Putty contest
Findings & Analysis Report

Report published: 2022-08-08. Updated with Mitigation Review: 2022-09-08.

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Putty smart contract system written in Solidity. The audit contest took place between June 29—July 4, 2022.

Following the C4 audit contest, warden hyh reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit contest report.

Wardens

144 Wardens contributed reports to the Putty contest:

1. hansfriese
2. hyh
3. minhquanym
4. csanuragjain
5. berndartmueller
6. xiaoming90
7. zzzitron
8. llllllll
9. unforgiven
10. horsefacts
11. kirk-baird
12. Lambda
13. shung
14. hubble (ksk2345 and shri4net)
15. Metatron
16. 0xsanson

17. reassor
18. Kenshin
19. sseefried
20. 0xNineDec
21. cccz
22. danb
23. PwnedNoMore (izhuer, ItsNio and papr1ka2)
24. auditor0517
25. 0x52
26. zishansami
27. sashik_eth
28. pedroais
29. exd0tpy
30. 0xc0ffEE
31. Treasure-Seeker
32. Alex the Entreprenerd
33. shenwilly
34. swit
35. BowTiedWardens (BowTiedHeron, BowTiedPickle, m4rio_eth, Dravee, and BowTiedFirefox)
36. TrungOre
37. 0xA5DF
38. Picodes
39. joestakey
40. 0x1f8b
41. ACai
42. ignacio
43. defsec
44. catchup

45. Critical
46. codexploder
47. zer0dot
48. dirk_y
49. antonttc
50. itsmeSTYJ
51. 0xDjango
52. 0xf15ers (remora and twojoy)
53. Bnke0x0
54. Chom
55. StErMi
56. ElKu
57. 0xNazgul
58. simon135
59. oyc_109
60. __141345__
61. MiloTruck
62. TomJ
63. JohnSmith
64. gogo
65. Funen
66. _Adam
67. cryptphi
68. rokinot
69. JC
70. Kaiziron
71. hake
72. robee
73. Limbooo

74. Waze
75. ReyAdmirado
76. MadWookie
77. fatherOfBlocks
78. durianSausage
79. datapunk
80. delfin454000
81. rajatbeladiya
82. Hawkeye (0xwags and 0xmint)
83. Yiko
84. Sm4rty
85. AmitN
86. 0x29A (0x4non and rotcivegaf)
87. peritoflores
88. samruna
89. async
90. GimelSec (rayn and sces60107)
91. Nethermind
92. saneryee
93. doddle0x
94. 0xSolus
95. aysha
96. David_
97. Sneakyninja0129
98. TerrierLover
99. chatch
100. 0xkatana
101. rfa
102. grrwahrr

103. 0xKitsune
104. RedOneN
105. UnusualTurtle
106. Aymen0909
107. saian
108. Tomio
109. c3phas
110. jayfromthe13th
111. z3s
112. 0v3rf10w
113. ajtra
114. ak1
115. Fitraldys
116. m_Rassska
117. mektigboy
118. mrpathfindr
119. natzuu
120. Randyyy
121. slywaters
122. sach1r0
123. cRat1st0s
124. ladboy233
125. zeesaw
126. 0xHarry
127. apostle0x01
128. asutorufos
129. codetilda
130. Haruxe
131. Ruhum

132. StyxRave
133. dipp

This contest was judged by **hickuphh3**.

Mitigations reviewed by **hyh**.

Final report assembled by **itsmetechjay**.

Summary

The C4 analysis yielded an aggregated total of 20 unique vulnerabilities. Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity and 16 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 82 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 94 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the **C4 Putty contest repository**, and is composed of 2 smart contracts written in the Solidity programming language and includes 357 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on **OWASP standards**.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling

- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (4)

[H-01] Fee is being deducted when Put is expired and not when it is exercised.

Submitted by zishansami, also found by 0x52, 0xsanson, auditor0517, berndartmueller, csanuragjain, and zzzitron

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L495-L503

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L451

Impact

Fee is being deducted when Put is expired and not when it is exercised in `PuttyV2.sol`. Comment section of the `setFee()` function mentions `"fee rate that is applied on exercise"` which signifies that the fee amount is meant to be deducted from strike only when a position is being exercised (or has been exercised).

But, in function `withdraw()` at PuttyV2.sol#L495-L503 the fee is being deducted even when the Put position is not exercised and has expired.

Also, in function `exercise()` there is no fee deduction from the `order.strike` when the Put position is exercised and the strike is being transferred to the caller (PuttyV2.sol#451).

This unintended deduction from assets of Put Shorter and the absence of fee deduction from strike when Put is exercised are directly impacting the assets and therefore marked as Medium Risk.

Proof of Concept

`if` condition present at PuttyV2.sol#L495 passes if `order.isCall` is `false` and `isExercised` is false.

`feeAmount` becomes positive if `fee > 0` and it gets deducted from the `order.strike` which gets transferred to `msg.sender` at line number PuttyV2.sol#503.

