Recoverable Wallet Audit

OPENZEPPELIN | SEPTEMBER 17, 2019

Security Audits

Micah Zoltu asked us to review and audit the recoverable wallet smart contract code. We looked at the code and now publish our results.

The audited commit is `2207f5a7cc3b2607f7f97a769fbca7fe9235952b` and the recoverable-wallet.sol contract in the recoverable-wallet/contracts/source folder was the contract in scope.

We note that as of the audit start date the project was still in development, and the testing infrastructure in particular was incomplete. We have hence refrained from assessing it and other general aspects of the project's health. We do recommend to have at least a rich documentation and a robust testing infrastructure prior to launch.

Here are our assessment and recommendations, in order of importance.

Update: *Micah applied several fixes to address the issues found below. These fixes can be found in the* `audit-response` *branch with the final commit being* `916ec835d504cd3fa1b7688ada5273a24d4ae298`.

Critical Severity

None.

High Severity

Contract does not conform to the ERC1820 specification

The ERC1820 standard defines a universal registry where addresses can register the interfaces they support. The specification requires that compliant contracts have a `canImplementInterfaceForAddress(bytes32 interfaceHash, address addr)` function that returns the `ERC1820_ACCEPT_MAGIC` constant *if and only if* the contract implements the interface (`interfaceHash`) for a given address (`addr`).

Importantly, EIP1820 specifies:

> If [the contract] does not implement the `interfaceHash` for a given address (`addr`), [the `canImplementInterfaceForAddress` function] MUST NOT return `ERC1820_ACCEPT_MAGIC` .

On line 12 of recoverable-wallet.sol the `Erc777TokensRecipient` contract implements a `canImplementInterfaceForAddress` function which returns the `ERC1820_ACCEPT_MAGIC` constant on all inputs. This would indicate to an external caller that the contract implements all interfaces for all addresses.

Consider modifying this function so that it returns `ERC1820_ACCEPT_MAGIC` only when `interfaceHash` is `keccak256(abi.encodePacked("ERC777TokensRecipient")` and `addr` is `address(this)` .

Additionally, the ERC1820 standard defines the `canImplementInterfaceForAddress(bytes32 interfaceHash, address addr)` function with `bytes32 interfaceHash` as the first parameter and `address addr` as the second parameter. However, the implementation on line 12 of recoverable-wallet.sol has these parameters reversed. We recommend changing the order of these parameters so the `canImplementInterfaceForAddress` function complies with the ERC1820 specification.

Update: *The `canImplementInterfaceForAddress` function has been modified to conform with the ERC1820 standard. Note that the comments in lines 18 and 21 are still using `addr` , yet the variable has been renamed to `_implementer` .*

register a list of recovery addresses using the `add_recovery_address` function on line 86. Each recovery address is associated with a positive `_recovery_delay_in_days`, and a recovery address with a strictly lower `_recovery_delay_in_days` is said to have a "higher priority" than a recovery address with a strictly higher `_recovery_delay_in_days`. The owner is able to remove recovery addresses using the `remove_recovery_address` function on line 92.

At any time, a recover address may begin the recovery process by calling `start_recovery` (line 97 of recoverable-wallet.sol), thereby becoming the `active_recovery_address`. This puts the contract in the "in recovery" state, during which several of the contract's functions cannot be called, including the `remove_recovery_address` function. The owner can abort the recovery process by calling the `cancel_recovery` function, thereby putting the contract back into the "out of recovery" state.

If a recovery address becomes compromised, the compromised address may call the `start_recovery` function, during which time the owner cannot remove the compromised address via the `remove_recovery_address` function due to the `only_outside_recovery` modifier. To remove the compromised address, the owner must first call the `cancel_recovery` function before calling `remove_recovery_address(<compromised_address>)`.

