Sherlock Statement of Coverage for:

Name of Protocol: Nifty Options (by Teller Org, Inc.)
Agreement Start Date: 9/24/2021
Length of Term: 12 months
Coverage Amount: The lesser of $10M or the TVL of Protocol Customer
Deductible Amount: 5k USDC
Frontend location: https://niftyoptions.org/
Claims to be paid in: USDC
Termination Fee: 0 USDC

Defined Terms

Sherlock - Sherlock Protocol
Client - Protocol Customer
SPCC - Sherlock Protocol Claims Committee

Maintaining Active Coverage

Coverage of a Client is only active and valid for as long as the Client has sent more funds to the Sherlock payment account than is equal to the accumulated premium debt of the Client. If the Client's balance of "sent funds" is less than the accumulated premium debt, the Client is no longer under coverage. The coverage will end at the first block where the balance of "sent funds" drops below the accumulated premium debt. Sherlock recommends that Clients keep greater than or equal to one week's worth of extra payment in the Sherlock payment account. Even if the Client is not currently under coverage, an exploit that occurred during a block before the coverage ended is still valid and Sherlock needs to properly assess and pay out that claim when necessary.

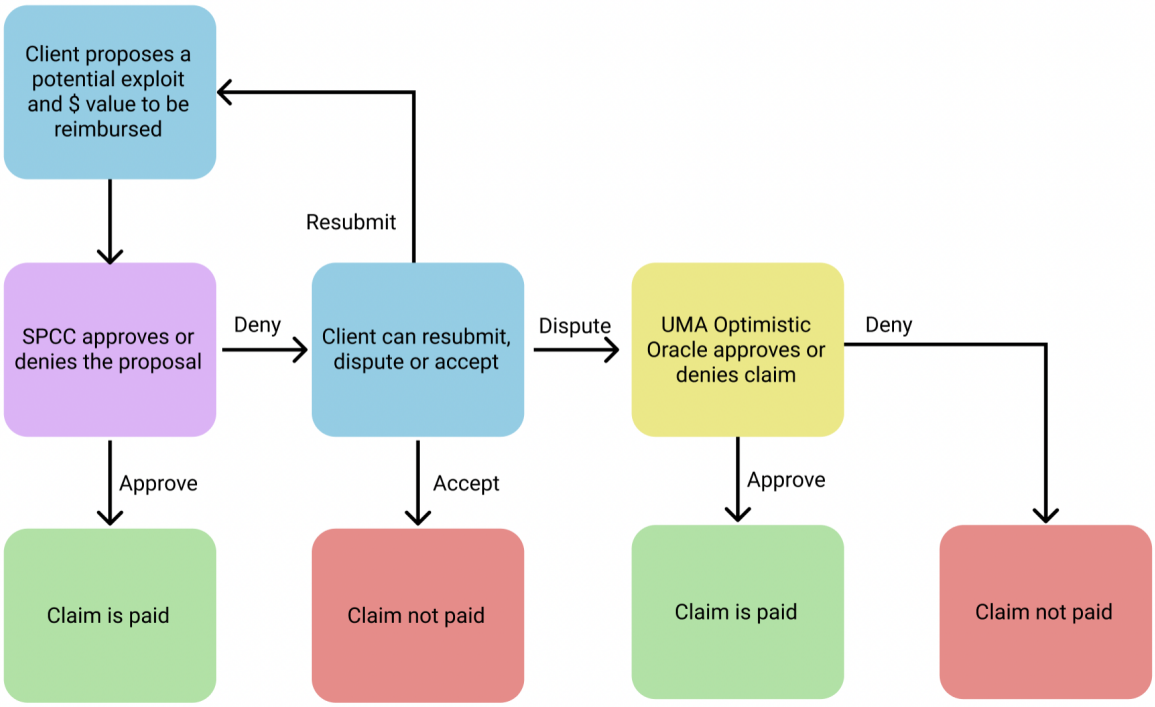
Claim Validity

A Client will bring a possible covered exploit in their protocol to the attention of Sherlock's security team. It is likely the security experts at Sherlock in charge of that Client will be involved in this process of discovering a possible exploit and understanding its nature. If there's a possibility that the exploit would be covered, the Client will be tasked with deciding the amount of the claim. It is likely the security experts at Sherlock in charge of that Client will also be heavily involved in advising the proper amount to create a claim for.

