Security plan for reopening and operation of Tropykus after June/23

Context

On the 14th of June 2023 at 7:46 am UTC-5, the Tropykus protocol suffered an attack which drained 158,000 USD from their DOC and kSAT (microsavings) market. The attacker stole 96k DOC and the equivalent to 62k USD in rBTC from the attack.

The Tropykus team led an exhaustive investigation to find out the root cause of the attack and design a safe reopening procedure. As part of this investigation some attack patterns were identified and a short and long term security plan was designed.

Security Plan for reopening

The security plan includes short, medium and long term tasks to ensure both a safe reopening of the protocol, as well as a continuous monitoring and mitigation risk plan. The plan can be summarized as follows:

1. De-listing of rBTC micro market in which the vulnerability was present
2. Going back to Compound original code base with this change
3. Repay the stolen DOC from the attacker to bring the protocol back the pre-attack interest rates
4. Deploy a new protocol controller for the markets without rBTC micro
5. Reaudit the protocol with automatic tools and AI
6. Deploy a quick response pause system
7. Connect this quick response system to be triggered automatically whenever a known vulnerability is known
8. Partner with other Compound-based protocols to share security information and collaborate
9. Deploy a continuous monitoring system to close the markets whenever a potential attack is detected
10. Scan regularly the code base for new reported vulnerabilities and update the internal security policies and protocols accordingly
11. Partner with an specialized consultant to review the protocols and policies

Mitigation measures for known vulnerabilities

After removing the Tropykus modifications of the Compound code from the protocol, Tropykus shall behave as Compound, with its advantages and inherited vulnerabilities. Some of these vulnerabilities are known and have been exploited in the past, and thus we have designed specific plans for each one of these known issues.

Migrating to Compound Code

When Tropykus was developed, we introduced a subsidized interest rate model to incentivize savings un rBTC up to a cap of 0.025 RBTC.

In Compound, an interest rate model is a smart contract that defines the interest rate curves of a market.

One of the modifications Tropykus did to the original code base of Compound was in the *InterestRateModel.sol* smart contract. In this contract,

When the custom interest rate model from Tropykus, included in the *HurricaneInterestRateModel.sol* contract is deployed, it sets the *isTropykusInterestRateModel* variable to true. The rest of the InterestRateModels do not perform this action.

```
pragma solidity 0.5.16;

import "./InterestRateModel.sol";

contract HurricaneInterestRateModel is InterestRateModel {
    using SafeMath for uint256;

    address public owner;
    uint256 public baseBorrowRatePerBlock;
    uint256 public promisedBaseReturnRatePerBlock;
    uint256 public optimalUtilizationRate;
    uint256 public borrowRateSlopePerBlock;
    uint256 public supplyRateSlopePerBlock;

    uint256 constant FACTOR = 1e18;
    bool public constant isTropykusInterestRateModel = true;

    constructor(
        uint256 _baseBorrowRate,
        uint256 _promisedBaseReturnRate,
        uint256 _optimalUtilizationRate,
        uint256 _borrowRateSlope,
        uint256 _supplyRateSlope
    ) public {
        blocksPerYear = 1051200;
        admin = msg.sender;
        baseBorrowRatePerBlock = _baseBorrowRate.div(blocksPerYear);
        promisedBaseReturnRatePerBlock = _promisedBaseReturnRate.div(
            blocksPerYear
        );
        borrowRateSlopePerBlock = _borrowRateSlope.div(blocksPerYear);
        supplyRateSlopePerBlock = _supplyRateSlope.div(blocksPerYear);
        optimalUtilizationRate = _optimalUtilizationRate;
        owner = msg.sender;
    }
```

After removing the kSAT market, any function in the protocol shall behave as if it was the original Compoound code.

Let's analyze the *mint* function in the CToken.sol contract

```
/**
 * @notice Sender supplies assets into the market and receives cTokens in exchange
 * @dev Accrues interest whether or not the operation succeeds, unless reverted
 * @param mintAmount The amount of the underlying asset to supply
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function mint(uint256 mintAmount) external returns (uint256) {
    (uint256 err, ) = mintInternal(mintAmount);
    return err;
}
```

The *mint* function then calls the *mintInternal* function.

```
/**
 * @notice Sender supplies assets into the market and receives cTokens in exchange
 * @dev Accrues interest whether or not the operation succeeds, unless reverted
 * @param mintAmount The amount of the underlying asset to supply
 * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), and the actual mint amount.
 */
function mintInternal(uint256 mintAmount)
    internal
    nonReentrant
    returns (uint256, uint256)
{
    uint256 error = accrueInterest();
    if (error != uint256(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow failed
        return (
            fail(Error(error), FailureInfo.MINT_ACCRUE_INTEREST_FAILED),
            0
        );
    }
    // mintFresh emits the actual Mint event if successful and logs on errors, so we don't need to
    return mintFresh(msg.sender, mintAmount);
}

struct MintLocalVars {
    Error err;
    MathError mathErr;
    uint256 exchangeRateMantissa;
    uint256 mintTokens;
    uint256 mintAmount;
    uint256 totalSupplyNew;
    uint256 accountTokensNew;
    uint256 actualMintAmount;
}
```

This function calls the *accrueInterest()* function and then the *mintInternal()* function

```
if (interestRateModel.isTropykusInterestRateModel()) {
    (mathErr, totalReservesNew) = newReserves(
        borrowRateMantissa,
        cashPrior,
        borrowsPrior,
        reservesPrior,
        interestAccumulated
    );
    if (mathErr != MathError.NO_ERROR) {
        return
            failOpaque(
                Error.MATH_ERROR,
                FailureInfo
                    .ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED,
                uint256(mathErr)
            );
    }
}
```

In the *accrueInterest()* function there is additional logic called if it is the Tropykus interest rate model, which is not executed for any other market besides kSAT.

The function *mintFresh()* calls the *exchangeRateStoredInternal()* function.

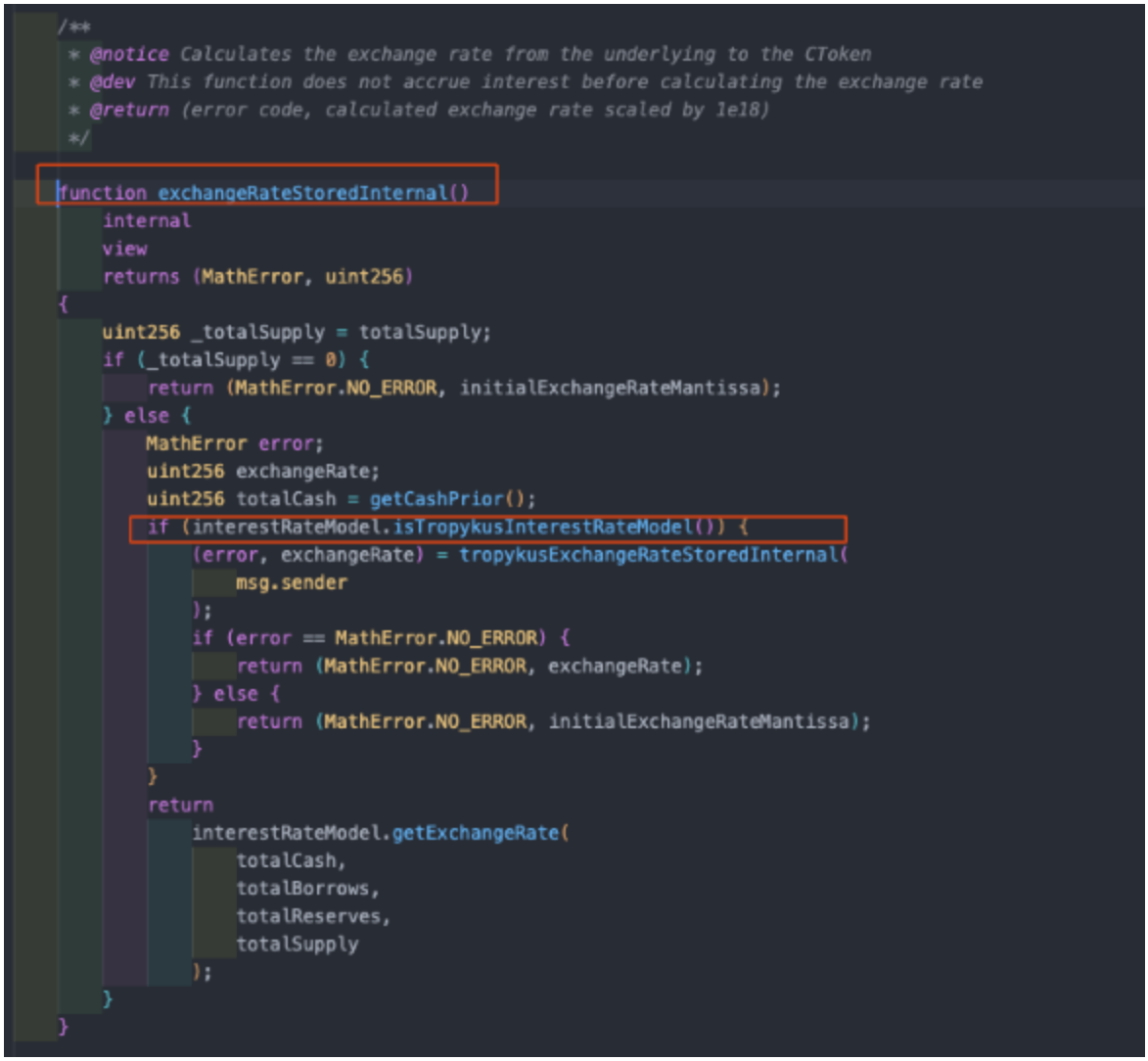
```
/**
 * @notice User supplies assets into the market and receives cTokens in exchange
 * @dev Assumes interest has already been accrued up to the current block
 * @param minter The address of the account which is supplying the assets
 * @param mintAmount The amount of the underlying asset to supply
 * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), and the actual mint amount.
 */
function mintFresh(address minter, uint256 mintAmount)
    internal
    returns (uint256, uint256)
{
    require(accountBorrows[minter].principal == 0, "T12");
    /* Fail if mint not allowed */
    uint256 allowed = comptroller.mintAllowed(…
    );
    if (allowed != 0) {…
    }

    /* Verify market's block number equals current block number */
    if (accrualBlockNumber != getBlockNumber()) {…
    }

    MintLocalVars memory vars;

    (…
    ) = exchangeRateStoredInternal();
    if (vars.mathErr != MathError.NO_ERROR) {…
    }
    vars.mintAmount = mintAmount;
    mintInternalVerifications(minter, vars);
```

The *exchangeRateStoredInternal()* function calls the *isTropykusInterestRateModel()* function from the associated market's interest rate model. If it is the interest rate model from Tropykus it performs custom calculations, in which the latest exploit happened. Otherwise it performs the standard calculation of Compound.



```
/**
 * @notice Calculates the exchange rate from the underlying to the CToken
 * @dev This function does not accrue interest before calculating the exchange rate
 * @return (error code, calculated exchange rate scaled by 1e18)
 */

function exchangeRateStoredInternal()
    internal
    view
    returns (MathError, uint256)
{
    uint256 _totalSupply = totalSupply;
    if (_totalSupply == 0) {
        return (MathError.NO_ERROR, initialExchangeRateMantissa);
    } else {
        MathError error;
        uint256 exchangeRate;
        uint256 totalCash = getCashPrior();
        if (interestRateModel.isTropykusInterestRateModel()) {
            (error, exchangeRate) = tropykusExchangeRateStoredInternal(
                msg.sender
            );
            if (error == MathError.NO_ERROR) {
                return (MathError.NO_ERROR, exchangeRate);
            } else {
                return (MathError.NO_ERROR, initialExchangeRateMantissa);
            }
        }
        return
            interestRateModel.getExchangeRate(
                totalCash,
                totalBorrows,
                totalReserves,
                totalSupply
            );
    }
}
```

As seen, by keeping an interest rate model contract different than the one developed by Tropykus, the code base performs the same as the original code of Compound.

Compound known issues and mitigation plan

Some of the reported issues in Compound and their forks and their mitigation plan will be detailed in the following sections:

Price oracle manipulation vulnerability

One of the most common issues in Compound forks and in any other lending protocol is the manipulation of price oracles. If the price of an asset is manipulated either by manipulating the price feed oracle or its proportion in a liquidity pool, the attacker can increase his collateral and get a loan in a different market which he could not be able to cover in a normal situation, such as the Mango Markets exploit.

In Tropykus, we use the price feeds from RSK and Money on Chain to provide the price of DoC and rBTC. We are monitoring these oracles continuously and we will connect these alerts with an automatic borrow pause mechanism to freeze loans if an anomaly in the oracles happen and restart the borrows once the situation has been studied.

ERC 677 / 777 tokens vulnerability

Another known issue in Compound happens when a token is transferred during a borrow operation. During this operation the funds are transferred before the state of the contract has been updated, which opens the door to a vulnerability if the underlying token of the market is a ERC677 or ERC777 token with fallback functions when the token is received.

```
    vars = borrowInternalValidations(borrower, vars);

    ////////////////////////
    // EFFECTS & INTERACTIONS
    // (No safe failures beyond this point)

    /*
     * We invoke doTransferOut for the borrower and the borrowAmount.
     *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
     *  On success, the cToken borrowAmount less of cash.
     *  doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
     */

    /* We write the previously calculated values into storage */
    accountBorrows[borrower].principal = vars.accountBorrowsNew;
    accountBorrows[borrower].interestIndex = borrowIndex;
    totalBorrows = vars.totalBorrowsNew;

    doTransferOut(borrower, borrowAmount);

    /* We emit a Borrow event */
    emit Borrow(
        borrower,
        borrowAmount,
        vars.accountBorrowsNew,
        vars.totalBorrowsNew
    );

    /* We call the defense hook */
    // unused function
    // comptroller.borrowVerify(address(this), borrower, borrowAmount);

    return uint256(Error.NO_ERROR);
}
```

This is the case of the rUSDT token in RSK, which has these functions inside the *transfer()* function of the token, and which was exploited in the past. As a result Tropykus only lists ERC-20 tokens without these fallback functions.

```
function transfer(address recipient, uint256 amount) external returns (bool) {
    require(recipient != address(0), "ERC777: transfer to zero address");

    address from = _msgSender();

    _callTokensToSend(from, from, recipient, amount, "", "");

    _move(from, from, recipient, amount, "", "");

    _callTokensReceived(from, from, recipient, amount, "", "", false);

    return true;
}
```

```
/**
 * @dev Call to.tokensReceived() if the interface is registered. Reverts if the recipient is a contract but
 * tokensReceived() was not registered for the recipient
 * @param operator address operator requesting the transfer
 * @param from address token holder address
 * @param to address recipient address
 * @param amount uint256 amount of tokens to transfer
 * @param userData bytes extra information provided by the token holder (if any)
 * @param operatorData bytes extra information provided by the operator (if any)
 * @param requireReceptionAck if true, contract recipients are required to implement ERC777TokensRecipient
 */
function _callTokensReceived(
    address operator,
    address from,
    address to,
    uint256 amount,
    bytes memory userData,
    bytes memory operatorData,
    bool requireReceptionAck
)
    private
{
     address implementer = _erc1820.getInterfaceImplementer(to, TOKENS_RECIPIENT_INTERFACE_HASH);
    if (implementer != address(0)) {
        IERC777Recipient(implementer).tokensReceived(operator, from, to, amount, userData, operatorData);
    } else if (requireReceptionAck) {
        require(!to.isContract(), "ERC777: token recipient contract has no implementer for ERC777TokensRecipient");
    }
}
```

As a mitigation, any new token to be listed shall pass a technical review inspired by the one in Compound and AAVE and any token with fallback functions shall not be listed.

New market support vulnerability

Another issue that has been identified in Compound is the rounding issue that happens when a CToken market is empty after deployment and has a collateral factor other than 0.
If an attacker deposits funds in an empty market and redeems most of his collateral he can redeem an infinite amount of CTokens which can be used as collateral to get loans in other markets. The most significant attack of this nature is the one that happened with Hundred Finance (https://twitter.com/HundredFinance/status/1647247792589471745 - https://www.comp.xyz/t/hundred-finance-exploit-and-compound-v2/4266) and most recently with Midas Capital (https://t.co/l9dSqXrWhs).

To mitigate this issue, any newly deployed market will have a collateral factor of 0 (which is the percentage of a deposit that can be used as a collateral to take a loan) and will have an automatic initial deposit during deployment to make sure the contract is initialized correctly. It shall be monitored for new deposits before listing it publicly and change its collateral factor to the one required by the market.

Continuous monitoring plan

One of the recent improvements in Tropykus was the monitoring of the price feeds to raise alarms to the Tropykus team in case the price of an asset had drastic changes in the previous block.

As part of our commitment to improve our security protocols we have designed an automatic mechanism to pause the borrows of the markets in case some specific events happen in the network such as the manipulation of price feeds or the deployment of a malicious smart contract.

Automatic borrows pause

Tropykus has always used a Multisig Wallet to perform administrative and security changes in the protocol by requiring at least 3 signatures to perform any change.

To secure the funds in case of an anomaly Tropykus will deploy a system that has a pre-approved market pause transaction that shall be executed automatically. In case the alarm is triggered, this system will add another signature to the multisig transaction and the pause action will take place. It will also notify the team about it in real time through several channels, ensuring a 24/7 quick response.

To restart the borrows a manual procedure shall take place to ensure that the technical team has given a green light after a security assessment.

Malicious smart contract identification

During the last attack we compared the attacking smart contract bytecode with the one used for a previous attack and we found common patterns in them that can be used to detect a potentially malicious smart contract. In most of the exploits a borrow function is used with other functions of the protocol in the same transaction, which have a characteristic hexadecimal signature which can be found in the bytecode.

Our team has listed some of these characteristic hexadecimal signatures and will be used as the base of a new system to identify newly deployed smart contracts and pause the markets as soon as a malicious smart contract is detected.

Integration of Forta Network

Forta Network is a tool to detect threats and anomalies in Real Time with Machine Learning compatible with any EVM chain like RSK, which is used by some of the largest DeFi protocols around such as LiDo, Compound, MakerDAO and AAVE, among others.

By integrating Forta into RSK we will be able to have a more robust alerting system, that can be used with other partners in RSK.

The first step on this integration will develop alerting bots for price feeds and evolve them to analyze malicious code and wallets.

Continuous improvement plan

Security is a non stop job. Every day a new issue is discovered and a new vulnerability can be exploited. Our commitment is to improve the security of our products continuously and partner with some of the most experienced teams in the industry to bring the best user experience and secure the funds of our users in all of our suite of products.

Regular monitoring of the smart contracts

With the latest advances in AI, we can perform regular scans on our code bases and detect potential issues before they can be exploited. One of the tools we will be using regularly is Solidity Scan and Mythx.

On any new identified issue we shall act promptly to take the required actions including alert our users if a critical issue is discovered.

Regular overview of Compound security checks

Given the fact that Tropykus was born as Compound fork, which has inherited their good practices and vulnerabilities, we will be monitoring constantly the findings of vulnerabilities from Compound in their own base code and follow their steps to prevent attacks from already known issues.

Partner with a security agency

For the last couple of weeks we have been in contact with experts in blockchain security to develop plans for continuous improvement, not only from a code perspective but also from an operations and development perspective. Some of the agencies we are looking forward to partnering with are Halborn and Sakundi. These deals shall be discussed and finalized in the next couple of weeks.

Partner with other Compound-based protocols to share knowledge and insights

One of the most important sources of information regarding Compound are the forks of this protocol. For the last few weeks we have been talking with some of these protocols to create a security and knowledge share community, not only to bring better standards and improve the security of Tropykus, but also the security of users of other protocols.