Dracula Protocol Security Review

by Valentin Mihov

16th October 2020

Summary

I did a security review of Dracula Protocol smart contract in the interval 13th-16th October 2020 for a total of 4 man/days. The review was done on the repository with SHA 9f1c590b0031c3628c4c67f2f8edb4e6b4defb11. The code of the protocol is a fork of the famous Sushiswap protocol with significant changes around some of the areas.

The review didn’t find any major security issues in the contract. There is one medium level issue, which allows to arbitrage the `drain` function market orders against Uniswap and several low level vectors of attack, but all of these are infeasible to execute as they need to go through a timelock embedded in the contract and can be detected in time.

The participants in the DRC/ETH pool should be aware there is a high concentration of `DRC` tokens into the dev treasure address, which can be potentially sold on the market, which will tank the price of the `DRC` token. This is what happened with the `SUSHI` tokens. The dev treasury is located at 0xa896e4bd97a733f049b23d2aceb091bce01f298d.

Everyone that participates in the protocol should monitor all timelock transactions and make sure they are not replacing the adapters of the pools with invalid/malicious contracts. The timelock transactions can be easily monitored here 0xd12d68fd52b54908547ebc2cd77ec6ebbefd3099.

About the author

Valentin Mihov is a software engineer with 25 years of coding experience. Participated in the International Olympiad in Informatics in 2004 as the captain of the bulgarian team. He's been the CTO of Santiment for 3 years, where he's been doing analytics of decentralized protocols and networks. Since July 2020 he is working as a consultant on decentralized protocols, risk assessment of smart contracts and investment opportunities in DeFi.

Security reviews vs audits

This document is a security review not a full security audit. The difference is that security audits are much more deep and performed by entities that are specializing into formal auditing. A security review is an informal version of an audit. In this particular case the review is performed by a software engineering professional, who is not specializing into Solidity security auditing, but performs such reviews for clients, who need security risk assessmet of protocols and can't afford the time and resources required for a full security audit.

Scope of the review

The scope of the review is the following:

- Check all verified contracts against the source code in GitHub and validate they match exactly, including any third party libraries
- Comparison of the main contract with the Sushi contracts and manual analysis of all the differences
- Run slither static analysis tool on the deployed contracts and analyze the warnings
- Inspection of the state of the contracts and making sure there is no configuration, which could lead to exploits
- Manual analysis of all contracts and checking them for backdoors and bugs

Overview of the protocol

The Dracula Protocol main contract (`MasterVampire`) is a fork of the SushiSwap `MasterChef` contract. The original contract can be found here:sushiswap/MasterChef.sol

The original MasterChef contract of Sushi was audited by PeckShield. You can read the audit here: publications/PeckShield-Audit-Report-SushiSwap-v1.0.pdf. The original contract does not contain any major issues. Many of the remarks in that audit apply also to the Dracula Protocol.

The Dracula Protocol allows users to stake tokens representing liqudity in different DEX protocols and earn rewards in DRC tokens. The rewards paid are determined by the rewards allocated to each pool and the percent of the funds that the user owns in the pool. When a deposit is done into a particular pool, the funds are forwarded into a "victim" contract, using a special proxy contract called "adapter". The victim contracts are other yield farming contracts. This makes the Dracula protocol a staker in the "victim" contracts and allows it to harvest the rewards from these protocols. There is a special "drain" function, which harvests the rewards from a given "victim" contract, sells the rewards on Uniswap for ETH, then purchases DRC with the acquired ETH and burns these tokens. This creates an "upwards" buy pressure on the DRC token and potentially rewards its holders as the amount of DRC tokens in circulation decrease and the price of the token potentially increases.

As of this writing, there are a total of 69.7 DRC given as rewards per block and on average 29.1 DRC being burned per block. This makes the token supply inflationary at the moment. The team has been decreasing the token rewards in the last days, so may be we will even see a deflationary period if the amount of tokens burnt becomes larger than the amount of tokens minted.

Risk level compared to yVault-like contracts

The architecture of Dracula Protocol is a simpler version of the yVault-like contracts, like pJars from the Pickle Finance and Harvest Finance. This is because there is no accounting for the earnings, which the contract is making from farming the underlying protocols. The way the earnings are distributed to the participants is through the potential appreciation of the price of the DRC token. This appreciation is achieved by buying back the tokens from the market and burning them. By adjusting the inflation rate of the DRC token, the authors of the protocol are able to potentially adjust the ratio between rewards paid and the buy pressure on the token.

