Badger Citadel contest
Findings & Analysis Report

2022-04-22

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Badger Citadel smart contract system written in Solidity. The audit contest took place between February 4—February 6 2022.

Wardens

40 Wardens contributed reports to the Badger Citadel contest:

1. Czar102
2. gellej
3. cmichel
4. pauliax
5. TomFrenchBlockchain

6. gzeon
7. pedroais
8. WatchPug (jtp and ming)
9. tqts
10. 0x1f8b
11. sirhashalot
12. hyh
13. harleythedog
14. hickuphh3
15. cccz
16. hubble (ksk2345 and shri4net)
17. defsec
18. csanuragjain
19. p4st13r4 (0x69e8 and 0xb4bb4)
20. 0x0x0x
21. lllllll
22. Dravee
23. OriDabush
24. Ruhum
25. NoamYakov
26. robee
27. floppydisk
28. samruna
29. wuwe1
30. kenta
31. peritoflores
32. Jujic
33. rfa
34. PostMan56

35. ych18
36. sabtikw
37. throttle

This contest was judged by 0xleastwood.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded 3 unique MEDIUM severity vulnerabilities. Additionally, the analysis included 24 reports detailing issues with a risk rating of LOW severity or non-critical as well as 22 reports recommending gas optimizations. All of the issues presented here are linked back to their original finding.

Notably, 0 vulnerabilities were found during this audit contest that received a risk rating in the category of HIGH severity.

Scope

The code under review can be found within the C4 Badger Citadel contest repository, and is composed of 1 smart contract written in the Solidity programming language and includes 384 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic

- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#).

Medium Risk Findings (3)

[M-01] The Owner and Proxy Admin can make users lose funds ("rug vectors")

Submitted by gellej, also found by WatchPug, Czar102, csanuragjain, p4st13r4, pedroais, TomFrenchBlockchain, defsec, hubble, gzeon, 0x1f8b, and sirhashalot

The contest explicitly asks to analyze the contract for "Rug Vectors", so that is what this issue is about.

I have classified this issue as "high risk" - although the vulnerability is considerable, the attacks themselves are not very likely to occur (they depend on the owner and/or the proxy admin to be compromised). The main reason why I believe the vulnerabiity is "high" is because the very fact that all these factors exist can make the sale fail, as informed users will avoid the contract completely one they realize the extent in which the contract is manipulable.

In the current implementation, there several ways that investors can lose funds if the owner of the contract is not well behaved. These risks can be divided into two kinds:

- owner becomes unable to act (for example, owner looses her private key, or the owner is a wallet or a DAO and signers cannot agree on the right action to take)
- owner is malicious (for example, the owner account gets hacked or the signers turn bad), and wants to steal as much as the funds as possible ("Rug Vectors"), or executes a griefing attack (i.e. acts in such a way to hurt the buyers and/or the project, without immediate financial gain)

The contract is vulnerable to all three types of vulnerabilities ("rug pull", "griefing" and "inactivity").

(1) Loss of funds due to owner inactivity: (1a) If the owner does never funds the contract, the buyers will not receive their tokens, and have no recourse to get their investment back (1b) If the owner does not call `finalize`, buyers will not receive their tokens, and and have no recourse to get their investment back

(2) Griefing attacks by the owner (attacks that that have no immediate gain for the attacker, but are either annoying or lead to loss of funds) (2a) the owner can change many essential conditions of the sale: for example, the price, the start time, the duration, the guest list, and the total amount of tokens that are allowed to be sold. The owner can do this at any moment, also **while the sale is in course**. This allows for all kinds of griefing attacks. It also voids the whole point of using a smart contract in the first place. (2b) Owner can pause the contract at any time, which will freeze the funds in the contract, as it also disallows users to claim their tokens

(3) Rug pull by owner (attacks with financial gain for the attacker, buyer loses money) (3a) The Owner can call `sweep` at any time and remove all unsold CTDL tokens while the sale is in progress. Future buyers will still be able to buy tokens, but the sale can never be finalized (unless the owner funds the contract) (3b) Owner can front-run buyers and change the price. I.e. the owner can monitor the mem pool for a large `buy` transaction and precede the transaction with her own transaction that changes the price to a very low one. If the price is low enough, `getAmountOut` will return `0`, and the buyers will lose her funds and not receive any CTDL tokens at all.