Once a possible exploit and the amount claimed by Client is brought to the attention of Sherlock, the process of deciding the validity of the claim begins. The first step is to bring the exploit and amount of the claim to the attention of the Sherlock Protocol Claims Committee (SPCC). The SPCC is made up of members of the core team of Sherlock as well as official advisors to Sherlock. These members will be well-versed in the general nature of exploits and events covered by Sherlock as detailed in this statement of coverage. This committee will be composed of some of the foremost security experts in the DeFi space. All of the members of the SPCC will have a stake in Sherlock (likely in the form of tokens or equity) and will have an interest in doing what is best for the long-term wellbeing of Sherlock. They will also have reputations and public identities existing outside of Sherlock that they will want to uphold. These factors will make it very likely that the members of the SPCC will see it in their best interest to make the most accurate claims decision possible.

The decision made by the SPCC will be binary (either a claim will be accepted or not). Once a decision is made on a claim by the SPCC, there are a few possible paths. The first path for a Client is to accept the decision. The second path is to revise the claim (usually the amount of the claim) and re-submit. A Client is limited to 3 submissions for each potential exploit (to be defined by the block number at which the potential exploit began). The third and last path for the Client is to escalate to arbitration. This would require the Client to "stake" up to 1% of the claim amount (amount will be decided by the SPCC) to escalate the claim above the SPCC. The escalation would move the claim decision from Sherlock's hands into the hands of UMA's Optimistic Oracle, more specifically UMA's Data Verification Mechanism. The claims decision will then be voted on by UMA tokenholders and the resolution of that vote will be the final claim decision (overruling the SPCC). If the Client is proven correct, then the amount specified by their claim will be paid out. They will also receive their 1% (or less) stake back in full. If the Client's escalation proves to be in vain, then the amount specified by the claim is not paid out and the 1% (or less) stake is kept by Sherlock.

Sherlock Claims Process



Paying a Claim

If the claims process results in a situation where funds need to be paid out, there are a few nuances to keep in mind. The default situation is to pay the Client back in the token specified at the top of this agreement. However, there may be situations where the Client's smart contracts may still be unstable or compromised. For example, an exploit could have occurred (which then triggered a payout) but the Client has not fully gained back control from the malicious party who caused the exploit. In this situation, Sherlock may pay out to the addresses of affected parties directly, instead of paying out to the Client. If a Client is deemed to no longer be compromised, it is much faster and more efficient (and likely more economical to end users) to pay out a claim to the Client directly. Then the Client can handle reimbursement for end users in a way that the Client sees fit. However, Sherlock's mission is to protect end users and keep end users' money safe, so Sherlock (through the SPCC) will have sole discretion as to whether a payout should be made to the Client or to affected users of the Client directly.

Deciding on Claims

When trying to decide if a claim falls under coverage or not, there are three main questions to ask (which will be explained in detail in the following pages):

1) Was there an unintended loss of user funds due to a flaw/oversight in the protocol? Basically did an exploit occur?
2) Does this exploit fall into the category of a "Known Economic Risk" explained below?
3) Does this exploit fall into a category under "Specific Events NOT Covered by Sherlock" listed below?

If 1) is true, meaning an exploit did occur, and 2) and 3) are false, then it is likely that this event should be covered and paid out by Sherlock. The reason for approaching the decision in this manner is that Sherlock provides some possibility for "unknown unknown" exploits occurring. And if this event is indeed an exploit, but Sherlock has not provided language around handling it in the letter or spirit of this document (specifically whether it should NOT be covered), then this new form of exploit should likely be covered by Sherlock.

