Provide

Counterfactual Audit Report

Monday, September 9, 2019

Summary

Connext engaged Provide to audit a subset of the smart contracts provided by the Counterfactual framework in advance of the mainnet release of Indra 2.0. This document provides a brief overview of the Counterfactual framework, the scope of the audit, issues discovered during the audit and recommendations for resolving such issues prior to deploying the contracts to the Ethereum mainnet.

Auditor

Kyle Thomas

Authenticity

The scope of the audit is limited to a subset of the Solidity contracts which can be found in this GitHub repository under `packages/cf-adjudicator-contracts` and `packages/cf-funding-protocol-contracts` at commit hash `cb6f9f6f7d46ff6bbce653f8a6a2815fd3254b43`.

Disclaimer

This document reflects the understanding of various issues and vulnerabilities as they are known to Provide Technologies Inc. in the context of the limited scope and specific version as outlined herein. This document makes no representations or warranties to any third party about the utility or safety of the project or its code, the fitness of the contracts for a specific purpose, the viability of the project or the suitability of any business model. This audit does not represent investment advice and should not be interpreted as such.

Scope

The final audit target reviewed contracts generated from the repository by this preprocessor script, which outputs two Solidity files containing the flattened in-scope `cf-adjudicator-contracts` and `cf-funding-protocol-contracts`, as outlined here and here, respectively, and their dependencies. The audit did not review any contracts individually; rather, it reviewed the contracts as they were rendered in the flattened Solidity files using the process described above.

The audit did not formally assess any scripts, tests, or other non-Solidity files within the project. The audit did not assess the usage or implementation of the Counterfactual framework from the perspective of the Indra project. The assessment furnished in this report is the product of an audit focused on reducing vulnerabilities to improve the overall security of the project; a lesser emphasis was placed on optimizing gas consumption. Suggestions made in this report regarding optimizing gas consumption should not be construed to be exhaustive since the primary focus of the audit was to eliminate exploitable vulnerabilities which could result in the loss of user funds and the erosion of trust in projects which depend on the Counterfactual framework.

A subset of two Solidity packages were considered in-scope for the audit: `cf-adjudicator-contracts` and `cf-funding-protocol-contracts`. The functionality of each package is described briefly in this section, and again in much greater detail here.

`cf-adjudicator-contracts` provide an interface to ensure an application can reach finality. See this flattened Solidity file. The `cf-adjudicator-contracts` package provides a subset of the audited contract surface area, as it is a dependency of `cf-funding-protocol-contracts`.

In-Scope Adjudicator Contracts

- `ChallengeRegistry.sol`
- `LibAppCaller.sol`
- `LibStateChannelApp.sol`
- `MChallengeRegistryCore.sol`
- `MixinCancelChallenge.sol`
- `MixinChallengeRegistryCore.sol`
- `MixinRespondToChallenge.sol`
- `MixinSetOutcome.sol`
- `MixinSetState.sol`
- `MixinSetStateWithAction.sol`

`cf-funding-protocol-contracts` consists of a (i) multisig wherein state deposits are held and (ii) interpreters which are responsible for checking the outcome for a particular application instance by way of the adjudicator registry. The funding protocol has a much smaller surface area than the adjudicator contracts but is much more error-prone due to the reliance on `delegatecall`. See this flattened Solidity file. The `cf-funding-protocol-contracts` package provides a superset of the audited contract surface area, as it has a dependency on `cf-adjudicator-contracts`.

In-Scope Funding Protocol Contracts

- `ConditionalTransactionDelegateTarget.sol`
- `MinimumViableMultisig.sol`
- `SingleAssetTwoPartyCoinTransferFromVirtualAppInterpreter.sol`
- `SingleAssetTwoPartyCoinTransferInterpreter.sol`

System Overview

A Counterfactual application is a smart contract which implements the `CounterfactualApp` interface and is represented in the adjudication layer of the Counterfactual framework by an `AppIdentity` which governs a plurality of instances of the `CounterfactualApp` within a channel. Each `CounterfactualApp` instance is uniquely identified by a `channelNonce` and allows for serial state transitions, affected by channel `participants`. Counterfactual plans to support concurrent state transitions in a future version of the framework.