(4) Rug pull by proxy Admin (4a) Although no deployment script is provided in the repo, we may assume (from the tests and the fact that the contracts are upgradeable) that the actual sale will be deployed as a proxy. The proxy admin (which may not be the same account as the owner) can change the implementation of the contract at any time, and in particular can change the implemention logic in such a way that all the funds held by the contract can be sent to the attacker.

Recommended Mitigation Steps

- In general, I would recommend to not write your own contract at all, but instead use OpenZeppelin's crowdsale contract: https://docs.openzeppelin.com/contracts/2.x/api/crowdsale#Crowdsale which seems to fit your needs pretty well
- To address 1a and 3a, enforce that the contract is funded with enough CTDL tokens *before* the sale starts (for example, as part of the initialize logic)

- To adress 1b, simply remove the “onlyOwner” modifier on the “finalize()” function so that it can be called by anyone
- To (partially) address 2a, reduce the extent to which the owner can change the sale conditions during the sale (in any case remove the setSaleStart, setSaleEnd, setTokenInLimit or limit their application to before the sale starts). Ideally, once the sale starts, conditions of the sale remain unchanged, or change in a predictable way
- To address 2b, leave the tokens of the buyer in the contract (instead of sending them to a `saleRecipient` and implement an `emergencyWithdraw` function that will work also when the contract is paused, and that allows buyers can use to retrieve their original investment in case something goes wrong
- To address 3a, allow the owner to call `sweep` only after the sale is finalized
- To address 3b, either do not allow to change the token price during the token sale, or, if you must have this functionality, have the price change take effect only after a delay to make front-running by the owner impossible
- To address 4a, do not deploy the contract as a proxy at all (which seems overkill anyway, given the use case)

Oxleastwood (judge) decreased to Medium severity and commented:

> Awesome write-up!

> Because the issue outlined by the warden covers several separate issues from other wardens, I’ll mark this as the primary issue and de-duplicate all other issues.

Oxleastwood (judge) commented:

> I’ve thought about this more and I’ve decided to split up distinct issues into 3 primary issues:
>
> - Owner rugs users.
> - Funds are transferred to saleRecipient before settlement.
> - Changing a token buy price during the sale by front-running buyers by forcing them to purchase at an unfair token price.
>
> This issue falls under the first primary issue.

[M-02] `saleRecipient` can rug buyers

Submitted by WatchPug, also found by gellej, Czar102, hyh, harleythedog, pauliax, cmichel, 0x1f8b, hickuphh3, cccz, and sirhashalot

In `TokenSaleUpgradeable.sol#buy()`, `tokenIn` will be transferred from the buyer directly to the `saleRecipient` without requiring/locking/releasing the corresponding amount of `tokenOut`.

This allows the `saleRecipient` to rug the users simply by not transferring `tokenOut` and finalizing the sale.

Proof of Concept

Given:

- tokenIn: `WBTC`
- _tokenOutPrice: `1e8`
- tokenOut: `CTDL`
- Alice `buy()` with `100e8`;
- Alice `buy()` with `200e8`;

A malicious `saleRecipient` can just not transfer any `CTDL` to the contract and `finalize()` and keep `300e8 WBTC` received.

As a result, Alice and Bob can not get the expected amount of `tokensOut`, and there is no way to retrieve the WBTC paid, in essence, lose all the funds.

Recommended Mitigation Steps

Instead of transferring the tokenIn directing to the `saleRecipient` in `buy()`, consider transferring the tokenIn into the contract (`address(this)`), and require a sufficient amount of `tokenOut` to be transferred into the contract first before the amount of `tokenIn` can be released to the `saleRecipient`.

shuklaayush (BadgerDAO) disagreed with severity

Oxleastwood (judge) decreased severity to Medium and commented:

> As this is also an issue regarding abuse of an owner's admin privileges, it fits the criteria of a `medium` severity issue.

