Drosera Trap Guidebook - Anthias Labs

charlie@anthias.xyz aaron@anthias.xyz vasu@anthias.xyz

November 20, 2024

Contents

1 Introduction

Decentralized Finance (DeFi) has emerged as a pivotal innovation in blockchain technology, with lending and borrowing protocols forming a core component of its infrastructure. However, these protocols face ongoing challenges related to security, particularly as adversarial actors develop new techniques to exploit vulnerabilities and siphon out user assets. A recurring theme in DeFi lending attacks is the ability to borrow more than what was originally deposited, highlighting systemic weaknesses in existing security measures.

This research introduces a novel and untapped security mechanism referred to as *Traps*. By analyzing historical exploits in lending protocols, the study demonstrates how Traps, if implemented, could have preempted and mitigated such attacks. Through comprehensive risk analysis of on-chain data from various liquidity pools, the research identifies vulnerabilities and presents several trap designs tailored to enhance the security posture of DeFi lending protocols. The proposed approach aims to anticipate and prevent a wide array of attack vectors, positioning Traps as a critical innovation for securing user assets in DeFi lending platforms.

2 Statistics on Exploits Involving Lending Protocols

This research assumes that the reader is already familiar with decentralized finance (DeFi) lending protocols and common smart contract vulnerabilities, including reentrancy, oracle manipulation, and liquidations. Before delving into specific protocols that could benefit from Drosera's trap technology to secure their treasuries, it is essential to examine key statistics on recent DeFi attacks.

The given plot indicates that lending and borrowing protocols are among the most targeted in DeFi security breaches, with significant financial losses. In fact, these protocols account for a staggering $377 million in treasury losses due to various security attacks. This makes them the most vulnerable category of DeFi protocols.

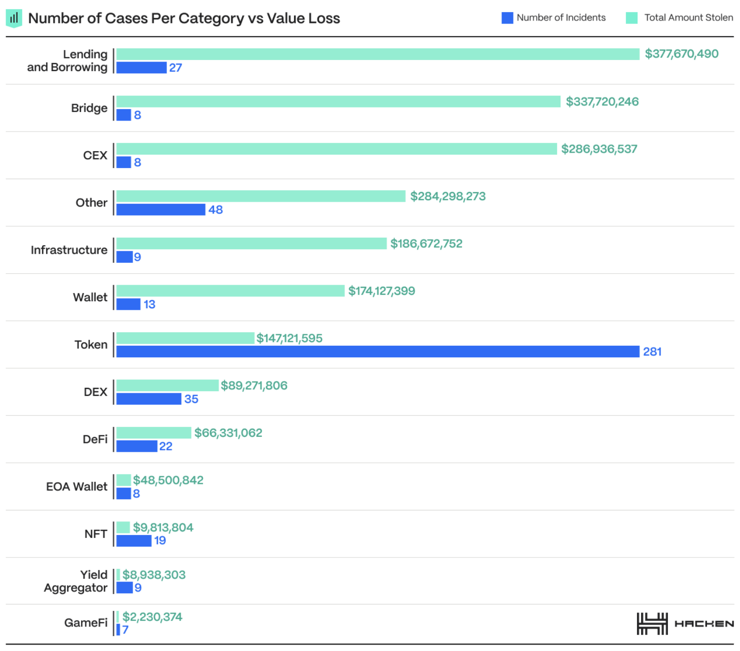


Figure 1: Plot 1 - Number of Cases Per Category vs. Value Loss

Further analysis reveals that DeFi protocols face persistent security breaches, as shown in statistics highlighting the average total loss per day across the ecosystem.

Despite the prevalence of such exploits, the current benchmark for protocol security, which often involves relying on audits, proves inadequate. Only 10% of the contracts that were exploited had undergone any form of audit, and even then, only half of these audits were relevant to the deployed blockchain code, demonstrating the limitations of audits as a security measure.

The chart below illustrates the composition of various types of exploits in lending protocols, providing a clearer understanding of which aspects attackers commonly exploit. This comprehensive breakdown allows for a better assessment of the security flaws in lending protocols and the critical areas that require stronger defenses.

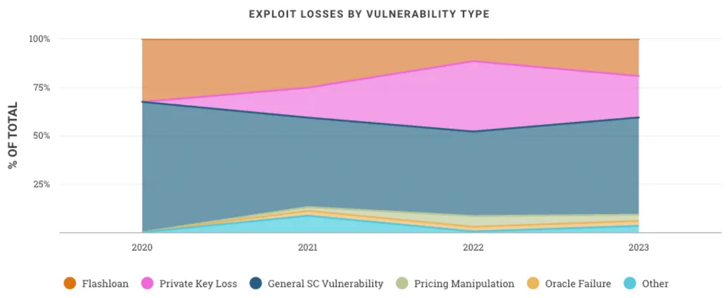


Figure 2: Plot 2 - Number of Cases Per Category vs. Value Loss

3 Lending Protocol Exploits – A Detailed Breakdown

3.1 Oracle Price Manipulation Attacks

A large number of exploits in lending protocols are associated with manipulation in prices quoted by various oracles. Lending protocols use oracles to quote prices needed while lending or borrowing assets on the protocol. These oracles are often used as a vulnerability by attackers to exploit the protocol. The general concept of borrowing on lending protocols involves the concept of an LTV (Loan to Value). Whenever a user borrows, that user's position has an LTV ratio. This value has to always be less than the maximum LTV allowed; if the LTV exceeds the maximum LTV allowed, liquidation occurs. An attacker tries to exploit the protocol in one of two ways: either by artificially increasing the maximum permissible LTV so that the attacker can borrow more with the same collateral, thus fooling the protocol into believing that the collateral deposited is of higher value than it actually is, or by artificially decreasing the LTV of the attacker's loan so they can still borrow more as the protocol is made to believe that the attacker's LTV is still lower than the maximum permissible LTV. Below are some famous exploits that occurred because of manipulation in the prices quoted by oracles:

3.1.1 Platypus Finance

This hack exploited a price manipulation vulnerability. These vulnerabilities exist if the swap price for an asset pair is calculated inside of a smart contract's code and is potentially manipulable by an attacker. In this case, both the cash and liability values in the protocol's smart contract had the ability to cause slippage in the swap price. If an attacker could create significant slippage, they could drain value from the protocol.

The attacker began by working to increase the liability. To do so, they deposited WAVAX tokens to LP-AVAX and sAVAX to the LP-sAVAX contract. Then, they used the LP-AVAX contract to swap sAVAX to WAVAX, reducing that contract's cash reserves. Finally, they eliminated that contract's cash reserves by withdrawing all WAVAX from the contract. The end result of these actions was that the swap price of the contract was increased, creating an opportunity for the attacker to drain value from the contract. In the end, they were able to steal about $2.2 million from the protocol.

As in the past, this Platypus Finance exploiter was far from professional. After stealing the funds from the project, they left them in an insecure wallet. As a result, an estimated $575K of the stolen tokens were retrieved by the protocol. However, the remaining $1.6 million remains under the control of the attacker.

3.1.2 Cream Finance

This is the second example of a price manipulation attack. In this exploit, the attacker used two accounts, A and B, to make the exploit profitable. Account A begins by flash borrowing \$500 million DAI from Maker, depositing it sequentially into various vaults: first into the yDAI Yearn Vault, then into the yUSD Curve pool, and finally into the yUSD Yearn Vault (yUSDVault). This action boosts the total supply of the yUSDVault from \$11 million to \$511 million. Account A then deposits the \$500 million from the yUSDVault into Cream, minting \$500 million cryUSD.

Meanwhile, Account B flash borrows \$2 billion ETH, mints cETH by depositing it into Cream, and borrows \$500 million yUSDVault as collateral. This borrowing is repeated, ultimately transferring \$1.5 billion cryUSD to Account A.

To efficiently gain control of the yUSDVault, Account A buys \$3 million DUSD, redeeming it for underlying yUSDVault collateral and then burning \$503 million yUSDVault shares, reducing the total supply to \$8 million, which raises each share's value to \$1. Next, Account A manipulates the price by transferring \$8 million yUSD into yUSDVault, doubling the price of each share to \$2. This maneuver instantly inflates the value of Account A's \$1.5 billion cryUSD to \$3 billion.

The price manipulation leads to a sudden debt increase for Account B, now owing \$3 billion cryUSD against \$2 billion in ETH collateral. This atomic price jump circumvents liquidation, as there is no way to buy out Account B's debt position without the sudden price change. Finally, Account A capitalizes on this inflated collateral by borrowing \$2 billion ETH, repaying the flash loans, and siphoning remaining assets in Cream, resulting in a profit of around \$130 million.

3.2 LP Token Price Manipulation Attacks

Many protocols allow users to deposit LP tokens as collateral. These attacks involve hackers exploiting weaknesses in the mathematics used in pricing LP tokens. This manipulation in the price of LP tokens again allows the attacker to artificially increase the maximum permissible LTV or decrease the LTV of the attacker's loan, thus allowing them to borrow more than they deposited. A classical example of this attack is the Warp Finance attack as detailed below:

3.2.1 Warp Finance

This exploit involved an LP token price manipulation vulnerability. The attacker began by taking out flash loans, using approximately \$5.8 million to provide liquidity to the DAI-WETH pool and receiving LP tokens in return. They then swapped the remaining flash loan amount in the same DAI-WETH pool, manipulating the pool's reserves and causing Warp Finance to miscalculate the LP token price. Exploiting this inflated valuation, the attacker used the LP tokens as collateral on Warp Finance, borrowing around \$7.8 million. They proceeded to repay the flash loans, leaving the inflated Warp Finance loan to default, effectively profiting from the manipulated collateral value.

3.2.2 OneRing Protocol

This is the second attack in the category of LP token price manipulation. On March 21, 2022, at 06:44:10 PM UTC, a flashloan-assisted attack led to the theft of \$1,454,672.24 USDC from the OneRing protocol, specifically targeting the OShare contract while leaving other tokens and liquidity pools unaffected. The hacker, identified as a professional due to the sophisticated methods used, employed a self-destructing contract for enhanced anonymity.

Before launching the attack, they moved gas funds via the Celer Network and deployed the exploit contract, which was active for a brief period. The hacker borrowed \$80 million USDC through Solidly flash loans to manipulate LP token prices within a single block, allowing them to drive a significant amount of OShare tokens out of the protocol. The total losses were approximately \$2 million, including swap and flash loan fees. Despite efforts to trace the hacker's address, the stolen funds were quickly moved back to Ethereum and into Tornado Cash, complicating further tracking efforts due to the wallet's clean history and Tornado Cash's anonymizing properties.

3.3 Attacks Involving Flash Loans

These attacks involve attackers taking up large sums of money as a loan in a single transaction and using that large sum to imbalance different liquidity pools, resulting in misquoted prices and exploiting the protocol. A classical example of this attack is the Warp Finance attack as detailed below:

3.3.1 Wise Finance

The attack involved the attacker utilizing a flash loan to accomplish the exploit. The team analyzed the attack and found that the exploiter had manipulated an almost empty PLP-stETH-Dec2025 market in Wise Lending, which had been deployed just a day earlier, to inflate the share price. By exploiting precision loss during share distribution calculations, the attacker took out a flash loan of 1,100 ETH, allowing them to borrow most of the funds from the lending markets.

The protocol's rounding-up mechanism for share withdrawals enabled the exploiter to repeatedly call the withdraw function with minimal amounts, creating a mismatch between token balance and shares and allowing profitable price manipulation. Despite built-in defenses, these measures were circumvented or even turned against the protocol itself. The exploiter performed a small initial deposit followed by a donation just below the threshold, effectively bypassing the checks. This donation exceeded the initial deposit by a vast percentage but passed verification. They then exploited the protocol's rounding bias, making a series of deposits and withdrawals that maximized rounding-related losses, which were integrated into the protocol's core figures. Ultimately, the exploiter transferred stolen assets totaling 178.9377 ETH, equivalent to approximately $455,666.

3.3.2 Sonne Finance

On May 4, 2024, the team behind Sonne Finance initiated Sonne Improvement Proposal 15 to add VELO, Velodrome Finance's native token, to their market. The proposal received overwhelming support, with 2.3 million SONNE tokens voting in favor. Following the vote, they scheduled transactions on their multisignature wallet, which had to pass through a 2-day timelock. This was intended to ensure a phased approach, including changes to collateral factors (c-factors). However, a vulnerability in the Timelock contract allowed anyone to execute the transaction to change the collateral factor and mint SONNE tokens in the newly established market.

Sonne had implemented preventive measures, such as sequential timelocks for market addition, user fund deposits, and eventual market activation. Unfortunately, these actions were planned as independent transactions rather than a single, linked sequence. When the time lock for the new market's creation expired, an attacker exploited this oversight by executing multiple transactions, including setting up the market with a new c-factor.

With the VELO market now empty, the attacker leveraged flash loans to exploit a rounding error inherent in Compound V2 forks. They borrowed VELO tokens, transferred them to the target soVELO contracts, and established several contracts to drain approximately $20 million. Another $6.5 million was salvaged by adding a small amount of VELO, while the attacker extended the exploit to Sonne USDC (soUSDC) and Sonne Wrapped Ether (soWETH) markets.

3.4 Attacks Involving TWAP Oracles

TWAP oracles, short for Time-Weighted Average Price oracles, are a type of oracle that calculates the average price of an asset over a specified period. Unlike instantaneous price oracles, which provide the current spot price, TWAP oracles mitigate price manipulation and volatility by averaging the price over time, often reducing the impact of sudden, short-term fluctuations. The price from a TWAP is hardly manipulated in one transaction and block. But being atomic is not a requirement for an attacker. TWAP is just a formula using on-chain sources, and it is a question of time and cost to attack TWAP. TWAP mostly works well with tokens having high liquidity but doesn't work well with low liquidity tokens. Below are two famous attacks involving TWAP oracles:

3.4.1 Inverse Finance

The vulnerable code resided in the project's YVCrvCrypto pool, where the inverse price oracle estimated the LP token's value based on the balance of current assets within the pool. The attacker exploited this by manipulating the asset balance through a series of deposits, swaps, and trades, thereby altering the perceived LP token value. They began by taking out a flash loan, then deposited collateral into the pool and performed swaps to artificially inflate the value of that collateral. This allowed them to borrow a significantly larger loan than should have been possible. After a few conversions, the attacker paid off the initial flash loan and walked away with a substantial profit.