Adjudication

The `ChallengeRegistry` is a singleton adjudication contract which is compatible only with `CounterfactualApp` implementations. `ChallengeRegistry` employs multiple inheritance to mixin various adjudication concerns; each mixin has access to (i) the `appChallenges` and `appOutcomes` storage mappings and (ii) a few internal helper methods in the derived `ChallengeRegistry` contract; these storage mappings and helper methods are provided by the `MChallengeRegistryCore` mixin. The `appChallenges` and `appOutcomes` storage mappings are each keyed on the `bytes32` `AppIdentity` hash:

```
mapping (bytes32 => LibStateChannelApp.AppChallenge) public appChallenges;
mapping (bytes32 => bytes) public appOutcomes;
```

Any `AppChallenge` published on-chain represents a failure by one party of a state channel contract (i.e., a `CounterfactualApp`) to adhere to the protocol.

The `ChallengeRegistry` contract publicly exposes the following methods which modify state and were prioritized in-scope of this audit:

- `MixinCancelChallenge.cancelChallenge`
- `MixinRespondToChallenge.respondToChallenge`
- `MixinSetOutcome.setOutcome`
- `MixinSetState.setState`
- `MixinSetStateWithAction.setStateWithAction`

This mixin pattern makes for excellent readability of the various `ChallengeRegistry` concerns.

Channel Funding Protocol

The `ConditionalTransactionDelegateTarget` contract publicly exposes the following methods which modify state and were prioritized in-scope of this audit:

- `ConditionalTransactionDelegateTarget.executeEffectOfFreeBalance`
- `ConditionalTransactionDelegateTarget.executeEffectOfInterpretedAppOutcome`

The `MinimumViableMultisig` contract publicly exposes the following methods which modify state and were prioritized in-scope of this audit:

- `MinimumViableMultisig.execTransaction`

Findings

The Counterfactual codebase is very clean; naming conventions are consistent and access modifiers are used properly. Kudos to the Counterfactual team for writing such clean, well-documented code.

Legend Critical High severity Medium severity Low severity Other

1. **Integer overflow can prevent finality.** A critical vulnerability in the `MixinSetState.setState` method exists such that an otherwise-authorized user can maliciously force a challenge to be finalized prematurely by providing a timeout value in her `SignAppChallengeUpdate` request to intentionally overflow the `finalizesAt` `uint256` stored on the `AppChallenge` instance. To exploit, the user simply needs to calculate the timeout such that:

```
uint256 max = 2**256 - 1;
uint256 maliciousTimeout = max - block.number; // sore loser
challenge.finalizesAt = block.number + maliciousTimeout + 1; // finalized!
```

The resulting adjudication state would be unrecoverable; the challenge status would be set to `ChallengeStatus.FINALIZES_AFTER_DEADLINE` and future calls to `isStateFinalized` would return `true`. All future calls to the following methods would uselessly revert:

- `MixinCancelChallenge.cancelChallenge`
- `MixinRespondToChallenge.respondToChallenge`
- `MixinSetOutcome.setOutcome`
- `MixinSetState.setState`
- `MixinSetStateWithAction.setStateWithAction`

2. **Integer overflow can prevent finality.** The same critical vulnerability as described in above item (1) also exists in the `MixinSetStateWithAction.setStateWithAction` method and its `SignedAppChallengeUpdateWithAppState` request.

3. **State channel application outcomes and funds transfers can be interpreted prematurely.** A critical bug in `ConditionalTransactionDelegateTarget.executeEffectOfInterpretedAppOutcome` unconditionally interprets channel application outcomes and initiates funds transfers without regard for application finality.

4. **Transaction replay possible via multisig.** The `MinimumViableMultisig.execTransaction` method exposes a transaction replay attack vector by failing to check computed transaction hashes against

`isExecuted`, the storage mapping which provides hash-based replay protection. Failing to ensure a hash has not already been executed allows `call` and `delegatecall` invocations to be repeatedly forwarded to `ConditionalTransactionDelegateTarget` on behalf of the multisig.