Oxleastwood (judge) commented:

> I've thought about this more and I've decided to split up distinct issues into 3 primary issues:
>
> - Owner rugs users.
> - Funds are transferred to `saleRecipient` before settlement.
> - Changing a token buy price during the sale by front-running buyers by forcing them to purchase at an unfair token price.
>
> This issue falls under the second primary issue.
>
> - Funds are transferred to `saleRecipient` before settlement.

[M-03] Owner can steal input tokens

Submitted by Czar102, also found by gellej, cmichel, tqts, gzeon, TomFrenchBlockchain, pauliax, and pedroais

TokenSaleUpgradeable.sol#L299-L309
TokenSaleUpgradeable.sol#L311-L324
TokenSaleUpgradeable.sol#L211-L224

Owner is in full control over the `saleRecipient` address. When a `buy()` transaction enters the mempool, an owner can frontrun the buy with a transaction that calls `setTokenOutPrice()` and sets the price to a very high value, effectively making bought tokens close to (if not usually equal) zero and consuming the tokens to the owner-selected address - `saleRecipient`. Thus, an owner has incentive to perform such attack as they may cause little or zero additional indebtnees to the contract and all tokens to the owner.

This can also be seen as a coincidence - an owner sets a price while a user broadcasts a `buy()` transaction. The user may buy for a significantly different price than they intended.

Recommended Mitigation Steps

Do not let changing sale price after the sale has started. Do not let changing sale start if the sale has already started.

GalloDaSballo (BadgerDAO) disagreed with severity and commented:

> I agree with the finding and the conclusion, we shouldn't let the price change after the sale has started.

> As for the example, that would only work once, because if we actually did that it would immediately warn every other user of our malicious behavior.

> I think the finding is valid and it's a clear example of admin privilege, so I believe medium severity to be more appropriate

0xleastwood (judge) decreased severity to Medium and commented:

> I agree with the sponsor on the above.

0xleastwood (judge) commented:

> I've thought about this more and I've decided to split up distinct issues into 3 primary issues:
>
> - Owner rugs users.
> - Funds are transferred to saleRecipient before settlement.
> - Changing a token buy price during the sale by front-running buyers by forcing them to purchase at an unfair token price.

> I believe this satisfies the third primary issue description.
>
> - Changing a token buy price during the sale by front-running buyers by forcing them to purchase at an unfair token price.

Low Risk and Non-Critical Issues

For this contest, 24 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by warden **Czar102** received the top score from the judge.

The following wardens also submitted reports: 0x0x0x, gellej, cmichel, Dravee, 0x1f8b, hubble, OriDabush, pauliax, IIIIIII, sirhashalot, NoamYakov, tqts, WatchPug, hyh, Ruhum, floppydisk, csanuragjain, defsec, gzeon, samruna, wuwe1, robee, *and* kenta.

Codebase Impressions and Summary

Several minor changes have been identified that can be applied in order to improve general security of the contract logic and code quality.

Among those, three out of four issues focus on code clarity, conventions and unambiguity of the comments. A possible attack vector has also been recognized, which makes the owner capable of griefing users and making their transactions revert, consuming gas fees.

[L-01] Ambiguous usage of `^` operator

Is Solidity `^` is used for `xor` operation, but in TokenSaleUpgradeable.sol:32 it is used to symbolize exponentiation. It is preferable to use `**` instead to avoid ambiguity or confusion.

```
// TokenSaleUpgradeable.sol:32
/// eg. 1 WBTC (8 decimals) = 40,000 CTDL ==> price = 10^8 / 40,
```

[L-02] Owner can frontrun `buy` function

Owner can frontrun `buy` function in order to cause transaction to fail (and as a consequence make someone lose gas fee) by invoking one of these functions:

- `setTokenInLimit`, with a small `_tokenInLimit` argument,

- `setSaleDuration`, with a small `_saleDuration` argument,
- `setSaleStart`, with a future timestamp.

Recommended Mitigation Steps

Revert calls to these functions after the sale has started.

[N-01] Open TODO

An open TODO is present in TokenSaleUpgradeable.sol:13. It is recommended to avoid open TODOs as they may indicate programming errors that still need to be fixed.

```
// TokenSaleUpgradeable.sol:13
TODO: Better revert strings
```

[N-02] Inconsistent naming conventions

The name of the variable `guestlist` (defined in TokenSaleUpgradeable.sol:53) and the event `GuestlistUpdated` (defined in TokenSaleUpgradeable.sol:76) should be changed to `guestList` and `GuestListUpdated` respectively in order to make them more readable and consistent with other parts of the code.

```
// TokenSaleUpgradeable.sol:53
BadgerGuestListAPI public guestlist;
```

```
// TokenSaleUpgradeable.sol:76
event GuestlistUpdated(address indexed guestlist);
```

GalloDaSballo (BadgerDao) commented:

> Appreciate the findings.
>
> Pretty sure `^` in a comment gives same level of clarity

> As for frontrun, I guess we could do that, but why?

Gas Optimizations

For this contest, 22 reports were submitted by wardens detailing gas optimizations. The report highlighted below by warden IIIIIII received the top score from the judge.

The following wardens also submitted reports: Czar102, TomFrenchBlockchain, Dravee, WatchPug, 0x0x0x, peritoflores, hyh, pauliax, Ruhum, Jujic, rfa, gzeon, defsec, tqts, NoamYakov, robee, OriDabush, PostMan56, ych18, sabtikw, *and* throttle.

[G-01] Use `!= 0` rather than `> 0` for unsigned integers in `require()` statements

When the optimizer is enabled, gas is wasted by doing a greater-than operation, rather than a not-equals operation inside `require()` statements. When Using `!=`, the optimizer is able to avoid the `EQ`, `ISZERO`, and associated operations, by relying on the `JUMPI` that comes afterwards, which itself checks for zero.

Examples

See markdown file within original submission for in-depth examples.

Tools Used

Hardhat Forge npx @remix-project/remix-lib

Recommended Mitigation Steps

Use `!= 0` rather than `> 0` for unsigned integers in `require()` statements. Note that the comparison in `claim()` results in no gas savings.

[G-02] - Use local variables to cache results of storage reads

Reading from storage is expensive whereas reading from the stack is cheap. If the result of a storage read in required multiple times, the code should cache the value in a local variable, and read from that variable, rather than fetching from storage again.

Examples

See markdown file within original submission for in-depth examples.

Tools Used

Hardhat Forge npx @remix-project/remix-lib

Recommended Mitigation Steps

Cache the result of storage reads in a stack variable for future reads

[G-03] - Pre-calculate repeatedly-checked offsets

Reading from storage is expensive, so it saves gas when only one variable has to be read versus multiple. If there is a calculation which requires multiple storage reads, the calculation should be optimized to pre-calculate as much as possible, and store the intermediate result in storage.

Examples

See markdown file within original submission for in-depth examples.

Tools Used

Forge

Recommended Mitigation Steps

Pre-calculate the end timestamp, and reference that rather than adding `saleStart` and `saleDuration` in multiple places. Note that due to packing, the variable is sensitive to the order in which it is defined in the contract, as well as its visibility. Also note that while I've split the separate instances, in reality you'd apply both of them and get a larger ammortization benefit

[G-04] - Use `unchecked` for operations not expected to overflow

If it is not possible for an operation to overflow, it should be wrapped in `unchecked {}` to save the gas that would have been used to check for an overflow.

Examples

See markdown file within original submission for in-depth examples.

Tools Used

Hardhat Forge npx @remix-project/remix-lib

Recommended Mitigation Steps

Add `unchecked { }` and casts to operations that cannot overflow

[G-05] - Pull tokens rather than pushing them

It wastes gas to push tokens to `saleRecipient` every time there is a buy.

Examples

1. TokenSaleUpgradeable.sol:L183

```
            tokenIn.safeTransferFrom(msg.sender, saleRecipient, _tok
```

Recommended Mitigation Steps

Create a new function or modify `sweep()` to send the current balance of the contract to `saleRecipient`

GalloDaSballo (BadgerDao) confirmed and commented:

> All findings are valid and appreciated, would recommend the warden to just post the specific gas differences instead of the whole table as there's 10s of thousands of lines in the readme, but the only lines that count are very few.

> That said, this may be the best report I've ever seen, as while it's a lot of superfluous information (again just delete the redundant lines), the information is all there and verifiable.

> I think the work from the warden is commendable and it is really appreciated.

> A breath of fresh air compared to the usual copy paste “use != instead of >”

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top