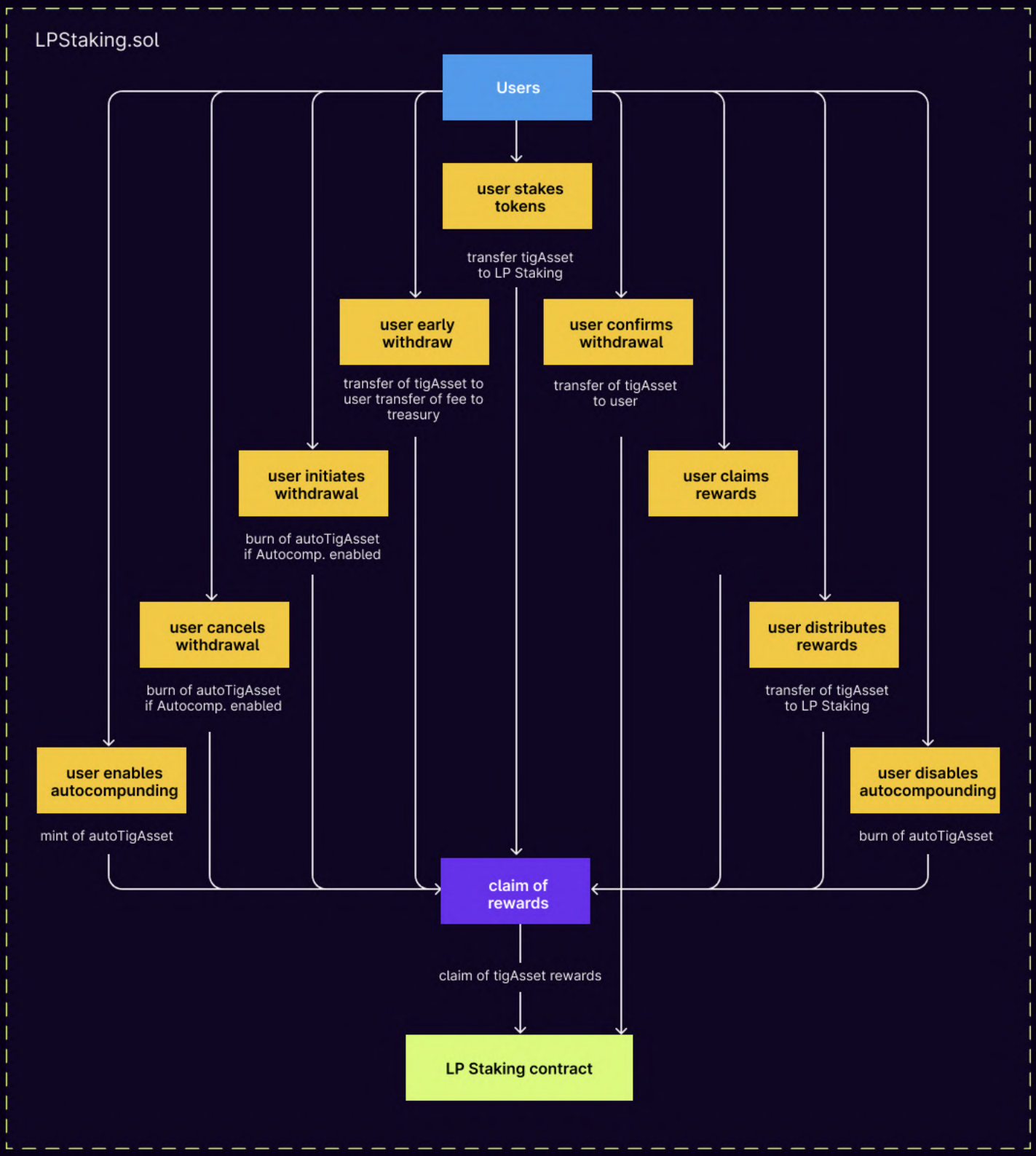
The audit aimed to verify that the contracts function correctly and with a known level of security and check the code line by line. The auditors also checked the contracts' code against their own checklist of vulnerabilities, validated the contracts' business logic, and ensured that best practices in terms of gas spending were applied.

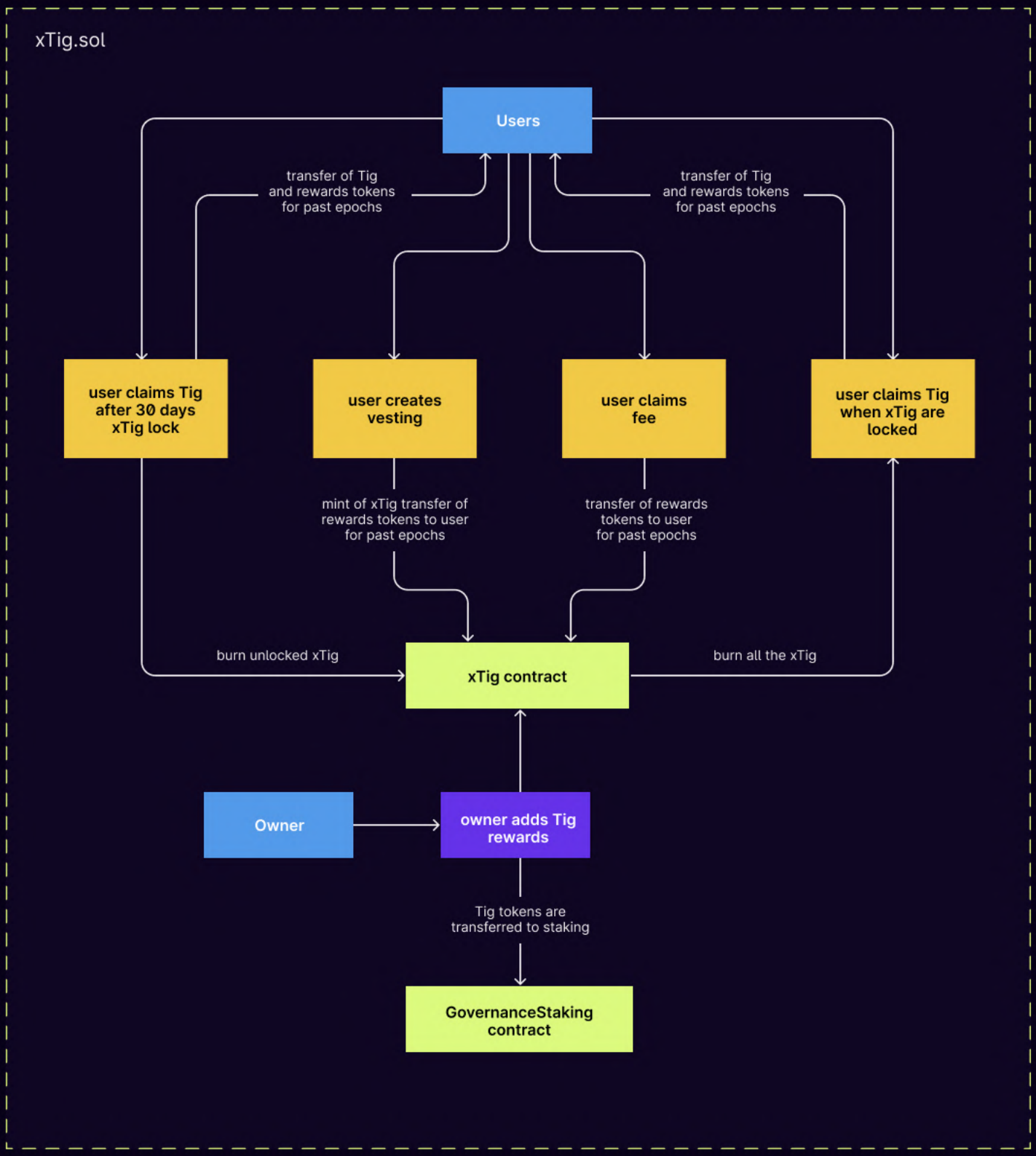
During the manual part of the audit, three critical severity issues, two high-severity issue, three medium-severity issues, five low-severity issues, and seven informational issues, were found.

