Security Audit Report

Simple Staker

March 25, 2022

1 Summary

Reach commissioned Ekasilicon to conduct a security audit of Simple Staker, a DApp which allows users to stake tokens and earn rewards.

Simple Staker 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 objective of this audit was to assess the security of Simple Staker for deployment as a TEAL program on the Algorand blockchain.

Security issues in scope included generic TEAL/Algorand security issues that afflict smart contracts generally and the failure of the compiler-produced TEAL implementation to uphold the invariants specified in the original Reach program.

The audit was performed with the assistance of the Jade program analyzer, which is designed specifically for the analysis of TEAL/Algorand smart contracts. 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 (i.e. trustworthy) static analysis techniques that are able not only to detect the presence of a particular security issue but also to fully demonstrate the absence of said issue once an ostensible resolution has been applied. Accordingly, the audit team pointed Jade toward security issues specific to a TEAL/Algorand deployment to prove (in the end) that Simple Staker was free of them.

Although Jade is built on sound static analysis techniques, its implementation and the audit process itself still relies on human judgement and is thus prone

to error. Consequently, although the audit team places great confidence in results obtain 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 focused primarily on demonstrating that the TEAL/Algorand manifestation of Simple Staker included sufficient safeguards to prevent disruption via universal attack vectors. This focus led to the discovery of major security issue and one informational issue. Once reported, the Simple Staker and Reach teams promptly addressed each issue and the audit team was able to verify with Jade that each was completely resolved.

The audit took place over the course of two weeks, a timeline enabled by two factors:

1. The use of automated analysis in Jade allowed ostensible resolutions to be verified with little effort.

2. The Simple Staker and Reach teams were extremely responsive. The audit team communicated closely with them throughout the course of the audit to raise, discuss, and resolve issues. Even when an issue required modification to the Simple Staker implementation or the Reach compiler, the respective team typically resolved it in the span of a single audit session.

Although Simple Staker proceeded through multiple iterations during this process, this audit report contrasts only its initial and final versions.

2 Scope

The audit team confined the audit to the following described scope.

codebase
https://github.com/DanBurton/simple-staker
initial
https://github.com/DanBurton/simple-staker/tree/e8cc225673f34a52c36014f5da94b6ea8007fdf8
final
https://github.com/DanBurton/simple-staker/tree/f3bda887eb6f48b5550e1c10f380d015c2f132f4
Reach implementation
https://github.com/DanBurton/simple-staker/src/index.rsh
TEAL assembly
https://github.com/DanBurton/simple-staker/src/build/index.main.appApproval.teal
TEAL bytecode
https://github.com/DanBurton/simple-staker/src/build/index.main.mjs#L2825

The target of the audit was the Simple Staker codebase (*codebase*). The initial review was conducted on a Simple Staker prototype at the initial tree (*initial*) of the codebase. The finial review was conducted on an updated Simple Staker at the final tree (*final*) of the codebase. The review centered around the Reach-implemented DApp (*Reach implementation*). Specifically, the review focused on the TEAL program produced by the Reach compiler (*TEAL assembly*). Because TEAL program assembly is not absolutely trivial, the audit team analyzed the assembled TEAL program (*TEAL bytecode*). The Reach

implementation, TEAL assembly, and TEAL bytecode artifacts are each qualified by a tree, either *initial* or *final*.

In this report, we contrast between the initial and final versions of Simple Staker, corresponding to the *initial* and *final* codebases, and omit intermediate versions. We use the term *final* only with respect to this audit process and not to Simple Staker's implementation generally.

3 Methodology

The purpose of the audit is to establish behaviors of the version of Simple Staker compiled for the TEAL/Algorand backend specifically. Accordingly, the audit begins at the level of the assembled TEAL program in order to reason about it on its own terms.

The Jade program analyzer provides a formal model of TEAL programs and was used in one mode to detect possible issues and in another to verify their existence. Jade was also used to reconstruct higher-level behaviors from the assembled TEAL program, approaching the level of the source Reach program.

Then audit was performed using an iterative process to incrementally account for increasingly higher-level behaviors. The process of the audit proceeded essentially as follows:

1. Produce an initial trivial model of the program that is conservative in the sense that it accounts for any behavior the program could exhibit. Because the model is trivial, it is typically both uninformative and imprecise. The purpose of this initial step is to establish the tractability of the program by the static analyzer.

2. Refine the current model to offer more information and capture particular program behaviors more precisely. The refinement is guided by the constraints the program imposes on itself which reflect both program-specific logic and TEAL/Algorand-specific environment selection. Constraints are interpreted against the increasingly-refined model and, when not accounted for by the model, are recorded for subsequent refinement steps.

3. The human-in-the-loop examines the different instances of the model reached by the program and, for each such instance, the additional constraints for which it does not account. The human devises useful refinements to the model and continues at step 2.

The choice of refinement must at once provide more and more-precise information about program behavior and keep the execution state space exploration tractable. Naive refinement leads to an explosion in the size of the state space which yields an intractable analysis.

The result of this iterative process was a program-specific model that captures important behavioral properties of the program. The analyzer used this model to produce summaries of every kind of successful transaction (i.e. transaction schemas) which account for the constraints on each transaction and its context. Jade reflects the conservative nature of the model in the summaries so that, for *every* successful transaction, there is a summary which accounts for it.

In principle, the model is formulated at such a level that its fidelity to the original program can be assessed by using the summaries as evidence in the proof of program invariants. Such a proof was considered outside of the scope of this audit however.

When a summary appeared to include unsafe transactions, the TEAL program is executed with concrete inputs derived from the constraints to determine whether the unsafe transaction is indeed realizable. If not, the precision of the model is refined to exclude such transactions and the results of a fresh analysis are assessed. If the transaction is realizable, the summary reports a true vulnerability and the audit team assesses its severity.

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 Findings

The audit team's primarily focused on the treatment of the `OnCompletion` transaction parameter by Simple Staker and identified the following two issues.

ID	Title	Severity	Status
ESS-1	Inadequate `OnCompletion` guard	● major	fully-resolved
ESS-2	Spurious control path	● informational	fully-resolved

Discussion of the nature and resolution of these issues follows.

5.1 Context

At the Algorand VM level, 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.

In some contexts, of course, a DApp may wish to allow such operations. Simple Staker, for instance, intends to allow contract deletion once the staking completes; the current state of the contract is explicitly encoded within the internal contract state. The high-level flow of the Simple Staker has essentially three states:

1. a pre-initialization state; once initialization is performed, Simple Staker enters
2. a staking state in which users can stake tokens and claim rewards; once the end of the staking period is reached and all rewards have been claimed, Simple Staker enters
3. a final state.

The intent is that Simple Staker allows itself to be deleted once it has reached its final state.

This portion of our analysis therefore centered on whether Simple Staker

1. allows any operations other than deletion in the final state,
2. allows any unintended operations in other states (including deletion), and, assuming these are both refuted,
3. can arrive at the final state through any unintended means.

5.2 Results

The Jade analyzer identified 6415 unique transaction schemas, differentiated by the set of constraints they impose on transactions and their contexts. Any transaction which would succeed at any point during the lifetime of Simple Staker and through the evolution of its internal state is an instance of one of these schemas. A significant fraction of the audit effort was to further analyze these schemas to deduce safety properties (also assisted by Jade).

The analysis uncovered two security issues, one the audit team considers major and one it considers informational. After the Reach compiler team addressed each issue, the audit team re-analyzed the code and considers the issues completely resolved.

5.2.1 ESS-1

The initial Simple Staker did not meaningfully constrain the `OnCompletion` parameter in the staking state. In particular, the initial Simple Staker permitted contracts which have not previously opted in to issue "view" transactions which do not affect internal contract state. However, the Algorand platform does not distinguish between such "pure" transactions and others which constitute genuine interaction and in both cases performs any specified post-transaction operation so long as the contract allows it.

The audit team considers this a major issue because it allows an attacker to craft a transaction which deletes Simple Staker or replaces it with code that puts Simple Staker on-chain state under the attacker's control.

This issue was reported to the Reach compiler team who updated the Reach compiler to generate programs which comprehensively guard the `OnCompletion` parameter.

The audit team repeated the analysis steps on the final Simple Staker and verified that it does indeed guard the `OnCompletion` parameter appropriately with respect to the internal contract state. In particular, a user can issue a transaction performing:

- an *opt-in* operation in any state,
- a *no-op* operation in any state, and
- a *deletion* operation in the final state.

That is, there exist acceptable transactions with such operations in each state. Because the unintended allowance of a *deletion* operation is potentially catastrophic, the audit team applied further analysis to ensure that the final state of the contract, in which *deletion* is possible, could not be prematurely

provoked; the succeeding State Flow section reports the details of this analysis.

The final analysis of the audit team is that this issue is completely resolved.

5.2.2 ESS-2

The audit team identified an issue in that the initial Simple Staker included a spurious control flow path on which the intended failure of a transaction was achieved in an incidental way. The existence of this path is harmless because the transition could still be traversed by a well-formed transaction and only such transactions are generated by the compiled front end, so the audit team classifies it as merely informational.

Although the issue was minor, the audit team nevertheless notified the Reach compiler team and they promptly resolved the issue within the compiler. A follow-up analysis determined that this issue is completely resolved.

5.3 State Flow

Simple Staker is designed to accept transactions which delete the Simple Staker DApp from the chain at a particular internal state. To ensure that Simple Staker is secure against malicious transactions to prematurely delete it from the chain, it is necessary to ensure that the internal state in which delete transactions are possible is reached only when intended.

Jade was able to produce the precise conditions under which an acceptable delete transaction is possible. Let X_i stand for the eight consecutive bytes of global storage at the empty key offset by i, $Y_{i,j}$ stand for the j consecutive bytes of the concatenation of global storage at key '0' and key '1' offset by i, A_i stand for the ith transaction argument, $|\cdot|$ give the number of bytes in a byte array, $\lfloor\cdot\rfloor$ interpret a byte array as an unsigned integer (implicitly

constraining it to no greater than eight bytes in length), and $\cdot[\cdot]$ look up a byte value in an array. Then an acceptable delete transaction is possible in Simple Staker only when

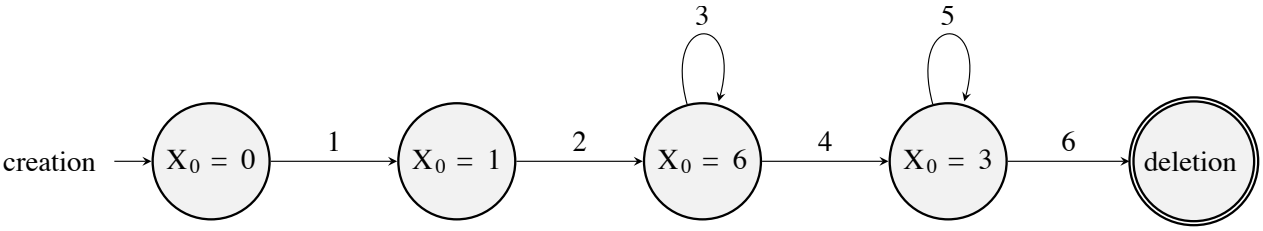
$$\lfloor X_0 \rfloor = 3 \wedge \lfloor A_0 \rfloor = 0 \wedge \lfloor A_1 \rfloor = 3 \wedge (\lfloor A_2 \rfloor = 0 \vee \lfloor A_2 \rfloor = \lfloor X_1 \rfloor)$$

and

$$|A_3| \leq 4092 \wedge A_3[0] \neq 0 \implies \mathsf{Sender} = Y_{0,32}$$

and the transaction is alone in its transaction group.

The only value out of the control of an attacker in this constraint is $\lfloor X_0 \rfloor$, a key component of the internal state of Simple Staker. It is therefore important that the global storage slot X_0 not take on the value 3 until it should. The audit team used Jade to extract the following state transition graph of X_0.



Jade's analysis demonstrated that no transitions other than those represented in the graph are possible. Its analysis was able to determine the precise conditions under which transitions 1, 2, and 6 occur, which were routine and matched expectations. It would be beneficial to determine the precise conditions under which transition 4 occurs and also to characterize transitions 3 and 5 by an invariant to demonstrate higher-level correctness properties. These transitions cover hundreds of transaction schemas and require more specialized analysis and so these results are out of the scope of this audit.

6 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.