Recommended Mitigation Steps

1. Update `if` condition at PuttyV2.sol#L498 with `(fee > 0 && order.isCall && isExercised)`
2. Add feeAmount calculation and deduction after put is exercised and strike is transferred at PuttyV2.sol#L451 as follows:

```
uint256 feeAmount = 0;
if (fee > 0) {
    feeAmount = (order.strike * fee) / 1000;
    ERC20(order.baseAsset).safeTransfer(owner(), feeAmount);
}
ERC20(order.baseAsset).safeTransfer(msg.sender, order.strike - f
```

outdoteth (Putty Finance) confirmed and commented:

> Report: Fees are only applied on puts if they are expired.

HickupHH3 (judge) increased severity to High and commented:

> Due to incorrect logic, there are 2 consequences of separate severities:

1. Expired put option being charged the admin fee. As @berndartmueller mentioned in #380, the fee should be charged on the premium (actually this is another issue, see #373). Since it is possible for the fee amount to be greater

than expected, I consider this to be a loss of assets and therefore given a high severity rating.

2. Put option not being charged fee upon exercising it. This can be considered to the "protocol leaked value" and thus be given a medium severity rating.

> Issues that mention (1) or both (1) and (2) will be given a high severity rating, those that mention only (2) will be given a medium.

outdoteth (Putty Finance) resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/4.

hyh (warden) reviewed mitigation:

> Fixed by changing the fee base to be `order.premium` `PR#4`, which is now paid uniformly for all option types on order filling. Utilizing `order.strike` as the fee base was the root cause for M-04, M-06, M-11, M-15, so the change to `order.premium` was a shared mitigation for all of them.

[H-02] `acceptCounterOffer()` May Result In Both Orders Being Filled

Submitted by kirk-baird, also found by csanuragjain, hansfriese, Lambda, and minhquanym

When a user is attempting to accept a counter offer they call the function acceptCounterOffer() with both the `originalOrder` to be cancelled and the new `order` to fill. It is possible for an attacker (or any other user who happens to call `fillOrder()` at the same time) to fill the `originalOrder` before `acceptCounterOffer()` cancels it.

The impact is that both `originalOrder` and `order` are filled. The `msg.sender` of `acceptCounterOffer()` is twice as leveraged as they intended to be if the required token transfers succeed.

Proof of Concept

acceptCounterOffer() calls `cancel()` on the original order, however it will not revert if the order has already been filled.

```
function acceptCounterOffer(
    Order memory order,
    bytes calldata signature,
    Order memory originalOrder
) public payable returns (uint256 positionId) {
    // cancel the original order
    cancel(originalOrder);


    // accept the counter offer
    uint256[] memory floorAssetTokenIds = new uint256[](0);
    positionId = fillOrder(order, signature, floorAssetToker
}
```

cancel() does not revert if an order has already been filled it only prevents future `fillOrder()` transactions from succeeding.

```
function cancel(Order memory order) public {
    require(msg.sender == order.maker, "Not your order");


    bytes32 orderHash = hashOrder(order);


    // mark the order as cancelled
    cancelledOrders[orderHash] = true;


    emit CancelledOrder(orderHash, order);
}
```

Therefore any user may front-run the `acceptCounterOffer()` transaction with a `fillOrder()` transaction that fills the original order. As a result the user ends up filling both `order` and `originalOrder`. Then `acceptCounterOffer()` cancels the `originalOrder` which is essentially a no-op since it's been filled and continues to fill the new `order` resulting in both orders being filled.

Recommended Mitigation Steps

Consider having `cancel()` revert if an order has already been filled. This can be done by adding the following line `require(_ownerOf[uint256(orderHash)] == 0)`.

outdoteth (Putty Finance) confirmed and commented:

> Report: It's possible to fill an order twice by accepting a counter offer for an already filled order.

outdoteth (Putty Finance) resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/2.

hyh (warden) reviewed mitigation:

> Fixed by requiring that order can't be in the filled state on cancel. This fully adheres to the original logic, but wasn't controlled for before.

[H-03] Create a short call order with non empty floor makes the option impossible to exercise and withdraw

Submitted by zzzitron, also found by danb, Kenshin, Metatron, minhquanym, and PwnedNoMore

HIGH - assets can be lost

If a short call order is created with non empty floorTokens array, the taker cannot exercise. Also, the maker cannot withdraw after the expiration. The maker will still get premium when the order is filled. If the non empty floorTokens array was included as an accident, it is a loss for both parties: the taker loses premium without possible exercise, the maker loses the locked ERC20s and ERC721s.

This bug is not suitable for exploitation to get a 'free' premium by creating not exercisable options, because the maker will lose the ERC20s and ERC721s without getting any strike. In that sense it is similar but different issue to the `Create a`

`short put order with zero tokenAmount makes the option impossible to exercise`, therefore reported separately.

Proof of Concept

- proof of concept
- reference case

The proof of concept shows a scenario where babe makes an short call order with non empty `floorTokens` array. Bob filled the order, and now he has long call option NFT. He wants to exercise his option and calls `exercise`. There are two cases.