The Spirit of Sherlock Exploit Protection

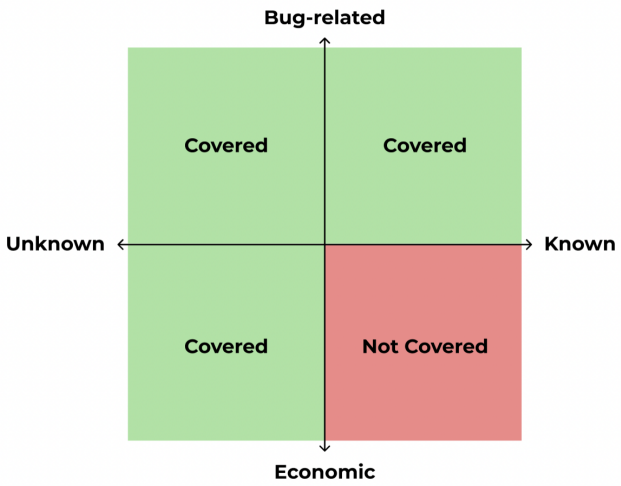
This document will outline in detail all of the areas of coverage by Sherlock against which claims can be made (or not made). Because there are always bound to be gaps in explicit wording, Sherlock also attempts to explain the "spirit" of what the later paragraphs will convey, so that unforeseen exploits can still be handled well.

Known Economic Risks

There are two important categories of coverage at Sherlock. The first is bug-related coverage. If a smart contract has a syntax error or otherwise fails to execute its logic as intended due to a mistake related to code being written improperly, that would likely be considered a bug-related incident. However, if there is still a loss of funds despite the code being technically correct in what it intended to do (as a third-party would observe), this would likely fall more in the category of an economic incident. The latter (economic incidents) are not so much a failure of code or syntax as they are a failure of economic design. The difference can be subtle and there are definitely gray areas, but generally if the literal code functions in the way a developer intended, that is likely an economic error. If the literal code does not function as expected, that is likely a bug-related error.

We can create a quadrant of coverage with four types of errors: unknown bug-related errors, known bug-related errors, unknown economic errors, and unknown economic errors. An unknown error is simply something that the developers/auditors are unaware

of (until it surfaces). This means a common bug (in a known class of bugs) can still be an unknown bug in a specific contract because it was not identified in that contract. Whether a bug-related error is known or unknown, an incident related to a smart contract bug should generally be covered. The onus is on Sherlock's security team to price known bugs properly or fix them. And unknown bugs are inherently unforeseeable so they should be covered. For unknown economic risks, the sentiment is the same. Because it was unknown (and therefore unforeseeable), it should be covered. But known economic risks are a bit different. Almost every protocol has some set of known economic risks. For example, if the value of Maker collateral falls below the value of the deposits made into the protocol, the depositors are at risk of losing funds. Same goes for almost anything related to token price volatility. If a token price goes down, holders of the token or parties who interact with that token are at risk of losing funds related to the price drop. These are examples of known economic risks. Sometimes, these risks are a large part of the reason APYs are so high for certain opportunities. These are not risks Sherlock intends to cover. The onus is on the end user to learn about and understand (as well as the protocol to teach) the known economic risks which drive the APY (or attractiveness, speculative or not) of the investment opportunity they are considering. Many of these economic risks exist in traditional finance and have existed for centuries in various markets. Sherlock's goal is to mitigate the risks that are uniquely specific to DeFi, especially the risk that code or economic designs do not function as intended. Therefore Sherlock covers only bug-related and unknown economic risks because these are the risks that users are not well-equipped to evaluate themselves.



