PartyDAO Security Review

Version 1.0

July 24, 2023

Conducted by:
Georgi Georgiev (Gogo), Independent Security Researcher

Table of Contents

1 About Gogo

Georgi Georgiev, known as Gogo, is an independent security researcher specializing in Solidity smart contract auditing and bug hunting. Having conducted numerous solo and team smart contract security reviews, he always strives to deliver top-quality security auditing services. For security consulting, you can contact him on Twitter, Telegram, or Discord - *@gogotheauditor*.

2 Disclaimer

Audits are a time, resource and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to find as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Overview

Project Name	PartyDAO
Repository	https://github.com/PartyDAO/party-protocol
Commit hash	b0c85da2a4e65df2afcefd9435ef25d3e826dc0f
Documentation	https://docs.partydao.org
Methods	Manual review

Scope

https://github.com/PartyDAO/party-protocol/pull/239

Issues Found

Critical risk	1
High risk	1
Medium risk	1
Low risk	2
Informational	6

5 Findings

5.1 Critical severity

5.1.1 Executor can steal funds from the Party due to an incorrect balance update check

Severity: *Critical Risk*

Context: ERC20SwapOperator.sol#L134-L137

Description: The `ERC20SwapOperationData` struct data is defined when the swap operation proposal is created and contains the `minReceivedAmount` property to prevent the Party from losing funds during the swap due to slippage or a malicious executor.

The following lines of code in `ERC20SwapOperator.execute` validate that the amount of output tokens received from the swap is at least the minimum specified by the Party members:

```
// Get the expected receiver of the tokens.
address payable receiver = ex.isReceivedDirectly
    ? payable(msg.sender)
    : payable(address(this));

// Get the received amount.
uint256 receivedAmount = op.toToken == ETH_TOKEN_ADDRESS
    ? receiver.balance
    : op.toToken.balanceOf(receiver);

// Check that the received amount is at least the minimum specified.
if (receivedAmount < op.minReceivedAmount) {
    revert InsufficientReceivedAmountError(receivedAmount, op.minReceivedAmount);
}
```

Since the function is called by the Party contract using a standard CALL opcode, the receiver can be specified to be either directly the Party (msg.sender) or the ERC20SwapOperator contract (address(this)). While the latter is not expected to hold any funds except during the transaction, the Party contract can have a balance of any tokens, including the `op.toToken`.

The problem here is that instead of checking the increase in the receiver's balance of the `op.toToken`, the validation is made on the whole receiver's balance. Therefore, the check can be easily passed, and the operation proposal executor could steal the difference between the actual amount of tokens that should have been received and the `op.minReceivedAmount`.

Recommendation: Calculate and use the difference in balance before and after the swap is executed instead of the whole receiver's balance.

Resolution: Resolved in commit b35407f.

5.2 High severity

5.2.1 Party contracts storage will get corrupted when a Zora auction is created

Severity: *High Risk*

Context: ListOnZoraProposal.sol#L61, PartyGovernance.sol#L197

Description: The ListOnZoraProposal contract was modified to use a new Zora auction contract due to a vulnerability reported in the previously used Zora AuctionHouse contract. The new version requires sellers of an NFT to use the "ZORA ERC-721 Transfer Helper" contract to manage approvals and transfers of the NFTs for sale.

The `zoraAuctionModuleApproved` storage variable is used in ListOnZoraProposal to check whether the approval on line 146 has already been made. As stated in a comment, this variable should be updated individually for each Party contract as the Party contracts execute this functionality via a DELEGATECALL and the variable is saved in and read from the contract's storage.

The issue here is that the `zoraAuctionModuleApproved` variable is not stored in a storage slot determined by the keccak256 hash of a string chosen for this specific purpose, but instead, it is stored at slot 0, which means that the value that will be read and modified is the one already stored in the Party contracts at slot 0.

This value is again of a boolean variable found in the PartyGovernance contract, serving a completely different purpose. Therefore, this storage collision can result in an unexpected behavior in both the ListOnZoraProposal and PartyGovernance contracts.

Recommendation: Consider removing the `zoraAuctionModuleApproved` storage variable and use `ZORA_TRANSFER_HELPER.ZMM().isModuleApproved` instead.

Resolution: Resolved in commit 15812f5.

