Bob's Repair Contract Audit
v2.4

New Alchemy

April, 2018

Introduction

During the week of March 26th, 2018 Bob's Repair engaged New Alchemy to perform an assessment of the smart contracts used for their Crowdsale and Token (BOB). BOB acts as a replacement for the Skilled Trade Workers Platform (STWP), which is an automated SMS system designed to improve the process of finding a contractor for various home repair/maintenance services.

New Alchemy performed the audit over three calendar days and focused on the implementation of several contracts that include the BOB Token, minting, crowdsale, transfers, and token dropping. Bob's Repair provided New Alchemy with access to relevant source code, whitepapers, and technical notes regarding the contracts and airdrop functionality.

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 bugfree status. The audit documentation is for discussion purposes only.

Executive Summary

New Alchemy's review of the Bob's Repair contracts did not uncover any issues of critical severity. However, these contracts do not implement the actual Crowdsale process, opting to manually control payment and token distribution. The contracts create the BOB Token and give Bob's Repair the ability to distribute initial tokens, increase the quantity of tokens in the network in a proportionate manner, and manage transfers once the Crowdsale is finished.

Overall, the contracts comprise a small set of functionality. However, since most of it is standard and copied from widely-used and reviewed contracts, the overall number of vulnerabilities that New Alchemy found was low–specifically the design and custom portions of the code.

Bob's Repair corrected minor issues after receiving the report. However, the company voluntarily elected not to implement protections against short-address attacks or two-phase ownership transfer, providing an opinion statement for the reason that they chose not to do so. Additionally, Bob's Repair decided not to execute the Crowdsale itself automatically via smart contracts. A token cap is created in the Crowdsale contract, but it must be invoked via a parameter in JavaScript manually. For this reason it must be understood that purchasers are putting great trust in Bob's Repair's ability and willingness to distribute tokens from the Crowdsale correctly and with great care, as outlined in the whitepaper.

Files Audited

The code reviewed by New Alchemy is in the GitHub repository https://github.com/bobsrepair/crowdsale-newalchemy-audit at commit hash `3762230506a00f6d442687e98c464793565216f5`.

Bob's Repair made corrections after receiving the initial report, located at commit hash `5c789a40044190de0959158738bfbf92a9b93c76`. New Alchemy used this version to confirm updates to the contracts.

New Alchemy's audit was additionally guided by the public Bob's Repair whitepaper[1].

General Discussion

The Bob's Repair contracts implement a framework that they can use to manage distribution, creation, and transfer of the BOB Token. BOB is a fairly standard ERC20 and ERC827-compliant token and incorporates code from the OpenZeppelin project[2]. Since the contracts do not implement a smart contract based Crowdsale, New Alchemy could not audit this process and how well it adheres to the guidelines set forth in the whitepaper. As such, token purchasers must place significant trust in Bob's Repair to perform the Crowdsale honestly and correctly.

The code specifies Solidity version 0.4.18, and the most recent version is 0.4.20. Bob's Repair should upgrade to the latest version as soon as is feasible. During the retest, Bob's Repair indicated that while 0.4.18 is used for now, the contracts will be compiled with the latest version upon release.

[1] https://bobsrepair.com/docs/White-Paper.pdf
[2] https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/ownership/Ownable.sol

Moderate Issues

Bob’s Repair Crowdsale is not implemented via smart contracts

Bob’s Repair has created contracts to handle the generation and distribution of BOB Tokens, but does so as library calls to be invoked directly. While the contracts appear to be functionally correct, this forces purchasers to place their trust in Bob’s Repair ability and willingness to execute the Crowdsale correctly. Any error in this process would require manual intervention by Bob’s Repair and forces purchasers to have to trust that Bob’s Repair handles the distribution correctly, with the potential for a significant loss of Ether.

To fix this issue consider creating an additional contract that will be used for the execution of the Crowdsale. This contract should outline the token cap, vesting duration, and distribution breakdown for the Crowdsale.

Re-test results: Bob’s Repair has voluntarily elected not to address this issue. The Crowdsale contract defines a cap for the number of tokens, but it is not set to a value. Bob’s Repair has provided a response to this issue. Their response is not in any way endorsed by New Alchemy and has been made expressly by Bob’s Repair:

We voluntarily opted out of using a smart-contract-controlled calculation of received funds because we want to allow token buyers to purchase BOB tokens with Bitcoin, Dash, Litecoin and Credit-Card transfers, instead of just Ethereum. We also want them all to have the same experience by visiting the same site–that is why Ethereum purchases go through the same workflow as other cryptocurrencies. Also, our distribution contract and administration software allows tokens to be distributed to the list of addresses and amounts generated by our Crowdsale backend. We have hired a firm to do an additional audit of this list before the distribution. Also, the BOBCrowdsale contract implements protection from double distribution errors; so we are sure that no errors will arise during this process.