Known Bug-Related Risks

If a developer / team understands the implications of a known bug-related risk, but deems it an "acceptable risk" for their protocol, it should still be paid out as long as the team disclosed it to (or at least did not make efforts to conceal it from) the Sherlock smart contract team. As long as the Sherlock team knows about the "acceptable" risk around the code, it can be priced properly in Sherlock's model.

However, if a team makes considerable effort to conceal (or obfuscate) a certain "acceptable risk" or known risk, Sherlock may have grounds to not pay out. This clause exists mainly to disincentivize protocol teams from concealing as many bugs/vulnerabilities as they can from the Sherlock smart contract team in order to get a lower rate for coverage.

With that in mind, Sherlock attempts to enumerate, in as clear terms as possible, the events that will or will not be covered by Sherlock coverage:

Specific Events NOT Covered by Sherlock

Token Price

Any event that is triggered by a change in token price should almost certainly not be covered by Sherlock. Any protocol should know exactly which tokens it could have the opportunity to interact with. And any protocol should have contingencies in their code for the price of all of these tokens dropping to zero or approaching infinity. The volatility of a token price is a perfect example of a "known economic risk" as recounted in the preceding section. This extends to "stablecoins" as well. However, attacks using flash loans and/or oracle manipulation are more sophisticated (but also result in large changes to a token's price) and so these types of events should usually be covered, despite a change in token price occurring as well.

Changes in token price especially apply on the user side. The risk of a token's price going down (or up in the case of short-selling) should always be considered a known risk and thus, unless there was some sophisticated manipulation accompanying it, a loss of funds caused by a change in the price of a token alone should not be a claimable event.

Collateral Shortfalls

This section is especially applicable to lending protocols and related protocols. Any lending protocol is well aware that one of the known economic risks is a shortfall in collateral, which would leave depositors unable to collect some of all of their principal. Of course, these collateral shortfalls could be caused by a bug in a smart contract, in which case Sherlock should cover the event. But a common, known economic risk of lending protocols is collateral shortfalls related to rapid and/or large changes in the price of tokens being used as collateral. This type of collateral shortfall would not be covered by Sherlock.

Unavailability of Funds

This section is especially applicable to lending protocols and related protocols. There may be situations where a depositor's tokens are not available to be withdrawn due to high utilization (on the borrowing side) of the depositor's tokens. This is a known economic risk related to lending protocols and thus would not be covered.

Approve Max / Approve Unlimited

The expectation for protocols covered by Sherlock is that they should discourage (or prevent) approving amounts (of tokens) to a contract above and beyond what is necessary for a specific transaction. Sometimes, it is not possible to entirely prevent this in the smart contracts, but it should at least be made impossible through the covered protocol's sponsored frontend/UI. Sherlock's goal is to protect end-users who may not be sophisticated users of crypto. Any user who goes against the recommendation of the sponsored UI and approves unlimited anyways can be thought of as a sophisticated user according to Sherlock. Users who approve more than they need for a specific transaction and then experience an exploit which drains funds held in their wallet (not at the covered protocol) will not be covered by Sherlock. To be clear, the user's funds that were in the protocol and lost due to an exploit would be reimbursed. Any funds taken from the user's wallet due to an exorbitantly high approval value will not be reimbursed by Sherlock.

Phishing attacks

Users affected by phishing attacks related to their wallet (Metamask, etc.) would not be covered by a specific protocol's policy. Even if the tokens involved were tokens related to or distributed by a specific protocol that has a policy with Sherlock.

Phishing attacks related to "fake" websites (i.e. websites hosted at domains other than the protocol's sponsored website/app) would also not be covered. The onus is on the user to ensure they are actually interacting with a covered protocol, not a duplicate, replica, or look-alike website or protocol.

Phishing attacks spawning from a covered protocol's sponsored website/app are also not covered (such as hijacking a DApp's DNS). Sherlock currently does not have the resources to ensure and monitor the security of website / frontend-related vulnerabilities, but this may change in the future. If getting coverage for this kind of attack is very high priority for a protocol team, we ask that the team to reach out to us.