The critical severity findings were improper storage updates about user staked balance when switching off autocompounding mode; another critical finding was due to wrong changes in the functionality for claiming Tig while fixing another issue. The possible denial of service in claiming Tig by running out of gas and unchecked transfer issues were addressed as medium-severity findings. Lack of validation, lack of documentation, vulnerabilities of wrong contracts' parameters setup, and other low-level results were addressed with low and informational severity.

Another part of the audit process involved checking the native tests prepared by the TigrisTrade team and preparing a set of custom test scenarios. It is worth mentioning that the TigrisTrade team provided a solid set of tests that covered all the logic of the contracts. However, Zokyo Security prepared its own set of unit tests. The complete set of unit tests can be viewed in the Code Coverage and Test Result sections.

The total security of the contracts is high enough. The contracts are well-written and tested. Tokens transfers, staking rewards distribution, vesting, and other system architecture components were carefully audited. Issues of contracts' parameters setup with the possible consequences for the protocol were addressed by auditors and fixed by the TigrisTrade team. TigrisTrade has also provided documentation on implemented functionalities. The link to the documentation is https://docs.tigris.trade/.


LPStaking.sol
Users
user stakes tokens
transfer tigAsset to LP Staking
user early withdraw
transfer of tigAsset to user transfer of fee to treasury
user confirms withdrawal
transfer of tigAsset to user
user initiates withdrawal
burn of autoTigAsset if Autocomp. enabled
user claims rewards
user cancels withdrawal
burn of autoTigAsset if Autocomp. enabled
user distributes rewards
transfer of tigAsset to LP Staking
user enables autocompunding
mint of autoTigAsset
user disables autocompounding
burn of autoTigAsset
claim of rewards
claim of tigAsset rewards
LP Staking contract


xTig.sol
Users
transfer of Tig and rewards tokens for past epochs
transfer of Tig and rewards tokens for past epochs
user claims Tig after 30 days xTig lock
user creates vesting
user claims fee
user claims Tig when xTig are locked
mint of xTig transfer of rewards tokens to user for past epochs
transfer of rewards tokens to user for past epochs
burn unlocked xTig
xTig contract
burn all the xTig
Owner
owner adds Tig rewards
Tig tokens are transferred to staking
GovernanceStaking contract

TIGRISTRADE LPSTAKING DESCRIPTION

A smart contract named `TigrisLPStaking`is used for staking tokens in return for rewards. The contract also includes a secondary contract named `AutoTigAsset`.

> The `TigrisLPStaking` contract allows users to stake specific tokens (assets) and earn rewards over time. The contract also supports an " autocompounding " feature where rewards are automatically reinvested to increase the staked amount. The contract also has a mechanism where users can withdraw their staked tokens and earn rewards.

The contract stores the staked tokens and the rewards. It interacts with the ERC20 tokens that are being staked and the `AutoTigAsset` contract for handling the autocompounding feature.

Users can interact with the contract in the following ways:

- `stake`: Users can stake their tokens by calling this function.
- `initiateWithdrawal`: Users can initiate the withdrawal of their staked tokens and rewards.
- `confirmWithdrawal`: Users can confirm their withdrawal after a certain period.
- `cancelWithdrawal`: Users can cancel their withdrawal request.
- `earlyWithdrawal`: Users can perform an early withdrawal before the withdrawal period, but this may incur a fee.
- `claim`: Users can claim their rewards.
- `setAutocompounding`: Users can enable or disable the autocompounding feature.

The `AutoTigAsset` contract is a token contract that represents the staked tokens in the `TigrisLPStaking` contract. When users enable autocompounding, their staked tokens are converted to `AutoTigAsset` tokens.

TIGRISTRADE LPSTAKING DESCRIPTION

Here is a brief explanation of how autocompounding works in this contract:

- When a user stakes their tokens, they can choose whether to enable autocompounding by calling the `setAutocompounding` function.
- If autocompounding is enabled, the contract will create a new token (AutoTigAsset) for the user. The amount of AutoTigAsset tokens minted for the user is proportional to the number of tokens they staked and the current value of the autocompounded assets.
- When rewards are distributed to the stakers, the contract will increase the value of the autocompounded assets proportionally to the rewards received.
- When the user wants to withdraw their stake, they can burn their AutoTigAsset tokens to get back the underlying tokens. The amount of underlying tokens they receive is proportional to the amount of AutoTigAsset tokens they burn and the current value of the autocompounded assets.

The contract also includes various administrative functions that can only be called by the contract owner, such as `whitelistAsset`, `unwhitelistAsset`, `setWithdrawalPeriod`, `setTreasury`, `setDepositFee`, `setEarlyWithdrawFee`, `setEarlyWithdrawalEnabled`, and `setDepositCap`. These functions are used to manage the contract settings and the list of stakable tokens.

When a user initiates a withdrawal of their staked assets, they cannot immediately get their assets back. Instead, they have to wait for a certain period.

However, the contract also provides an option for users who want to withdraw their assets immediately, without waiting for the cool down period. This is known as an "early withdrawal".

The `earlyWithdrawal` function allows users to withdraw their assets before the cool down period ends. But this comes at a cost. An early withdrawal fee is charged for this convenience.

This fee is transferred to the treasury, which is an address set by the contract owner. The remaining amount after deducting the fee is transferred to the user.

TIGRISTRADE XTIG DESCRIPTION

The xTIG contract is a vesting contract designed to reward users for participating in the platform. It is part of a DeFi ecosystem and interacts with other contracts such as a governance staking contract, an ERC20 token contract, and potentially other reward contracts.

> The contract allows users to earn rewards over time if permissioned address staked TIG token for them. The rewards are distributed in the form of xTIG tokens, which represent a claim on a portion of the total rewards pool. The contract also includes a vesting period for rewards, during which users cannot claim their rewards. If users choose to claim their rewards before the end of the vesting period, they are charged an early unlock penalty.

The contract also allows for the collection of fees from traders. These fees are used to increase the rewards pool further. The contract includes a mechanism to track the fees generated by each trader and allocate rewards proportionally. Reward allocation is spread among different epochs. Each epoch lasts one day.

The contract also includes a number of administrative functions that can only be called by the contract owner. These include the ability to set the vesting period, set the early unlock penalty, whitelist and unwhitelist reward tokens, and recover unclaimed TIG tokens.

The xTIG contract interacts with the following external contracts:

- The TIG token contract (IERC20) is the token that permission address stakes for users to earn rewards. The contract needs to be able to transfer and approve these tokens.
- The governance staking contract (IGovernanceStaking) is used to stake the TIG tokens and distribute the rewards.
- The extra rewards contract (IExtraRewards) is optional and can be used to distribute additional rewards.
- The treasury contract is where the contract sends any unclaimed rewards or penalties collected.

The contract holds the staked TIG tokens, the generated fees, and the distributed rewards. It also keeps track of the vesting schedule for each user's rewards.

In terms of user interaction, users can claim their rewards (either after the vesting period or early with a penalty) and view their pending rewards. Traders can also generate fees that contribute to the rewards pool.