However, once the owner has called the `cancel_recovery()` function, there exists a race condition during which the compromised address attempts to call `start_recovery()` again before the owner can call `remove_recovery_address(<compromised_address>)`. If successful, a compromised recovery address can keep the contract in a recovery state, thereby preventing any calls to any function with the `only_outside_recovery` modifier—including the `deploy` and `execute` functions.

If compromised recovery addresses are within the scope of the threat model, consider adding a function with the `only_owner` and `only_during_recovery` modifiers that cancels the recovery process and removes the compromised address in a single function call.

Update: *A new function (`cancelRecoveryAndRemoveRecoveryAddress`), has been added which allows the owner to cancel the recovery process and remove the offending recovery*

The `RecoverableWallet` contract implements an ownership transfer system whereby the owner may offer ownership of the `Ownable` contract to an address referred to as the `pending_owner`. This is achieved by calling the `start_ownership_transfer` function on line 38. Afterwards, the `pending_owner` may accept ownership by calling the `accept_ownership` function on line 50. If the current owner calls the `cancel_ownership_transfer` function *before* the `pending_owner` calls the `accept_ownership` function, then the offer is rescinded and the `pending_onwer` may no longer claim ownership.

After the `start_ownership_transfer` function has been called by the owner, the owner may wish to call the `cancel_ownership_transfer` to prevent ownership transfer. After the owner has broadcasted a transaction calling the `cancel_ownership_transfer` function—but before that transaction is mined—the `pending_owner` may call the `accept_ownership` function using a transaction with a higher gas price, thereby accepting ownership before the transaction to rescind the offer has been mined.

Whether or not this is a concern depends upon the intention of the owner when calling `cancel_ownership_transfer`. If the intention is to cancel the transfer of ownership to a non-responsive address, then this poses no threat. If the intention, however, is to prevent the transfer of ownership to a `pending_owner` that is no longer trusted, then this may pose a threat.

Update: *This is intentional functionality and a comment has been added to the code to indicate this.*

Recovery cancellation can be front-run

Similar to the issue **"Ownership transfer cancellation can be front-run"**, cancellation of the recovery process can also be front-run. After the `start_recovery` function has been called by a recovery address, and after the `active_recovery_end_time` has passed, the owner may wish to call the `cancel_recovery` function to prevent finalization of recovery before someone calls the `finish_recovery` function. After the owner has broadcasted a transaction calling the

Whether or not this is a concern depends upon the intention of the owner when calling `cancel_recovery`. If the intention is to cancel the transfer of ownership to a non-responsive recovery address, then this poses no threat. If the intention, however, is to prevent the transfer of ownership to an `active_recovery_address` that is no longer trusted, then this may pose a threat.

Update: *This is intentional functionality and a comment has been added to the code to indicate this.*

Low Severity

uint256/uint16 type mismatch

There is a parameter type mismatch in the `recovery_address_added` event. The `recovery_delay_in_days` parameter in the `recovery_address_added` event (line 59) is of type `uint256`. This event is emitted on line 89 with its second parameter being passed a `uint16`.

Consider using `uint256` rather than `uint16` everywhere.

Update: *The type of the `recovery_delay_in_days` parameter has been changed to `uint16` to resolve the type mismatch.*

Mixed usage of the recovery and ownership transfer systems

On line 122 the `finish_recovery` function sets `pending_owner = active_recovery_address;`. This requires the recovery address to then call `accept_ownership` in order to become the new owner. While this works fine, it requires two on-chain transactions and may result in an `ownership_transfer_finished` event being emitted without a corresponding `ownership_transfer_started` event preceding it.

If this is undesirable behavior, consider setting the `owner` instead of the `pending_owner` on line 122, and then modifying the `recovery_finished` event accordingly. This would require one fewer on-chain transaction to change owners via the recovery process, and it would keep the recovery process and the ownership transfer system independent of one another.