Front-end bugs

In the same vein as phishing attacks, Sherlock currently does not have the resources to ensure and monitor the security of website / frontend-related vulnerabilities, but this may change in the future. So Sherlock cannot cover any unintended loss of funds resulting from an exploit/bug in the frontend of a protocol customer. This means that code related to libraries like Web3.js or Ethers.js cannot be covered even if it is interacting with smart contracts. The code covered must be deployed on a blockchain and frontend code does not meet this criteria.

Transaction Ordering Attacks / Frontrunning / Sandwich Attacks / MEV-Related Attacks

These types of attacks involve malicious addresses (often controlled by bots) that spot profitable transactions in the mempool and then execute the transaction themselves in order to capture the profit. Or the malicious address sees a certain state change that will be caused by a transaction in the mempool, and calls a function or executes a transaction to take advantage of that state change. The biggest reason that Sherlock cannot cover these types of attacks is because the potential for fraud is too high. If a user or protocol "loses" funds because their transaction is front-run in the mempool, it is very difficult for Sherlock to know that the address doing the frontrunning is not also controlled by the same user or protocol.

However, in certain cases, these types of events would be covered by Sherlock. If, for whatever reason, a protocol tries to pass private or randomness-reliant information through the mempool, this should be covered by Sherlock (see "Specific Known Bug-Related Attacks" below) because the Sherlock security team should catch these types of bugs and in those cases it is fairly clear that unsound logic was being used in the code. In other cases, it's not always clear what the intentions of the developers were and therefore Sherlock cannot cover those cases.

Another area where this would be covered is simply bad logic in the protocol which doesn't check for certain conditions. The specific example here is the ERC20 *approve* race-condition exploit.

Rug Pulls / Admin Rights / Off-limits functionality

The unauthorized accessing of any function or contract where access is white-listed or entirely disallowed is NOT covered. Sherlock strongly recommends multi-signature admin functionality for all accounts and admin contracts.

The risk of funds being lost in a single signature setup is too high for Sherlock to cover. And in a multi-signature setup, the preponderance of evidence related to a loss of funds points to a rug pull, which is a situation Sherlock does not intend to cover. Therefore Sherlock cannot cover ANY “admin”-related exploits. Sherlock is not able to accurately assess these types of exploits currently and so the price of premiums would be far too high if these risks were covered by Sherlock. Sherlock is working to expand its coverage in this area. But for now, any exploit related to privileged access (without an accompanying covered exploit), will not be covered by Sherlock.

This also applies to any governance-induced loss of funds. If a majority of token holders decides to vote maliciously in any way, that cannot be covered. For example, if a majority of token holders decide to transfer a minority’s share of tokens to themselves, this would not be covered. And if a malicious party somehow acquires enough tokens to make a malicious change through governance, this also should not be covered.

Note: A bug related to a missing (or incorrect) access control check (such as a missing modifier) would be covered. This is a mistake in the code, not a “rug pull” necessarily.

Specific Events That Should Not Be Relied on for Decisions

Flash Loan

A flash loan by itself is simply a way to acquire more tokens. Any attack that can be accomplished with a flash loan can also be accomplished without a flash loan (by a whale, etc.). Therefore, the presence of a flash loan does not necessarily mean that an exploit has occurred. However, flash loans are often accompanied by other events (oracle manipulation, etc.) which are exploits. And, of course, if a flash loan is a part of a broader unknown economic attack, then the event should be covered. If the flash loan is simply taking advantage of a known economic attack (liquidation may occur if a token price drops), then it would not be covered by Sherlock. The presence of flash loans by themselves in a potential exploit event are not good indicators of whether an event should be covered or not.

