Contract Audit: Kranz

Preamble

This audit report was undertaken by @adamdossa for the purpose of providing feedback to the UniTrade team. It has been written without any express or implied warranty.

This audit was done on the code committed to Github as:
https://github.com/UniTradeApp/kranz-coin/tree/c467da3ab6c20ce6c1508d1f9806d86dd0bca617
which matched the branch `audit-may17` at the time of writing.

Disclosure

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. There is no owed duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

Classification

- **Comment** - A note that highlights certain decisions taken and their impact on the contract.
- **Minor** - A defect that does not have a material impact on the contract execution and is likely to be subjective.
- **Moderate** - A defect that could impact the desired outcome of the contract execution in a specific scenario.
- **Major** - A defect that impacts the desired outcome of the contract execution or introduces a weakness that may be exploited.
- **Critical** - A defect that presents a significant security vulnerability or failure of the contract across a range of scenarios.

Audit

This audit covers the contract `Kranz.sol`.

Contract Behaviour

Kranz.sol implements an ERC20 token, which takes a variable (depending on the target address) fee on each transfer, which is partially distributed to three wallets, with the remainder burnt.

Issues

Moderate - User can transfer less than expected

On each transfer, Kranz takes a fee (defaulted to 1% for non-DEX addresses, 3% for DEX addresses). It then splits this amongst the reward, developer, liquidity and burn addresses.

When calculating the split, each amount is calculated independently. Since Solidity will round down multiplications to the nearest integer, it is possible that the sum of the amounts transferred to the reward, developer, liquidity and burn addresses do not match the expected fee.

As a simple example, if the user tries to transfer 100 base units of Kranz, the expected fee would be 1 unit (1%).

Since 1 multiplied by any percentage (strictly less than 100%) will result in 0, the user will end up transferring 99 units, and no fee will be levied.

Mitigation

In the `getFees` function, replace:
`toBurn = amount.mul(_burnPercent).div(1e18);`
with:
`toBurn = amount.sub(toReward).sub(toDeveloper).sub(toLiquidity);`

Client Response

Resolved by client using the above mitigation - see commit:
https://github.com/UniTradeApp/kranz-coin/commit/7beb29dfa18bd8be9303a88da1fb85e2ddb822a7

Minor - Contract only permits a single DEX address

Kranz has a `address private _uniswapAddress;` which is matched against for each transfer, to determine whether a 1% or 3% fee should be levied.

The intention is that this address would be set to the Uniswap V2 pool for the `[KRANZ, WETH]` pair (or equivalent on BSC).

However, it is possible for there to be multiple Uniswap V2 pools for KRANZ (e.g. `[KRANZ, USDC], [KRANZ, DAI]`), and in addition there may be other DEX addresses (e.g. Uniswap V3 pool) that should also attract the higher 3% fee.

Mitigation

Rather than storing a single address `_uniswapAddress`, consider storing a mapping:
`mapping (address => bool) dexAddresses;`

This would allow you to specify multiple DEX addresses which would all attract the 3% fee.

I did check whether it would be possible to dynamically look up whether a particular address is a Uniswap pool, but it doesn't seem like this is possible since the Uniswap V2 Factory stores pool addresses in an array rather than a mapping.

Client Response

Resolved by client using the above mitigation - see commit:
https://github.com/UniTradeApp/kranz-coin/commit/7beb29dfa18bd8be9303a88da1fb85e2ddb822a7

Minor - Contract owner can front-run transfers with fee changes

The contract allows the owner to modify the contract transfer fees. The owner could monitor the transaction pool for a large transfer, then front-run it with a change to increase the fees, leaving the user to pay a higher than expected fee. In general when interacting with the token, the user doesn't have any guarantee that the fees won't be modified before their transfer is included in a block.

It should be noted that the fee changing functions do limit the maximum fee to 5% of the transferred amount.

Mitigation

Consider modifying fees with a time lock approach. An initial transaction could specify the new fees, and a subsequent transaction (which must be more than a fixed number of blocks later) can then update the fees.

Client Response

Resolved by client using the above mitigation - see commit:
https://github.com/UniTradeApp/kranz-coin/commit/7beb29dfa18bd8be9303a88da1fb85e2ddb822a7

Minor - Tokens sent to the BURN address are not really burnt

A small percentage of each token transfer is intended to be burnt (50% of either 1% or 3% by default).

However, rather than burning the tokens, the tokens are instead sent to an unreachable address (`0x000000000000000000000000000000000000dEaD`). This is not the same as burning tokens, as the total supply is not correspondingly reduced.

This issue depends on what you expect the behaviour to be in this instance - it may be that you want to keep the total supply constant, but just permanently reduce the circulating supply, in which case the issue can be ignored.

Mitigation

Consider burning tokens by using the `ERC20Burnable` contract from openzeppelin. This function reduces the total supply as well as reducing the targets balance by the specified amount.

Client Response

Resolved by client using the above mitigation - see commit:
https://github.com/UniTradeApp/kranz-coin/commit/7beb29dfa18bd8be9303a88da1fb85e2ddb822a7

Comment - Consider allowing a 0% fee

Currently the `setTransactionFee` and `setTransactionFeeDex` enforce that the fee is strictly greater than 0%. I don't see any particular reason to not allow the owner of KRANZ in the future to specify a 0% fee.

Mitigation

Remove the check that the fee is strictly greater than 0 in:

```
require(
    fee > 0 && fee <= 5e16,
    "Krans: sell transaction fee should be > 0 and <= 5%"
);
```

Client Response

Resolved by client using the above mitigation - see commit:
https://github.com/UniTradeApp/kranz-coin/commit/7beb29dfa18bd8be9303a88da1fb85e2ddb822a7

Testing

The repo contains detailed test cases that cover the majority of the contract functionality, with 3 unit tests included which all pass.

The tests don't currently cover the admin functions which modify fees, DEX addresses and wallet values - you should consider adding test cases for these, even though the functions are largely trivial.