- case 1: he calls exercise with empty `floorAssetTokenIds` array
- case 2: he calls exercise with non-empty `floorAssetTokenIds` array with matching length to the `orders.floorTokens`

In the case1, the input `floorAssetTokenIds` were checked to be empty for put orders, and his call passes this requirement. But eventually `_transferFloorsIn` was called and he gets `Index out of bounds` error, because `floorTokens` is not empty which does not match with empty `floorAssetTokenIds`.

```
// case 1
  // PuttyV2.sol: _transferFloorsIn called by exercise
  // The floorTokens and floorTokenIds do not match the lenghts
  // floorTokens.length is not zero, while floorTokenIds.length

      ERC721(floorTokens[i]).safeTransferFrom(from, address(thi
```

In the case2, the input `floorAssetTokenIds` were checked to be empty for put orders, but it is not empty. So it reverts.

```
// case2
// PuttyV2.sol: exercise
// non empty floorAssetTokenIds array is passed for put option,

        !order.isCall
            ? require(floorAssetTokenIds.length == order.floorTo
```

```
            : require(floorAssetTokenIds.length == 0, "Invalid f
```

After the option is expired, the maker - babe is trying to withdraw but fails due to the same issue with the case1.

```
    // maker trying to withdraw
    // PuttyV2.sol: withdraw

      _transferFloorsOut(order.floorTokens, positionFloorAssetToken1
```

Note on the PoC:

- The test for case1 is commented out because foundry could not catch the revert. But by running the test with un-commenting these lines will show that the call reverts with `Index out of bounds`.
- For the same reason the withdraw also is commented out
- The reference case just shows that it works as intended when the order does not contain non-empty `floorTokens`.

Tools Used

Foundry.

Recommended Mitigation Steps

It happens because the `fillOrder` does not ensure the `order.floorTokens` to be empty when the order is short call.

STYJ (warden) commented:

> Note that it is possible to cause loss of funds for others through this.
>
> Assume that maker (A) creates a long call and taker (B) fills it, transferring floor tokens (XYZ) into putty.
>
> If maker (C) creates a short call with floorTokens (XYZ), taker (D) is able to fill and exercise his long call since XYZ already resides on Putty. This will however

> invalidate the options pair that was created between A and B since A cannot exercise and B cannot withdraw.

outdoteth (Putty Finance) commented:

> Agree that this should be marked as high severity given the exploit scenario provided by @STYJ above.

outdoteth (Putty Finance) confirmed and commented:

> Report: Short call with floorTokens will result in a revert when exercising.

HickupHH3 (judge) commented:

> Agreed, all wardens gave the same scenario that leads to a direct loss of NFTs and premium, but @STYJ's exploit scenario raises the gravity of the situation since users can be griefed.

outdoteth (Putty Finance) resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/1.

hyh (warden) reviewed mitigation:

> Fixed by prohibiting non-empty `order.floorTokens` for short calls.

> Other option types do need `floorTokens` : long calls' taker provides floor tokens on filling, while long put owner brings in the floor tokens on exercise, taking the strike. Short put owner can thereafter retrieve the tokens on withdraw.

[H-04] Zero strike call options can be systemically used to steal premium from the taker

Submitted by hyh, also found by hansfriese

Some non-malicious ERC20 do not allow for zero amount transfers and order.baseAsset can be such an asset. Zero strike calls are valid and common enough derivative type. However, the zero strike calls with such baseAsset will not be

able to be exercised, allowing maker to steal from the taker as a malicious maker can just wait for expiry and withdraw the assets, effectively collecting the premium for free. The premium of zero strike calls are usually substantial.

Marking this as high severity as in such cases malicious maker knowing this specifics can steal from taker the whole premium amount. I.e. such orders will be fully valid for a taker from all perspectives as inability to exercise is a peculiarity of the system which taker in the most cases will not know beforehand.

Proof of Concept

Currently system do not check the strike value, unconditionally attempting to transfer it:

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L435-L437

```
            } else {
                ERC20(order.baseAsset).safeTransferFrom(msg.senc
            }
```

As a part of call exercise logic:

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L422-L443

```
    function exercise(Order memory order, uint256[] calldata flc
        ...

        if (order.isCall) {
            // -- exercising a call option

            // transfer strike from exerciser to putty
            // handle the case where the taker uses native ETH i
            if (weth == order.baseAsset && msg.value > 0) {
                // check enough ETH was sent to cover the strike
                require(msg.value == order.strike, "Incorrect ET
```

```
            // convert ETH to WETH
            // we convert the strike ETH to WETH so that the
            // - because withdraw() assumes an ERC20 interfa
            IWETH(weth).deposit{value: msg.value}();
        } else {
            ERC20(order.baseAsset).safeTransferFrom(msg.senc
        }

        // transfer assets from putty to exerciser
        _transferERC20sOut(order.erc20Assets);
        _transferERC721sOut(order.erc721Assets);
        _transferFloorsOut(order.floorTokens, positionFloorA
    }
```

Some tokens do not allow zero amount transfers:

https://github.com/d-xo/weird-erc20#revert-on-zero-value-transfers

This way for such a token and zero strike option the maker can create short call order, receive the premium:

