Smart Contract Auditing for ZenPool

Testers

1. Or Duan
2. Omri Shdaimah

Table of Contents

Table of Contents 2

Management Summary 3

System Overview 4

Vulnerabilities by severity 5

Scope and Contracts 7

Smart Contract Auditing Findings 8

- Users Can Withdraw Funds From Other Balances Until the Pool Is Empty 8
- Unsafe selfdestruct in Proxy Contract 10
- User’s Pools are Subjected to MEV Attacks 11
- Lack of Guardian Mitigation 12
- Disallow Zero Amount Transfers 16
- Insufficient Logging for Privileged Functions 17
- Redundant Condition Checking 18
- Naming Inconsistency 19

Management Summary

ZenPool team contacted Sayfer Security in order to perform a full security audit for all their contracts.

Before assessing these services we held a kickoff meeting with ZenPool's technical team and received an overview of the system and the goals for this assessment.

The following audit took 20 man-days to complete, all on-chain contracts were reviewed line by line with at least 2 auditors per contract. Due to time constraints from the client-side, we reviewed off-chain components and took a best-effort approach to make sure they do not contain any critical vulnerabilities.

We found a total of 9 findings, 3 of which were classified as "high" risk vulnerabilities and could be exploitable by malicious attackers, who could empty the pool's funds completely.
We documented our process and our suggestions for how each vulnerability should be fixed. ZenPool's team implemented the fixes which were documented in every vulnerability section.

System Overview

ZenPool is an open-source, non-custodial token and lending market protocol.

Users can deposit their crypto assets to earn interest or borrow other tokens to pay interest in ZenPool's market.
ZenPool has its own token called ZEN.

ZEN is a free-floating currency backed by the stable coin BUSD treasury supply. ZEN tokens can only be minted and burned by the protocol, only in response to price does the protocol do so. Each ZEN is supported by at least one BUSD.
If the price of ZEN falls below 1 BUSD, the protocol buys and burns ZEN, pushing the price back up to 1 BUSD.

ZenPool also supports bonds which are another way to increase its treasury. The protocol sells bonds in exchange for various assets, and in exchange, the buyer receives ZEN at a significantly reduced price. This boosts the treasury and allows ZeenPool to provide incredible yields to its customers.

Finally, a portion of all ZenPool product fees will be used as additional backing, potentially providing ZenPool’s token with an infinite runway for staking rewards.

Vulnerabilities by severity

Low	Medium	High	Informational
1	2	3	3

High

- Transaction DoS
- direct loss of funds
- permanent freezing of funds

Medium

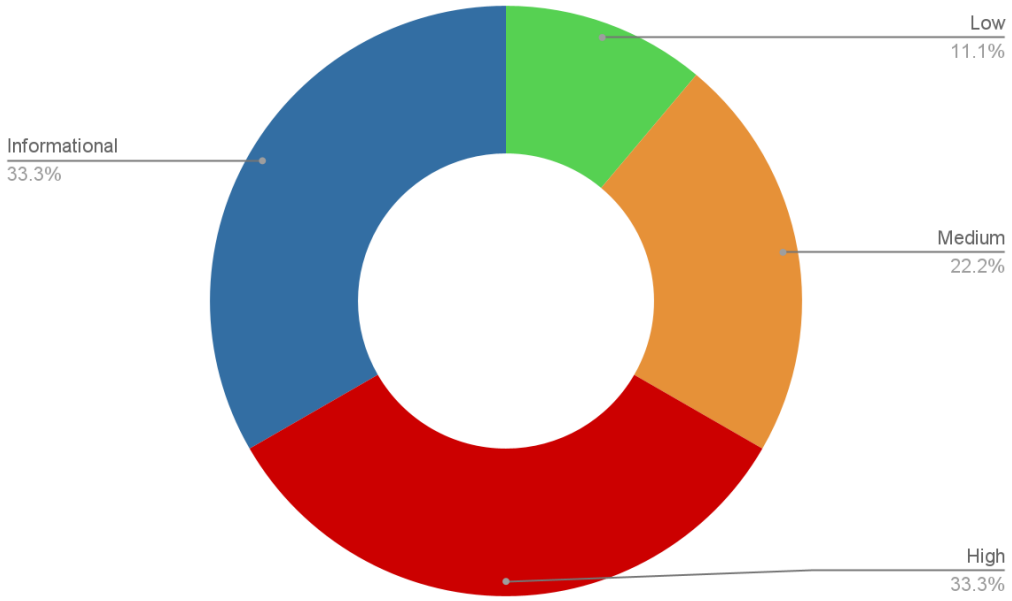
- Attacks against thin clients
- Partially DoS
- Gas attacks

