OpenZeppelin

InstaDApp Audit

OPENZEPPELIN SECURITY | SEPTEMBER 17, 2019

Security Audits

About InstaDApp

InstaDApp is an autonomous banking portal that runs on top of emerging blockchain-based financial protocols such as MakerDAO, Uniswap, or Compound. Its mission is to simplify everyday banking needs, such as taking loans, lending money, swapping tokens, and taking leveraged positions.

InstaDApp provides an interface on top of other DeFi protocols. Its target userbase is users who lack advanced technical or financial experience. It helps users manage CDPs, including providing the ability to wipe or add collateral to 3rd-party CDPs, keep track of CDP activity via email or Telegram, and save gas by bundling transactions. For example, a user with a CDP can withdraw Dai, buy ETH, and lock that ETH, all with a single transaction.

For each user, InstaDapp creates a unique smart contract – called a "UserWallet." This wallet contract allows the user to execute external code (using `delegatecall`) from logic contracts that have been whitelisted by InstaDapp admins. This way, new functionality can be added even after a wallet has been created.

To keep a record of existing wallets and whitelisted logic contracts, InstaDapp uses the InstaRegistry contract. The admins of this contract have the ability to "enable" or "disable" logic contracts (that is, the ability to add or remove logic contracts from the whitelist). Logic contracts may be whitelisted in one of two ways: They can be classified as standard logic contracts, or they can be classified as "static" logic contracts.

that the user can always exit InstaDapp even if the InstaRegistry admins become malicious or compromised.

The InstaRegistry contract also allows anyone to create new wallets (via either of its two `build` functions). It provides functionality for the owner of a UserWallet to transfer ownership to another account. It also gives the admins of the InstaRegistry contract the ability to change the InstaRegistry admin addresses and register other privileged roles.

Audit context and scope

InstaDapp is a live project on Ethereum's mainnet. The code we audited is the code located at the address `0x498b3bfabe9f73db90d252bcd4fa9548cd0fd981`, as verified by Etherscan. This consists of the two top-level contracts: `UserWallet` and `InstaRegistry`. We audited these two contracts and the contracts from which they inherit.

The `InstaRegistry` contract is used to deploy new `UserWallet` instances, track the ownership of `UserWallet` instances that it deploys, and approve/disapprove logic proxy contracts that can be used by the `UserWallet` instances that it has deployed. It is not intended to hold any valuable assets.

The `UserWallet` contracts are the only contracts intended to hold any valuable assets. `UserWallet` contracts are built using a proxy pattern that allows their functionality to be extended using proxy logic contracts that have been approved by the `InstaRegistry` contract.

We did not audit any logic proxy contracts as part of this audit. Since proxy logic contracts can manipulate the state of `UserWallet` contracts when called by the owners of the `UserWallet` (including transferring valuable assets out of the wallet), we recommend that any proxy logic contracts also be audited before they are invoked by users.

The code verified by Etherscan is the same as the code found in the `UserWallet.sol` and `InstaRegistry.sol` files at commit `4863c0c4156af7ded9cdb38b66e5f5e527c4a6d0`.

- RegistryInterface (interface)
- AddressRecord
- UserAuth
- UserNote
- UserWallet
- AddressRegistry
- LogicRegistry
- WalletRegistry
- InstaRegistry

Out of scope

The following contracts were explicitly not in scope:

- Any logic proxy contracts — static or otherwise. This includes any logic contracts currently being used on Ethereum's mainnet and any logic contracts planned for future use. Any logic proxy contracts that will be used with InstaDApp should undergo a separate audit before they are enabled by the InstaDApp admins or used by InstaDApp users.
- Any contracts found in the InstaDApp Github repository, other than the ones we explicitly listed above.
- The Uniswap, Compound, MakerDAO, or other DeFi project code.
- Any old versions of the InstaDApp project.

Findings

Here we share our findings and recommendations.

A quick note for clarification. The WalletRegistry contract has two separate functions that share the same name: the `build()` function and the `build(address _owner)` function. These two functions do not have identical parameter types, so there is no risk of function signature collision. However, it would be confusing to readers of this report if we were to refer to "the `build` function" ambiguously. To avoid such confusion, we will disambiguate by always including the input latter [illegible] function.

Critical

None. 🙂

High

None. 🙂

Medium

Transfer of wallet ownership can be DoSed — Method 1

An active attacker can prevent the owner of a wallet from transferring wallet ownership. The attack works as follows:

The attacker can front-run all transactions calling `setOwner(X)` with a transaction that calls `build(X)`.

This makes `proxies[X]` not equal to `UserWallet(0)` when the `setOwner(X)` transactions is processed, causing the require statement on line 139 to revert. The result is that the attacker prevents the transfer of wallet ownership.