Given the above I would put the risk level of the Dracula Protocol to be somewhere in the middle between simple synthetix-based staking protocols and the yVault-like systems.

List of deployed contracts

- `MasterVampire` - 0xd12d68fd52b54908547ebc2cd77ec6ebbefd3099
- `DraculaToken` - 0xb78b3320493a4efaa1028130c5ba26f0b6085ef8
- `SushiAdapter` - 0xc9e87b02b5e9526b1a11d54abafbde4e74928375
- `LuaAdapter` - 0x5085ac38d03ef07665f9b1adebd14b6e32cab11d
- `YfvAdapter` - 0x5f389d2f80cdf4824350f8f9fba04123e5e0730a

- `UniswapAdapter` - 0x6bb3ac9a221de66e27fa524a110242c522eaf88c
- `PickleAdapter` - 0x2736680f6292897fcaaeb69c0178cf226e67485b
- `DrainController` - 0x2e813f2e524dB699d279E631B0F2117856eb902C

I verified that all of these contracts match the contracts from github repository tree 9f1c590b0031c3628c4c67f2f8edb4e6b4defb11 by comparing the verified contracts on Etherscan with the flattened contracts from the repository.

Potential vectors for attack

In this section I am describing potential issues and vectors to attack the protocol. These are low probability issues either because it is not practically possible to execute them or they are attacks that need to happen through the timelock, which allows the users to detect and react to them.

Sandwich price arbitrage in the drain function

The drain function harvests the rewards from the victim's staking pool, sells that reward for ETH, then it purchases DRC using the ETH and burns that DRC. It is possible to manipulate the price of the DRC/ETH pair before calling the `drain` function and make the contract purchase DRC tokens at an inflated price. Such an attack can be executed using a flash loan of ETH and it has actually been executed already. You can see an analysis of such an arbitrage using a 2300 ETH flash loan: Ethtx.info Analysis

The arbitrage will be bigger if the amount of funds being sold in the drain function is larger. The attack can be mitigated and will probably never reach large proportions as the arbitrageurs are competing for getting the arbitrage opportunity as it grows bigger and bigger. The above flash loan arbitrage earned 0.46 ETH and required 0.29 ETH for gas, so the total earnings was 0.17 ETH.

The team fixed the arbitrage opportunity by adding an oracle check, which is currently rejecting the drain if the current price is bigger than 30% of the average price in the last hour. The fix was done in this commit Dracula-Protocol/contracts@9f1c590 and then the adapters of all pools were updated through the timelock.

Drain or block funds through a malicious adapter

There is no migration logic in the `MasterVampire` contact. This is a standard attack vector for Sushi clones. The removal of the migrator logic ensures that this can't be used for draining the funds.

Instead of the migrator, there is a similar attack vector, by using the adapters for each victim pool. All the staked funds are going through the adapter of each pool, so the admins of the contract can set the adapter to a syphon contract, which just transfers them to an EOA. In order for this to happen the admins will need to call the `updateVictimInfo` function, which is protected by a timelock. The existence of the timelock makes executing this attack much more difficult.

The participants in the pools are encouraged to monitor the timelock transactions and make sure there are no transactions that try to replace the adapter with a malicious one.

Mint a lot of DRC tokens through high rewards per block

There is a potential infinite mint attack, which requires to do the following actions:

- Set the `poolRewardLimiter` to a very high number (ex. 2^32)
- Add a pool with a very high `rewardPerBlock`
- Supply tokens to the pool
- Immediately harvest the rewards and use the dev funds + the newly minted DRC tokens to market sell on the DRC/ETH uniswap pool and drain more of the ETH

There is a very low probability such an attack can be carried out undetected, as it will require to do the first 2 actions through the timelock of the contract and users will be able to withdraw within 24 hours from when the attack is initiated and protect their funds. This makes the attack not feasible in practice.