Roles and Responsibilities (LPStaking):

1. Contract Owner (Ownable.sol)

Responsibilities:

- Correct deployment of the smart contract.
- Setting up the initial parameters such as treasury address, deposit fee, early withdrawal fee, withdrawal period, etc.
- Whitelisting and unwhitelisting of assets.
- Updating the treasury address, deposit fee, early withdrawal fee, withdrawal period, and deposit cap.
- Enabling and disabling early withdrawals.

Possibilities:

- Can call the `setTreasury`, `setDepositFee`, `setEarlyWithdrawFee`, `setWithdrawalPeriod`, `setEarlyWithdrawalEnabled`, `setDepositCap`, `whitelistAsset`, `unwhitelistAsset` functions.
- Has access to change the treasury address which is a valuable asset.
- Can enable or disable early withdrawals which can affect the users' funds.

2. Users

Responsibilities:

- Correct interaction with the smart contract such as staking, initiating withdrawal, confirming withdrawal, cancelling withdrawal, early withdrawal, claiming rewards, setting autocompounding.
- Replenishment of the reward balance by calling the `distribute` function.

Possibilities:

- Can call the `stake`, `initiateWithdrawal`, `confirmWithdrawal`, `cancelWithdrawal`, `earlyWithdrawal`, `claim`, `setAutocompounding`, `distribute` functions.
- Can stake their assets and earn rewards.
- Can initiate, confirm, cancel withdrawals.
- Can enable or disable autocompounding for their assets.Can distribute rewards to the stakers.
- It's important to note that users are responsible for their own funds and must take action to claim rewards and withdraw their assets.

3. AutoTigAsset Contract

Responsibilities:

- Correct minting and burning of AutoTigAsset tokens.

Possibilities:

- Can call the `mint` and `burn` functions.
- Can mint and burn AutoTigAsset tokens which represent the user's stake in the contract.

4. Treasury

Responsibilities:

- Receives fees from deposit and early withdrawal operations.

In this smart contract, the contract owner has the most responsibilities and possibilities as they are in charge of setting up the contract and maintaining it. The users interact with the contract to stake their assets and earn rewards.

Deployment (LPStaking):

The deployment script seems correct and follows the standard procedure for deploying smart contracts using the Hardhat deployment plugin. Here's the breakdown of the script:

1. It imports the necessary modules and gets the named accounts for `deployer` and `treasury`.

2. It retrieves the address of the previously deployed `StableToken` contract.

3. It deploys the `TigrisLPStaking` contract with the `treasury` address as an argument. The contract is deployed from the `deployer` account.

4. If the `TigrisLPStaking` contract is newly deployed, it calls the `whitelistAsset` function of the contract to whitelist the `StableToken` with an initial deposit of `0`.

TIGRISTRADE LPSTAKING DESCRIPTION

The script does not seem to have any issues. However, here are a few things to consider:

- Ensure that the `deployer` account has enough funds to deploy the contract.

- The `treasury` address should be correctly set and controlled by a trusted entity since it has important permissions in the contract.

- The `StableToken` contract should be correctly deployed before running this script.

- The `whitelistAsset` function is called with an initial deposit of `0`. If the contract requires an initial deposit for whitelisting an asset, this value should be adjusted accordingly.

- The script does not set important parameters such as `depositFee`, `depositCap`. These parameters need to be set during deployment, so they should be added to the script.

- The script assumes that the `StableToken` is the only asset to be whitelisted. If there are other assets to be whitelisted, they should be added to the script.

Settings (LPStaking):

1. Treasury Address (`treasury`): This is the address where the protocol sends the deposit and early withdrawal fees. It's important to set this to a secure and controlled address. If set incorrectly, the collected fees could be lost or stolen.

2. Withdrawal Period (`withdrawalPeriod`): This setting determines the period a user has to wait before they can confirm their withdrawal. Changing this setting affects the liquidity of the staked assets. If set too high, it could discourage users from staking their assets due to the long withdrawal period.

3. Deposit Fee (`depositFee`): This is the fee charged when a user stakes their assets. If set too high, it could discourage users from staking their assets. If set too low, it could affect the revenue of the protocol.

4. Early Withdrawal Fee (`earlyWithdrawFee`): This is the fee charged when a user withdraws their assets before the withdrawal period. If set too high, it could discourage users from staking their assets. If set too low, it could encourage users to withdraw their assets early, affecting the liquidity of the staked assets.

5. Early Withdrawal Enabled (`earlyWithdrawalEnabled`): This setting determines whether users can withdraw their assets early. Disabling early withdrawal could discourage users from staking their assets due to the lack of liquidity.

6. Deposit Cap (`depositCap`): This is the maximum amount of assets that can be staked. If set too low, it could limit the growth of the protocol. If set too high, it could expose the protocol to risks if a large amount of assets are staked and then withdrawn.

7. Whitelisted Assets (`whitelistAsset`): This setting determines which assets can be staked. It's important to whitelist only trusted and secure assets to prevent any vulnerabilities or risks.

8. Auto Compounding (`setAutocompounding`): This setting allows users to enable or disable auto compounding for their assets. If autocompounding is disabled, users will need to manually claim and restake their rewards.

These settings are crucial for the operation and security of the TigrisLPStaking protocol. They should be set carefully and reviewed regularly to ensure the protocol operates efficiently and securely.

Roles and Responsibilities (xTIG):

1. Owner:
The owner is the account that deploys the smart contract. It has the most control over the contract, including the ability to set various parameters and manage other roles. The owner is responsible for the correct deployment of the contract and the initial configuration of the system. The owner also has the responsibility to replenish the reward balance to ensure the protocol works correctly.

Functions and Permissions:

- `addTigRewards(uint256 _epoch, uint256 _amount)`: This function allows the owner to add TIG rewards to the contract. The owner must ensure that the epoch is not in the past and must transfer the specified amount of TIG from their account to the contract.
- `setTigAssetValue(address _tigAsset, uint256 _value)`: This function allows the owner to set the value of a TIG asset.
- `setCanAddFees(address _address, bool _allowed)`: This function allows the owner to set which addresses are allowed to add fees.
- `setTreasury(address _address)`: This function allows the owner to set the treasury address.
- `setExtraRewards(address _address)`: This function allows the owner to set the extra rewards address.
- `setVestingPeriod(uint256 _time)`: This function allows the owner to set the vesting period.
- `setEarlyUnlockPenalty(uint256 _percent)`: This function allows the owner to set the early unlock penalty.
- `whitelistReward(address _rewardToken)`: This function allows the owner to whitelist a reward token.
- `unwhitelistReward(address _rewardToken)`: This function allows the owner to unwhitelist a reward token.
- `recoverTig(uint256 _epoch)`: This function allows the owner to recover unclaimed TIG rewards from past epochs.

Roles and Responsibilities (xTIG):

2. Permissioned Accounts:
These are accounts that have been given permission by the owner to add fees. They are responsible for adding fees to the contract.

Functions and Permissions:
- `addFees(address _trader, address _tigAsset, uint256 _fees)`: This function allows permissioned accounts to add fees to the contract. The fees are added to the value generated by a trader during an epoch.

3. Traders (Users):
Traders are users who generate fees. These fees are added to the contract by the permissioned accounts. Traders can create vests and claim their rewards.

Functions and Permissions:
- `createVest(uint256 _from)`: This function allows traders to create a vest. The vest is created for the fees generated by the trader during the specified epochs.
- `claimTig()`: This function allows traders to claim their vested TIG rewards.
- `earlyClaimTig()`: This function allows traders to claim their vested TIG rewards early, with a penalty.
- `claimFees()`: This function allows users to claim their pending rewards.

