Zellic

Tokenbound

Smart Contract Security Assessment

November 13, 2023

Prepared for:

Jayden Windle

Future Primitive

Prepared by:

Kuilin Li and Ulrich Myhre

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Future Primitive from August 29th to September 8th, 2023. During this engagement, Zellic reviewed Tokenbound's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is it possible to do unauthorized executions using an account?
- Is it possible to do an unauthorized signing on behalf of an account?
- Is it possible to obtain write access to the storage of an account, without going through the sandbox?
- Can there be arbitrary writes to storage despite the sandbox?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

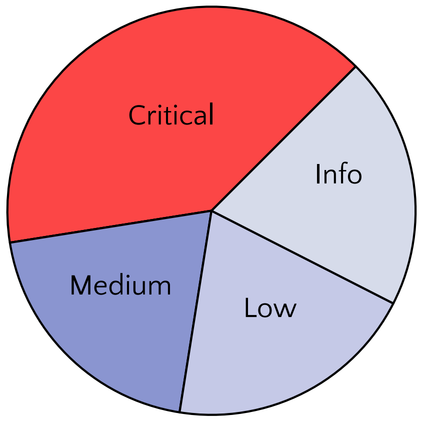
1.3 Results

During our assessment on the scoped Tokenbound contracts, we discovered five findings. Two critical issues were found. One was of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Future Primitive's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	0
Medium	1
Low	1
Informational	1



2 Introduction

2.1 About Tokenbound

The ERC-6551 registry contract is responsible for assigning account addresses to all non-fungible tokens (NFTs). It is deployed to the same address on each chain using Nick's Factory. The registry is essentially an opinionated create2 factory, where every contract deployed is an ERC-1167 proxy with appended constant data. The account address for an NFT is derived from the NFT's chain ID, contract address, and token ID as well as a salt and an implementation address.

Tokenbound provides an opinionated ERC-6551 account implementation that supports ERC-4337 execution, upgradability (both ERC-1967 and diamond-style overrides), delegation, and multiple execution models.

A core part of the Tokenbound implementation is ensuring accounts can be deployed and used across multiple chains. This means that the contract for an account should be deployable at the same address on every chain and that the holder of the token that owns the account should be able to use the account on any chain. Currently this is supported on Optimism (via address aliasing). Future bridges will be supported by whitelisting on the account guardian.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3 Scope

The engagement involved a review of the following targets:

Tokenbound Contracts

Repositories	https://github.com/tokenbound/contracts https://github.com/erc6551/reference
Versions	contracts: `48155498e2a291b6890a6e2ea5ccdd40046bab12` reference: `fe246b7b603241fc5c073b74ed18d4c46d88eed7`
Programs	AccountGuardian.sol AccountProxy.sol AccountV3.sol AccountV3Upgradable.sol BaseExecutor.sol BatchExecutor.sol ERC4337Account.sol ERC6551Account.sol ERC6551Executor.sol Lockable.sol NestedAccountExecutor.sol Overridable.sol Permissioned.sol SandboxExecutor.sol Signatory.sol TokenboundExecutor.sol ERC6551Registry.sol ERC6551AccountLib.sol ERC6551BytecodeLib.sol
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Kuilin Li, Engineer
kuilin@zellic.io

Ulrich Myhre, Engineer
unblvr@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

August 29, 2023	Kick-off call
August 29, 2023	Start of primary review period
September 8, 2023	End of primary review period
September 11, 2023	Draft report delivered
November 13, 2023	Final report delivered

3 Detailed Findings

3.1 AccountProxy can be reinitialized

- **Target**: AccountProxy
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Critical
- **Impact**: Critical

Description

The `initialize` external function of AccountProxy is used to upgrade the account to the initial implementation after deployment:

```
function initialize(address implementation) external {
    if (!guardian.isTrustedImplementation(implementation))
        revert InvalidImplementation();
    ERC1967Upgrade._upgradeTo(implementation);
}
```

However, there is no access control or reinitialization guard on this function.

Impact

Anyone can upgrade any deployed AccountProxy to any trusted implementation.

Recommendations

Add a reinitialization guard to this contract.

Remediation

This issue has been acknowledged by Future Primitive, and a fix was implemented in commit 69af6f0e.

3.2 NestedAccountExecutor can bypass locks and state changes

- **Target**: NestedAccountExecutor
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The abstract contract NestedAccountExecutor is used to allow the owner of an NFT whose Tokenbound Account (TBA) owns an NFT whose TBA owns an NFT and so forth, any levels deep, to act upon any of the NFTs as an executor without having to expensively nest calls.