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L327-L339

```
        if (weth == order.baseAsset && msg.value > 0) {
            // check enough ETH was sent to cover the premiu
            require(msg.value == order.premium, "Incorrect E

            // convert ETH to WETH and send premium to maker
            // converting to WETH instead of forwarding nati
            // 1) active market makers will mostly be using
            // 2) attack surface for re-entrancy is reduced
            IWETH(weth).deposit{value: msg.value}();
            IWETH(weth).transfer(order.maker, msg.value);
        } else {
            ERC20(order.baseAsset).safeTransferFrom(msg.senc
        }
```

Transfer in the assets:

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L366-L371

```
// filling short call: transfer assets from maker to cor
if (!order.isLong && order.isCall) {
    _transferERC20sIn(order.erc20Assets, order.maker);
    _transferERC721sIn(order.erc721Assets, order.maker);
    return positionId;
}
```

And wait for expiration, knowing that all attempts to exercise will revert:

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L435-L437

```
    } else {
        ERC20(order.baseAsset).safeTransferFrom(msg.senc
    }
```

Then recover her assets:

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L508-L519

```
// transfer assets from putty to owner if put is exercis
if ((order.isCall && !isExercised) || (!order.isCall &&
    _transferERC20sOut(order.erc20Assets);
    _transferERC721sOut(order.erc721Assets);

    // for call options the floor token ids are saved ir
    // and for put options the floor tokens ids are save
    uint256 floorPositionId = order.isCall ? longPositic
    _transferFloorsOut(order.floorTokens, positionFloor/

    return;
```

```
            }
```

Recommended Mitigation Steps

Consider checking that strike is positive before transfer in all the cases, for example:

```
            } else {
+               if (order.strike > 0) {
                    ERC20(order.baseAsset).safeTransferFrom(msg.
+               }
            }
```

Alex the Entreprenerd (warden) commented:

> Seems contingent on token implementation, however certain ERC20 do revert on 0 transfer and there would be no way to exercise the contract in that case.

outdoteth (Putty Finance) confirmed and commented:

> Report: Cannot exercise call contract if strike is 0 and baseAsset reverts on 0 transfers.

HickupHH3 (judge) commented:

> There is a pre-requisite for the ERC20 token to revert on 0 amount transfers. However, the warden raised a key point: zero strike calls are common, and their premium is substantial. The information asymmetry of the ERC20 token between the maker and taker is another aggravating factor.

outdoteth (Putty Finance) resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/3.

hyh (warden) reviewed mitigation:

> Fixed by conditioning call's logic on `order.strike > 0`. There is no use case for zero strike puts and so this case remains unconditioned, i.e. still always require successful `order.strike` transfer.

Medium Risk Findings (16)

[M-01] Malicious Token Contracts May Lead To Locking Orders

*Submitted by kirk-baird, also found by 0xA5DF, cccz, chatch, csanuragjain, Alex the Entreprenerd, hansfriese, hyh, itsmeSTYJ, Kenshin, pedroais, sashik*eth, unforgiven, and xiaoming90_

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L79

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L80

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L81

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L72

Impact

It is possible to prevent an order from executing `exercise()` or `withdraw()` by having a malicious token contract included in the order as part of the any of the following fields.

- `baseAsset`
- `floorTokens[]`
- `erc20Assets[]`
- `erc721Assets[]`

An attacker as a maker may create an order and set one of these addresses to a malicious contract in the attackers control. The attacker allows the user to fill the order then toggles a variable on the malicious contract which always causes it to revert.

The attacker benefits by preventing orders from being `exercise()` if they are in an undesirable position (e.g. if they have gone short and the price has gone up). The attacker waits for either the time to expire or the price to go down and allows transfers to occur on their malicious token.

Similar attacks can also be performed over the `withdraw()` function since this also makes calls to untrusted external addresses. This would allow an attacker to exercise an option then prevent the other user from claiming any of the NFTs or ERC20 tokens that are owed to them.