5. Treasury:
The treasury is an account set by the owner. It receives unclaimed rewards and any excess rewards that are not distributed to users including penalty.

Deployment (xTIG):

The deployment script seems to be correct and follows the common pattern for deploying smart contracts using the Hardhat deployment plugin. Here's how it works:

1. It imports the necessary modules and gets the named accounts for `deployer` and `treasury`.

2. It retrieves the deployments of `StableToken`, `GovernanceStaking`, and `TigrisToken` contracts.

3. It deploys the `xTIG` contract using the `deployer` account. The arguments for the constructor of `xTIG` contract are correctly provided - the name of the token, the symbol, the address of `TigrisToken`, the address of `GovernanceStaking`, and the `treasury` address.

4. It executes the `whitelistReward` function of the `xTIG` contract to whitelist the `StableToken` as a reward token.

5. It executes the `setTigAssetValue` function of the `xTIG` contract to set the value of the `StableToken` to 1 ETH.

> The addresses with which contracts are deployed and the setting of important parameters seem to be correct. However, it's important to note that the correctness of these settings largely depends on the specific requirements and configurations of your project.

Also, make sure that the `StableToken`, `GovernanceStaking`, and `TigrisToken` contracts are correctly deployed before running this script, and the `deployer` account has sufficient funds to cover the gas costs of the deployments and function executions.

You need to consider the setting of important parameter such as `canAddFees`. This parameter needs to be set during deployment or immediately after, so they should be added to the script or another script that is run immediately after deployment.

Lastly, remember to verify the contract on Etherscan or a similar platform after deployment, and always test your deployment scripts on a testnet before deploying to mainnet.

Settings (xTIG):

The following are the important settings for the protocol:

1. `vestingPeriod`: This setting determines the period over which the vested rewards are locked up and gradually released. If this period is set too short, it might not encourage long-term participation in the protocol. If it's too long, it might discourage users from participating due to the long lock-up period.

2. `earlyUnlockPenalty`: This setting determines the penalty for early withdrawal of vested rewards. If the penalty is too high, it might discourage users from participating due to the risk of losing a significant portion of their rewards if they need to withdraw early. If it's too low, it might not discourage early withdrawals, undermining the purpose of vesting.

3. `treasury`: This is the address where unclaimed rewards and any excess rewards that are not distributed to users are sent. It is important to set this to a secure and trusted address. If the treasury address is compromised, the attacker could drain the unclaimed and excess rewards.

4. `canAddFees`: This is a mapping that determines which addresses are allowed to add fees. It's important to only allow trusted addresses to add fees to prevent malicious actors from manipulating the protocol.

5. `tigAssetValue`: This is a mapping that sets the value of a TIG asset. It's important to set this correctly to ensure fair distribution of rewards.

6. `rewardTokens`: This is a mapping that determines which tokens are whitelisted as reward tokens. It's important to only whitelist trusted tokens to prevent users from receiving worthless or malicious tokens as rewards.

7. `extraRewards`: This is the address of the contract that can claim extra rewards from the protocol. It's important to set this to a secure and trusted contract. If the extra rewards contract is compromised, the attacker could drain the extra rewards.

8. `epochAllocation`: This is a mapping that sets the allocation of rewards for each epoch. It's important to set this correctly to ensure fair distribution of rewards.

TIGRISTRADE XTIG DESCRIPTION

The settings in the smart contract are designed to prevent several potential vulnerabilities:

1. `vestingPeriod` and `earlyUnlockPenalty`: These settings prevent the vulnerability of short-term manipulation. By requiring users to lock their rewards for a certain period, it discourages users from joining the protocol, earning rewards, and immediately leaving, which could destabilize the protocol.

2. `treasury`: By setting the treasury to a secure and trusted address, it prevents the vulnerability of an attacker draining the unclaimed and excess rewards.

3. `canAddFees`: This setting prevents unauthorized addresses from adding fees, which could be used to manipulate the protocol.

4. `tigAssetValue`: By correctly setting the value of TIG assets, it prevents the vulnerability of unfair reward distribution.

5. `rewardTokens`: By whitelisting only trusted tokens as reward tokens, it prevents the vulnerability of users receiving worthless or malicious tokens as rewards.

6. `extraRewards`: By setting the extra rewards contract to a secure and trusted contract, it prevents the vulnerability of an attacker draining the extra rewards.

7. `epochAllocation`: By correctly setting the allocation of rewards for each epoch, it prevents the vulnerability of unfair reward distribution.

These settings are crucial for the security and fairness of the protocol. Incorrect settings could expose the protocol to various attacks and manipulations, so it's important to carefully set and manage these settings.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

CRITICAL-1 | RESOLVED

Staking amount not added to total staking amount and user staked amount when user disables autocompounding.

xTig.sol: setAutocompounding().
When disabling autocompounding, autoTigAssets are burned, but the _toStakedAssets value is not added to the total staking and user-staked amounts. In that case, users, who switched autocompounding off, will not be able to withdraw tokens from Staking because they were not added to their staked amount.

Recommendation:
Add _toStakedAssets to the total staking amount, and the user staked amount on autocompounding disabling.

Post-audit:
_toStakedAssets is now added to the total staking amount and the user staked amount when the user turns autocompounding off.

CRITICAL-2 | RESOLVED

Users will lose rewards if they claim not the whole range of reward batches.

xTig.sol: claimTig(), earlyClaimTig().
When claiming Tig (both early and the other way), the user puts a limit on how many reward batches he is willing to claim. But in the functions for claiming Tig, the rewards array is deleted from storage, and the reward batches, which are out of that limited range, will be inaccessible.

Recommendation:
Do not delete the storage array of user rewards or do the calculation for claiming Tig in the other way.

Post-audit:
Storage information about users' rewards is not being deleted no more.

CRITICAL-3 | RESOLVED

Staking amount is not added to the user-paid amount when the user disables autocompounding.

LPStaking.sol: setAutocompounding().
When disabling autocompounding, autoTigAssets are burned, but the _toStakedAssets value is not added (but set) to the user-paid amount. In that case, users, who switched autocompounding off, will be able to withdraw all the tigAsset balance of the contract by repeatedly disabling autocompounding if they have some tokens staked. This issue arises because disabling autocompounding while not having autoTig will result in the userPaid set being zero on every setAutocompounding call and repeatedly withdrawing rewards.

Recommendation:
Add _toStakedAssets to the user paid amount instead of setting it.

Post-audit:
Staking account is now added to paid user's amount.

User is able to create a vest from the withdrawn TIG.

xTIG.sol: recoverTig(), createVest().
Function recoverTig() allows the owner to withdraw all unallocated TIG from a certain past epoch for which users haven't created a vest and minted xTIG. However, these changes are updated neither in `epochAllocation[_epoch]` nor in `epochAllocationClaimed[_epoch]` storage variables. As a result, the user can create a vest from TIG which the owner recovered.
Example:

- epochAllocation[_epoch] = 100.

epochAllocationClaimed[_epoch] = 80.
feesGenerated[_epoch][trader] = 20.
epochFeesGenerated[_epoch] = 100.
20 TIG is staked and trader is eligible to create a vest and mint 20 xTIG out of it.

- Owner calls recoverTig() and withdraws epochAllocation[_epoch] - epochAllocationClaimed[_epoch] = 100 - 80 = 20.

epochAllocation[_epoch] = 100.
epochAllocationClaimed[_epoch] = 80.
feesGenerated[_epoch][trader] = 20.
epochFeesGenerated[_epoch] = 100.
20 TIG are withdrawn and trader should no longer be eligible to create a vest and mint 20 xTIG out of it.