Consider restricting access to the `build(address _owner)` function so that it may be called only from the InstaRegistry address and (optionally) a set of whitelisted third-party addresses.

Transfer of wallet ownership can be DoSed — Method 2

Even if the calls to `build(address _owner)` are restricted (as mentioned in the issue "Transfer of wallet ownership can be DoSed — Method 1"), an attacker may prevent owners of wallets from transferring ownership. This second method of preventing ownership transfers works as follows.

The attacker can front-run each transaction of the form `setOwner(X)` with a single transaction that performs both of the following functions atomically:

(These two actions can be performed atomically in a single transaction by creating an exploit contract that performs both functions in a single function call.)

As before, this results in `proxies[X]` not being equal to `UserWallet(0)` when the `setOwner(X)` transaction is processed — causing a `revert` at line 139 of InstaRegistry.sol, and thus preventing the honest user from transferring ownership.

Consider implementing an "active ownership acceptance" scheme that requires newly proposed owners to accept ownership actively. For example, ownership transfer would be a two-step process where:

1. The current owner sets a "pending owner."
2. The "pending owner" calls a function (ie: `acceptOwnershipTransfer()`) in order to claim ownership.

For an audited example of this pattern, see Micah Zoltu's recoverable wallet contract.

This active ownership acceptance scheme would prevent this method of DoS'ing ownership transfer, as well as prevent accidental burning of UserWallets by inexperienced users transferring their wallet ownership to an inactive address (or to a contract that is not equipped to interact with UserWallets).

Admin/Owner roles may be burned by mistake

If the admin or owner of the InstaRegistry contract mistakenly calls the `setAddress` function without explicitly entering an input for the `_userAddress` parameter (a common user error when interacting with contracts), some front-ends may interpret the missing `address` parameter as the zero address having been passed as the `_userAddress` parameter. This would result in the admin or owner mistakenly burning their admin/owner rights.

Consider adding a `require` statement in the `setAddress` function that would revert if the `_userAddress` parameter is the zero address. This would protect against the accidental burning of admin/owner rights. (Intentional burning of admin/owner rights would still be possible.)

`note` modifier and the second one is in the `excecute` function.

While this does not pose a security risk *per se*, these are a critical part of the system and should be better documented. For assembly blocks, we highly recommend that *every line* of assembly be explained using code comments. This helps improve the readability and understandability of these critical sections of code.

Also note that the use of assembly discards several important safety features of Solidity, which can render the code less safe and more error-prone. Consider implementing thorough tests to cover all potential use cases of these functions to ensure they behave as expected.

Logging a memory pointer instead of the actual data

The `UserNote` contract implements the `note` modifier, which logs information when the `execute` function is called. Inside of the modifier, there is an `assembly` block which stores information from the transaction in the local variables `foo` and `bar`, which are then emitted during the `LogNote` event.

The `bar` variable is set via `bar := calldataload(36)`, which contains the *position* of data in memory instead of the actual data itself. If the intention is to log the actual *data* and not just the *position* of the data, then consider changing this block of code and writing tests to ensure it is behaving as expected. Also, consider updating the comments/documentation for this assembly block to explain its purpose.

Various NatSpec issues

There are various issues related to the NatSpec documentation in the code. We list several of them here as a single issue. We consider the NatSpec documentation to be part of the contract's public API, so NatSpec-related issues are assigned a higher severity than other code-comment related issues.

- The @notice and @dev tags are empty on the `AddressRegistry` contract.
- The @notice tag is empty on the `LogicRegistry` contract.
- The @dev tag repeats the @title tag on the `LogicRegistry` contract.

- The @dev tag of the [illegible] function is empty / does not explain what the `logicStatic` function does.
- The @dev tag of the user wallet `constructor` function is incorrect / misleading.
- The @dev tag of the `record` function contains a misspelling.
- The NatSpec for the `isAuth` modifier should have a @return tag.
- The NatSpec for the `execute` function should have a @return tag.

Consider removing empty tags, adding @return tags where needed, using @dev tags to describe the intended use of contracts/functions, and otherwise addressing the above-listed issues. See the Solidity Documentation for more information.

Low

Wallet creation transactions can be made to fail — Method 1

An attacker can cause all honest users' wallet-creation transactions to fail. The honest user will still end up with a functioning wallet that they control, but their wallet software (i.e., MetaMask) will display an error saying that their wallet-creation transaction was unsuccessful (reverted). The attack works as follows:

An attacker can front-run all calls to `build()` or `build(address _owner)` with their own call to `build(address _owner)` with the `_owner` parameter set to the honest user's address. This results in the honest user's transaction being reverted at line 127 of InstaRegistry.sol.

The honest user will still end up with a functioning wallet that they control because the attacker's transaction will have created it for them. However, the failed transaction message could be confusing for unsophisticated users, and could also increase the complexity for front-end developers who would have to handle that edge case.

Consider restricting access to the `build(address _owner)` function so that it may be called only from the InstaRegistry address and (optionally) a set of whitelisted third-party addresses.

Wallet creation transactions can be made to fail — Method 2

fail. As before, the honest user will still end up with a functioning wallet that they control, but their wallet software (i.e., MetaMask) will display an error saying that their wallet-creation transaction was unsuccessful (reverted). This second method of causing wallet creation transactions to fail works as follows.

An attacker can front-run all calls to `build()` or `build(address _owner)` with a single transaction that performs both of the following actions:

1. Calls `build()` to create a new UserWallet instance that the attacker controls.
2. Transfers ownership of the attacker's new UserWallet instance to the honest user's address by calling `setOwner(address nextOwner)` on the attacker's new UserWallet instance and passing the honest user's address as the `nextOwner` parameter.

(These two actions can be performed atomically in a single transaction by creating an exploit contract that performs both functions in a single function call.)

The honest user will still end up with a functioning wallet that they control because the attacker's transaction will have created one and transferred ownership to the honest user. However, the failed transaction message could be confusing for unsophisticated users, and could also increase the complexity for front-end developers who would have to handle that edge case.

To prevent this, consider implementing the "active ownership acceptance" scheme recommended in the "Transfer of wallet ownership can be DoSed — Method 2" issue.

UserWallet owners may mistakenly burn ownership

A common mistake for users to make is calling a function while forgetting to pass an explicit value for one or more of the function's parameters. When this occurs, the missing input may be interpreted as the default, uninitialized value for its given type. For example, failure to provide a `uint256` parameter when calling a function through Remix results in the `uint256` parameter being interpreted as `0`.

If a user makes this kind of mistake when calling the `setOwner` function of their UserWallet, they may mistakenly burn ownership of their wallet by assigning ownership to the zero address.

the `nextOwner` parameter is *not* the zero address.

Alternatively, consider implementing the “active ownership acceptance” scheme recommended in the “Transfer of wallet ownership can be DoSed — Method 2” issue.

A third alternative, which would not require redeploying any of the live contracts, would be to call `build(address _owner)` with the zero address passed as the `_owner` parameter. This will create a new UserWallet for the zero address, which will make `proxies[0] != UserWallet(0)`, which will prevent any future user from transferring ownership of their wallet to the zero address mistakenly.

Privileged roles can have only one member

The `AddressRegistry` contract has a mapping named `registry` that maps (hashes of) a role to the address that has that role. Importantly, this means that each role can have only one address that has that role.

Two important roles are the `admin` and `owner` roles, which are the only two roles who have access to functions with the `isAdmin` modifier. The `setAddress` function has the `isAdmin` modifier and can be used to create new roles. It can also be used to change the address to which an existing role maps. Any revocation of roles must be done via the `setAddress` function — for example, by setting the address of a role to the zero address.

These access controls work as intended but are limiting in the sense that only one address can have each role. The `admin` and `owner` roles have identical power in the existing contracts. It seems as though having these two different roles is a way to hack around the limitation that each role type can have only one address associated with it.

Consider using a more flexible role-management pattern that would allow more than one address of each role type, such as OpenZeppelin’s Role Library. You can see an example of how to use it in the MinterRole contract.

This approach would allow for more flexibility when you want to have multiple addresses with the same role.

the image of `proxies` mapping.

This can happen if the owner of the UserWallet calls `setOwner` with the InstaRegistry address as the `nextOwner` parameter. Then, the next time anyone calls the `build()` or `build(address _owner)` functions again, the original UserWallet will be overwritten in the `proxies` mapping at line 129 of InstaRegistry.sol, and thus no longer appear in the image of the `proxies` mapping at all.

If this is considered undesirable behavior, consider requiring that the `record` function cannot have its `_nextOwner` parameter be equal to the InstaRegistry address. Alternatively, consider implementing the "active ownership acceptance" scheme" recommended in the "Transfer of wallet ownership can be DoSed — Method 2" issue.

Hardcoded gas remainder

The `execute` function of the `UserWallet` contract uses a `delegatecall` to a user-inputted logic proxy contract. This appears to be a clone of MakerDAO's delagate call pattern.

In that `delegatecall`, the gas parameter is set to `sub(gas, 5000)`. The reason for this value is not documented. Presumably, this is to ensure there is enough gas (*i.e.* at least 5000) remaining to finish executing the rest of the assembly block. This could potentially be a problematic approach because the gas cost of opcodes can change (see EIP 1884 as an example), which could result in insufficient gas issues in the future.

Consider setting the value to `sub(gas, minRemainingGas)`, where `minRemainingGas` is either a user-inputted parameter or a user-settable global variable.