5.3 Medium severity

5.3.1 Approvals for non-standard ERC20 tokens do not work

Severity: *Medium Risk*

Context: ERC20SwapOperator.sol#L117, ERC20SwapOperator.sol#L148

Description: A well-known issue regarding interactions with non-standard ERC20 tokens, such as USDT, that do not implement the EIP20 interface correctly.

Since USDT does not return a boolean when `.approve` is called, but the ERC20 interface used defines that there should be a boolean returned, the compiler will check whether the `returndatasize()` is exactly 32 bytes long (one word size) and revert the call if this is not the case.

Recommendation: Use OpenZeppelin's SafeTransferLib `safeIncreaseAllowance` method, or alternatively, implement a custom `compactApprove` method in the `LibERC20Compat` library.

Resolution: Resolved in commit 93cc2ac.

5.4 Low severity

5.4.1 NFT token allowance is not removed when auction is canceled

Severity: *Low Risk*

Context: ListOnZoraProposal.sol#L148

Description: The new Zora auction contract does not transfer the NFT when the auction is created, but instead when the first bid is made. If a first bid is never made, the seller can cancel the auction, and the NFT will remain in their wallet (Party).

Therefore, the allowance given to the Zora auction contract on line 148 could be removed when `_settleZoraAuction` is executed to prevent from unexpected behavior, such as the losing the NFT in case of an unexpected behavior or a vulnerability in the external auction contract.

Recommendation: Set the allowance to 0 when an auction is canceled.

Resolution: Resolved in commit 29a5317.

5.4.2 The zoraAuctionModuleApproved boolean is never set to true

Severity: *Low Risk*

Context: ListOnZoraProposal.sol#L145-L147

Description: As mentioned in issue 5.2.1., the `zoraAuctionModuleApproved` storage variable is used in ListOnZoraProposal to check whether the approval on line 146 has already been made.

However, this variable is never set to true.

Recommendation: Set the `zoraAuctionModuleApproved` boolean flag to true after line 146.

Resolution: Resolved in commit 15812f5.

5.5 Informational

5.5.1 Users should not be able to swap a token for the same token

Severity: *Informational*

Context: ERC20SwapOperator.sol

Description: The `fromToken` and `toToken` parameters in ERC20SwapOperator should never be the same token.

Recommendation: Implement a check to revert the transaction if the `fromToken` is the same as the `toToken`.

Resolution: Resolved in commit b2b8b2e.

5.5.2 minReceivedAmount should never be 0

Severity: *Informational*

Context: ERC20SwapOperator.sol#L139-L142

Description: `minReceivedAmount` should not be set to 0 because it would mean that either 0 amount of the `fromToken` was spent or the entire amount/value of the `fromToken` was lost.

Recommendation: Consider whether receiving 0 amount of the `toToken` should be a valid scenario. If not, add a check to revert the transaction if `minReceivedAmount` is set to 0.

Resolution: Acknowledged.

5.5.3 compatTransferFrom may revert with an unexpected revert message if abi.decode fails

Severity: *Informational*

Context: LibERC20Compat.sol#L50

Description: The `r` variable in `LibERC20Compat.compatTransferFrom` can have a length specified by the token contract, for example 1 byte. In such cases, if the length is less than 32 bytes, the `abi.decode` operation on line 50 will revert with am unexpected revert message.

Recommendation: Implement a check on line 50 to ensure that the length of `r` is exactly 32 bytes.

Resolution: Acknowledged.

5.5.4 Redundant imports

Severity: *Informational*

Context: ListOnZoraProposal.sol#L7-L8

Description: The `LibRawResult` and `LibSafeERC721` libraries are no longer used in the ListOnZoraProposal contract.

Recommendation: Remove the mentioned imports.

Resolution: Resolved in commit 115c819.

5.5.5 Incorrect comment

Severity: *Informational*

Context: ZoraHelpers.sol#L7

Description: A comment in ZoraHelpers incorrectly states that the contract is used by ListOnOpenseaProposal, while it is actually used by the ListOnOpenseaProposalAdvanced contract.

Recommendation: Correct the aforementioned comment.

Resolution: Resolved in commit 088cc7f.

5.5.6 Typographical mistakes

Severity: *Informational*

Context: ListOnZoraProposal.sol#L43-L54

Description: The `tokenid` parameter in some of the events in ListOnZoraProposal does not follow the camelCase naming convention.

Recommendation: Consider using `tokenId` consistently instead of `tokenid` in ListOnZoraProposal.

Resolution: Acknowledged.