- Trader calls createVest() and mints 20 xTIG, though the owner already recovered the corresponding TIG.

As a result, a trader receives tokens that are supposed to be distributed among traders from other epochs. This can lead to insufficient TIG when other traders try to withdraw funds, and the only way to ensure enough TIG is to allocate additional tokens.

Thus, the issue is marked as critical because it may prevent traders from withdrawing eligible TIG and break the tokenomic of the project.

Recommendation:
Consider marking `epochFeesGenerated[_epoch]` as 0 during recoverTig(). Other possible solution is to add a validation in createVest() that `epochAllocationClaimed[_epoch]` is not equal to `epochAllocation[_epoch]` and assign `epochAllocation[_epoch]` to `epochAllocationClaimed[_epoch]` in recoverTig().

This will prevent traders from creating xTIG out of recovered TIG and also prevent pointless execution of createVest() when all xTIG is allocated in a particular epoch.

Post-audit:

At first, the issue was marked as “Critical”, but after communication with the TigrisTrade team, the severity was downgraded to “High” because, according to TigrisTrade, the issue can only be caused by the owner of the contract, and the owner can add that TIG back into the contract into a far-in-the-future epoch to make sure affected users can redeem xTIG for TIG. Nevertheless, the Tigris Trade team has successfully resolved the issue by removing the function recoverTig() as it was deprecated.

HIGH-2 RESOLVED

Rewards might get stuck on contract for some time.

xTig.sol: claimTig()
Due to conditions for breaking the while-loop, users cannot claim rewards for some time. When iterating over an array of reward batches, iterator “i” is incremented when unlock time is not met. So if vestingPeriod was changed (specifically - if it was decreased) and there will be a situation when the reward with index “k” was not unlocked and the reward with index “k+p” was unlocked, in this case, iteration will go through else-branch and iterator “i” will be incremented. The problem is that the condition for continuing the while-loop is that iterator “i” should be less than the amount of reward batches the user is willing to claim.

Recommendation:

Change the loop in a way that the user will always be able to claim their rewards.

Post-audit:

Ability to claim rewards with any index was implemented.

MEDIUM-1 | RESOLVED

Unchecked transfers.

xTig.sol: claimTig(), earlyClaimTig(), addTigRewards(), recoverTig(), _claim(), _distribute();
LPStaking.sol: stake(), confirmWithdrawal(), earlyWithdrawal(), _claim(), distribute().
There are transfers of ERC20 tokens without result verifying a returned value, which indicates whether the operation succeeded, i.e., it is not reverted on and does not handle errors, and transfer may not be performed. It is therefore recommended to handle the return value to prevent unintended cases and to use `SafeERC20` by OpenZeppelin, which supports both tokens' implementations: those with a return boolean value and those without.

Recommendation:
Use the method `safeTransfer()` from the library `SafeERC20` by OpenZeppelin.

Post-audit:
SafeERC20 library is now used.

MEDIUM-2 | RESOLVED

Possible denial of service in claiming Tig.

xTig.sol: claimTig(), earlyClaimTig().
Users can create vesting every epoch. In this case, when users have significant reward batches when executing claimTig() and earlyClaimTig(), the transaction might run out of gas, and users will not be able to claim their Tigs. The problem is that it is impossible to state the range of epochs for claiming Tig. Thus, claimTig() and earlyClaimTig() functions are iterating over all the epochs the particular user has vested in.

Recommendation:
Add “from” and “to” limits when claimingTig in both claimTig() and earlyClaimTig() functions.

Post-audit:
Mechanism for claiming Tig was optimized and now the difficulty of such an operation is less than O(n).

MEDIUM-3 | VERIFIED

Maximum approval for Staking contract.

xTig.sol: constructor().
Approval for maximum unit value is considered an insecure practice as it allows the target to access the contract's balance permanently. The approval is granted for the GovernanceStaking smart contract. Although the contract is part of the non-upgradable protocol, granting an unlimited allowance may lead to losing funds if the GovernanceStaking contract is exploited. Thus, auditors cannot verify the security of granting a total allowance to it since the GovernanceStaking is out of scope.

Recommendation:
Make approvals for other contracts only for certain amounts, necessary for particular transfers.

Post-audit:
TigrisTrade stated that the GovernanceStaking contract does not contain functionality for transferring TIG from the xTIG contract, except the one called by xTIG, so there is no additional risk for approving maximum value.

LOW-1 | RESOLVED

Vesting period can be set to large values.

xTIG.sol: setVestingPeriod().
Parameter `_time` is not validated for any limits, meaning that vesting period can be set to large values, preventing users from ever withdrawing their TIG tokens.

Recommendation:
Add a constant representing a time limitation and validate parameter `_time` to be less or equal to it.

Post-audit:
Vesting period is now capped to 30 days.

Early unlock penalty can be set to 100%.

xTIG.sol: setEarlyUnlockPenalty().
Parameter `_percent` is validated as less or equal to the constant `DIVISION_CONSTANT`.
The continuous represents 100%, meaning the penalty can be set to a maximum value, and the user will lose all the TIG during early withdrawal.

Recommendation:
Add a certain limit for the early unlock penalty so that users won't lose all the TIG.
Otherwise, validate that users will be notified about the penalty, especially in case it is equal to 100%.

Post-audit:
Maximum early unlock penalty is now capped to 75%.

Pending Tig might be calculated in the wrong way.

xTIG.sol: pendingTig().
When calculating pending Tig, the loop iterates over all reward batches, adding the batch's reward amount to the pending Tig amount for all the unlocked vesting. When the first batch with the current block.timestamp is greater than unlock time is met, loop "breaks" and pendingTig returns calculatedTig amount. This method works fine when vestingPeriod is immutable and can not be changed. However, vestingPeriod can be reset. If its value is decreased, for example, from 30 days to 10 days, users might get their unlocked reward which is not calculated as pending Tig. For example, the User createVest on day 1, then the Owner changes vestingPeriod to 10 days. Then, the User creates a second vest, 15 days later, the user's double vest will be unlocked, but pendingTig will return 0 in this way because the loop will break on the first iteration.
If pendingTig is used in the front-end and returns the wrong pendingTig, the user will be misled, and given incorrect information about his unlocked rewards.

Recommendation:

Do not break the loop when the first unlock time is not met, but iterate over all the reward batches and add only unlocked rewards, similarly to the calculation in claiming function.

Post-audit:
Loop for calculating pending Tig is now not breaks when first locked vesting is met.

LOW-4 | VERIFIED

Asset value isn't validated when adding fees.

xTIG.sol: addFees().
Asset value (tigAssetValue[_tigAsset]) can equal zero when addFees() is called, and the function won't revert. If tigAssetValue doesn't set, the permissioned user can successfully call addFess(), while nothing will happen effectively. This can be misleading.

Recommendation:

In addFess() check that tigAssetValue[_tigAsset] is greater than zero.

Post-audit:
TigrisTrade decided not to implement validation for asset value, as, according to them, fees being added for xTIG isn't critical for the protocol to function.

LOW-5 | RESOLVED

Possible inflation attack due to lack of validation.

LPStaking.sol: whitelistAsset()
_initialDeposit argument isn't validated, so it can be set to a minimal value or 0, thus leading to the possibility of running an inflation attack. You can read more about this type of attack here. Though the inflation attack is usually specified as a high issue, the severity of the current issue is set to low since the mechanism against it is already present. It needs to have additional validations to ensure its security.