Proof of Concept

Any of the transfers in **exercise** make external calls to untrusted addresses.

```
function exercise(Order memory order, uint256[] calldata flo
    /* ~~~ CHECKS ~~~ */

    bytes32 orderHash = hashOrder(order);

    // check user owns the position
    require(ownerOf(uint256(orderHash)) == msg.sender, "Not

    // check position is long
    require(order.isLong, "Can only exercise long positions'

    // check position has not expired
    require(block.timestamp < positionExpirations[uint256(or

    // check floor asset token ids length is 0 unless the po
    !order.isCall
        ? require(floorAssetTokenIds.length == order.floorTo
        : require(floorAssetTokenIds.length == 0, "Invalid f
```

```
/* ~~~ EFFECTS ~~~ */


// send the long position to 0xdead.
// instead of doing a standard burn by sending to 0x000.
// to 0xdead ensures that the same position id cannot be
transferFrom(msg.sender, address(0xdead), uint256(orderH


// mark the position as exercised
exercisedPositions[uint256(orderHash)] = true;


emit ExercisedOrder(orderHash, floorAssetTokenIds, order


/* ~~~ INTERACTIONS ~~~ */


if (order.isCall) {
    // -- exercising a call option


    // transfer strike from exerciser to putty
    // handle the case where the taker uses native ETH i
    if (weth == order.baseAsset && msg.value > 0) {
        // check enough ETH was sent to cover the strike
        require(msg.value == order.strike, "Incorrect ET


        // convert ETH to WETH
        // we convert the strike ETH to WETH so that the
        // - because withdraw() assumes an ERC20 interfa
        IWETH(weth).deposit{value: msg.value}();
    } else {
        ERC20(order.baseAsset).safeTransferFrom(msg.send
    }


    // transfer assets from putty to exerciser
    _transferERC20sOut(order.erc20Assets);
    _transferERC721sOut(order.erc721Assets);
    _transferFloorsOut(order.floorTokens, positionFloorA
} else {
```

```
            // -- exercising a put option

            // save the floor asset token ids to the short posit
            uint256 shortPositionId = uint256(hashOppositeOrder(
            positionFloorAssetTokenIds[shortPositionId] = floorA

            // transfer strike from putty to exerciser
            ERC20(order.baseAsset).safeTransfer(msg.sender, orde

            // transfer assets from exerciser to putty
            _transferERC20sIn(order.erc20Assets, msg.sender);
            _transferERC721sIn(order.erc721Assets, msg.sender);
            _transferFloorsIn(order.floorTokens, floorAssetToker
        }
    }
```

The attacker must control one of these contracts and have it set as a malicious ERC20 / ERC721 function that fails under attacker controlled conditions.

Recommended Mitigation Steps

Consider whitelisting approved ERC20 token or ERC721 address contracts to prevent users setting malicious token contracts. However, this remediation will have a significant admin input / gas trade-offs.

outdoteth (Putty Finance) acknowledged and commented:

> Technically this is a valid finding. However we don't intend to fix this at the contract level. Instead there will be adequate warnings on the UI to inform a user that they should be vigilant for any tokens that are not verified by putty (in addition, the UI will show the unverified token's logo as a question mark instead of as the token's logoURI).

> Report: Setting malicious or invalid erc721Assets, erc20Assets or floorTokens prevents the option from being exercised.

HickupHH3 (judge) commented:

> It's contingent on the external requirement for the attacker to be in control of a malicious ERC20 or NFT. Hence, medium severity is appropriate: `2-Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

Pedroais (warden) commented:

> I will argue why this issue should be high severity instead of medium.
>
> "It's contingent on the external requirement for the attacker to be in control of a malicious ERC20 or NFT. "
>
> Anyone can deploy a malicious contract and pass it as an ERC20 or NFT. This is not an external requirement, anyone can do it. Any malicious contract deployed by the attacker will work.
>
> This issue imposes a risk of asset loss to users without external requirements. The sponsor states unknown tokens will be shown with a question mark in the UI. This is ok but I think the attack is high severity since the user would be reasonable to think if he is accepting an offer for a BAYC (example of an expensive NFT) and some unknown token that in the worse case he should at least get the BAYC. This attack doesn't require the user to be dumb or act recklessly but just normal functioning of the protocol. The fake token shouldn't prevent the user from exercising the real BAYC.
>
> A user would be reasonable to expect to at least be able to exercise the real NFT in a case with an option that includes a real NFT and a malicious one. The problem is the malicious NFT can block the exercise of the real NFT. An option can be created using many real and valuable tokens with just 1 malicious token that prevents exercising the real ones.
>
> I hope the judge can consider these arguments and make his decision.

HickupHH3 (judge) commented:

> I should have phrased it better. The external requirement isn't on the attacker being in control of the malicious ERC20 / NFT. As rightfully pointed out, it can be

> easily done.

> The external requirement here is the user deciding to fill an option containing malicious assets. Such options can be considered to be honeypots that users should be made aware of (eg. through documentation, PSAs or warnings to the user). There's only so much the protocol can do to protect users, with tradeoffs against centralisation risks if the suggestion of whitelisting assets is adopted.

> This attack doesn't require the user to be dumb or act recklessly but just normal functioning of the protocol. The fake token shouldn't prevent the user from exercising the real BAYC.

> Partial exercising of options could be a feature, but opens up new attack surfaces and would be a non-trivial to implement. It is a limitation of the protocol that should be clearly communicated to users.

hyh (warden) reviewed mitigation:

> Now addressed on UI/DB level.

[M-02] Unbounded loops may cause `exercise()` s and `withdraw()` s to fail

*Submitted by lllllll, also found by 0xNineDec, sashik*eth, shung, and xiaoming90_

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L636-L640

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L646-L650

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L657-L661

Impact

There are no bounds on the number of tokens transferred in an order, and gas requirements can change (especially since orders can have a duration of [27 years](), so orders filled at time T1 may not be exercisable/withdrawable at time T2, or with the provided assets if the assets use a lot of gas during their transfers (e.g. aTokens and cTokens). The buyer of the option will have paid the premium, and will be unable to get the assets they are owed.

Proof of Concept

There are no upper bounds on the number of assets being transferred in these loops:

```
File: contracts/src/PuttyV2.sol   #1

636         function _transferERC20sOut(ERC20Asset[] memory assets
637             for (uint256 i = 0; i < assets.length; i++) {
638                 ERC20(assets[i].token).safeTransfer(msg.sender
639             }
640         }
```

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L636-L640

```
File: contracts/src/PuttyV2.sol   #2

646         function _transferERC721sOut(ERC721Asset[] memory asse
647             for (uint256 i = 0; i < assets.length; i++) {
648                 ERC721(assets[i].token).safeTransferFrom(addre
649             }
650         }
```

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L646-L650

```
File: contracts/src/PuttyV2.sol   #3
```

```
657       function _transferFloorsOut(address[] memory floorToke
658           for (uint256 i = 0; i < floorTokens.length; i++) {
659               ERC721(floorTokens[i]).safeTransferFrom(addres
660           }
661       }
```

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L657-L661

Recommended Mitigation Steps

Have an upper bound on the number of assets, or allow them to be transferred out one at a time, if necessary

outdoteth (Putty Finance) acknowledged, but disagreed with severity and commented:

> Adding a hardcoded check at the contract level is not a viable fix given that gas costs and limits are subject change over time. Instead, there already exists a limit of 30 assets on the frontend/db level.

outdoteth (Putty Finance) commented:

> Report: Unbounded loop can prevent put option from being exercised.

HickupHH3 (judge) commented:

> Medium severity is justified because, while very unlikely to happen, there could be a loss of assets.

hyh (warden) reviewed mitigation:

> Now addressed on UI/DB level.

[M-03] Put option sellers can prevent exercise by specifying zero amounts, or non-existant tokens

Submitted by IIIIIII, also found by 0xNineDec, exd0tpy, and zzzitron

Put option buyers pay an option premium to the seller for the privilege of being able to 'put' assets to the seller and get the strike price for it rather than the current market price. If they're unable to perform the 'put', they've paid the premium for nothing, and essentially have had funds stolen from them.

Proof of Concept

If the put option seller includes in `order.erc20Assets`, an amount of zero for any of the assets, or specifies an asset that doesn't currently have any code at its address, the put buyer will be unable to exercise the option, and will have paid the premium for nothing:

```
File: contracts/src/PuttyV2.sol   #1

453                 // transfer assets from exerciser to putty
454                 _transferERC20sIn(order.erc20Assets, msg.sende
```

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L453-L454

The function reverts if any amount is equal to zero, or the asset doesn't exist:

```
File: contracts/src/PuttyV2.sol   #2

593         function _transferERC20sIn(ERC20Asset[] memory assets,
594             for (uint256 i = 0; i < assets.length; i++) {
595                 address token = assets[i].token;
596                 uint256 tokenAmount = assets[i].tokenAmount;
597
598                 require(token.code.length > 0, "ERC20: Token i
599                 require(tokenAmount > 0, "ERC20: Amount too sm
600
601                 ERC20(token).safeTransferFrom(from, address(th
602             }
603         }
```

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L593-L603

Recommended Mitigation Steps

Verify the asset amounts and addresses during `fillOrder()`, and allow exercise if the token no longer exists at that point in time.

outdoteth (Putty Finance) confirmed and commented:

> At the contract level there exists 2 possible mitigations;
>
> 1. Remove the zero amount check (not feasible because it will cause another DOS issue for tokens that revert on 0 transfer).
> 2. Check all erc20 assets are valid in `fillOrder` (gas tradeoff because it requires an O(n) loop to check).
>
> Instead, the best mitigation imo is to add a check on the frontend/db level to ensure that all erc20 assets have a token amount greater than 0 and that it exists as a contract.
>
> If users want to go lower level than the db/frontend then they must exercise their own diligence.
>
> edit: decided to go with a 3rd option instead.
>
> Simply skip the ERC20 transfer if the amount is 0.
>
> Report: Setting an erc20Asset with a zero amount or with no code at the address will result in a revert when exercising a put option.

outdoteth (Putty Finance) resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/8.

hyh (warden) reviewed mitigation:

> Fixed zero amount part by introducing the noop for zero amount transfers in both `_transferERC20sIn` and `_transferERC20sOut` ERC20 transfer functions. The second part of the issue, fake tokens, is similar to M-01, M-02.

[M-04] Put options are free of any fees

Submitted by berndartmueller, also found by 0xsanson, hubble, Lambda, Metatron, and swit

Fees are expected to be paid whenever an option is exercised (as per the function comment on L235).

Put options

If a put option is exercised, the exerciser receives the strike price (initially deposited by the short position holder) denominated in `order.baseAsset`.

Call options

If a call option is exercised, the exerciser sends the strike price to Putty and the short position holder is able to withdraw the strike amount.

However, the current protocol implementation is missing to deduct fees for exercised put options. Put options are free of any fees.

Proof of Concept

The protocol fee is correctly charged for exercised calls:

PuttyV2.withdraw

```
    // transfer strike to owner if put is expired or call is exercis
    if ((order.isCall && isExercised) || (!order.isCall && !isExerci
        // send the fee to the admin/DAO if fee is greater than 0%
        uint256 feeAmount = 0;
        if (fee > 0) {
            feeAmount = (order.strike * fee) / 1000;
            ERC20(order.baseAsset).safeTransfer(owner(), feeAmount);
        }
```

```
        ERC20(order.baseAsset).safeTransfer(msg.sender, order.strike

        return;
    }
```

Contrary, put options are free of any fees:

[PuttyV2.sol#L450-L451](#)

```
    // transfer strike from putty to exerciser
    ERC20(order.baseAsset).safeTransfer(msg.sender, order.strike);
```

Recommended Mitigation Steps

Charge fees also for exercised put options.

outdoteth (Putty Finance) commented:

> Fees are only applied on puts if they are expired.

HickupHH3 (judge) commented:

> Making this the primary issue for the med severity issue, as per my comment in #269:
>
> "Put option not being charged fee upon exercising it. This can be considered to the "protocol leaked value" and thus be given a medium severity rating."

outdoteth (Putty Finance) confirmed and resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/4.

hyh (warden) reviewed mitigation:

> The same fix as in H-01.

[M-05] `fillOrder()` and `exercise()` may lock Ether sent to the contract, forever

Submitted by IllIIII, also found by 0x29A, 0xc0ffEE, 0xDjango, AmitN, auditor0517, berndartmueller, BowTiedWardens, cccz, danb, dipp, dirky, hansfriese, horsefacts, hyh, joestakey, kirk-baird, oyc*109, peritoflores, rfa, sashik*eth, simon135, sseefried, StErMi, swit, xiaoming90, and zzzitron_

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L324

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L338

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L436

Impact

`fillOrder()` and `exercise()` have code paths that require Ether to be sent to them (e.g. using WETH as the base asset, or the provision of the exercise price), and therefore those two functions have the `payable` modifier. However, there are code paths within those functions that do not require Ether. Ether passed to the functions, when the non-Ether code paths are taken, is locked in the contract forever, and the sender gets nothing extra in return for it.

Proof of Concept

Ether can't be pulled from the `order.maker` during the filling of a long order, so `msg.value` shouldn't be provided here:

```
File: contracts/src/PuttyV2.sol   #1

323             if (order.isLong) {
324                 ERC20(order.baseAsset).safeTransferFrom(order.
325             } else {
```

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L323-L325

If the `baseAsset` isn't WETH during order fulfillment, `msg.value` is unused:

```
File: contracts/src/PuttyV2.sol   #2

337                 } else {
338                     ERC20(order.baseAsset).safeTransferFrom(ms
339                 }
```

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L337-L339

Same for the exercise of call options:

```
File: contracts/src/PuttyV2.sol   #3

435                 } else {
436                     ERC20(order.baseAsset).safeTransferFrom(ms
437                 }
```

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L435-L437

Recommended Mitigation Steps

Add a `require(0 == msg.value)` for the above three conditions.

Alex the Entreprenerd (warden) commented:

> Why would the caller send ETH when they don't have to?

sseefried (warden) commented:

> User error is one possibility.

outdoteth (Putty Finance) confirmed and commented:

> Report: Native ETH can be lost if it's not utilised in exercise and fillOrder.

outdoteth (Putty Finance) resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/5.

hyh (warden) reviewed mitigation:

> Fixed with native funds amount control added to strike transfer logic of `fillOrder`. Zero strike and zero premium corner cases are yet unhandled as described in M.M-01 and M.M-02 in the Mitigation Review below.

[M-06] [Denial-of-Service] Contract Owner Could Block Users From Withdrawing Their Strike

Submitted by xiaoming90, also found by berndartmueller

When users withdraw their strike escrowed in Putty contract, Putty will charge a certain amount of fee from the strike amount. The fee will first be sent to the contract owner, and the remaining strike amount will then be sent to the users.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L500

```
function withdraw(Order memory order) public {
        ..SNIP..

        // transfer strike to owner if put is expired or call is
        if ((order.isCall && isExercised) || (!order.isCall && !
                // send the fee to the admin/DAO if fee is great
                uint256 feeAmount = 0;
                if (fee > 0) {
                        feeAmount = (order.strike * fee) / 1000;
```

```
                    ERC20(order.baseAsset).safeTransfer(owne
                }

                ERC20(order.baseAsset).safeTransfer(msg.sender,

                return;
            }
            ..SNIP..
    }
```

There are two methods on how the owner can deny user from withdrawing their strike amount from the contract

Method #1 - Set the `owner()` to `zero` address

Many of the token implementations do not allow transfer to `zero` address ([Reference](#)). Popular ERC20 implementations such as the following Openzeppelin's ERC20 implementation do not allow transfer to `zero` address, and will revert immediately if the `to` address (recipient) points to a `zero` address during a transfer.