Function/Modifier/Variable naming could be improved for readability

Several variables, functions, and parameters have names that do not represent their purpose clearly. This is most severe in the UserNote contract. To improve readability, consider renaming them to reflect their purposes more clearly. Our suggestions for renaming are:

In AddressRecord:

- `registry` to `registryAddress`,

- `LogSetOwner` to `OwnerChanged`,
- `auth` to `onlyAuthorized`,
- `setOwner` to `changeOwner`,
- `isAuth` to `isAuthorized`.

In UserNote:

- `foo` to `targetAddress`,
- `bar` to `dataInput`,
- `guy` to `caller`,
- `wad` to `value`,
- `fax` to `data`,
- `note` to `receipt`.

In AddressRegistry:

- `LogSetAddress` to `UpdatedRole`,
- `registry` to `rolesRegistry`,
- `getAddress()` to `getRoleAddress()`,
- `setAddress()` to `setRoleAddress()`,
- `isAdmin` to `hasPermission`,
- `_name` to `_roleName`,
- `_userAddress` to `_actorAddress`.

In LogicRegistry:

- `logicProxiesStatic` to `staticLogicProxies`,
- `logic` to `hasLogic`,
- `logicStatic` to `hasStaticLogic`.

In WalletRegistry:

- `Created` to `WalletCreated`,
- `LogRecord` to `TransferedOwnership`,
- `proxies` to `walletRegistry`,

Some functions that return values do so explicitly. For example, the `logic` and `logicStatic` functions both explicitly return their return values.

However, other functions do not return their return values explicitly. For example, the `build(address _owner)` and `record` functions do not use explicit return statements.

Consider using explicit returns instead of implicit returns on all functions with a return value. This helps prevent regressions that may occur when code changes over time.

Function visibilities could be more restrictive

Many functions are given `public` visibility when they could be restricted to `external`. We recommend restricting function visibility as much as possible because it saves gas and simplifies security analysis. Consider setting the visibility of the following functions to `external`:

- The `setAddress` function of the `AddressRegistry` contract.

- The `setOwner` function of the `UserAuth` contract.
- Every function in the `LogicRegistry` contract.
- The `build` function of the `WalletRegistry` contract.
- The `record` of the `WalletRegistry` contract.

Notes

Benign reentrancy in setOwner

The violation of the checks-effects-interactions pattern in the `setOwner` function of `UserWallet.sol` could allow for benign reentrancy if the `registry` address were ever the address of a malicious contract.

False comment

The comment on line 124 of InstaRegistry.sol suggests that the `build(address _owner)` function throws if `msg.sender` is not a proxy contract created by the InstaRegistry contract. However, this is not true. It appears as though this comment was intended for the `record` function.

Consider moving this comment immediately above the `record` function.

Implicit variable visibility

The `registry` mapping variable in the `AddressRegistry` contract is implicitly public. Consider making the visibility explicit to improve readability.

Inconsistent parameter indicators

In most cases, function and modifier parameter names are indicated with an underscore. But in `logicAuth`, `setOwner`, and `isAuth` they are not. Consider making this consistent for readability.

Lack of native ETH withdrawal method

The `UserWallet` contract relies upon the logic proxy contracts to perform ETH withdraws. Consider providing a native ETH-withdrawal function in the UserWallet contract. Otherwise, consider adding an ETH-withdrawal function via a static logic proxy.

Update: *The InstaDApp team has informed us that the deployed `InstaRegistry` contract has approved an ETH-withdrawal function as a static logic proxy. We did not audit this ETH-withdrawal function because it was out of scope.*

Poor test coverage

The test coverage for the contracts is quite sparse. Consider writing more tests for better coverage.

Avoid the use of type aliases

The `logicStatic` function in the `LogicRegistry` contract is unnecessary, as it duplicates the public `logicProxiesStatic` getter.

Consider removing the `logicStatic` function and using the public `logicProxiesStatic` mapping getter.

Return variable declared in function definition

To favor explicitness and readability, declaring the return variables during the body of the function instead of doing it in the function definition is preferred.

Consider declaring the output `response` inside the function body, and explicitly return the variable once the code flow finishes.

Web3 version

`web3` is currently included as a dependency in the `package.json` file as `"web3": "^1.0.0-beta.37"`. Developers should be aware of issue #2266 introduced in web3 v1.0.0-beta.38, where custom providers are not accepted, which could introduce unexpected bugs in the project.

Consider pinning the `web3` dependency to a fixed version (*e.g.* `"web3": "1.0.0-beta.37"`) or update to the latest version where this issue is solved.

Conclusion

No critical or high severity issues were found. Some changes were proposed to follow best practices, improve extensibility, and reduce potential attack surface.

Summary

If you are interested in a non-technical overview of this audit, we present a summary of the system as it relates to the audit as well as a couple of interesting findings in our summary article.

Related Posts

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

OpenZeppelin