Recommendation:
Add validation for _initialDeposit. It's better to set the minimum value for _initialDeposit higher (e.g., 1e18), so performing an inflation attack without an unrealistically huge amount of tokens won't be possible.

Post-audit:
Validation for _initialDeposit was added.

Lack of documentation.

xTig.sol, LPStaking.sol.
The contracts are missing NatSpec documentation for many of their externally callable functions, variables with public visibility (that have externally callable getters), events, structs, and themselves.
Documentation helps to:

- make the code clearer and more readable, giving a fuller view;
- ensure code quality when written and modified;
- keep developers in sync and minimize unnecessary interaction;
- make it easier to integrate with other protocols.

It is therefore recommended to fully describe their action, calling requirements, features, formal parameters, and return values, applying NatSpec where required.
It is worth noting that the Solidity documentation recommends contracts to be fully annotated using NatSpec for all public interfaces (everything in the ABI).

Recommendation:
Add the full NatSpec documentation.

Post-audit:
Comprehensive NatSpec documentation was added.

INFO-2 | RESOLVED

Lack of validation.

TigrisLPStaking.sol: constructor(), setTreasury(), stake();
TigrisLPStaking.sol(AutoTigAsset.sol): constructor();
xTIG.sol: constructor(), setTreasury(), setExtraRewards().
Either not all or none of the parameters passed to these functions are validated. This can lead to unexpected behavior in further use, which is why it is recommended to validate parameters. For example, addresses should be validated not to be equal to zero addresses, and numeric parameters should be validated not to be equal to `0`. The point is especially essential for immutable variables, which are only set once during deployment.

Recommendation:
Add the necessary validation.

Post-audit:
Validations for all needed variables were added.

INFO-3 | VERIFIED

Rewards are not claimed when users confirm withdrawal.

LPStaking.sol: confirmWithdrawal()
_claim() function is not called when the user confirms withdrawal and users are not claiming their rewards in that operation. As the cancelWithdraw() _claim call is present, it also should be presented in the confirmWithdrawal() function.

Recommendation:
Add claim of rewards when users confirm withdrawal.

Post-audit:
TigrisTrade verified that behavior, stating that the claim of rewards was not supposed to happen when the user confirmed the withdrawal.

INFO-4 | RESOLVED

Code duplicating in pending Tig calculation .

LPStaking.sol: earlyClaimTig()
_unstakeAmount += reward.amount is duplicating in 118 and 121 lines of code in both “in” and “else” branches, which is making code readability more complex.

Recommendation:
Put duplicating line of code off the if-statement.

Post-audit:
Duplicating line code was taken off the if-statement.

INFO-5 | RESOLVED

Lack of events.

Although most functions emit events, there are still functions that should emit them to provide a way to track all the changes on the smart contracts.
The following functions lack the event:
xTIG.sol: setTigAssetValue(), recoverTig().

Recommendation:
Add events to the functions.

Post-audit:
Events were added and not they are emitted in the corresponding functions.

INFO-6 | VERIFIED

Users are forced to withdraw TIG in case vesting period is set to 0.

xTIG.sol: createVest(), lines 78-80.
In case the variable `vestingPeriod` is set to 0, users are forced to withdraw their TIG in exchange for xTIG immediately. However, based on the documentation, users are encouraged to hold xTIG to continue earning the reward tokens. Thus, it is assumed that users shouldn't be forced to withdraw it.

Recommendation:
Remove the mandatory claiming of TIG or verify that it corresponds to the protocol's logic and update the documentation accordingly.

Post-audit:
TigrisTrade verified that behavior and stated that xTIG earns the same rewards as normal staked TIG, so the user won't miss out on rewards by forcing the withdrawal. Forced withdrawal is a design decision made by the team.

INFO-7 RESOLVED

Iterator is declared before the loop but its value is not being changed.

xTig.sol: earlyClaimTig()
Iterator "i" not being changed in a loop for claiming the user's reward. Its value is zero all over the loop iterations.

Recommendation:
Remove the iterator if it is not being changed.

Post-audit:
Loop was modified in a way that iterator is now being changed during the loop iterations.

	TigrisLPStaking.sol xTIG.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting TigrisTrade in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the functionality of the code, as well as a review of the TigrisTrade contract requirements for details about issuance amounts and how the system handles these.

LPStaking

Operations

- ✓ Should whitelist (1061ms)
- ✓ Should unwhitelist asset (880ms)
- ✓ Shouldn't unwhitelist asset with invalid parameters (786ms)
- ✓ Shouldn't whitelist with invalid params (483ms)
- ✓ Shouldn't set with invalid params (605ms)

Staking

- ✓ Should stake with autocompounding (1205ms)
- ✓ Should stake without autocompounding (1417ms)
- ✓ Shouldn't stake with invalid parameters (877ms)
- ✓ Shouldn't stake when deposit cap exceeded (927ms)

Withdraw

- ✓ Should initiate withdraw with autocompounding (1569ms)
- ✓ Shouldn't initiate withdraw with invalid parameters (1480ms)
- ✓ Should initiate withdraw without autocompounding (499ms)
- ✓ Should initiate max withdraw without autocompounding (428ms)
- ✓ Should withdraw without autocompounding + without fee + without withdrawal period (527ms)
- ✓ Should withdraw with autocompounding + without fee + without withdrawal period (777ms)
- ✓ Should confirm withdrawal without autocompounding (1002ms)
- ✓ Shouldn't confirm withdrawal with invalid parameters (412ms)
- ✓ Should cancel withdrawal without autocompounding + with fee (493ms)
- ✓ Shouldn't cancel withdrawal with invalid params (776ms)
- ✓ Should cancel withdrawal with autocompounding (1510ms)
- ✓ Should make early withdrawal with autocompounding + with withdrawal fee (1400ms)
- ✓ Should make early withdrawal without autocompounding + without withdrawal fee (1040ms)
- ✓ Shouldn't make early withdrawal with invalid parameters (980ms)
- ✓ Should make early max withdrawal with autocompounding (853ms)

Distribution

- ✓ Should distribute rewards (3 users) (867ms)
- ✓ Should distribute rewards (2 users) (659ms)
- ✓ Shouldn't distribute rewards (2 users) if transfer failed (423ms)
- ✓ Should distribute rewards (2 users + only autocompound) (702ms)

✓ Should distribute rewards (2 users + only staking) (449ms)
✓ Shouldn't distribute rewards with invalid params

Claim

✓ Should claim (3 users) (713ms)

Scenarios

✓ Should withdraw if deposit cap is set to be lower than total deposits (1095ms)
✓ Should claim rewards correctly without deposit funds touching (1471ms)
✓ Should withdraw correctly if the user changes his autocompound mode while he has pending withdrawal (959ms)
✓ Shouldn't cancel withdrawal with invalid params (776ms)
✓ Should claim rewards correctly (1107ms)
✓ Should withdraw correctly if user changes his autocompound mode (on - off) while he has pendi withdrawal (945ms)
✓ Should withdraw correctly if user changes his autocompound mode (off - on) while he has pendi withdrawal (945ms)
1) Should not allow to perform inflation attack
2) Should withdraw correctly if autotig transferred to user when he staked
3) Shouldn't allow user to drain the contract's balance by repeatedly disabling autocompounding