Specific Events Covered by Sherlock

Oracle Manipulation

Although oracle manipulations should be fairly “known” economic risks by now, this type of exploit is so new (and somewhat technical) that, in most cases, Sherlock still considers

unexpected oracle behavior to be an “unknown economic risk” to end users and protocol developers. However, Sherlock security experts are well aware of this risk, so it should be priced properly with the knowledge that it will usually be a covered event at Sherlock. The industry seems to be learning its lesson and moving away from “spot price” oracles to time-weighted average price oracles but exceptions still exist. Events where the oracle (or collection of oracles) actually gives an incorrect price should definitely be covered, assuming losses are sustained. And although they are more foreseeable, events where a malicious party “leans on the scale” in order to manipulate a certain oracle should also be covered.

Specific Known “Bug-related” Attacks

- Integer underflow/overflow
- Reentrancy including cross-function reentrancy
- Silent failing sends / unchecked sends / unchecked low-level calls / delegatecall to untrusted callee
- Unbound loops
- Self-destruct-related exploits / forcibly sending Ether to a contract
- Absence of required participants
- Denial-of-service due to fallback function, gas limit reached, unexpected throw, unexpected kill
- False randomness / reliance on “private” information being sent through the mempool
- Time manipulation / timestamp dependence
- Short address attacks
- Insufficient gas griefing
- Authorization through tx.origin
- Uninitialized storage pointer
- Floating pragma / outdated compiler version / compiler-related bugs
- Missing checks / callable initialization function
- Missing variables / using the wrong variable
- Proxy/upgradability-related attacks (such as the OpenZeppelin UUPS bug)
- External dependencies (such as OpenZeppelin libraries)

Known “bug-related” attacks not listed here

The list of specific, known bug-related attacks above is surely incomplete, but is provided mainly for convenience. Any attack that can be classified as bug-related but is not listed under “Specific Events NOT Covered By Sherlock” should inherently be covered by Sherlock.

Events that Combine Different Attacks

Many exploits combine multiple types of events to disrupt a protocol. As long as just one of the events in the combined attack is determined to be covered by Sherlock, then the entire aggregated attack should be covered.

Attributes Specific to Client

Test Suite

Test coverage: Good
Quality of tests: High

Best Practices

Proper formatting: Yes
Readability: High
Commenting: Good

Composition

Size of Codebase: Very small
Code Complexity: Low

Composability

Composability with other protocols: None
Use of oracles: None

Blockchains

Blockchain: Ethereum
Multi-chain: No
L2s: No

Libraries/Contracts

Upgradeable: Yes
Use of battle-tested libraries where possible: Yes
Tokens used: WETH
External contracts/interfaces used: IERC721 (OpenZeppelin), IERC721Metadata (OpenZeppelin), IERC1155 (OpenZeppelin), IERC1155MetadataURI (OpenZeppelin), IERC721ReceiverUpgradeable (OpenZeppelin), IERC1155ReceiverUpgradeable (OpenZeppelin), IERC20 (OpenZeppelin), SafeERC20 (OpenZeppelin), ERC721Upgradeable

(OpenZeppelin), Initializable (OpenZeppelin), OwnableUpgradeable (OpenZeppelin), Strings (OpenZeppelin)

Security Suggestions Made to Client

Lead Security Expert: Janbro
Support team: Evert Kors

Contracts reviewed:

- Teller-Options-Master-Repo/packages/hardhat/contracts/Option.sol
- Teller-Options-Master-Repo/packages/hardhat/contracts/OptionURIFetcher.sol

Commit hash:

- 58daac0c15a6531689d46493747dcd9becbb80a0

Total Suggestions: 9

Critical: 0
High: 3
Medium: 1
Low: 2
Note: 3

ISSUE 1

Summary

line 155:

```
function cancelOption(uint256 optionId) public onlyOptionOwner(optionId) {
```

cancelOption does not return option creators `incentiveAmountWei` when option is cancelled in undefined state (before the option is filled).