Lack of a timelock on the dev treasury

The dev treasury currently holds 740000 DRC tokens, which are valued at $180000. The owner of this address is an externally owned address (EOA), which allows the holder of the address to sell the tokens on the market without a warning. This is what happened with the SUSHI dev treasury and caused a large price decrease of the token. It will be better if the dev address is migrated to a timelocked smart contract. This has its own problems as knowing upfront that a large amount of tokens will be sold on the market opens the possibility for abitrage. A possible solution will be to put the treasury in a timelock and when certain amount of tokens need to be sold, they are transfered to an EOA and this address sells the tokens in batches in time which is not known upfront.

Other findings and potential improvements

In this section I'll highlight some observations I had while reviewing the contracts. These are not major issues, but only interesting observations and potential improvements.

Difficulty to monitor the Timelock transactions

The current implementation makes it hard to monitor the timelock transactions, as the timelock is embedded in the main staking contract. This makes it hard to easily see the transactions that interact with the timelock in etherscan. My recommendation is to use an alternative tool, which allows to list all the `queueTransaction` events and examine the decoded arguments: 0xd12d68fd52b54908547ebc2cd77ec6ebbefd3099

A potential improvement for future versions of the protocol will be to make the timelock an external contract, which will make it easier to monitor the transactions directly on etherscan.

Different reward calculation compared to Sushi

The reward calculation logic is changed. In the original SUSHI contract, the reward logic relied on allocation points to compute what percentage of the total rewards each pool is getting. In the `MasterVampire` contract this is changed and the reward logic relies on a fixed reward per block, which is set for each pool. There is no bonus period, which makes the calculations simpler. There is no need to compute multipliers across block ranges, which mitigates problems around times when the multipliers changes.

Built-in Compound timelock

There is a timelock built in. The timelock is a fork of the Compound standard timelock contract with a change of the syntax of the `call` method, that conforms with the latest version of Solidity. You can read more here Release Version 0.6.4 · ethereum/solidity

The timelock allows to set the admin one time without waiting for the timelock delay. This functionality is still not used, as the `admin_initialized` flag is `False`. This flag will allow to change the address, which is able to submit delayed transactions to the contract. There are no security implications of this, as all the admin actions will need to go through the timelock before coming into force.

The current delay of the timelock is 24h and this is the minimal delay of the timelock. This is a sane value, which should allow participants to have enough time to react if the owner decides to do some malicious action.

Small gas optimizations to prevent 0 transfers

There are few places where checks are added for 0 amount transfers. These checks ensure that no 0 amount transfers will be initiated. This will save gas in some cases.

Proxy pattern for deposits/withdrawals

Each registered pool has an address to which the deposits/withdrawals are forwarded. The forwarding is done using `delegatecall` for the mutating calls and `staticcall` for the non-mutating function calls. The calls are delegated to adapters, which are implemented for each supported farming protocol. This is the largest change in the protocol. It is important to note that the proxy pattern used in this protocol is much simpler than using a yVault-like approach, where each deposit issues shares representing a portion of all locked funds in the protocol. Using the proxy pattern allows to delegate the accounting to the underlying farming protocol, keeping the `MasterVampire` contract simpler.

Unlimited approval for spending the `MasterVampire` funds by the underlying farms

The `deposit` function in the `MasterVampire` contract is issuing an unlimited approval for the underlying contract to spend its tokens. Given that the approval is done every time, it will be safer to issue an approval only for the amount being deposited. That could prevent potential security issues if the underlying farm contract is compromised, while keeping the gas cost the same.

The emergencyWithdrawal is not using the corresponding function in the underlying protocols

The `MasterVampire` contract has an `emergencyWithdrawal` function, which can be used to get your staked funds back in case something goes wrong with the reward computation logic. The adapters of the underlying farms are implementing this function in their interface, but that function is not used. Instead a regular withdraw and transfer of the locked funds is done. This is not going to protect the Dracula staker from potential bugs in the underlying farm reward logic bugs.

The current implementation of the `emergencyWithdraw` is also going to leave reward tokens in the contract. These tokens will be sold on the next `drain` call, so these rewards will eventually be used for buying back `DRC` tokens.

My recommendation will be to use the `emergencyWithdraw` function in the adapters, so that the reward bug protection to be included in the Dracula protocol. In case there is no emergency withdraw function in the underlying farm, the implementation in the adapter can use a regular withdraw.

Special priviledge actors

The contract contains 3 special roles that have elevated privileges. In the following sections I am outlining each role and its privileges.