37 passing (36s)
3 failing

xTIG

addFees

✓ Should work correctly
✓ Should revert if not permissioned
1) Should revert if tigAssetValue equals 0 for token

addTigRewards

✓ Should work correctly (55ms)
✓ Should revert if not owner
✓ Should revert if epoch is in the past

createVest

✓ Should work correctly (91ms)
✓ Should revert if no fees generated (43ms)

claimTig

✓ Should work correctly (153ms)
✓ Should work correctly if called from createVest when vestingPeriod set to 0 (131ms)
✓ Should revert if no TIG to claim (170ms)

earlyClaimTig

✓ Should work correctly (156ms)
✓ Should work correctly if already unlocked (149ms)
✓ Should revert if no TIG to claim (143ms)

Scenarios

✓ [day 0] User1 vests 100 xTIG -> [day 5] user1 vests 100 xTig -> [day 35] user claims

✓ [day 0] User1 vests 100 xTIG -> [day 15] user1 vests 100 xTig -> [day 30] user early claims Tig and has 150 Tig on balance (216ms)
✓ [day 0] User1 vests 100 xTig -> [day 100] User1 early claims Tig end gets 100 Tig (150ms)

16 passing (1s)
1 failing

FILE	% STMTS	% BRANCH	% FUNCS
TigrisLPStaking.sol	99.11	83.33	100
xTIG.sol	70.59	54.05	62.5

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by the TigrisTrade team

As a part of our work assisting TigrisTrade in verifying the correctness of their contract code, our team has checked the complete set of tests prepared by the TigrisTrade team.

We need to mention that the original code has a significant original coverage with testing scenarios provided by the TigrisTrade team. All of them were also carefully checked by the team of auditors.

TigrisLPStaking

Zero address checks

✓ Should not allow staking zero address asset (38ms)
✓ Should not allow withdrawal initiation with zero address asset
✓ Should not allow withdrawal confirmation with zero address asset
✓ Should not allow withdrawal cancellation with zero address asset
✓ Should not allow claiming zero address asset
✓ Should not allow distributing zero address asset
✓ Should not allow toggling zero address autocompounding
✓ Should not allow whitelisting zero address asset
✓ Should not allow unwhitelisting zero address asset
✓ Should not allow setting zero address deposit cap

onlyOwner checks

✓ Should not allow non-owners to set a new treasury
✓ Should not allow non-owners to whitelist an asset
✓ Should not allow non-owners to unwhitelist an asset
✓ Should not allow non-owners to set withdrawal period
✓ Should not allow non-owners to set deposit cap
✓ Should not allow non-owners to set deposit fee
✓ Should not allow non-owners to set early withdrawal fee
✓ Should not allow non-owners toggle early withdrawal

Deposit cap

✓ Should not allow deposits above the deposit cap (761ms)

Staking

✓ Should not allow setting deposit fee over 1%
✓ Should not allow staking non-whitelisted assets (47ms)
✓ Asset whitelisting (186ms)
✓ Should not allow staking more than the user balance (183ms)
✓ Should reduce stake amount by deposit fee and transfer fee to treasury (211ms)

Withdrawal

✓ Should not allow setting withdrawal period over 30 days
✓ Should not allow initiating a withdrawal of zero
✓ Should not allow initiating a withdrawal of more than the user deposited with staked balance (263ms)
✓ Should not allow initiating a withdrawal of more than the user deposited with autocompounding balance (304ms)
✓ Should allow max withdrawal of user's autocompounding balance (465ms)
✓ Should allow max withdrawal of user's staked balance (419ms)
✓ Withdrawal should be confirmed in the same transaction as initiation if withdrawal period is zero (359ms
✓ Should not allow confirming a withdrawal of zero
✓ Should not allow confirming a withdrawal before the withdrawal period (355ms)
✓ Should not allow cancelling a withdrawal of zero
✓ Should correctly cancel a withdrawal for an autocompounding user (407ms)
✓ Should correctly cancel a withdrawal for a non-autocompounding user (420ms)

Early withdrawal

✓ Should not allow early withdrawal if disabled
✓ Should not allow early withdrawal of zero
✓ Should not allow early withdrawal of zero address
✓ Should not allow early withdrawal of more than the user deposited with staked balance (307ms)
✓ Should not allow early withdrawal of more than the user deposited with autocompounding balance (334ms)
✓ Should not allow early withdrawal if withdrawal is already initiated (227ms)
✓ Early withdrawal should send the correct amount to the staking user and fee to the treasury (206ms)
✓ Early withdrawal should send the correct amount to the autocompounding user and fee to the treasury (286ms)
✓ Owner should be able to set early withdrawal fee of 25% or less
✓ Setting an early withdrawal fee over 25% should rever

Distributio

✓ If total deposited is zero, distribution should simply return (132ms)
✓ If amount is zero, distribution should simply return (164ms)
✓ If asset is unwhitelisted, distribution should simply return (232ms)
✓ Failing to transferFrom in distribution should simply return (251ms)

AutoTigAsset

onlyFactory modifier

✓ Should not allow non-factory to mint
✓ Should not allow non-factory to burn

Trading

Constructor

✓ Deployment should revert if any address is zero (252ms)

Check onlyOwner and onlyProtocol

✓ Set max win percent

✓ Set LP distributio
✓ Set fees
✓ Set trading extenstio
✓ Set block delay
✓ Set allowed vault
✓ Set limit order execution price range
✓ Set valid signature timer
✓ Set paused
✓ Set max gas price
✓ Set chainlink enabled
✓ Set node
✓ Set allowed margin
✓ Set min position size
✓ Modify short oi
✓ Modify long oi
✓ Set referral

Setters

✓ Setting LP distribution over 100% should revert
✓ Set max win percent (56ms)
✓ Set valid signature timer
✓ Set paused
✓ Set max gas price
✓ Set fees (98ms)

Signature verification

✓ Valid signature should work as expected (329ms)
✓ Using an expired signature should revert (422ms)
✓ Using an invalid signature should revert (233ms)
✓ Using a signature from a non-node address should revert (330ms)
✓ Using a future signature should revert (308ms)
✓ Trying to trade a closed market should revert (294ms)
✓ Trying to trade with asset that doesn't match signed message should revert (241ms)
✓ Trying to trade an asset with zero price should revert (233ms)

Market trading

✓ Opening a market long with a bad SL should revert (253ms)
✓ Opening a market short with a bad SL should revert (249ms)
✓ Opening a market short with no SL should open position (401ms)
✓ Opening a position while trading is paused should revert (87ms)
✓ Opening a position with < min leverage should revert (82ms)
✓ Opening a position with > min leverage should revert (81ms)
✓ Opening a position with a non-allowed margin asset should revert (66ms)
✓ Opening a position on a non-allowed pair should revert (56ms)

✓ Opening a position with < min position size should revert (71ms)
✓ Opening a position with an unapproved stablevault should revert (43ms)
✓ Trading should revert if margin asset isn't listed in stablevault (56ms)
✓ Opening a position with a bad stablevault should revert (240ms)
✓ Opening a market position with tigAsset (460ms)
✓ Closing over 100% should revert (497ms)
✓ Closing 0% should revert (388ms)
✓ Closing with a bad stablevault should revert (711ms)
✓ Closing a limit order should revert (204ms)
✓ Closing someone else's position should revert (408ms)
✓ Partially closing a position should revert if position size would go below minimum position size, fully closing should not revert (1209ms)
✓ Partially closing a position should have correct payout and new margin (776ms)

Trading using <18 decimal token

✓ Opening and closing a position with tigUSD output (658ms)
✓ Opening and closing a position with <18 decimal token output (681ms)