Risk Rating

Medium

Vulnerability Details

If a user calls `createOption` with a non zero `msg.value`, that amount will be locked in the contract if the option is cancelled before it is filled.

Impact

Locked funds

Tools Used

Manual code review

Recommended Mitigation Steps

Refund the `opt.incentiveAmountWei` when the option is cancelled in the `undefined` state

```
if (originalStatus == OptionStatus.undefined) {
    // Refund opt.incentiveAmountWei
    payable(ownerOf(optionId)).transfer(
        opt.incentiveAmountWei
    );
}
```

Mitigated by client

Yes, the `incentiveAmountWei` variable is renamed to `premiumAmountWei` and returned to the option creator in WETH.

ISSUE 2

Summary

line 285:

`_transferOptionBundle(optionId, opt.optionFiller);`

When `exerciseOption` is called, the token bundle is transferred to the `optionFiller`. According to ERC1155 standard, `safeTransferFrom` calls the `onERC1155Received` of the receiver if it is a contract.

Risk Rating

High

Vulnerability Details

This puts execution flow into the receiver and allows a malicious receiver to revert in their `onERC1155Received` function, causing a DOS on the exerciseOption function. This could allow a `optionFiller` to revert if the underlying token bundle decreases in price below the option `buyoutPriceWei` until they are able to expire the option and withdraw their WETH.

Impact

The option creator would not be able to exercise their option.

Tools Used

Manual Code Review

Recommended Mitigation Steps

A pull pattern should be utilized for option fillers to receive their token bundle after an option is exercised.

Mitigated by client

Yes, WETH is being used instead of native ETH, this mitigates the issue as reverts can not be triggered on the receival of WETH.

ISSUE 3

Summary

line 209:

```
function expireOption(uint256 optionId) public {
    ...
    payable(opt.optionFiller).transfer(
        opt.buyoutPriceWei
    );
    ...
}
```

Option can be prevented from expiring and therefore lock NFTs by opt.optionFiller reverting in fallback payable function.

Risk Rating

High

Vulnerability Details

An option owner can be prevented from expiring their option and receiving their token bundle if the option filler implements a fallback payable function and reverts. This means a option filler can hold an option creators token bundle hostage with a smart contract.

Impact

Option creators token bundle is locked in the contract

Proof of Concept

```
pragma solidity 0.6.12;

import './Option.sol';

contract Ransom {
   bool internal ransomEnabled = true;
   address internal optionContract;

   constructor(address _optionContract) public {
       optionContract = _optionContract;
   }

   function fillOption(uint256 optionId) public payable {
       optContract = Option(optionContract);
       optContract.fillOption(optionId).value(msg.value);
   }

   function unlockNFT() public payable {
      if(msg.value >= 1 ether) {
         ransomEnabled = false;
      }
   }

   fallback() external payable {
      require(!ransomEnabled);
   }
}
```

Tools Used

Manual code review

Recommended mitigation

Use pull payment rather than push payment pattern

Mitigated by client

Yes, WETH is being used instead of native ETH, this mitigates the issue as reverts can not be triggered on the receival of WETH.

ISSUE 4

Summary

line 170:

```
function cancelOption(uint256 optionId) public onlyOptionOwner(optionId)  {
    ...
    payable(opt.optionFiller).transfer(
        opt.buyoutPriceWei
    );
    ...
}
```

Option can be prevented from being cancelled and therefore lock NFTs by opt.optionFiller reverting in fallback payable function.

Risk Rating

High

Vulnerability Details

An option owner can be prevented from cancelling their option and receiving their token bundle if the option filler implements a fallback payable function and reverts. This means a option filler can hold an option creators token bundle hostage with a smart contract.

Impact

Option creators token bundle is locked in the contract

Proof of Concept

```
pragma solidity 0.6.12;

import './Option.sol';

contract Ransom {
   bool internal ransomEnabled = true;
   address internal optionContract;

   constructor(address _optionContract) public {
       optionContract = _optionContract;
   }

   function fillOption(uint256 optionId) public payable {
       optContract = Option(optionContract);
       optContract.fillOption(optionId).value(msg.value);
   }
```

```
    function unlockNFT() public payable {
        if(msg.value >= 1 ether) {
            ransomEnabled = false;
        }
    }

    fallback() external payable {
        require(!ransomEnabled);
    }
}
```

Tools Used

Manual code review

Recommended mitigation

Use pull payment rather than push payment pattern

Mitigated by client

Yes, WETH is being used instead of native ETH, this mitigates the issue as reverts can not be triggered on the receival of WETH.

ISSUE 5

Summary

line 170:

`payable(opt.optionFiller).transfer(`

line 209:

`payable(opt.optionFiller).transfer(`

line 249:

`payable(_msgSender()).transfer(opt.incentiveAmountWei);`

line 280:

`payable(ownerOf(optionId)).transfer(`

Gas costs are subject to change and can cause `.transfer()` to fail in the future. Additionally, there are better methods to prevent reentrancy such as reentrant guards or to utilize the checks-effects-interactions pattern.

Risk Rating

Low

Vulnerability Details

`.call.value(...)` should be used instead of `.transfer(...)` or `.send(...)`.

https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/

Impact

`.transfer()` could fail in the future if gas costs are updated.

Tools Used

Manual Code Review

Recommended Mitigation Steps

Utilize low level `.call{value: amount }()`

Mitigated by client

Yes, WETH is being used instead of native ETH, this mitigates the issue.

ISSUE 6

Summary

line 134:

```
...
opt.buyoutPriceWei = buyoutPriceWei;
opt.incentiveAmountWei = msg.value;
...
```

If `opt.buyoutPriceWei` is less than `opt.incentiveAmountWei`, a user can fill an option for a net positive amount of ETH with no regard for the option.

Risk Rating

Low

Impact

A user could create an unfavorable option.

Tools Used

Manual code review

Recommended Mitigation Steps

Ensure `opt.buyoutPriceWei > opt.incentiveAmountWei`

eg.

line 125:

```
require(buyoutPriceWei > msg.value);
```

Mitigated by client

Yes

ISSUE 7

Summary

line 33:

```
enum OptionStatus {
    undefined,
    filled,
    exercised,
    cancelled
}
```

`undefined` is a risky and undescriptive name for OptionStatus state.

Risk Rating

Note

Impact

Readability

Tools Used

Manual code review

Recommended Mitigation Steps

Rename to `open`

Mitigated by client

Yes

ISSUE 8

Summary

line 44:

```
// TODO: support ERC20
```

line 169:

```
// TODO: change buyout token from ETH to WETH (ERC20)
```

line 208:

```
// TODO: change buyout token from ETH to WETH (ERC20)
```

line 247:

```
// TODO: change buyout token from ETH to WETH (ERC20)
```

Implement TODO functionality or remove comments

Risk Rating

Note

Impact

Readability

Tools Used

Manual code review

Recommended Mitigation Steps

Implement TODO functionality or remove comments

Mitigated by client

Yes

ISSUE 9

Summary

line 91:

```
function bundleOf(uint256 optionId) public view returns (TokenBundle memory)
{
```

line 119:

```
createOption(Option.TokenBundle,uint256,uint32) public payable returns
(uint256 optionId_) {
```

line 155:

```
function cancelOption(uint256 optionId) public onlyOptionOwner(optionId) {
```

line 189:

```
function expireOption(uint256 optionId) public {
```

line 228:

```
function fillOption(uint256 optionId) public payable {
```

line 264:

```
function exerciseOption(uint256 optionId) public onlyOptionOwner(optionId) {
```

Public functions can be declared external

Risk Rating

Note

Impact

Increased gas usage

Tools Used

Manual code review

Recommended Mitigation Steps

Declare functions which do not need to be called internally as external

Mitigated by client

Yes