3.4.2 Moola Markets

The attacker manipulated the TWAP oracle pricing formula used by Moola Markets. They began by swapping CELO tokens for MOO tokens and then locked the MOO tokens to obtain additional CELO tokens, repeating this process multiple times. This allowed them to manipulate the price of MOO on Ubeswap, which subsequently affected the MOO TWAP price oracle used by the Moola Protocol.

The attacker initially funded their wallet with 182,000 CELO tokens from Binance. They then swapped CELO for MOO tokens by calling the Swap function. Next, the attacker locked the MOO tokens as collateral and borrowed CELO tokens, using the borrowed funds to buy more MOO tokens. Repeating this process allowed the attacker to drive up the price of MOO against CELO, ultimately increasing the MOO token's value from 0.02 CELO to 0.73 CELO. Once the price had been sufficiently inflated, the attacker borrowed the remaining assets on the protocol, draining all liquidity and stealing approximately $8.4 million in assets.

These were the types of attacks possible in lending protocols.

4 Drosera, Traps, and How They Can Tackle Exploits

"Drosera aims to leverage the decentralized nature of Ethereum consensus to create a base layer for security. Protocols define incident response logic for Operators to carry out. Slashing and reward mechanisms ensure honesty and accountability. This approach to security expands monitoring and bug bounty programs to a dynamic model. Drosera maintains an open and efficient platform, encouraging collaboration and shared knowledge among the participating parties. This could result in the evolution of advanced security solutions and best practices, contributing to the overall enhancement and resilience of the Ethereum ecosystem" *Drosera Litepaper*.

Traps are a set of smart contracts that define the conditions for detecting invariants and performing on-chain responses. Traps have an on-chain and off-chain component that are described below:

- **Trap**: An off-chain smart contract that performs data collection and analysis to signal the execution of an on-chain response function.
- **Trap Config**: An on-chain smart contract that configures the trap and defines the on-chain response callback function. Example: `pause(uint256)`, `react(address)`, etc.

Below is a detailed breakdown of how each of the above-mentioned attacks could either have been completely avoided or the damage could have at least been limited using Drosera Traps.

The Drosera trap contract has two primary functions:

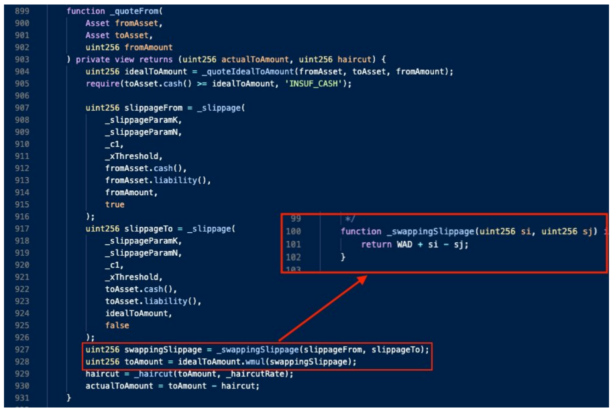
- `collect()` function, which contains the data collection and monitoring logic.
- `shouldRespond()` function, which contains the validation logic to decide when to trigger a response.

Joint modification of these two functions together can be used for various protocols to define various possible incident-response strategies that can help secure these protocols.

4.1 Platypus Finance

The swap slippage in Platypus depends on the convergence of the coverage cash liquidity ratio in the asset contract. The attacker was able to manipulate the cash and liability in the contract to gain a handful of incentives from this swap slippage manipulation. This caused an increase in the ‘`slippageFrom`‘ value, which is the first parameter in the ‘`_swappingSlippage`‘ function, which resulted in the manipulation of the ‘`actualToAmount`‘ in the ‘`_quoteFrom`‘ function. The attacker then swapped the underlying assets to take away a huge profit from this manipulated slippage.

The `collect()` function of a Drosera Trap could here be used to track the two slippage parameters `si` & `sj` on a block by block basis & trigger a response to pause the pool if one of the two values suddenly increases or decreases sharply. This would prevent the attacker from manipulating the pool by misquoting the slippage for a swap.



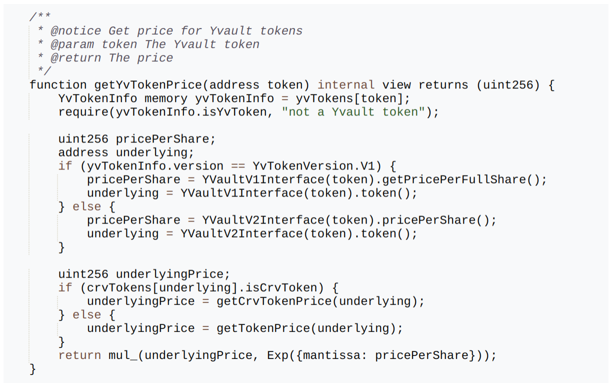
```
function _quoteFrom(
    Asset fromAsset,
    Asset toAsset,
    uint256 fromAmount
) private view returns (uint256 actualToAmount, uint256 haircut) {
    uint256 idealToAmount = _quoteIdealToAmount(fromAsset, toAsset, fromAmount);
    require(toAsset.cash() >= idealToAmount, 'INSUF_CASH');

    uint256 slippageFrom = _slippage(
        _slippageParamK,
        _slippageParamN,
        _c1,
        _xThreshold,
        fromAsset.cash(),
        fromAsset.liability(),
        fromAmount,
        true
    );
    uint256 slippageTo = _slippage(
        _slippageParamK,
        _slippageParamN,
        _c1,
        _xThreshold,
        toAsset.cash(),
        toAsset.liability(),
        idealToAmount,
        false
    );
    uint256 swappingSlippage = _swappingSlippage(slippageFrom, slippageTo);
    uint256 toAmount = idealToAmount.wmul(swappingSlippage);
    haircut = _haircut(toAmount, _haircutRate);
    actualToAmount = toAmount - haircut;
}
```

```
*/
function _swappingSlippage(uint256 si, uint256 sj)
    return WAD + si - sj;
}
```

Figure 3: Platypus Finance

4.2 Cream Finance

The attacker manipulated the reported exchange rate of the vault using a Flash Loan, subsequently altering the value the Oracle reports for the yUSD token. The `collect()` function of the trap contract could here be set to monitor the critical value of the `exchangeRateStored` needed for swap & `totalReserves` in the pool. A sudden change in their value(s) can trigger a response to pause the pool.



```
/**
 * @notice Get price for Yvault tokens
 * @param token The Yvault token
 * @return The price
 */
function getYvTokenPrice(address token) internal view returns (uint256) {
    YvTokenInfo memory yvTokenInfo = yvTokens[token];
    require(yvTokenInfo.isYvToken, "not a Yvault token");

    uint256 pricePerShare;
    address underlying;
    if (yvTokenInfo.version == YvTokenVersion.V1) {
        pricePerShare = YVaultV1Interface(token).getPricePerFullShare();
        underlying = YVaultV1Interface(token).token();
    } else {
        pricePerShare = YVaultV2Interface(token).pricePerShare();
        underlying = YVaultV2Interface(token).token();
    }

    uint256 underlyingPrice;
    if (crvTokens[underlying].isCrvToken) {
        underlyingPrice = getCrvTokenPrice(underlying);
    } else {
        underlyingPrice = getTokenPrice(underlying);
    }
    return mul_(underlyingPrice, Exp({mantissa: pricePerShare}));
}
```

Figure 4: Cream Finance

4.3 Warp Finance

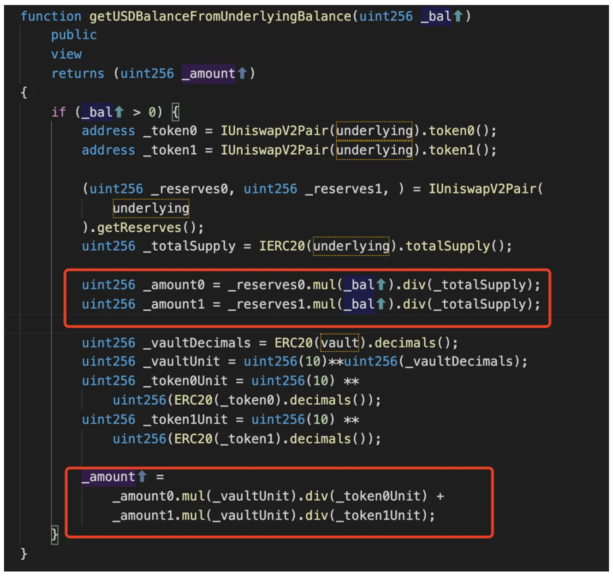
The problem here lies with the assumption of TVL to be constant for calculating LP Price. The `collect()` function of Trap should track the TVL of the market here which is used for LP price calculation. Sudden change in TVL should trigger a response to pause the market.

```
function _calculatePriceOfLP(
    uint256 supply,
    uint256 value0,
    uint256 value1,
    uint8 supplyDecimals
) public pure returns (uint256) {
    uint256 totalValue = value0 + value1;
    uint16 shiftAmount = supplyDecimals;
    uint256 valueShifted = totalValue * uint256(10)**shiftAmount;
    uint256 supplyShifted = supply;
    uint256 valuePerSupply = valueShifted / supplyShifted;

    return valuePerSupply;
}
```

Figure 5: Warp Finance

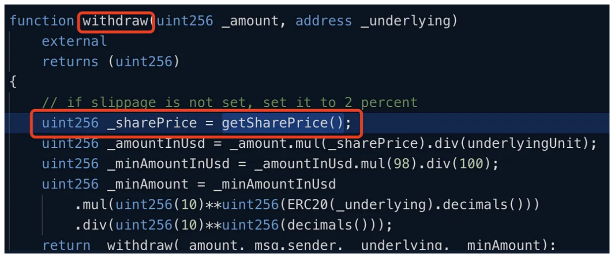
4.4 OneRing Protocol

The attacker inflated the calculation of LP Token price (OShare price here) by manipulating the reserves of the pool. The attacker deposited a large sum of USDC into the pool which increased the `_amount` value in turn increasing the current OShare price which is obtained by the `getSharePrice()` function. The `collect()` function of the trap contract here should track the value of `_amount0` & `_amount1` & trigger a response as soon as any of the two changes drastically. This would pause the `deposit` & `withdraw` function thus preventing any attacker from manipulating the market using incorrect price values.



```
function getUSDBalanceFromUnderlyingBalance(uint256 _bal)
    public
    view
    returns (uint256 _amount)
{
    if (_bal > 0) {
        address _token0 = IUniswapV2Pair(underlying).token0();
        address _token1 = IUniswapV2Pair(underlying).token1();

        (uint256 _reserves0, uint256 _reserves1, ) = IUniswapV2Pair(
            underlying
        ).getReserves();
        uint256 _totalSupply = IERC20(underlying).totalSupply();

        uint256 _amount0 = _reserves0.mul(_bal).div(_totalSupply);
        uint256 _amount1 = _reserves1.mul(_bal).div(_totalSupply);

        uint256 _vaultDecimals = ERC20(vault).decimals();
        uint256 _vaultUnit = uint256(10)**uint256(_vaultDecimals);
        uint256 _token0Unit = uint256(10) **
            uint256(ERC20(_token0).decimals());
        uint256 _token1Unit = uint256(10) **
            uint256(ERC20(_token1).decimals());

        _amount =
            _amount0.mul(_vaultUnit).div(_token0Unit) +
            _amount1.mul(_vaultUnit).div(_token1Unit);
    }
}
```

Figure 6: OneRing - 1

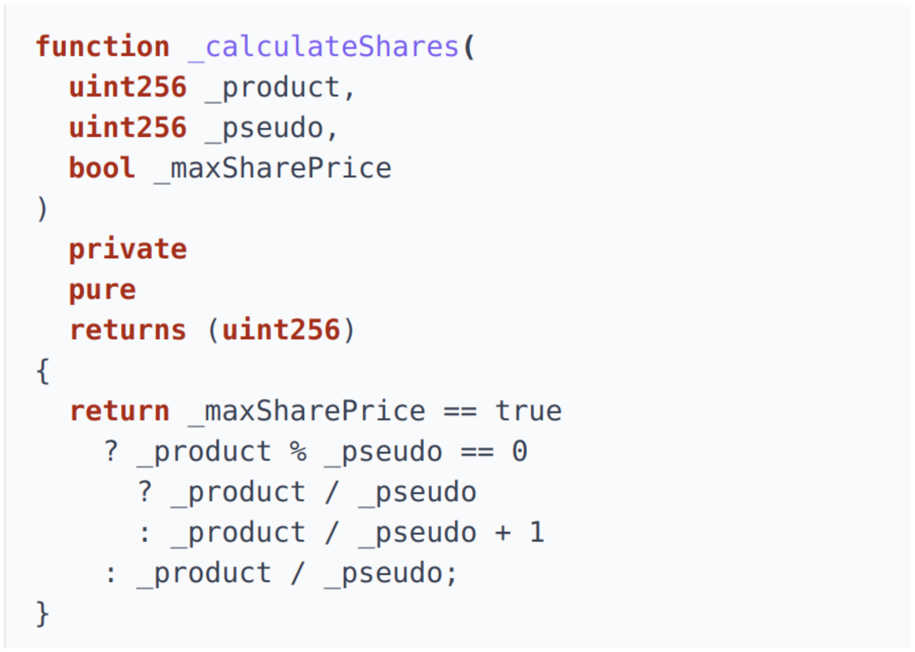


```
function withdraw(uint256 _amount, address _underlying)
    external
    returns (uint256)
{
    // if slippage is not set, set it to 2 percent
    uint256 _sharePrice = getSharePrice();
    uint256 _amountInUsd = _amount.mul(_sharePrice).div(underlyingUnit);
    uint256 _minAmountInUsd = _amountInUsd.mul(98).div(100);
    uint256 _minAmount = _minAmountInUsd
        .mul(uint256(10)**uint256(ERC20(_underlying).decimals()))
        .div(uint256(10)**uint256(decimals()));
    return _withdraw(_amount, msg.sender, _underlying, _minAmount);
```

Figure 7: OneRing - 2

4.5 Wise Lending

The attacker leveraged precision loss in the share distribution calculation, exploiting the protocol's rounding-up mechanism in withdrawals. This allowed the attacker to repeatedly call the withdraw function with small amounts, creating a discrepancy between token balance and shares, ultimately manipulating prices for profit. The `collect()` function of the trap contract could be used to keep a track of `_amount` deposited into a market. This would allow markets to become imbalanced by a large ratio thus removing the chances of wrong calculation of `calculateShares()`.

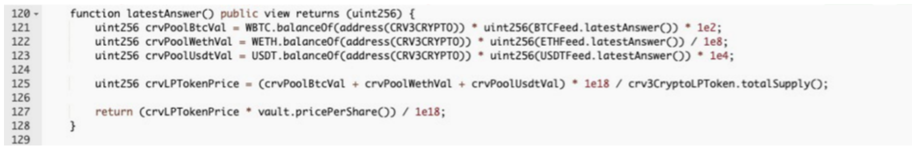


```
function _calculateShares(
  uint256 _product,
  uint256 _pseudo,
  bool _maxSharePrice
)
  private
  pure
  returns (uint256)
{
  return _maxSharePrice == true
    ? _product % _pseudo == 0
      ? _product / _pseudo
      : _product / _pseudo + 1
    : _product / _pseudo;
}
```

Figure 8: Wise Lending

4.6 Inverse Finance

The attacker here exploited the market using a flash loan to deposit a large sum of WBTC (27K). The `collect()` function of the trap contract here could again be made to track the reserves in the pool or basically the balance of the three tokens in the pool, if any value suddenly changes, a response to pause the `withdraw` function could be made.



```
function latestAnswer() public view returns (uint256) {
    uint256 crvPoolBtcVal = WBTC.balanceOf(address(CRV3CRYPTO)) * uint256(BTCFeed.latestAnswer()) * 1e2;
    uint256 crvPoolWethVal = WETH.balanceOf(address(CRV3CRYPTO)) * uint256(ETHFeed.latestAnswer()) / 1e8;
    uint256 crvPoolUsdtVal = USDT.balanceOf(address(CRV3CRYPTO)) * uint256(USDTFeed.latestAnswer()) * 1e4;

    uint256 crvLPTokenPrice = (crvPoolBtcVal + crvPoolWethVal + crvPoolUsdtVal) * 1e18 / crv3CryptoLPToken.totalSupply();

    return (crvLPTokenPrice * vault.pricePerShare()) / 1e18;
}
```

Figure 9: Inverse Finance

4.7 Guide on Trap Designs

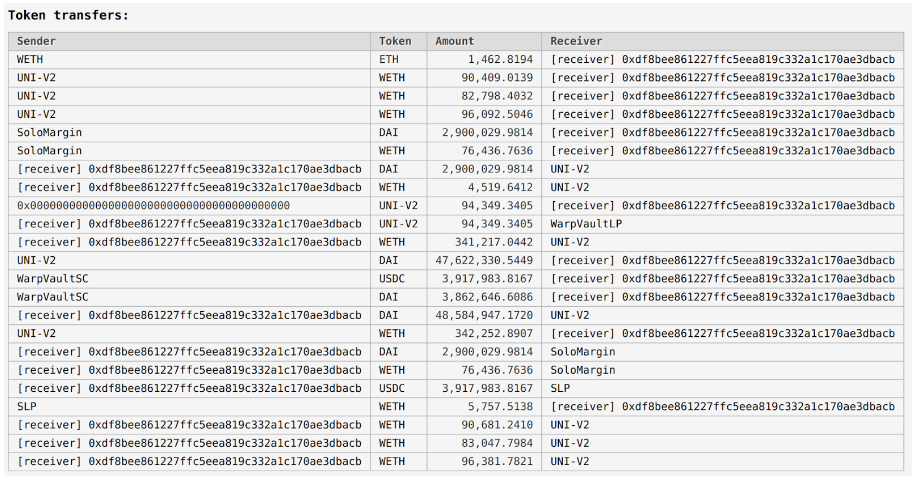
This document had a detailed thorough analysis of 8 different protocols & the various attacks that exploited these protocols. This analysis leads us to a number of conclusions as detailed below:

4.7.1 Traps are Mutable and Adaptable

Traps can be updated over time, allowing for changes in data collection methods and validation logic. This flexibility makes traps a reliable and versatile tool for protecting lending protocols, as the dynamics of these markets (total supply, total assets, TVL, LP price, etc.) are constantly evolving. The fact that these parameters, often exploited in lending protocol attacks, fluctuate continuously highlights the need to keep traps updated based on market conditions.

4.8 No Single Trap Design is Sufficient

The diversity and complexity of attacks on various protocols lead us to conclude that no single trap design can comprehensively protect a protocol. For instance, consider the Warp Finance exploit. The attached images illustrate the changes in balances and token transfers that occurred during the exploit. These transfers underscore the difficulty a protocol would face in designing a trap that could prevent such an attack. The Warp vault, relying on UNI-V2 oracles, would not have been able to implement a single trap to avoid the vulnerabilities associated with the Uniswap oracles. Preventing this attack would have required multiple traps working together to protect the treasury without unnecessarily pausing the market's operations.



Token transfers:

Sender	Token	Amount	Receiver
WETH	ETH	1,462.8194	[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb
UNI-V2	WETH	90,409.0139	[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb
UNI-V2	WETH	82,798.4032	[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb
UNI-V2	WETH	96,092.5046	[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb
SoloMargin	DAI	2,900,029.9814	[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb
SoloMargin	WETH	76,436.7636	[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb
[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb	DAI	2,900,029.9814	UNI-V2
[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb	WETH	4,519.6412	UNI-V2
0x0000000000000000000000000000000000000000	UNI-V2	94,349.3405	[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb
[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb	UNI-V2	94,349.3405	WarpVaultLP
[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb	WETH	341,217.0442	UNI-V2
UNI-V2	DAI	47,622,330.5449	[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb
WarpVaultSC	USDC	3,917,983.8167	[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb
WarpVaultSC	DAI	3,862,646.6086	[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb
[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb	DAI	48,584,947.1720	UNI-V2
UNI-V2	WETH	342,252.8907	[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb
[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb	DAI	2,900,029.9814	SoloMargin
[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb	WETH	76,436.7636	SoloMargin
[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb	USDC	3,917,983.8167	SLP
SLP	WETH	5,757.5138	[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb
[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb	WETH	90,681.2410	UNI-V2
[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb	WETH	83,047.7984	UNI-V2
[receiver] 0xdf8bee861227ffc5eea819c332a1c170ae3dbacb	WETH	96,381.7821	UNI-V2

Figure 10: Warp Finance Exploit Transactions

4.9 External Dependencies Require Complex Trap Logic

Many markets rely on external services for quoting prices, estimating slippage, and other functions. This external reliance highlights the necessity for protocol developers to design traps with these dependencies in mind. A single trap may not suffice, and complex trap logic or multiple traps may be required to ensure robust protection.

4.10 Subjective Nature of 'Sudden Changes'

The concept of a "sudden change" is highly subjective and varies significantly among markets. For example, a deposit or withdrawal of $500K may be considered sudden in a market with a TVL of $2M, as it represents 25% of the market's total value. In contrast, the same transaction would have minimal impact on a market with $500M in liquidity. This variation in lending protocol dynamics makes it clear that no fixed value can define a sudden change across all markets. Instead, trap design must involve thorough historical analysis of a market's transactions to determine appropriate triggering conditions, which must be constantly monitored and updated as needed. At Anthias Labs, we offer monitoring services for these markets and propose regularly updated trap designs. Once approved by a protocol, these designs can be applied to maintain security, regardless of the protocol's changing dynamics.

4.11 Using Traps to Determine Trap Updates

Finally, Anthias Labs proposes that trap updates themselves be determined using traps. Currently, many protocols rely on their core teams to set crucial values in smart contracts, such as constants for LP price, slippage, TVL, or limiting values like maxMint and maxWithdraw. These values are often decided by the protocol team of just a few people, which can lead to poor risk analysis or incorrect configurations. By allowing Drosera operators to decide the validation logic of a trap using a trap itself, the community can be involved in the risk analysis process through consensus. This approach will lead to better-adjusted and more frequently updated traps, ensuring adaptability to changing market conditions. This is a concept we refer to as meta-traps, which we will expand on in further research.

This formalized guide outlines how traps can be designed, implemented, and updated to enhance the security of DeFi lending protocols. With the continuous involvement of the community and a dynamic approach to trap logic, protocols can be better equipped to withstand a wide range of attacks.

5 Conclusions

The in-depth analysis of the various types of hacks in lending protocols mentioned earlier allows us to draw the following conclusions regarding the efficient usage of traps in these protocols:

- **Traps as a Preventive Mechanism:** Traps can play a crucial role in avoiding exploits targeting lending protocols by addressing the primary vulnerabilities that are exploited in almost all such attacks. By continuously monitoring and adapting to changing conditions, traps can act as a proactive defense mechanism.
- **Key Parameters to Monitor Using the Trap Contract's collect() Function:** To effectively track and mitigate risks, the most common parameters that traps should monitor are the following:
  - TVL of the market
  - Reserves in the market
  - Amount deposited in a transaction
  - Amount withdrawn in a transaction
  - LP Token price
  - Quantity of LP Tokens minted/burned in a single transaction
  - Collateralization Ratio
  - Utilization Ratio of the market
  - Funding Rate
- **Rate Rapid Consensus-Based Response:** Traps can trigger a response when two-thirds of the nodes reach consensus, which on Ethereum Layer 1 takes approximately 15 minutes. This time frame is significantly faster compared to the time it takes most protocol teams to become aware of an exploit and respond. For instance, it took 25 minutes for Sonne Finance and 40 minutes for OneRing Protocol to recognize and react to their respective attacks. This makes traps a more reliable and efficient safeguard during emergencies.

About Anthias Labs

Anthias Labs is a boutique on-chain advisory firm focused on DeFi risk management and system design. We tackle mission-critical problems for select partner protocols.

Legal Disclaimer