Obscure error message provided upon contract creation failure

The `deploy` function attempts to create a contract based on user-provided parameters and checks whether contract creation was successful by checking whether the output of `create2` is a non-zero address. If the wallet contract doesn't have enough balance to fund the newly created contract with the amount specified in the `_value` parameter, the transaction will revert with the obscure "Contract creation failed." error message issued by the non-zero address check.

Consider splitting this into two checks. First, we suggest adding a `require` statement to the beginning of the `deploy` function that checks whether the `RecoverableWallet` contract has a balance of at least `_value` ETH, and returns a meaningful error message otherwise. Second, rather than checking that the output of `create2` is a non-zero address, we recommend the more stringent requirement that code actually exists at the address returned by the call to `create2` using `extcodesize`.

Update: *The `deploy` function has been modified to output more detailed messages in the event of a failed contract deployment. The `extcodesize` option was not used because "a valid `create2` call may not result in any code actually being deployed at the destination address (e.g., `self-destruct` in constructor or deployment code returning an empty bytecode array)".*

Unnecessary setting of the `active_recovery_end_time` variable

The variable `active_recovery_end_time` is consumed only in the `finish_recovery` function (on line 120), which has the `only_during_recovery` modifier. The `only_during_recovery` modifier passes only when `active_recovery_address != address(0)`. This occurs only after `start_recovery` has been called and before either `cancel_recovery` or `finish_recovery` has been called.

Since `start_recovery` sets `active_recovery_end_time`, there is no need to set `active_recovery_end_time = uint256(-1)` on line 67. Consider leaving the `active_recovery_end_time` variable uninitialized on line 67.

For the same reason, it is unnecessary to set `active_recovery_end_time = uint256(-1);` in the `reset_recovery` function on line 146. Consider removing the code at line 146.

Unnecessary call to reset_recovery() in constructor

Whether or not the recommendations from the issue **"Unnecessary setting of the active_recovery_end_time variable"** are taken into consideration, the call to `reset_recovery` on line 80 is unnecessary since `active_recovery_address` and `active_recovery_end_time` will be set correctly when `start_recovery` is called. Consider removing this function call.

Update: *The call to `reset_recovery` has been removed from the constructor.*

Duplicate code in the finish_recovery() function

Line 119 duplicates the code in the `only_during_recovery` modifier, which is already being applied to the `finish_recovery` function. Consider removing this superfluous line.

Update: *The duplicate code has been removed from the `finish_recovery` function.*

False comment in the start_recovery() function

The comment on line 103 states:

> NOTE: the recovery address cannot change during recovery, so we can rely on this being != 0

The second half of the statement—that "we can rely on this being != 0" is true. However the first part of the statement—that "the recovery address cannot change during recovery" is false. The function that contains the comment (the `start_recovery` function) is the means by which a recovery address can change during recovery. In particular, a recovery address can change during recovery when a recovery address with a higher priority than the current `active_recovery_address` calls the `start_recovery` function.

Update: *The comment has be updated to correctly reflect contract behavior.*

Ownership transfer can be started while another is underway

The `start_ownership_transfer` function (line 38) in the `Ownable` contract can be called when the `pending_owner` is NOT `address(0)` (that is, when an ownership transfer is already underway). This can result in two or more `ownership_transfer_started` events

If this is not intended functionality, consider adding a `require(pending_owner == address(0))` to the `start_ownership_transfer` function to prevent this.

Update: *The `start_ownership_transfer` function (since renamed to `startOwnershipTransfer`) has been modified to first call the cancelOwnershipTransfer function before beginning a new ownership transfer. This ensures that the correct sequence of events is emitted.*

Ownership transfer can be cancelled before one is underway

The `cancel_ownership_transfer` function (line 44) in the `Ownable` contract can be called when the `pending_owner` is `address(0)` (that is, when an ownership transfer is not already underway). This can result in an `ownership_transfer_cancelled` event being emitted without a corresponding `ownership_transfer_started` event having been emitted.

If this is not intended functionality, consider adding a `require(pending_owner != address(0))` to the `cancel_ownership_transfer` function to prevent this.