Minor Issues

Two-phase ownership transfer

In contracts that inherit the common `Ownable` contract from the OpenZeppelin project[3], a contract has a single owner. This owner can unilaterally transfer ownership of the entire contract to a different owner. However, if the owner of a contract makes a mistake in entering the address of an intended new owner, then the contract can become irrecoverably unowned.

In order to preclude this, New Alchemy recommends implementing two-phase ownership transfer. In this model, the original owner designates a new owner, but does not actually transfer ownership. The new owner then accepts ownership and completes the transfer. This can be implemented as follows:

```
contract Ownable {
    address public owner;
    address public newOwner

    event OwnershipTransferred(address indexed oldOwner, address indexed newOwner);

    function Ownable() public {
        owner = msg.sender;
        newOwner = address(0);
    }

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address _newOwner) public onlyOwner {
        require(address(0) != _newOwner);
        newOwner = _newOwner;
    }

    function acceptOwnership() public {
        require(msg.sender == newOwner);
        OwnershipTransferred(owner, msg.sender);
        owner = msg.sender;
        newOwner = address(0);
    }
}
```

Re-test results: Bob's Repair has voluntarily elected not to address this issue and has provided a response to it. This response is not in any way endorsed by New Alchemy and has been made

[3]https://bobsrepair.com/docs/White-Paper.pdf

expressly by Bob's Repair:

As stated in the audit report, the only situation when this two-phase transfer can help is in the event that a mistake is made by the owner of the token contract. To prevent such mistakes we only use software compatible with EIP55 (Mixed-case checksum address encoding) and in the event of a theoretically possible manual transfer of the ownership of said contract, we will triple-check the new owner's address.

We have decided, that in our case, the possibility of such a mistake, which can lead to the accidental loss of the control of the token contract is very low. In the case that we are required to implement other logic by any future partners to mitigate the risks involved in token ownership control and management, we can always publish a complementary contract to implement required logic.

Short-address attack protections

The following issue is one that many, including those behind the OpenZeppelin project, have explicitly chosen not to do address. However, it is New Alchemy's position that there is value in protecting users by incorporating low-cost mitigations into likely target functions.

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", an attacker generates an 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 effects methods that transfer tokens to destination addresses, where the method parameters include a destination address followed immediately by a value. In the Bob's Repair contracts, such methods include

- `BOBCrowdsale.mint`
- `BOBCrowdsale.mintTokens`
- `BOBPExchange.tokenTransferNotify`
- `BOBPToken.transfer`
- `BOBPToken.transferFrom`
- `BOBPToken.approve` (inherited from StandardToken)
- `BOBPToken.increaseApproval` (inherited from StandardToken)
- `BOBPToken.decreaseApproval` (inherited from StandardToken)
- `BOBToken.transfer` (two versions)
- `BOBToken.transferFrom` (two versions)
- `BOBToken.approve` (two versions, inherited from ERC827Token and StandardToken)
- `BOBToken.increaseApproval` (two versions, inherited from ERC827Token and StandardToken)

- `BOBToken.decreaseApproval` (two versions, inherited from ERC827Token and StandardToken)
- `BobToken.mint` (inherited from MintableToken)

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 least* the minimum length of the method's expected arguments (for instance, `msg.data.length` for an external call to `BOBToken.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, Bob's Repair would not stand out from the community if they also choose not to do so.

Re-test results: Bob's Repair has voluntarily elected not to address this issue and has provided a response to it. This response is not in any way endorsed by New Alchemy and has been made expressly by Bob's Repair:

We at Bob's Repair have examined currently available solutions to a potential short-address attack and the possible consequences of implementing a protection against this attack. We came to the conclusion that adding this protection may lead to a situation that some 3rd-party contracts fail to work with our token. This is discussed in OpenZeppelin Issue 261 thread . For example, a person or entity could try to send a zero amount of tokens. If an amount of zero tokens is sent it could cause an unknown error that we cannot predict. One other consequence of "manually" checking the length of each argument is that our contracts may become incompatible with future versions of EVM.

We believe that argument verification should happen on some other stages of User-to-Contract interaction. We are confident in the ability of major exchanges to correctly process accidental or deliberate errors caused by user input.

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.

BOBCrowdsale.sol

Line 45

Commented out parameter name results in an invalid function declaration. This should be either uncommented or removed entirely.

Re-test results: Bob's Repair fixed this problem by removing the comment.

Line 50

It is possible to overflow this check, given both parameters are uint256. Based on documentation, this is unlikely to happen but should be considered nonetheless. Use SafeMath for all arithmetic if possible.

Re-test results: Bob's Repair fixed this by switching to SafeMath for the addition operation.

AirdropToken.sol

Line 72

If this function is ever invoked with a value for `amount` that is not divisible by `PERCENT_DIVIDER`, the remainder will be truncated.

Re-test results: Bob's Repair opted not to fix this issue, since the potential remainder is likely to be very small.

BurnableToken.sol

Line 19

This check is superfluous since the contract is using .sub.

Re-test results: Bob's Repair elected to leave this check in for the sake of readability.