eka SILICON

Security Audit Report

Humbleswap

June 6, 2022

1 Summary

Reach commissioned Ekasilicon to conduct a security audit of Humbleswap, which comprises three dApps: `triumvirate`, a token swapper, and `staker`. The token swapper has two variants: one which allows users to swap between a native token and a non-native token (`net-tok`), and one which allows users to swap between two non-native tokens (`tok-tok`). This audit considers each variant of the token swapper separately, and so considers four dApps in total.

Humbleswap is implemented in Reach (https://reach.sh). From a single Reach program, the Reach compiler produces each tier of the implemented dApp for several back-end chains. The audit considered the deployable artifacts of all four dApps generated from two Reach source files for the Algorand network.

The objective of this audit was to determine whether the high-level dApp behaviors were preserved at the network level. This determination requires the consideration of Algorand-specific network capabilities in conjunction with the behavior of the dApp.

The audit was performed with the assistance of the Jade program analyzer, which is designed specifically for the analysis of smart contracts which target the Algorand network. Jade is developed by Ekasilicon and this development has been supported by the Algorand Foundation; it is open source (https://github.com/ekasilicon/jade).

Jade is built on sound static analysis techniques that are able not only to detect the presence of a particular vulnerability but also to fully demonstrate the absence of said vulnerability once an ostensible resolution has been applied. Thus, within this report, claims that “the dApp traverses flow X” are claims that “the dApp traverses *only* flow X” and no adversary can craft a transaction to cause the dApp to deviate from it.

Although Jade is built on sound static analysis techniques, its implementation and the audit process itself still relies on human judgement and is thus susceptible to error. Consequently, although the audit team places great confidence in results obtained with Jade’s assistance, it can offer no guarantee that they are unassailable. Furthermore, Jade is used to analyze specific key facets of the smart contract’s execution. The demonstrated confinement of a particular facet to its intended limits applies precisely to that facet and does not say anything about other facets or the smart contract more generally.

The audit took place over the course of two and a half weeks, a timeline enabled by the use of automated analysis with Jade.

The audit was able to formally establish two properties:

1. The deployed dApp has the same protocol flow as the source program.

2. All extended arithmetic operations are checked for overflow.

2 Scope

codebase	https://github.com/reach-sh/duoswap-core
tree	https://github.com/reach-sh/duoswap-core/tree/a70f3ed822b6a4fbc758d00d25157d595db159f0
Reach triumvirate and swapper	https://github.com/reach-sh/duoswap-core/index.rsh
triumvirate bytecode	https://github.com/reach-sh/duoswap-core/build/index.triumvirate.mjs#L2569
net-tok bytecode	https://github.com/reach-sh/duoswap-core/build/index.net_tok.mjs#L2805
tok-tok bytecode	https://github.com/reach-sh/duoswap-core/build/index.tok_tok.mjs#L2842
Reach staker	https://github.com/reach-sh/duoswap-core/staker.rsh
staker bytecode	https://github.com/reach-sh/duoswap-core/build/staker.main.mjs#L3440

The target of the audit was the Humbleswap codebase (*codebase*) as it exists at *tree*. The focus was the relationships between (1) the source file *Reach triumvirate and swapper* and the generated bytecode *triumvirate*, *net-tok*, and *tok-tok*, and (2) the source file *Reach staker* and the generate bytecode *staker*).

3 Methodology

To gain comprehensive assurance about a program written in a language other than the native bytecode of the platform on which it's run, one must ensure that

1. the program as written matches the intent of the developers, and
2. the meaning of the compiled artifact is faithful to original.

Because the environment model of the platform may differ than that of (the language of) the source program, one must additionally ensure that the compiled artifact does not permit interactions at the platform level that are not intended at the source level. (Here, *interaction* is used to refer to both an action that a contract participant takes with respect to the contract and the particular combination of the implementation and its execution environment.)

Instantiating this schema, comprehensive assurance about a Reach dApp is obtained by ensuring:

1. The Reach program, including declared assertions and invariants, matches the developers' intent.
2. The compiled artifacts are faithful to the original Reach program, including that

   a. the meaning of the program as written is preserved through compilation and

   b. the embedded assertions actually hold.

3. The compiled artifacts are secure with respect to the interactions allowed by their respective platforms.

The Reach compiler is capable of verifying embedded assertions at the Reach source level, in which case 2a would subsume 2b. However, it is possible to disable this verification, in which case 2a and 2b would be considered somewhat independently.

This audit does not deliver comprehensive assurance in the preceding sense. (Doing so constitutes full formal verification which is extremely labor-intensive.) However, it does address each identified aspect, in whole or in part.

Aspect 1, that the Reach dApps as written reflect the intent of the Humble team, is established by communication with the Humble team. The primary audit effort is devoted to establishing aspects 2 and 3 for dApp deployment on the Algorand network specifically.

The approach to establishing these aspects is, for each dApp-derived bytecode program, proceeds as follows:

1. Use the Jade program analyzer to construct a network-level model which can be used both to examine network-specific behaviors and to relate to its source Reach program.

2. Use automated analysis to extract network-level transaction schemas and categorize them according to the effect they have on the encoded state of the Reach source program. This categorization induces a state transition graph which itself models the protocol flow of the program.

3. Use the state transition graph to reason about high-level flows, including reachability of particular states and actions and protocol invariants.

The Jade program analyzer (simply *Jade* in the sequel) provides automated analysis to produce the network-level model. By default, Jade produces a naive model which is often both imprecise and intractable to construct. However, precision and tractability can be obtained by customizing the model constructor, in part or in full, with respect to each program—Jade is designed to easily accommodate this customization. This custom model is the result of an iterative process, which proceeds essentially as follows:

1. Given an analyzer specialization (i.e. a custom model constructor, initially naive), attempt to produce the naive model of the program to assess precision and tractability. Even if the attempt is unsuccessful, examining Jade's effort typically reveals a promising aspect to specialize.

2. Examine the produced model (or attempt) to identify refinements to the specialization to account for particular program behaviors at a higher abstraction level. By choosing the appropriate abstraction, the analysis becomes at once more precise and requires less effort by the analyzer (perhaps bringing the analysis into tractability, if it isn't already). The choice of abstraction is determined by the compilation model and the internal logic of the program. Proceed to step 1.

Of course, the human-in-the-loop breaks at step 1 if the produced model is sufficiently precise. Significant audit effort was devoted to devising effective custom models which were both precise and tractable.

Even when its production is tractable, the size of a Jade-produced model typically exceeds the size of the program. However, the form of the model is no longer code but, essentially, a set of valid transaction schemas which include the constraints under which valid instances can be constructed and the external effects the transaction instance produces.

Where resources permit, the edges are summarized as a logical formula of constraints and effects which fully characterizes the transition. With these constraints, it is possible in principle to prove high-level invariants about dApp behavior.

4 Disclaimer

This report does not constitute legal or investment advice. The results documented in this report are provided for informational purposes to document the due diligence involved in the development of only the analyzed contracts. The report authors assume no liability for any and all potential consequences of the deployment or use of these contracts.

This report makes no claims that its presented results are fully comprehensive in the sense that every facet of the smart contract is covered. Furthermore, this report concerns only one, albeit critical, component of the deployed dApp and can therefore make no claims about the correct operation of the system it supports. The report authors recommend using a variety of techniques to assess a program’s fitness, including those not undertaken in the reported audit process.

Although the tools used to perform this audit are designed and intended to provide sound information, the possibility of human error in implementation or utilization remains. We therefore offer confidence but no guarantee about the claims made in this report, and again recommend the use of a variety of techniques to increase assurance.

5 Context

5.1 OnCompletion

On the Algorand network, the `OnCompletion` transaction parameter dictates which post-transaction operation is performed with respect to the contract and the transaction sender. The operation may be to “opt in” the sender to the contract so that it can begin interacting with it more meaningfully, to update the contract code, to delete the contract code, or do nothing (among a few other possibilities not relevant here). The operations to update or delete the contract are particularly sensitive because a contract which does not constrain them appropriately is susceptible to attacker-crafted transactions which disrupt or hijack the dApp and its assets.

Reach can express dApps with indefinite lifetimes; such dApps which allow an `OnCompletion` parameter which updates or deletes the contract code are in error. Reach can also express dApps with finite lifetimes, which may be manifest as the culmination of a protocol flow or as a protocol abort. These dApps necessarily allow the contract code to be deleted (or irrevocably block the dApp’s function) and so must permit transactions which have irrevocable effects.

For dApps in the latter category, which are typical, it is imperative that the preconditions for the irrevocable action be met before the action is acceptable. A major portion of the audit effort was devoted to characterizing the preconditions and ensuring, as much as resources permit, that the preconditions cannot be satisfied against the intended dApp design.

In terms of the state transition graph extracted from the analyzer-produced model, the audit considered, for each dApp, whether

1. it allows any unintended operations in non-final states (e.g. deletion), and,
2. it can arrive at the final state through any unintended means.

The sections that follow present state transition graphs which depict the final state of the protocol flow as the removal of the dApp from the network by definition. That is, any transition which leads to the dApp being removed from the network reaches a final state. In addition, the state transition graphs reflect every possible state transition (with one documented and irrelevant exception). Thus, the results establish that no path through the state space outside the depicted edges is possible and thus no deletion of the dApp can occur unless as the result of the depicted state evolution.

5.2 Arithmetic Overflow

Algorand programs provide arithmetic over 64-bit unsigned integers and 64-byte arrays interpreted as unsigned integers. However, if the result of an operation over the former exceeds 64 bits (i.e. overflows), the program fails. Similarly, if an arithmetic operation is attempted on a byte array larger than 64 bytes, the program fails. These behaviors mean that Algorand programs are natively safe with respect to arithmetic overflow; there is no possibility that an arithmetic operation could overflow and "wrap around" the result with the program proceeding unwittingly.

This behavior, though safe, is conservative. It restricts actually-safe arithmetic sequences that exceed bounds transitorily. To circumvent this restriction, Reach implements 256-bit integers in Algorand programs using 32-byte arrays. The subject dApps of this audit use these integers to carry a set of 64-bit integers through formulas whose intermediate values exceed 64 bits but whose final values ostensibly don’t. The dApps then extract the lower 64 bits of the 256-bit result to propagate. If the implementation allows the lower 64 bits to be extracted without ensuring that the upper 192 bits are zero, then it does not protect against overflow and is unsafe.

6 Results

The following sections discuss the results of the audit/tool analysis of each dApp. The flow of each dApp is characterized by a state transition graph. The audit/tool analysis demonstrated that no transitions other than those represented in the graph are possible. It was also able to closely inspect the precise conditions under which "trivial" transitions occur (representing 12 or fewer transaction schemas); the conditions for each of these were routine and matched expectations. Nontrivial transitions, which represent dozens, hundreds, or thousands of transaction schemas, can be used to prove higher-level invariants, but such results are out of the scope of this audit. Nevertheless, the schemas underlying these transitions will be provided.

For all dApps, all non-creation transitions require $\mathrm{ApplicationID} \neq 0$. The audit/tool analysis also verified, for all dApps, that

1. All except the delete transition require $\mathrm{OnCompletion} = 0$.
2. All transitions (except a single noted and accounted transition) explicitly check their source state.
3. All 256-bit arithmetic operations are checked for overflow and all casts from 256-bit integers to 64-bit integers are checked to ensure the value can be represented in 64 bits.

6.1 net-tok

The symbolic analyzer (customized to this specific dApp) produced 34 transaction schemas. In the state transition graph (below), the schemas partition with

- one (1) for creation,
- one (1) for transition 1,
- 27 for transition 2,
- two (2) for transition 3,
- one (1) for transition 4, and
- two (2) for transition 5.

$$\begin{tikzcd}
\text{creation} \arrow[r] & X_0 = 0 \arrow[r, "1"] & X_0 = 4 \arrow[loop above, "2"] \arrow[r, "3"] & X_0 = 2 \arrow[loop above, "4"] \arrow[r, "5"] & \text{deletion}
\end{tikzcd}$$

6.2 tok-tok

Because `net-tok` and `tok-tok` are the derived from the same source program and distinguished only be their type parameterization, their high-level flow is identical. The symbolic analyzer (customized to this specific dApp) produced 34 transaction schemas. In the state transition graph (below), the schemas partition with

- one (1) for creation,
- one (1) for transition 1,
- 27 for transition 2,
- two (2) for transition 3,
- one (1) for transition 4, and
- two (2) for transition 5.

$$\begin{tikzcd}
\text{creation} \arrow[r] & X_0 = 0 \arrow[r, "1"] & X_0 = 4 \arrow[loop above, "2"] \arrow[r, "3"] & X_0 = 2 \arrow[loop above, "4"] \arrow[r, "5"] & \text{deletion}
\end{tikzcd}$$

6.3 triumvirate

The symbolic analyzer (customized to this specific dApp) produced 125 transaction schemas. In the state transition graph (below), the schemas partition with

- one (1) for creation,
- one (1) for transition 1,
- 111 for transition 2, and
- 12 for transition 3.

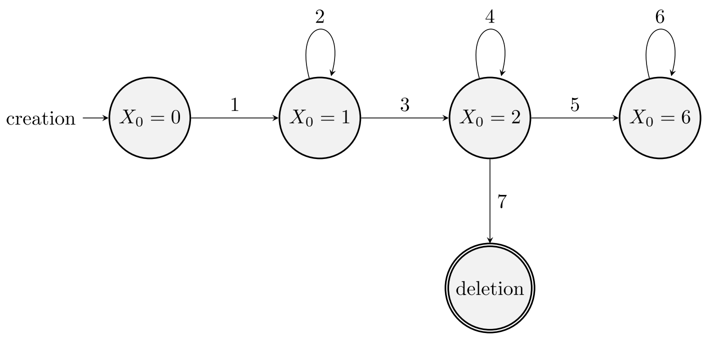
$$\begin{tikzcd}
\text{creation} \arrow[r] & X_0 = 0 \arrow[r, "1"] & X_0 = 3 \arrow[loop above, "2"] \arrow[r, "3"] & \text{deletion}
\end{tikzcd}$$

6.4 staker

The symbolic analyzer (customized to this specific dApp) produced 13905 transaction schemas. In the state transition graph (below), the schemas partition with

- one (1) for creation,
- one (1) for transition 1,
- one (1) for transition 2,

- one (1) for transition 3,
- one (1) for transition 4,
- two (2) for transition 5,
- 13,895 for transition 6, and
- two (2) for transition 7.



There is one transition not reflected in the diagram which transitions each state to itself and opts the sender into participation with the staker.

7 Conclusion

The audit documented in this report was able to characterize the precise transaction schemas which effect the state transitions in the high-level Reach program. So characterized, the audit showed that these schemas do not allow attackers to travel outside the intended state flow of each dApp.

However, this description is not a claim that the dApps in question are fully secure. In particular, the audit did not formally prove that the major transitions (constituting between dozens and thousands of transaction schemas) protect the protocol appropriately. But, these schemas have been formulated logically for a higher-level prover to use for that and similar purposes.

The methodology used in this audit has been used to analyze other Reach programs and subsequently uncover serious vulnerabilities. One reason that no such vulnerabilities were found in this audit is that the Reach language and compiler have been altered to prevent the previously-found vulnerabilities from manifesting.

8 About

Ekasilicon strives to offer high-quality assurances of important smart contract properties using a combination of versatile tools and contract-specific abstractions. If you are in need of a custom verification solution, contact info@ekasilicon.io to schedule a consult.