The `executeNested` function only checks the ownership of the intermediary tokens when checking the proof:

```
function executeNested(
    address to,
    uint256 value,
    bytes calldata data,
    uint256 operation,
    ERC6551AccountInfo[] calldata proof
) external payable returns (bytes memory) {
    // ...

    for (uint256 i = 0; i < length; i++) {
        accountInfo = proof[i];
        address tokenContract = accountInfo.tokenContract;
        uint256 tokenId = accountInfo.tokenId;

        address next = ERC6551AccountLib.computeAddress(
            erc6551Registry, __self, block.chainid, tokenContract,
    tokenId, accountInfo.salt
        );

        if (tokenContract.code.length == 0) revert InvalidAccountProof();
        try IERC721(tokenContract).ownerOf(tokenId)
    returns (address _owner) {
            if (_owner ≠ current) revert InvalidAccountProof();
            current = next;
        } catch {
            revert InvalidAccountProof();
        }
    }

// ...
}
```

The `Lockable` feature ensures that when a contract is locked, the assets of its TBA cannot change. The `state` feature ensures that if something is executed by a TBA, the state will change.

However, if NestedAccountExecutor is used, it only checks `Lockable` and mutates `state` for the TBA that is currently executing the operation, in the `_beforeExecute()` later in the function. The intermediary NFTs can be used in the proof even if they are locked and even without mutating `state` for the intermediary TBAs.

Impact

A malicious seller on a decentralized marketplace can list a locked and state-committed NFT with a TBA that owns a valuable NFT, and then still, before finalizing a sale, execute anything on behalf of the valuable NFT's TBA, bypassing the lock and without mutating the committed state.

Recommendations

Add functionality to check if the intermediary TBA is locked, or specify clearly that marketplaces should ensure all TBAs in the tree are locked and have their state committed.

Remediation

This issue has been acknowledged by Future Primitive, and a fix was implemented in commit 70c768e6.

3.3 The `initData` parameter in `createAccount` can be modified by front-runners

- **Target**: ERC6551Registry
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The `createAccount` function in ERC6551Registry allows anyone to deploy the TBA for an NFT.

However, the `initData` parameter allows the deployer, who can be anyone, to specify the initial call to the contract.

Since anyone can provide this parameter, a front-runner can arbitrarily modify the `initData` parameter and deploy a contract before a legitimate user has a chance to.

Impact

This parameter cannot be used safely, because anyone can front-run a transaction that includes a call to `createAccount` with data to substitute any other data in there.

The impact is limited in the intended use case because when an AccountProxy is the first parameter to `createAccount`, the implementation, the proxy is not yet initialized, so the only thing that can be called on it is `initialize`.

Currently, anyone can call that even after initialization, as described in Finding 3.1. However, even after that is fixed, as long as there are multiple trusted implementations, a malicious front-runner can front-run a user to get an unexpected implementation in the TBA. The user's transaction will still go through because `createAccount` returns the account as if the deployment had succeeded if the account already existed.

Recommendations

Only allow a constant `initData`, or include the `initData` as a part of the creation code so that different `initData` contents will produce different TBAs. Additionally, this function should revert if the account already exists.

Remediation

This issue has been acknowledged by Future Primitive.

3.4 Function `isERC6551Account`, without expected implementation parameter, can be forged

- **Target**: ERC6551AccountLib
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

Description

The `isERC6551Account` function in ERC6551AccountLib allows contracts to check if an address is an ERC-6551 account with a particular registry.

However, an attacker can create an ERC-6551 account with an attacker-controlled implementation that passes this check and then later self-destruct the implementation, making the check fail. Later, in another transaction, they can then reinstantiate the implementation if it was deployed with create2. Thus, they can arbitrarily forge the result of this call.

Impact

Currently, there is no impact since the two-argument form of this function is not used anywhere.

Recommendations

This form of the function should be removed in order to demonstrate to users of this library that it is unsafe to determine whether an account is an ERC-6551 account without an expected implementation in mind.

Remediation

This issue has been acknowledged by Future Primitive, and a fix was implemented in commit ba77e72f.

3.5 Token owner can forge `_rootTokenOwner`

- **Target**: AccountV3
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

The `_rootTokenOwner` function is used across the project to find the root owner, for authorization and authentication. More specifically, it is used in `Overridable` to key into the `overrides` to determine which root owner context to set and use overrides, in a similar way in `Permissioned`.

However, a malicious token owner can effectively arbitrarily set the root token owner to whoever they want by transferring an NFT in the chain to a victim. Ordinarily, they lose control over the NFT. However, if one of the tokens in the ownership chain is controlled by the attacker, such that the attacker can arbitrarily pull the NFT away from the victim afterwards, they can do this and not give away the NFT.

Impact

Currently, there is no impact, but only because both `Overridable` and `Permissioned` check if the sender is the root token owner and, only then, allow overrides and permissions to be set.

Recommendations

It is risky to use `_rootTokenOwner` for authentication. Consider an attacker-deployed NFT that returns whatever the attacker sets when `ownerOf` is called on it. If assets are transferred to that NFT's TBA, then the `_rootTokenOwner` of those assets and its children can return whatever the attacker wishes, even if it is a victim address that has never been authenticated during this whole process. Using it for authorization is okay since this exploit requires ownership of the token anyways, but using it for authentication is flawed.

A warning should be added to `_rootTokenOwner` to ensure it is never used for authentication.

Remediation

This issue has been acknowledged by Future Primitive, and a fix was implemented in commit b2cc0944.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Receive hooks are not static for override purposes

The receive hooks `onERC721Received`, `onERC1155Received`, and `onERC1155BatchReceived` should use `_handleOverride` rather than `_handleOverrideStatic` because these functions are not static, and overrides that implement receive hooks should be able to set state.

This issue has been acknowledged by Future Primitive, and a fix was implemented in commit a75a3123.

4.2 Overrides do not have `msg.sender`, so they cannot easily authenticate sender

When an override is found, it is called by the account like this:

```
(bool success, bytes memory result) =
    sandbox.call(abi.encodePacked(implementation, msg.data));
```

This means that the address of the sender, as seen by the override implementation, is going to be the account, which means the actual sender is not known. The override implementation cannot use `tx.origin` because it may be a relayed transaction and it has no way of inspecting the sender of intermediary stack frames. So, if it needs to authenticate the sender, it has to take in a signature or some other method of authentication, which is inconvenient.

This issue has been acknowledged by Future Primitive, and a fix was implemented in commit a971ebae.

5 Threat Model

This provides a comprehensive threat model that addresses various security concerns related to the protocol's safety. As time permitted, we examined each area of concern, and created a threat model that outlines potential threats and how the protocol defends itself against them.

Please note that not all threats might have been included in this model. The absence of a threat model in this section does not necessarily imply that the protocol has considered it.

5.1 Area of concern: Arbitrary write access

Rationale

Here we are trying to assess the contracts' resilience against bugs that enable arbitrary write access to an account's storage. It is crucial that storage access is tightly controlled, as an account should not be able to write to the storage of any other account. Account holders are allowed to execute arbitrary delegatecalls, but they should not obtain arbitrary storage access. The current mechanism to disallow access is to call into a dedicated sandbox contract, but only if the sender is the account itself. This sandbox is deployed to a known address and given low-level execution permissions on the account.

The pattern of limiting storage access through a sandbox is fairly new and novel, and it does not have much prior work to compare against. The sandbox contract Sandbox.yul is also completely implemented in assembly, which does not benefit from the same type of checks and validations that the compiler is able to do for code written in Solidity.

Analysis

The sandbox contract is fairly short and concise. It immediately reverts if the caller is not the owner, before any calldata is copied or any executions have taken place. There are test cases in place that try to write to the storage as the wrong address, and this is expected to fail.

Inside the account contracts, the glue for the sandbox code is found in LibSandbox.sol, which calculates the expected sandbox address. This address depends on the hash of the text string `org.tokenbound.sandbox` as salt and the hash of the bytecode for a given address. However, the bytecode is not read from the contract itself but assumed to be the address sandwiched between a static header and footer.

The code(s) used to generate the header and footer are located in CompileLibSandbox.s.sol and are the result of the aforementioned Sandbox.yul with a hardcoded storage slot equal to the owner address.

LibSandbox.sol also provides a deployment function for the same bytecode, and this functionality is used in Overridable.sol and LibExecutor.sol if, and only if, there is no contract already deployed at that address. Calculating this address could be a bit expensive, and it could be a good idea to keep a map of known-existing sandbox contracts to reduce this cost.

Overridable.sol will call into the implementation address using a sandbox, but only if an override is set for the current function selector (`msg.sig`). By default, this is set up for the payable `receive()` and `fallback()` functions.

LibExecutor.sol only uses the sandbox whenever the requested operation is `OP_DELEGATECALL` and restricts any other operation that is not `OP_CALL`, `OP_CREATE`, or `OP_CREATE2` by making the execution function revert. As the only way to write to storage is through `OP_DELEGATECALL`, it being ran through the sandbox should limit writes to the storage of the account contract itself.