Owner

Currently this address is the contract itself, which implements a timelock with 24h delay. This is a special privilege which allows to call the following functions:

- `add` - add a reward pool to the contract, specifying the adapter that needs to be used and the rewards per block, which the pool is paying.
- `updatePoolRewardLimiter` - there is an upper limit on the amount of rewards being paid per pool. This limit can be changed by the admin.
- `updateVictimInfo` - this is a very powerful function as it allows to update the destination where the staked funds can end up. This could allow for stealing the funds by the admins, if they set a siphon contract, which is under their control. The issue is mitigated by the timelock. The users of the contract should monitor the timelock for suspicious calls to this function.
- `updatePoolDrain` - adjusts the percents of the victim rewards that will be sold for ETH and the percent of the sold ETH, which will be transferred to a drain address. The drain address has a similar idea as the dev treasury. Part of the earnings of the protocol can be directed to that address.
- `updateDrainAddress` - sets the address to which part of the drained rewards will be redirected. Currently this address is the same as the address of the admin of the contract. It is the same address which deploys and manages the whole system
- `updateRewardUpdaterAddress` - sets the address which is allowed to update the rewards of the pools

Reward updater

This is an address which is allowed to set the reward size of each pool. There is an upper limit for the max size of the rewards which this address can set. The role can call the following functions:

- `updateRewardPerBlock` - update the rewards per block for a given pool
- `updateRewardPerBlockMassive` - update the rewards per block for a list of pools

Developer

This is an address to which all the dev rewards are redirected to. The amount of dev rewards is hardcoded to 8% of all minted `DRC`. This role can call the following functions:

- `updateDevAddress` - updates the dev address to a new address

Lack of reentrancy guards

It will be good to put re-entrancy guards on the main public functions. The reentrancy danger is elevated for this contract, as it calls external contracts which can be changed by the owners. It is easy for an reentrancy bug to slip in while adding new adapters.

Static analysis findings

The static analysis tool Slither was used on the deployed contracts. The tool tries to find common problems in solidity contracts. There are no significant issues found by the tool. Here is a list of the relevant warnings:

- No check of the return value of the `approve` action in the adapters, before the reward tokens are swapped on Uniswap. The check is not needed as if the approval fails, the subsequent swap will also fail and the whole transaction will fail, thus there is no need to handle that particular error case. Relevant code: https://github.com/Dracula-Protocol/contracts/blob/9f1c590b0031c3628c4c67f2f8edb4e6b4defb11/adapters/lua/LuaAdapter.sol#L49
- Use of `delegatecall` on a dynamic address in the `MasterVampire` contract. This opens the possibility for executing a `delegatecall` on an address containing a malicious contract. The threat is mitigated by making sure the `victim` address cannot be changed to a malicious contract. The change of this address is behind a 24h timelock, so users can see the change and have time to withdraw funds before the malicious contract enters into force. Relevant code: https://github.com/Dracula-Protocol/contracts/blob/9f1c590b0031c3628c4c67f2f8edb4e6b4defb11/VampireAdapter.sol#L52
- Reentrancy warnings on the `MasterVampire` contract. These warnings are quite relevant and I think it will be a very good improvement to add reentrancy guards to the functions that do fund movement. The current adapters does not have a possibility to callback the main contract, but as new adapters are added, it's possible such situation to happen. There are no obvious reentrancy explotations at the moment, but it is definitely something to be aware of. Relevant code: https://github.com/Dracula-Protocol/contracts/blob/9f1c590b0031c3628c4c67f2f8edb4e6b4defb11/MasterVampire.sol#L189
- Ignored return value for a bunch of `transfer` calls in the `MasterVampire` contract. These are either transfers, which transfer reward tokens and are not mission critical or transfers, which if are not carried out successfully will fail subsequent operations. Relevant code: https://github.com/Dracula-

Protocol/contracts/blob/9f1c590b0031c3628c4c67f2f8edb4e6b4defb11/MasterVampire.sol#L300

Disclaimer

This security review does not give any warranties that there are no bugs in the reviewed contracts. This security review is different from a security audit, as it is not so thorough and comprehensive as an official audit. The author did his best to review the contract fairly, independently and to the best of his knowledge. It is advisable to perform several security audits/reviews and to create a bug bounty program. This security review is not an investment advice.