5. **Unchecked user input.** A minor bug in the `LibStateChannelApp.verifySignature` method exists as a result of unchecked user input. It is possible to cause an out-of-bounds error by passing `signers` and `signatures` arrays of different lengths.

6. **Predisposition to non-optimal gas consumption.** The codebase relies on `ABIEncoderV2` to make it easy for developers to declare a `struct` to represent channel state and then conveniently encode and decode those state objects to and from `bytes`. This is great for readability and developer experience. The caveat of `ABIEncoderV2` is its usage can result in non-optimal gas consumption.

Recommendations

Our recommendation is that all identified issues SHOULD be fixed prior to using the contracts in any production application. Issues identified as Critical or High severity should be considered “MUST fix.”

Update: all recommendations outlined below have been addressed.

Critical

1. Fix a vulnerability in the `MixinSetState.setState` method which allows a malicious party to intentionally overflow the `finalizesAt` `uint256` stored on an `AppChallenge` instance which the user is a participant. This is a critical vulnerability as it can result in the loss of funds and break the channel protocol. **Update: this issue was fixed in `d37c62d`.**

2. Fix a vulnerability in the `MixinSetStateWithAction.setStateWithAction` method which allows a malicious party to intentionally overflow the `finalizesAt` `uint256` stored on an `AppChallenge` instance which the user is a participant. This is a critical vulnerability as it can result in the loss of funds and break the channel protocol. This vulnerability is identical to the critical vulnerability discovered in the `MixinSetState.setState` method as described in above item (1). **Update: this issue was fixed in `d37c62d`.**

3. Fix a bug allowing premature interpretation and funds transfers for non-finalized applications via `ConditionalTransactionDelegateTarget.executeEffectOfInterpretedAppOutcome`. The `ChallengeRegistry` should be consulted to ensure the application is finalized before its outcome is interpreted. **Update: this issue was fixed in `4465ffe`.**

High severity

1. Fix a vulnerability in the `MinimumViableMultisig.execTransaction` method which exposes a replay attack vector by failing to check `isExecuted`, by simply checking computed transaction hashes against the `isExecuted` storage mapping. **Update: this issue was fixed in `f4770c9`.**

Low severity

1. Fix a potential out-of-bounds error caused by failing to check `signers` and `signatures` arrays to be of equal length in the `LibStateChannelApp.verifySignature` method. **Update: this issue was fixed in `bfd8b55`.**

Other

1. The primary focus of this audit was to reduce vulnerabilities, not gas consumption. Due to the use of `ABIEncoderV2`, our recommendation is to perform a sanity check on each `struct` declaration to ensure values are properly packed to fit in the smallest number of 32-byte slots as possible.

Futureproofing

The following recommendations are out-of-scope of the audit itself but are included below as “friendly advice” in the spirit of improving the future maintainability of the Counterfactual project:

1. The `ConditionalTransactionDelegateTarget` needs better comments; comments are either missing or documented function parameters no longer match the actual function signatures.

2. The test suite could be more readable. Consider taking the time to refactor such that the test suite mirrors the cleanliness and modularity of the Solidity packages. For example, it would make the codebase even more readable if each mixin as provided by the adjudicator package had a corresponding spec file. Currently, `challenge-registry.spec.ts` seems like an attempt to cover everything and this approach will become increasingly hard to read and maintain as more coverage is added. This isn’t the end of the world by any means but refactoring the test suite could result in solid gains in both coverage and readability and force a few shared context helpers to be written which will make the test harness a lot more scalable.

3. This utility script was used to generate a flattened Solidity artifact including all contracts which were in-scope for this audit and will be helpful when kicking off targeted audits in the future.

Conclusion

The Counterfactual project is awesome! It provides a great foundation for building state channel protocols and applications.

Provide ❤️ Connext and we look forward to the imminent mainnet release of Indra 2.0 following minor updates to the Counterfactual framework.

Links

Connext Counterfactual Provide