The SandboxExecutor.sol contract glues the outgoing flow from the sandbox with the other contracts. The calls here require the caller to be the sandbox itself, or it will revert. Accepted inputs go to LibExecutor.sol. Exposed functions in SandboxExecutor.sol are limited to `sload` and `call/create/create2`, where the last three go to LibExecutor. This should adequately disallow arbitrary write access but allow storage reads from external contracts.

Conclusion

Given the critical nature of the sandbox, the documentation for and the amount/variety of test cases could be increased.

The attack surface of the sandbox feature is likely limited to the interface specified by `ISandboxExecutor`. This is because, without that interface and without the sandbox feature in the contract itself, an executor of the TBA can already deploy a similar or identical sandbox using `OP_CREATE2` and then use it to call arbitrary delegatecalls into other contracts using `OP_CALL`. So, the addition of the sandbox feature implemented in this way cannot increase the capabilities of an executor of the TBA, except for the functionalities that whitelist the address of the sandbox, namely those in SandboxExecutor.sol. The SandboxExecutor in turn whitelists the additional capabilities to `call/create/create2`, which an authorized executor is already capable of doing, and `sload`, which can only read the storage. Theoretically speaking, instead of reading the storage, off-chain code could instead obtain any storage slot via state trie inspection (`eth_` getStorageAt) and then submit a Merkle proof-of-inclusion for that particular storage slot. So, even the sload does not give more capabilities to the executor beyond saving them the gas of verifying such a proof.

5.2 Area of concern: Unauthorized execution/signing

Rationale

Bugs that allow unauthorized execution using an account are always critical, because they allow seemingly valid signers to execute arbitrary operations on behalf of the account, like transferring funds out of the account.

Any bug or vulnerability that allows unauthorized signing using an account are also very likely critical. Being able to sign on behalf of an account gives an attacker direct access to the execution interface by calling into the Entrypoint as the contract. Being able to replay, inject, reorder, or modify existing signatures can also lead to loss of funds, denial of service, or an overall bad user experience.

Analysis

The contracts define a wide variety of executors. Executor functions are external by default, but they call _isValidExecutor(address executor) to check if the caller is allowed to execute. The default, virtual function for this verification is accepting the ER C-4337 EntryPoint but also cross-chain execution from trusted chain bridges and the L1 account on OPStack chains. In addition to these callers, the token owner, the root token owner (iterated up the token ownership tree), and any permissioned address for the given account are whitelisted.

For nested accounts, an array of ERC-6551 account information has to be provided, and it has to specify an ownership path from the current account to its parent. Finally, the parent has to be a valid executor. The calculations required for this are very heavy, and very long ownership chains could prove to be too costly to verify before hitting a gas limitation. A cap on this length could be worth thinking about, especially since this ownership chain is traversed for every call into any override. As all the executors are verifying the caller, this is as safe as the whitelisting logic.

For whitelisting callers, the root token owner can use the Permissioned subcontract to call setPermissions to whitelist any other address. These whitelist capabilities are stored in the permissions mapping, keyed first by the root token owner (so that if the root token owner changes, the permissions change) and then by the permissioned caller. We note in Finding 3.5 that a root token owner can temporarily spoof the root token owner to any address, but for the purposes of these calls, this spoofing is currently not exploitable.

When it comes to unauthorized signing, EIP-6551 defines an interface that includes an `isValidSigner(address signer, bytes calldata context)` function, which returns a `bytes4` magic value `0x523e3260` if the given signer is valid. By default, the holder of the non-fungible token the account is bound to must be considered a valid signer. Accounts may implement additional authorization logic (e.g., for invalidating the holder as a signer or granting signing permissions to other nonholder accounts).

The contract AccountV3.sol has a default for `_isValidSigner`, which allows a token owner to be a valid signer. The same is true for the root owner of an account tree and any permissioned addresses for the given account. This is similar to the protections for execution. The signature check itself has two possible branches, where one of them uses OpenZeppelin's ECDSA contract to run the `ecrecover` functionality and verify that the signer is valid. There is also support for ERC-1271 smart contract signatures, which are checked by OpenZeppelin's SignatureChecker contract.

Conclusion

Although the contract features a wide variety of authorized executors, that variety is documented and highly intentional. In this audit, each type of authorization was considered from the perspective of an external attacker who never held authorization on the account.

6 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Tokenbound contracts, we discovered five findings. Two critical issues were found. One was of medium impact, one was of low impact, and the remaining finding was informational in nature. Future Primitive acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.