Limit orders and liquidations

✓ Executing long order where TP would go past open price should remove TP (507ms)
✓ Executing short order where TP would go past open price should remove TP (524ms)
✓ Executing limit order before a second has passed should have no bot fees (494ms)
✓ Creating and executing limit buy order, should have correct price and bot fees (483ms)
✓ Creating and executing limit sell order, should have correct price and bot fees (498ms)
✓ Creating and executing buy stop order, should have correct price and bot fees (587ms)
✓ Creating and executing sell stop order, should have correct price and bot fees (510ms)
✓ Creating a limit with zero price should revert (58ms)
✓ Creating a limit with orderType 0 should revert (43ms)
✓ Executing an open position should revert (469ms)
✓ Creating and executing an unmet limit buy order should revert (305ms)
✓ Creating and executing an unmet limit sell order should revert (337ms)
✓ Creating and executing an unmet buy stop order should revert (379ms)
✓ Creating and executing an unmet sell stop order should revert (404ms)
✓ Creating and executing a limit order with price out of range (too low) should revert (376ms)
✓ Creating and executing a limit order with price out of range (too high) should revert (310ms)
✓ Executing TP before block delay has passed should revert (347ms)
✓ Executing an unmet long TP should revert (395ms)
✓ Executing an unmet short TP should revert (549ms)
✓ Executing an unmet long SL should revert (538ms)
✓ Executing an unmet short SL should revert (495ms)
✓ Executing a TP on an open position with no TP should revert (441ms)
✓ Executing an SL on an open position with no SL should revert (461ms)
✓ Limit closing a limit order should revert (325ms)
✓ Using too much gas should revert (63ms)

✓ Executing a TP/SL should have no bot fees if a second hasn't passed since last update (1084ms)
✓ Executing a long TP should have correct fees and payout (981ms)
✓ Executing a short TP should have correct fees and payout (983ms)
✓ Executing a long SL should have correct fees and payout (995ms)
✓ Executing a short SL should have correct fees and payout (1000ms)
✓ Executing limit order while trading is paused should revert (243ms)
✓ Cancelling a an open position should revert (501ms)
✓ Cancelling a limit order should give collateral back (415ms)
✓ Opening a limit order with tigAsset (153ms)
✓ Liquidating a limit order should revert (268ms)
✓ Liquidating a non-liquidatable position should revert (517ms)
✓ Liquidating long position, should have correct rewards (595ms)
✓ Liquidating short position, should have correct rewards (604ms)

Modifying functions

✓ Updating TP/SL on a limit order should revert (208ms)
✓ Updating TP (477ms)
✓ Updating SL (693ms)
✓ Add margin should revert if on limit order (315ms)
✓ Add margin should revert if leverage goes below min leverage (458ms)
✓ Add margin with non-tigAsset (523ms)
✓ Add margin with tigAsset (486ms)
✓ Add margin with non-tigAsset with permit (678ms)
✓ Remove margin should revert if on limit order (249ms)
✓ Remove margin should revert if leverage goes above max leverage (527ms)
✓ Remove margin should revert if it would go into liquidation threshold (819ms)
✓ Remove margin with non-tigAsset output (528ms)
✓ Remove margin with tigAsset output (594ms)
✓ Adding to position should revert on limit order (298ms)
✓ Adding to position on long should combine margin and open price proportionally, accInterest should work as expected (753ms)
✓ Adding to position on short should combine margin and open price proportionally, accInterest should work as expected (695ms)
✓ Add margin should revert if current PnL >= maxPnL% - 100% (436ms)
✓ Remove margin should revert if current PnL >= maxPnL% - 100% (493ms)
✓ Adding to position should revert if current PnL >= maxPnL% - 100% (544ms)

PnL calculations

✓ Long and PnL is positive (907ms)
✓ Short and PnL is positive (775ms)
✓ Long and PnL is negative (651ms)
✓ Short and PnL is negative (631ms)
✓ Max win should be capped to 10x on full close (647ms)
✓ Max win should be capped to 10x on partial close relative to margin being closed (791ms)

✓ Max win should be unlimited if max win is set to zero (843ms)
✓ Total loss should pay out zero (594ms)
✓ Fees calculations (887ms)

Order minimum delay check

✓ CheckDelay on opening + interaction should work as expected (336ms)

Spread calculations

✓ Open price should be correct after spread (803ms)

Liquidation price view calculations

✓ Long liquidation price without funding rate (561ms)
✓ Short liquidation price without funding rate (452ms)
✓ Long liquidation price with funding rate (445ms)
✓ Short liquidation price with funding rate (427ms

Referral calculation

✓ Referrer should receive the correct amount of referral rewards and decreased tradin fee should be correct on both opening and closing and referral should be locked (1390ms)
✓ Referral tiers (1522ms)

Comparing node's prices to Chainlink price feed

✓ Setting Chainlink feed for an asset and enabling Chainlink should work as expected
✓ Opening a trade with a price that doesn't match Chainlink's price should revert (319ms)
✓ Opening a trade when Chainlink price feed returns zero shouldn't revert (387ms)
✓ Opening a trade with a price that matches Chainlink's price shouldn't revert (356ms)

Open interest calculations

✓ Should work correctly on long, short, full and partial close (1798ms)

Trading through a prox

✓ Approval should revert if sending ETH to contract that can't receive ETH
✓ Should revert if proxy isn't approved
✓ Should work as expected if proxy is approved (414ms)

TRADING INCENTIVES

✓ xTIG should not be transferable (64ms)
✓ Owner can set a new vesting period
✓ Setting an early unlock penalty reverts
✓ Owner can set early unlock penalty percent
✓ Whitelisting a reward twice should revert
✓ Unwhitelisting a reward twice should revert
✓ An address that isn't the trading contract can't store fee
✓ Adding rewards to concluded epochs should revert
✓ User creating a vest with no trading fees generated by the user should continue
✓ Claiming TIG early with no xTIG should revert
✓ Claiming TIG normally with no xTIG should revert (94ms)
✓ Pending returns zero if staked xTIG supply is zero
✓ Upcoming xTIG returns zero if no fees are generated in epoc
✓ Should have correct contract staked TIG balance

✓ Upcoming TIG and storing epoch fees
✓ Vesting should result in correct xTIG balances and supply (153ms)
✓ Vesting should result in correct RewardBatch array (93ms)
✓ Early unlock pending TIG amount and claim (521ms)
✓ Full unlock pending TIG amount and claim normally (412ms)
✓ Full unlock pending TIG amount and claim with earlyClaimTig (427ms)
✓ Earning staking rewards from xTIG should be 1:1 with TIG (1275ms)
✓ If an user claims TIG with some xTIG locked and some unlocked, only claim the unlocked (622ms)
✓ Unclaimed xTIG can be recovered into treasury (312ms)
✓ Treasury should receive excess tigUSD rewards (241ms)
✓ Pending returns zero if staked TIG balance is zero (108ms)
✓ Can only recover from concluded epochs (87ms)
✓ Optional extra rewards module (568ms)
✓ onlyOwner: Can allocate TIG rewards
✓ onlyOwner: Can set tigAsset val
✓ onlyOwner: Can set contract to store fee
✓ onlyOwner: Can set extra rewards
✓ onlyOwner: Can set vesting period
✓ onlyOwner: Can set early unlock penalty
✓ onlyOwner: Can whitelist a reward
✓ onlyOwner: Can unwhitelist a reward
✓ onlyOwner: Can recover unclaimed TI

224 passing (3m)

FILE	% STMTS	% BRANCH	% FUNCS
TigrisLPStaking.sol	100	100	100
xTIG.sol	94.85	93.24	93.75

We are grateful for the opportunity to work with the TigrisTrade team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the TigrisTrade team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo