LootSafe Smart Contract Audit
v1.0

New Alchemy

August, 2018

New Alchemy

Copyright 2018, New Alchemy Limited LLC

Introduction

During August 2018, LootSafe LLC engaged New Alchemy to audit the LootSafe Platform smart contracts. These contracts consist of three custom source files, reference two standardized EIP/ERC20 files and are supported by a testing environment containing several test cases.

The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts, finding differences between the contract's implementation and their behavior as described in public documentation, and finding any other issues with the contracts that may impact their trustworthiness. LootSafe provided New Alchemy with access to a detailed whitepaper and a GitHub repository containing the source code with an associated `README.md` file.

The audit was performed over three days. This document describes the issues discovered in the audit.

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug-free status. The audit documentation is for discussion purposes only.

Executive Summary

The LootSafe platform smart contracts consist of three custom source files, reference two standardized EIP/ERC20 files and are supported by a testing environment containing several test cases. The contract code was very well structured, very well commented, leveraged standardized and well tested libraries, and all 13 tests ran and passed out of the box.

The audit uncovered no critical issues, one moderate issues and six minor issues, primarily involving:

- Insufficient input validation that may allow internal contract data pollution.
- Unused contract variable and function modifier code.
- Several significant state-changing functions that do not emit events.
- Inadequate dependency management for standardized libraries.
- Several known EIP/ERC20 weaknesses.

While the test cases were not in scope, they were helpful to understand and explore issues. The whitepaper and `README.md` were considered the primary documentation for critical constants and intended functionality. Since large sections of base functionality is standard and leverages widely-used

New Alchemy

and reviewed contracts, most of the findings pertained to the code found in the custom contract files.

New Alchemy believes the way forward should include contract source corrections corresponding to the issues identified in this review, adoption of a robust dependency management strategy, and user documentation for mitigating the known EIP/ERC20 weaknesses. The absence of critical issues is a very positive leading indicator.

Files Audited

The code reviewed by New Alchemy is in the GitHub repository https://github.com/LootSafe/lootsafe.marketplace.contracts at commit hash `d87b8c89a8851971c90248e8b5415137320053d6`.

`contracts/Market.sol`	`ac147a333fe9b08067bb41e63fd425ecaed2e92b`
`contracts/Migrations.sol`	`19a7c00c1123db1875e307e7186c323b4092434b`
`contracts/Vault.sol`	`56cde2b4bfeb365e43a9667037f39f3c56e1c0f3`
`contracts/lib/Cellar.sol`	`3479c87b345a616bd64df157520402b64ecea254`
`contracts/lib/EIP20.sol`	`f74bcd48457fa08f890beb394b3dd5009b4a026a`
`contracts/lib/EIP20Interface.sol`	`3daf1c7e479a85f6a1b3e230429f568c081e4bcd`

New Alchemy's audit was additionally guided by the detailed LootSafe Whitepaper 2018[1]

[1] https://lootsafe.io/app/LootSafe_Whitepaper.pdf shasum = fed2cc84fdc9ded040866dbcf568f4c24bd8493e

New Alchemy

General Discussion

The LootSafe platform primarily revolves around the `Vault.sol` and `Market.sol` contracts. Each will be discussed below.

The `Vault.sol` contract serves as an account owned by a merchant and co-managed with a market (contract). An instance is created by a parent `Market.sol` contract which is then able to lock, unlock and transfer assets. The merchant owner is able to make withdrawals and redeem accidental ETH sent to the contract. Both parties are able confirm asset balances. This contract is well structured, well written and tested via `1_market_test.js`.

The assets managed by `Vault.sol` are EIP/ERC20 tokens provide by code leveraged from Consensys. The specific code is an interface file (`EIP20Interface.sol`) and the token contract itself (`EIP20.sol`). Leveraging battle-tested contract code is considered a best practice. However, the current code repository is not structured to retain the source and version of the external code. This makes tracking versions and outstanding vulnerabilities more difficult than necessary.

The `Market.sol` contract creates, cancels and fulfills listings. The listing structure is defined in `lib/Cellar.sol`. The `fallback()` function creates a new vault for a merchant who then creates a listing for a particular asset. This contract is also well structured, well written and tested via `0_vault_test.js`.

Note that the contracts do not use a SafeMath library. As a result, all calculations may be subject to overflow or underflow conditions depending upon supplied operands. This audit specifically explored this concern and found only one tangential issue (#1) where this was relevant. However, future code development could easily insert vulnerabilities that impact contract functionality. New Alchemy recommends a defensive practice of utilizing a SafeMath library.

New Alchemy

Critical Issues

There were no critical issues found.

New Alchemy

Moderate Issues

1. Insufficient Input Validation

Insufficient input validation may allow an attacker to pollute the contract's internal data structures, exploit downstream logic vulnerabilities and/or facilitate simple and easily preventable user mistakes.

The `lock_asset()` function in the `Market.sol` contract is shown below. This function records an amount of locked assets as shown below.

```
60 function lock_asset (address asset, uint256 amount) public only_parent {
61     locked_assets[asset] += amount;
62 }
```

The above function does not check the asset balance prior to locking assets. The asset balance should provide an upper limit for the `amount` parameter. As a result, this may cause subsequent issues when the `withdrawal()` function shown below is called:

```
75 function withdrawal (address asset, uint256 value) public only_merchant {
76     EIP20Interface i_asset = EIP20Interface(asset);
77     uint256 balance = i_asset.balanceOf(this);
78     uint256 locked = locked_assets[asset];
79     require(balance - locked >= value, "INSUFFICIENT OR LOCKED VAULT BALANCE");
80     i_asset.transfer(merchant, value);
81 }
```

If the amount of locked assets is larger than the balance, the left side of the inequality on line 79 will silently underflow, due to the absence of a SafeMath library. This will result in the success of the now nonsensical `require` statement. The protection this `require` statement was intended to provide is effectively bypassed.

New Alchemy recommends adding a balance check to the `lock_asset()` function. Utilizing a SafeMath library across all the contracts would also be very beneficial.

New Alchemy

Minor Issues

2. Unused Contract Variable

In the `Market.sol` contract, the contract 'owner' variable is declared, initialized at construction-time, but never subsequently utilized. It is possible that this variable is intended for use by external functionality, but this may also be an oversight (e.g. it may be intended for use in a subsequent function modifier). New Alchemy recommends a careful review and either delete the variable, implement related code or comment its intended use.

3. Unused Function Modifier

In the `Vault.sol` contract, the function modifier `multi_auth` is declared but never used. This modifier is shown below

```
36 /// @notice Allows both parties to execute
37 modifier multi_auth {
38     require(msg.sender == address(parent) || \
39     msg.sender == merchant, "UNAUTHORIZED MULTIAUTH SENDER");
40     _;
41 }
```

This may be an extraneous remnant from prior versions that ought to be removed. Alternatively, the non-usage may be an oversight that ought to be remedied. New Alchemy recommends a careful review and either deletion or application of the function modifier.

New Alchemy

4. Log All Significant State-Changing Events

The common security mantra of "deter, detect, delay and respond" provides a useful mental model for consideration during smart contract development. The 'detect' and 'respond' aspects are very dependent upon logging and auditing functionality. For these reasons, New Alchemy recommends emitting events whenever significant state-changing functionality is invoked (and perhaps further differentiating between success and failure for particularly sensitive operations).

This strategy is very well followed in the `Market.sol` contract.

In the `Vault.sol` contract, the `withdrawal()` and `transfer()` functions utilize the underlying EIP20 asset `transfer()` function to emit the `Transfer` event. However, the `lock_asset()` and `unlock_asset()` functions do not emit events.

New Alchemy recommends the addition of emitted events to the the `lock_asset()` and `unlock_asset()` functions in `Vault.sol`.

5. Inadequate Dependency Management

The primary contracts make excellent use of standardized functionality. However, a robust strategy for version control and dependency management for external files is lacking. As a result, it is difficult to definitively determine file versions, heritage and freshness. This will impact vulnerability awareness and resolution.

The `contracts/lib` directory contains standardized functionality sourced from Consensys. Since edits have been made to some files, it is no longer possible to easily compare hashes with the source project. Further, version history has been lost and it will be difficult to track the emergence of new vulnerabilities, their applicability and their resolution.

An attractive way forward is to consider the use of Git submodules. Github provides useful documentation on this topic[2]. *Note that this is an informational issue.*

6. Lack of Short-Address Attack Protection

Some Ethereum clients may create malformed messages if a user is persuaded to call a method on a contract with an address that is not a full 20 bytes long. In such a "short-address attack"[3], an attacker generates an address (or in this case an asset address) whose last byte is 0x00, then sends the first 19 bytes of that address to a victim. When the victim makes a contract method call, it appends the 19-byte address to `msg.data` followed by a value. Since the high-order byte of the value is almost certainly 0x00, reading 20 bytes from the expected location of the address in `msg.data` will result in the correct address. However, the value is then left-shifted by one byte, effectively multiplying it by 256 and potentially causing the victim to transfer a much larger number of tokens than intended. `msg.data` will be one byte shorter than expected, but due to how the EVM works, reads past its end will just return 0x00.

This attack affects methods that handle tokens and destination addresses, where the method parameters include a destination address followed immediately by a value. In the LootSafe contracts, there are several methods of concern, including:

```
Vault.sol:59  function lock_asset (address asset, uint256 amount) public only_parent {
Vault.sol:66  function unlock_asset (address asset, uint256 amount) public only_parent {
Vault.sol:75  function withdrawal (address asset, uint256 value) public only_merchant {
Vault.sol:87  function transfer (address to, address asset, uint256 value) public onl...t {
```

While the root cause of this flaw is buggy serializers and how the EVM works, it can be easily mitigated in contracts. When called externally, an affected method should verify that `msg.data.length` is *at*

[2] https://blog.github.com/2016-02-01-working-with-submodules/

[3] How to Find $10M Just by Reading the Blockchain https://blog.golemproject.net/how-to-find-10m-by-just-reading-blockchain-6ae9d39fcd95

least the minimum length of the method's expected arguments (for instance, `msg.data.length` for an external call to `transfer()` should be at least 68: 4 for the hash, 32 for the address (including 12 bytes of padding), and 32 for the value; some clients may add additional padding to the end). This can be implemented in a modifier. External calls can be detected in the following ways:

- Compare the first four bytes of `msg.data` against the method hash. If they don't match, then the call is internal and no short-address check is necessary.
- Avoid creating `public` methods that may be subject to short-address attacks; instead create only `external` methods that check for short addresses as described above. `public` methods can be simulated by having the external methods call `private` or `internal` methods that perform the actual operations and that do not check for short-address attacks.

Whether or not it is appropriate for contracts to mitigate the short-address attack is a contentious issue among smart-contract developers. Many, including those behind the OpenZeppelin project, have explicitly chosen not to do so. While it is New Alchemy's position that there is value in protecting users by incorporating low-cost mitigations into likely target functions, LootSafe would not stand out from the community if they also choose not to do so.

7. ERC20 Double-Spend Attack Vulnerability

The standard ERC20 interface, implemented in `EIP20Interface.sol`, has a design flaw: if some user Alice wants to change the allowance granted to another user Bob, then Alice checks if Bob has already spent his allowance before issuing the transaction to change Bob's allowance. However, Bob can still spend the original allowance before the transaction changing the allowance is mined, which thus allows Bob to spend both the pre-change and post-change allowances[4]. In order to have a high probability of successfully spending the pre-change allowance after the victim has verified that it is not yet spent, the attacker to waits until the transaction to change the allowance is issued, then issues a spend transaction with an unusually high gas price to ensure that the spend transaction is mined before the allowance change. More details on this flaw are available at https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/ and https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729.

Due to this flaw, safely using the ERC20 `approve()` interface to change allowances requires that:

1. Allowances only change between zero and non-zero (and never non-zero to another non-zero)
2. Allowances change at most once in a block

These restrictions allow Alice to safely change Bob's allowance by first setting it to zero, waiting for that transaction to be mined, verifying that Bob didn't spend its original allowance, and then finally setting the allowance to the new value. Requiring this sequence of operations in `approve()` would violate the ERC20 standard, though users can still take this approach even without changes in the `approve()` implementation.

There are two alternative approaches that contracts can take to mitigate this flaw:

[4] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#approve

- To make it more convenient to change allowances, New Alchemy recommends providing `increaseApproval()` and `decreaseApproval()` functions that add or subtract to the existing allowances rather than overwriting them. This effectively moves the check of whether Bob has spent its allowance to the time that the transaction is mined, removing Bob's ability to double-spend. Clients that are aware of this non-standard interface can use it rather than `approve`; using `approve()` remains open to abuse. This is approach is implemented by the current version of the OpenZeppelin library[5].
- Only allow `approve()` to change allowances between zero and non-zero and don't allow multiple changes to a user's allowance in the same block. In order to change Bob's allowance, Alice must first set it to zero, wait until that transaction is mined, verify that the original allowance was not spent, then finally set the allowance to the new value. Requiring this sequence of operations by implementing restrictions in `approve()` would violate the ERC20 standard, though users can still take this approach even without changes in the `approve()` implementation.

Since both approaches are outside of the ERC20 standard, both approaches require user cooperation to work properly. Accordingly, LootSafe should provide documentation advising its users on how they should manage other users' allowances.

[5] https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol

New Alchemy

Smart Contract ABI Reference

This section provides the fully-elaborated application binary interface (ABI) for the primary contracts as seen at runtime. The ABI defines how the contracts may be interacted with while running inside the EVM. The tables make for very easy inspection of top-level specifics including:

- The definitive reference of all exposed functions
- Correct input and output function signatures
- Presence of constructors and fallbacks as expected
- Correct state mutability (pure, view, nonpayable, payable)
- Extraneous functions which should be private

`Market.sol` ABI Reference

Function	Inputs Type:Name	Outputs Type:Name	State Mutability
base	-NONE-	address:–	view
cancel_listing	uint256:id	-NONE-	nonpayable
constructor()	address:_base	-NONE-	nonpayable
create_listing	address:asset uint256:amount uint256:value	-NONE-	nonpayable
fallback()	-NONE-	-NONE-	payable
fulfill_listing	uint256:id	-NONE-	nonpayable
listing_count	-NONE-	uint256:–	view
listings	uint256:–	uint256:id uint256:date address:merchant address:asset uint256:amount uint256:value uint256:status	view
owner	-NONE-	address:–	view
vaults	address:–	address:–	view

New Alchemy

`Vault.sol` ABI Reference

Function	Inputs Type:Name	Outputs Type:Name	State Mutability
constructor()	address:_merchant	-NONE-	nonpayable

Function	Inputs Type:Name	Outputs Type:Name	State Mutability
ether_saver	-NONE-	-NONE-	nonpayable
has_balance	address:asset uint256:value	bool:–	nonpayable
lock_asset	address:asset uint256:amount	-NONE-	nonpayable
merchant	-NONE-	address:–	view
parent	-NONE-	address:–	view
transfer	address:to address:asset uint256:value	-NONE-	nonpayable
unlock_asset	address:asset uint256:amount	-NONE-	nonpayable
withdrawal	address:asset uint256:value	-NONE-	nonpayable

Line by line comments

This section lists comments on design decisions and code quality made by New Alchemy during the review. They are not known to represent security flaws.

A. Market.sol

Line 1

It is recommended to lock pragma to a specific compiler version, it is seen as best practice to enable that feature on every smart contract. To remediate this issue modify the code to remove the caret symbol ^ as such:

```
1 pragma solidity 0.4.24;
```

Line 33

Contract logic involves the `block.timestamp` globally available variable. Note that this value can be manipulated by the block miner by approximately 30 seconds. It does not appear that fundamental contract integrity is dependent upon precise timestamps, but this vulnerability should be kept in mind as the code evolves.

New Alchemy

B. Vault.sol

Line 1

It is recommended to lock pragma to a specific compiler version, it is seen as best practice to enable that feature on every smart contract. To remediate this issue modify the code to remove the caret symbol ^ as such:

```
1 pragma solidity 0.4.24;
```

Line 19

The mapping should explicitly specify "public", "private", or "internal".

Line 52

Function `has_balance()` should be declared as a "view" function.

C. lib/EIP20.sol

Lins 59 and 69

Functions `balanceOf()` and `allowance()` should be declared as "view" functions.

D. lib/Cellar.sol

Line 13

It is recommended to use enums rather than numeric values to increase compile-time checking, avoid errors from incorrect values, and increase readability of code.

New Alchemy