Low

- Best practices

Informational

- Does not harm the system, and we don't have enough data or knowledge to prove it will ever do any harm, yet it is important to share our concerns.



Scope and Contracts

As part of the project scoping and to understand our clients' focus and needs, we defined the contracts we should test on this audit. Together we found the best balance that suits this specific project.

The scope is a soft scope by definition, meaning we could test on a local environment other contracts that might be interacting with the contract that is defined as the audit's scope. This gives us the flexibility to find security issues that might be overlooked otherwise.

The list of contracts and their Github commit:

Commit Hash	Contract
5d253cb3c544a17399e7d2f4c381a1065bcfa0a0	https://github.com/zenpoolproject/zenpool/tree/master/contracts/createGovernor.sol
3887cd307163d130477d4805e74ade11f84d7a4d	https://github.com/zenpoolproject/zenpool/tree/master/contracts/ZenPoolUserManager.sol
a50062dfc63a0e82ec43de98865420193270236a	https://github.com/zenpoolproject/zenpool/tree/master/contracts/ZenPoolManager.sol
c4d0db2f9fbfc3146a1a1a797e55c3f3d7a19899	https://github.com/zenpoolproject/zenpool/tree/master/contracts/ZenTickets.sol
844132ae5210cb27101bf4a415d9c16e201e1afc	https://github.com/zenpoolproject/zenpool/tree/master/contracts/ZenPoolFunds.sol
a70146c5d276f933a79c567d2e96512302a6a940	https://github.com/zenpoolproject/zenpool/tree/master/contracts/ZenGovernorAlpha.sol
d476137af592fe71cae2578a62e2f8f92b335d8c	https://github.com/zenpoolproject/zenpool/tree/master/contracts/ZenLendingPool.sol
8c5d858ad7fbfe856cd23160fd8810997f3fdf70	https://github.com/zenpoolproject/zenpool/tree/master/contracts/ZenGovernorAlpha.sol

Smart Contract Auditing Findings

Users Can Withdraw Funds From Other Balances Until the Pool Is Empty

Contract	contracts/ZenPoolFunds.sol
Risk	High
Fixed	Yes
Found by	Manual Testing
Description	The `withdrawFunds` function validates that the user can only withdraw the maximum amount in his balance, checked by its address and later compared within the token pool.

```
function withdrawFunds(address _addrs, address _tkn, uint256 _amount) external returns(uint256)
{
   require(_amount <= getMaxWithdrawForAddress(_tkn, _addrs), "Balance too low.");
   uint256 TLV = config.tokenInfo().getLTV(_tkn);
```

Later the withdrawn amount is deducted from the user's current max borrow at the current price. If the total amount of borrowed funds by the user exceeds the new max borrow, the method fails because the user no longer has enough collateral to support their borrow position. This requirement, however, is only checked if the user is not already over-leveraged:

```
  if(getBorrowFunds(_addrs) <= getMaxBorrow(_addrs))
    require(
       getBorrowFunds(_addrs) <= getMaxBorrow(_addrs).sub(
```

```
            _amount.mul(config.tokenInfoRegistry().priceFromAddress(_tkn))
            .mul(borrowLTV).div(Utils.getDivisor(address(config), _tkn)).div(100)
        ), "Insufficient funds when withdraw.");
```

An attacker could use this functionality to exploit the withdrawFunds function to withdraw more than the max borrow amount.
He could do that because the getMaxBorrow will only be checked if the user borrows less than the maximum amount he is allowed, in a variety of scenarios the attacker could abuse the flow to skip this specific require, allowing him to call the withdrawFund multiple times until the pool will get emptied.

Mitigation	Change the require to be called before line 145 so it does not depend on the if statement. This way the require will always be executed.

Unsafe selfdestruct in Proxy Contract

Contract	contracts/ZenTickets.sol contracts/ZenPoolManager.sol
Risk	High
Fixed	Yes
Found by	Manual Testing
Description	When the main `ZenPoolManager` is deployed, it also deploys multiple contracts as well. One of them is the `ZenTickets` contract which is mostly secured, except for the `destoryContract` function which has no ACL mechanism in place like the rest of the functions: `function destroyContract(address payable addr) external { selfdestruct(addr); }` By exploiting this functionality an unauthenticated attacker could call the `destoryContract` function and choose where the ETH of the contract will be transferred to. This is easy to exploit from the attacker's perspective.
Mitigation	Use the `onlyOwner` modifier like it's used in the reset of the functions. Another more specific solution would be to use a custom ACL mechanism like openzeppelin's roles ACL that will only allow specific roles to access the `destoryContract` functions.

User's Pools are Subjected to MEV Attacks

Contract	contracts/ZenPoolUserManager.sol
Risk	High
Fixed	No - Risk Taken
Found by	Manual Testing
Description	The main `ZenPoolUserManager` is handling custom user pools that have been created in the system. The user can call different actions on these pools if he has the right permissions to do so, a user can also grant access via role mechanism based on OpenZeppelin's access control contracts The `ZenPoolUserManager` has a proper ACL mechanism, but at the same time, it has 2 functions that are subjected to front/back running or any MEV attacks. `function destroyContract(address payable addr) external { selfdestruct(addr); }` While the business logic itself is right, and seems like it can not do much harm as it has a proper ACL, a user with the same role could exploit it via MEV attacks.
Mitigation	Use the `onlyOwner` modifier like it's used in the reset of the functions. Another more specific solution would be to use a custom ACL mechanism like openzeppelin's roles ACL that will only allow specific roles to access the `destoryContract` functions.

Lack of Guardian Mitigation

Contract	contracts/ZenGovernorAlpha.sol
Risk	Medium
Fixed	Fixed
Found by	Legacy Code Analysis and Manual Testing

Description	During our audit we performed Original Code Analysis, and compared the client's current codebase to the open-source projects that the client used to develop the code. If we found changes were made in the open-source code we took a deeper look at these to make sure they were done wisely.
	Most clients consider 3rd party open-source projects secure because they come from major vendors (E.G Uniswap pool). This assumption is mostly true. However, major security bugs occur when clients perform minor changes in the code and assume these changes don't affect the overall security of the product.
	
	During our Original Code Analysis, we found that ZenPool uses code from STRIKE protocol in order to create governance token, the original code from STRIKE repo is:
	`require(msg.sender == guardian ||`
	`strk.getPriorVotes(proposal.proposer, sub256(block.number, 1)) < proposalThreshold(),`
	`"GovernorAlpha::cancel: proposer above threshold");`
	
	While the code in ZenGovernorAlpha is:
	`require(zen.getPriorVotes(proposal.proposer, sub256(block.number, 1)) < proposalThreshold(),`
	`"ZenGovernorAlpha::cancel: proposer above threshold");`
	
	The following statement was removed `msg.sender == guardian`, this means guardian can not cancel orders if a threshold is reached. Eliminating the power of guardians can cause very dangerous situations. For instance, if a compromised proposal that transfers all the contract money to a malicious actor was performed (by a private key

	hack for example) the threshold protection mechanism is redundant and can't help, because there is no guardian that can stop the proposal from happening.
Mitigation	Add `msg.sender == guardian` to the `require` .

Lack of Reenetrancy Guard

Contract	contracts/ZenLendingPool.sol
Risk	Medium
Fixed	Fixed
Found by	Slither and Manual Testing
Description	The following borrow function does not contain a reentrancy guard nor the checks-effects-interactions pattern, this is currently not exploitable as the pool only supports ERC20 tokens, but if in the future new ERC-777 will be added a reentrancy exploit can occur and cause full drawing of contract funds.

```
function borrow(uint256 borrowAmount) external {
    uint256 depositOfZenRequired = calculateDepositOfZenRequired(
        borrowAmount
    );

    require(
        _token.balanceOf(address(this)) >= borrowAmount,
        "Not enough token balance"
    );

    zenToken.transferFrom(msg.sender, address(this), depositOfZenRequired);
```

In addition, the code does not follow the checks-effects-interactions pattern. In this specific case, the `require` is checked after some business logic is already implemented, this could cause reentrancy bugs if side effects will be added before the `require`.

Mitigation	Add a reentrancy guard for same function reentrancy and also use the checks-effects-interactions pattern for complex cross-function reentrancy.

Disallow Zero Amount Transfers

Contract	contracts/ZenPoolFunds.sol
Risk	Low
Fixed	Yes
Found by	Manual Testing
Description	The `ZenPoolFunds.sol` contract allows zero amount transfers between users. While by itself it is not a security vulnerability, this is a code sample that can expand the attack vector of a malicious attacker. The vulnerability exist because `transfer` on ZenPool emit events, by doing zero amount `transfer` an attacker could emit multiple events that can in some scenarios trigger off-chain business logic, without even holding any token in the pool.
Mitigation	A deeper understanding of the business is needed. If possible, remove this functionality. If there are use cases where it makes sense to use zero amount transfers, implement another layer of checks to limit the use to the users who are part of this group.

Insufficient Logging for Privileged Functions

Contract	contracts/ZenGovernorAlpha.sol
Risk	Info
Fixed	Yes
Found by	Manual Testing

Description	The following privileged `onlyOwner` function does not emit events.
	Events are a common practice for privileged functions to announce to the public that a change has been made.
	Without events, changes are harder to detect and users get surprised by the contract behavior.
	
	The owner can call the `chainId` by executing `setChainId()` and modifying the `chaindId` parameter without any event being emitted in the process.
	
	`function setChainId(uint256 _chainid) external onlyOwner{`
	`uint256 chainId;`
	`chainId = _chainid;`
	`}`

Mitigation	Add code that emits events.

Redundant Condition Checking

Contract	contracts/createGovernor.sol
Risk	Info
Fixed	Yes
Found by	Manual Testing

Description	When creating a new governor token the `_timelockDelay` parameter is being validated twice. once in all checks when the function starts and then as part of another `require`, this is redundant and should be removed.
	
	`require(_timelockDelay > 0, "Time lock delay parameter is 0");`
	`ZenTimelock ZenTimeLock = new ZenTimelock(msg.sender, _timelockDelay);`
	`require(address(ZenTimeLock) != address(0) && _timelockDelay > 0, "Time lock contract not created");`
	`ZenGovernorAlpha governor = new ZenGovernorAlpha(_governorName, address(ZenTimeLock), zenTokenAddr, msg.sender, _votingDelay,`
	`_votingPeriod, _proposalThreshold, _votingThreshold);`
	`require(address(governor) != address(0), "Governor Contract not created");`
	
	Duplicate code requires more gas to run and can lead to future bugs when the business logic changes.

Mitigation	Remove the second redundant `require`.

Naming Inconsistency

Contract	- Multiple -
Risk	Info
Fixed	Risk Taken
Found by	Slither and Manual Testing

Description	There are multiple occurrences of different styles of capitalizations (lower camel case, snake case etc.).
	
	`function deposit_token(...)`
	`function checkEther(...)`
	`function getConfig(...)`
	`function get_batch_user_setting(...)`

Mitigation	Review the code for the code styling. Add a code styling guide, and enforce it using a CI task.