Update: *A require statement has been included that prevents the function from being called when no ownership transfer is underway.*

Overloaded functionality of the start_recovery() function

The `start_recovery` function provides two functionalities. One is to allow a recovery address to initiate the recovery process when no recovery is underway. The other is to allow a recovery address with higher priority to usurp the current `active_recovery_address` when a recovery is already underway. These differing intentions both leverage the same function and both emit the same events.

Consider disentangling these two functionalities by breaking the existing `start_recovery` function into two separate functions: a `startRecovery` function with the `only_outside_recovery` modifier that provides the first functionality, and a `usurpRecovery` function with the `only_during_recovery` modifier that provides the second functionality.

emitted when escaping. This ensures that the correct events are emitted in the event

Notes & Additional Information

- The `recovery_cancelled` event (line 62) accepts no parameters. This may make `recovery_cancelled` events harder to track or interpret. Consider adding a parameter to this event.

  Update: *A parameter has been added to this event.*
- The `start_ownership_transfer` function can be called with the current `owner` passed as the `_pending_owner`. Similarly, the current `owner` can be added as a recovery address and so also may become the `pending_owner` by way of completing the recovery process. If this is undesirable behavior, consider adding require statements in the `start_ownership_transfer` and `start_recovery` functions to prevent this.

  Update: *This is intended behavior.*
- For all variables that store time values, consider changing their variable names to include the units of time they encode. For example, consider naming `_proposed_recovery_delay` to `_proposedRecoveryDelayDays`. Additionally, to make the code less error-prone during future changes, consider using only seconds throughout the code and using days only in the function interface.

  Update: *Variables in units of days had their names changed to reflect this, but* `activeRecoveryEndTime` *is in units of seconds and its name still does not reflect it.*
- On line 3 and line 9, consider renaming `Erc1820Registry` to `ERC1820Registry`.
- Following the Solidity Style Guide can improve code readability. For example, consider naming events using the CapWords style, naming functions and modifiers with the mixedCase style, keeping to a maximum line length of 79 (or 99) characters, etc.

  Update: *Several changes were made to bring the code closer inline with the Solidity style guide.*
- The code `active_recovery_address != address(0)` is used in several places. Consider moving it to an internal view function, `_inRecovery()`, and referencing it on line 70, line 75, line 101, and line 119. If applying the suggestion from the issue **"Duplicate code in the finish_recovery() function"** and also splitting the current `start_recovery` function into two functions, then this recommendation will not provide any benefit.

- ~~Update: Natspec documentation was added throughout the code.~~
- Consider adding a comment above the `exists` function in the `RecoverableWalletFactory` contract to explain its purpose.
  Update: *A comment was added above the `exists` function explaining its purpose.*
- Two or more owners of recovery addresses with the same priority may find themselves in a race to be the first to call `start_recovery`.
  Update: *This is intended functionality.*
- We observed that it is not easy for the owner to unilaterally relinquish ownership of the wallet. The owner may *offer* ownership to another address, but that other address must actively accept the offer in order for ownership to transfer. It is possible, however, for the owner to provably relinquish control of the wallet by (1) failing to add any recovery addresses and (2) transferring ownership to a contract that calls `accept_ownership()` but provides no other functionality.
  Update: *This is intended functionality.*
- The function `add_recovery_address` will accept `address(0)` as a `_new_recovery_address`. This can result in `recovery_delays[address(0)] > 0` and a `recovery_address_added` event being emitted with `address(0)`. This is benign, but one may consider adding a `require` statement to `add_recovery_address` to prevent this.
  Update: *A require statement has been added to prevent passing address(0) as the `_new_recovery_address`.*

Conclusion

No critical or high severity issues were found. Some changes were proposed to follow best practices, reduce the potential attack surface, and comply with the ERC1820 specification.

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the audited contracts. The above should not be construed as investment advice. For general information about smart contract security, check out our thoughts here.

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits