AdEx Network

AdEx Protocol - Solidity Security Review

Version: 2.0

December, 2018

Contents

Introduction 2

Disclaimer . . . 2

Document Structure . . . 2

Overview . . . 2

Review Summary 4

Per-Contract Vulnerability Summary . . . 5

Detailed Findings 5

Summary of Findings 6

Tokens Can Be Locked if Channel Creators Lose Their Keys . . . 7

Publishers Unable to Withdraw from Fast Expiring Channels . . . 8

Inadequate and Misleading On-chain Channel State . . . 9

Miscellanenous Comments and Gas Optimisation Suggestions . . . 10

A Edge Cases (non-exhaustive list) 12

B Vulnerability Severity Classification 14

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the current Ethereum implementation of the AdEx Protocol. AdEx is a blockchain-powered advertisement exchange aiming at disrupting the existing online advertising landscape and addressing its significant problems, including advertising fraud, privacy, and consent to receiving sponsored messages.

This review focused on AdEx's payment channel implementation named OUTPACE via `AdExCore`, libraries `ChannelLibrary`, `SignatureValidator`, `MerkleProof`, and the `SafeERC20` token wrapper.

The review focused solely on the security aspects of the Solidity implementation of the smart contracts, but also includes general recommendations and informational comments relating to minimising gas usage, along with general code quality observations.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review regarding, the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the AdEx protocol contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an open/closed status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as "informational".

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the AdEx contracts under review, along with a list of edge-cases that were verified as passing spec (see Edge Cases (non-exhaustive list)).

Overview

The Ethereum implementation of the AdEx protocol performs multiple functions, namely it:

- Allows advertisers to use ERC20 tokens to create state-based payment channels. Each channel represents an ad campaign, where publishers can subsequently withdraw payment upon fulfilling the advertiser's specifications.
- Optimizes for scale and cost by allowing all supply-side bidding and end-user fulfillment events to transact and to be observed by validators off-chain.
- Ensures these off-chain events are then aggregated (see `OCEAN` introduction) and signed off by a committee of designated validators with a 2/3 *supermajority*.
- Governs on-chain payouts to approved users through the use of `merkle tree` proofs, where the *leaves* contains account addresses and the approved amounts.

- Allows advertisers to withdraw remaining campaign balance upon channel expiry.

AdEx's `SafeERC20` aims to be compatible with all major versions of ERC20 [1] tokens. Specifically, this particular implementation aims to (i) prevent tokens from being stuck when sent to incompatible, older ERC20 contracts, and (ii) to accurately assess the transaction success state from newer ERC20 contracts.

AdEx acknowledges that validator consensus can break, in the event that all parties or the majority of validators become malicious. It is emphasized that such scenarios can be mitigated through publisher consortiums, reputation monitoring, and adding 3rd party tie-breakers as additional validators.

AdEx acknowledges that with AdEx Protocol, traditional digital advertising fraud and Sybil attacks would still be possible. It is emphasized that future development will include anti-fraud tooling, end-user level proof of work requirements, identify verification, and other mitigations. The current implementation of the AdEx protocol heavily relies on off-chain components as part of its security model (particularly the AdEx SDK).

Review Summary

This review was initially conducted on commit 98c65a3, which contains the `contracts` folder. This folder contains a number of contracts, all of which are inherited (directly or indirectly) by the main `AdExCore` contract. The complete list of contracts is as follows:

```
Contracts
├── AdExCore.sol
├── libs
│   ├── ChannelLibrary.sol
│   ├── MerkleProof.sol
│   ├── SafeERC20.sol
│   ├── SafeMath.sol (Out of Scope)
│   └── SignatureValidator.sol
└── Migrations.sol (Out of Scope)
```

Retesting activities targeted commit caa872c.

This security assessment targeted exclusively the following contracts:

- `AdExCore`
- `ChannelLibrary`
- `MerkleProof`
- `SafeERC20`
- `SignatureValidator`

To support this review, the testing team used the following automated testing tools:

- Rattle: `https://github.com/trailofbits/rattle`
- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Per-Contract Vulnerability Summary

The testing team identified a total of five (5) issues during this assessment, all of which are classified as informational. These findings have all been acknowledged by the development team.

AdExCore (`AdExCore.sol`)

Some informational notes are given.
Some gas-saving modifications are suggested.
No potential vulnerabilities have been identified.

ChannelLibrary (`ChannelLibrary.sol`)

Some informational notes are given.
Some gas-saving modifications are suggested.
No potential vulnerabilities have been identified.

SignatureValidator (`SignatureValidator.sol`)

Some informational notes are given.
No potential vulnerabilities have been identified.

MerkleProof (`MerkleProof.sol`)

Some informational notes are given.
Some gas-saving modifications are suggested.
No potential vulnerabilities have been identified.

SafeERC20 (`SafeERC20.sol`)

Some informational notes are given.
No potential vulnerabilities have been identified.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the AdEx Protocol's smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
ADX-01	Tokens Can Be Locked if Channel Creators Lose Their Keys	Informational	Closed
ADX-02	Publishers Unable to Withdraw from Fast Expiring Channels	Informational	Closed
ADX-03	Inadequate and Misleading On-chain Channel State	Informational	Closed
ADX-04	Miscellanenous Comments and Gas Optimisation Suggestions	Informational	Resolved

ADX-01	Tokens Can Be Locked if Channel Creators Lose Their Keys
Asset	AdExCore.sol
Status	**Closed:** See Resolution
Rating	Informational

Description

In the event that a channel becomes stale and cannot update/progress its state (for example, the *supermajority* cannot be reached, or more than $\frac{1}{3}$ lose their keys or fail to participate) the tokens can only be withdrawn once the channel expires. In such an event, only the channel `creator` can withdraw the original deposited tokens.

It can be the case that a creator creates a channel and disposes or loses the related private key over time, not expecting to have to withdraw the channel's tokens. In this scenario, tokens are lost forever in the `AdExCore` contract.

This is simply an informational point and may be the intentional design of the related smart contract.

Recommendations

A safety mechanism could be easily implemented which allows anyone to withdraw lost tokens using a timeout-based approach. For example, on line [55] the require could be modified as follows:

```
require(msg.sender == channel.creator || now > channel.validUntil + 1 year);
```

Resolution

The development team acknowledges this issue, see below response:

"After review, we decided to pass on implementing the recommendation, since it constitutes unintuitive behavior for us; the recommendation would allow anyone to claim the tokens after a period; in terms of decision making, this is the same debate as whether lost coins should be "recycled" from bitcoin/ethereum. In our protocol papers, we mention that using an arbitor validator is a helpful, since if one of them loses the keys, the channel can still be drained".

ADX-02	Publishers Unable to Withdraw from Fast Expiring Channels
Asset	ChannelLibrary.sol
Status	**Closed:** See Resolution
Rating	Informational

Description

The AdEx Protocol documentation states [2], *"OUTPACE channel should have 2-3 times as long of a duration, in order to allow extra time for publishers to withdraw their revenue."*

However, in the current core implementation, there is no minimum time requirement for each channel's `validUntil` duration. Thus, publishers have to rely on the *Market* (a RESTful service) to compute and communicate all time constraints. This Market layer cannot be entirely trusted.

Assuming the time interval is erroneously communicated to publishers, or simply not accounted for by publishers, there exists a realistic edge case where channels expire before publishers are able to withdraw earned revenues.

Similarly, a malicious advertiser can also create a channel with a small active time window, with just enough time for publishers to confirm a bid, but not enough time to allow payment withdrawal. After the channel expires, the advertiser is able to withdraw the remaining balance without paying publishers.

Recommendations

AdEx acknowledges this potential **stuck revenue** problem. The current implementation suggests publishers will reliably check for expiration times off-chain and will account for this potential issue.

Another recommendation is to impose a reasonable `MIN_VALIDITY` time interval (as a `constant` in `ChannelLibrary`) for each channel upon channel creation. This constant could be used to restrict channels to have minimum lengths.

Resolution

The development team acknowledges this issue, see below response:

"Again, we consider this to be a defined behavior, although we will consider implementing the recommendation of a minimum channel duration. However, as with many things in the AdEx ecosystem (token used, who are the validators) this is one of the things that will be governed on-chain: short lived channels would be excluded/filtered".

ADX-03	Inadequate and Misleading On-chain Channel State
Asset	ChannelLibrary.sol
Status	**Closed:** See Resolution
Rating	Informational

Description

The AdEx Protocol documentation specifies that [2]: *"the market needs to track all on-chain OUTPACE channels and needs to constantly monitor their liveness (>=2/3 validators online and producing new states) and state."*.

Furthermore, the `AdexCore` maintains a public `states` mapping that signals whether channels are `Unknown`, `Active`, or `Expired`.

Current design and implementation seem to permit clumsy providers or market-builders to mistake `states` for a reliable source of channel *state*. It is possible that users will rely on the `getChannelState` getter function to read this information on-chain and see if the channel has expired.

This is problematic since the `states` mapping does not accurately describe the latest actual channel state. For example, the only way for a channel state to become `Expired` is if the advertiser withdraws their balance upon channel expiry. Expired channels, whose advertisers have not yet withdrawn, will show up as `Active` in the related on-chain `states`.

Recommendations

A getter function could be implemented, which would leverage the `isValid()` function on `Active` states with the current blocktime to return a more accurate description of whether a channel's state is truly active and valid.

Resolution

The development team acknowledges this issue, see below response:

"We acknowledge this issue and it's a very good point, however we think it's an issue of naming. There is a separate system to track the overall channel state (the validator stack), which is also aware of campaign health and whether all funds in the channel are distributed (Exchausted state) that takes care of that. Our internal recommendation would be to rename the state getter".

ADX-04	Miscellanenous Comments and Gas Optimisation Suggestions
Asset	All Contracts in Scope
Status	Resolved: See inline comments and Resolution
Rating	Informational

Description

This section outlines miscellaneous comments and suggestions that were found as a by-product of this review. We include this section as it may be useful for the authors to improve readability of the code.

- **Gas Saving** - In `ChannelLibrary.sol` on lines [9] and [10], a `uint8` is used. The EVM prefers type sizes of 32 bits (i.e. `uint256`). In order to read/write smaller types the EVM uses masks which costs more gas. Setting these to `uint` types will reduce computation and save gas.
  ✓ Resolved in commit [da725ba]
- **Gas Saving - For-loop Convention** - In `MerkleProof.sol` line [8], the for-loop terminates when `i!=proof.length`. It may be better to terminate the for-loop as of `i < proof.length` for better readability. This would also save gas, as the EVM takes one additional opcode to perform a "not equals" check than it takes to perform a “less than” check.
  ✓ Resolved in commit [46a8e48]
- **Gas Saving - Public Functions** - In `AdExCore.sol`, the following functions can be declared `external` instead of `public` to save gas: `channelOpen`, `channelWithdraw` and `channelWithdrawExpired`.
- **Comment Inaccuracy** - In `MerkleProof.sol` line [7], developer comment notes the future need to *“compare valueHash == root if the proof is empty?”*. This is not necessary as this base case is already handled by the function. A merkle tree with only 1 leaf uses the leaf as the root itself.
  ✓ Resolved in commit [caa872c]
- **Comment Inaccuracy** - In `SafeERC20.sol` line [10], developer comment suggests the `checkSuccess()` function is *“definitely not a pure fn but the compiler complains otherwise”*. This function is indeed a pure function as the assembly doesn’t include any impure opcode (see a non-exhaustive list [3]).
  ✓ Resolved in commit [caa872c]

Recommendations

Ensure these are as expected.

Resolution

The items listed above have been resolved, with the exception of the "Gas Saving - Public Functions". See related response below from the development team:

"Everything addressed in latest master except "gas saving - public functions". Using "external" causes various types of issues related to the new ABIEncoderv2 within the solidity/ethereum ecosystem; the most prominent one we encountered was the C++ version of solc v0.4.25 crashing with a segfault when the functions are external; this has not

happened ever since we simplified a particular function signature, but still, truffle test fails with an obscure solc JS error when we try to change it now; should be investigated further".

Appendix A Edge Cases (non-exhaustive list)

A non-exhaustive list of edge cases were identified to support this security review and are given along with this document. Each edge case was manually stepped-through and validated, with the expected behaviors listed below.

AdExCore: function `channelOpen()`

- Advertisers cannot override existing campaign channels with new channels, regardless of the state of existing channels ✓
- The channel creator must be the `msg.sender`, i.e. malicious users cannot designate a random address to the campaign creator ✓
- Advertisers can stake token amounts in the range $[0, 2^{256} - 1]$ ✓
- Advertisers can input any `bytes32` value to represent the campaign spec ✓
- Advertisers using non-malicious, correctly-implemented ERC20 tokens can expect failed token transfers to revert ✓
- Advertisers can use malicious ERC20 tokens and successfully create channels. This leaves it to the network, publishers, and other users to verify the token standards used for a each campaign ✓

AdExCore: function `channelWithdraw()`

- Users except channel creators can only withdraw from active campaigns ✓
- Given honest validators and the correct channel hash, users cannot spoof the `signatures` array to withdraw from channel. ✓
- Given honest validators and the correct channel hash, users cannot spoof the `stateroot` to withdraw from channel. ✓
- Given honest validators and the correct channel hash, users cannot spoof the withdrawal amount (used to calculate `balanceleaf`) such that it correctly points back to the stateroot ✓

AdExCore: function `channelWithdrawExpired`

- Given correct timestamps, advertisers can only withdraw from expired channels that they created ✓
- Advertisers are only able to successfully withdraw remaining funds from each expired channel once ✓
- Expired (in real time) channels can have `active` states if advertisers failed to withdraw or have not yet withdrawn funds ✓

ChannelLibrary: function `isValid`

- Advertisers have to provide at least 2 validators addresses (honest or not) upon channel creation ✓
- Advertisers have to provide at most 25 validators addresses (honest or not) upon channel creation ✓

- Given correct block timestamps, advertisers can only create channels which expire after channel creation, but before 365 days from channel creation ✓
- Channel variables that should not change throughout the campaign are used to calculate the channel hash, i.e. channelID ✓

ChannelLibrary: function `isSignedBySupermajority`

- Given honest validators and the correct input state, the two thirds majority vote rule is enforced ✓
- Validation fails if users submitted a different number of signatures than the number of channel validators ✓

SignatureValidator: function `isValidSignature`

- Function validates that a hash was signed by the input signer address, given AdEx's specifications ✓

MerkleProof: function `isContained`

- Given a valid `proof` array and `valueHash`, function returns true upon traversing a valid `balanceleaf` to its merkle root ✓

SafeERC20: function `checkSuccess`

- Advertisers can use previous, potentially non-compliant versions of ERC20 tokens that do not return success/failure states on token transfer ✓
- In the event that ERC20 tokens return a value upon transfer, function only checks 32 bytes of the returned values for a boolean response ✓

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] ERC-20 Token Standard. Github, Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[2] AdEx. AdEx Protocol. Github, 2018, Available: https://github.com/AdExNetwork/adex-protocol. [Accessed 2018].

[3] Sigma Prime. Purity in the EVM. Blog, 2018, Available: https://blog.sigmaprime.io/evm-purity.html. [Accessed 2018].