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/5fbf494511fd522b931f7f92e2df87d671ea8b0b/contracts/token/ERC20/ERC20.sol#L226

```
    function _transfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {
        require(from != address(0), "ERC20: transfer from the zero a
        require(to != address(0), "ERC20: transfer to the zero addre

        _beforeTokenTransfer(from, to, amount);

        uint256 fromBalance = _balances[from];
        require(fromBalance >= amount, "ERC20: transfer amount excee
        unchecked {
            _balances[from] = fromBalance - amount;
            // Overflow not possible: the sum of all balances is cap
            // decrementing then incrementing.
```

```
            _balances[to] += amount;
        }

        emit Transfer(from, to, amount);

        _afterTokenTransfer(from, to, amount);
    }
```

It is possible for the owner to transfer the ownership to a `zero` address, thus causing the fee transfer to the contract owner to always revert. When the fee transfer always reverts, no one can withdraw their strike amount from the contract.

This issue will affect all orders that adopt a `baseAsset` that reverts when transferring to `zero` address.

Method #2 - If `baseAsset` is a ERC777 token

> Note: `owner()` could point to a contract or EOA account. By pointing to a contract, the contract could implement logic to revert whenever someone send tokens to it.

ERC777 contains a `tokensReceived` hook that will notify the recipient whenever someone sends some tokens to the recipient .

Assuming that the `baseAsset` is a ERC77 token, the recipient, which is the `owner()` in this case, could always revert whenever `PuttyV2` contract attempts to send the fee to recipient. This will cause the `withdraw` function to revert too. As a result, no one can withdraw their strike amount from the contract.

This issue will affect all orders that has ERC777 token as its `baseAsset` .

Impact

User cannot withdraw their strike amount and their asset will be stuck in the contract.

Recommended Mitigation Steps

It is recommended to adopt a withdrawal pattern for retrieving owner fee.

Instead of transferring the fee directly to owner address during withdrawal, save the amount of fee that the owner is entitled to in a state variable. Then, implement a new function that allows the owner to withdraw the fee from the `PuttyV2` contract.

Consider the following implementation. In the following example, there is no way for the owner to perform denial-of-user because the outcome of the fee transfer (succeed or fail) to the owner will not affect the user's strike withdrawal process.

This will give users more assurance and confidence about the security of their funds stored within Putty.

```
mapping(address => uint256) public ownerFees;

function withdraw(Order memory order) public {
        ..SNIP..
    // transfer strike to owner if put is expired or call is exe
    if ((order.isCall && isExercised) || (!order.isCall && !isEx
        // send the fee to the admin/DAO if fee is greater than
        uint256 feeAmount = 0;
        if (fee > 0) {
            feeAmount = (order.strike * fee) / 1000;
            ownerFees[order.baseAsset] += feeAmount
        }

        ERC20(order.baseAsset).safeTransfer(msg.sender, order.st

        return;
    }
    ..SNIP..
}

function withdrawFee(address baseAsset) public onlyOwner {
        uint256 _feeAmount = ownerFees[baseAsset];
        ownerFees[baseAsset] = 0;
        ERC20(baseAsset).safeTransfer(owner(), _feeAmount);
}
```

outdoteth (Putty Finance) disagreed with severity

HickupHH3 (judge) commented:

> The scenarios provided are valid, especially for baseAssets that revert on zero-address transfer.

> While the likelihood is low, assets are lost and cannot be retrieved.
> `3 — High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).`

HickupHH3 (judge) decreased severity to Medium and commented:

> Thinking about it further, the external conditions / requirements needed for the DoS to happen are somewhat strong.
>
> - the ERC777 attack requires `owner()` or the token to be engineered to be malicious and adopted.
> - DoS via revoking ownership requires `fee` to be non-zero first, which is unlikely to happen. I can classify this as a "user-prone" bug, which would be similar to cases like including ETH when WETH is intended to be used (#226).
>
> Hence, I think medium severity is more appropriate: `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

outdoteth (Putty Finance) confirmed and resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/4.

hyh (warden) reviewed mitigation:

> The same fix as in H-01: as the platform fee is now transferred on order filling, any owner griefing can only yield a denial of service. There will be no loss of funds as this way position is only about to be created when the fee is transferred.

[M-07] An attacker can create a short put option order on an

NFT that does not support ERC721 (like cryptopunk), and the user can fulfill the order, but cannot exercise the option

Submitted by cccz, also found by lllllll and minhquanym

An attacker can create a short put option on cryptopunk. When the user fulfills the order, the baseAsset will be transferred to the contract.

However, since cryptopunk does not support ERC721, the user cannot exercise the option because the safeTransferFrom function call fails. Attacker can get premium and get back baseAsset after option expires.

Proof of Concept

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L343-L346

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L628-L629

Recommended Mitigation Steps

Consider adding a whitelist to nfts in the order, or consider supporting exercising on cryptopunk.

STYJ (warden) commented:

> Putty uses solmate's `ERC721.safeTransferFrom` which requires that the NFT contract implements `onERC721Received`. For the case of OG NFTs like punks and rocks, this will fail, https://github.com/Rari-Capital/solmate/blob/main/src/tokens/ERC721.sol#L120

thereksfour (warden) commented:

> The user does not need to send cryptopunk to the contract when fulfilling the short put option order, but the user will pay a premium to the order creator. Later,

> when the user wants to exercise the option, since the cryptopunk does not support safetransferfrom, the user cannot exercise the option.

STYJ (warden) commented:

> The user does not need to send cryptopunk to the contract when fulfilling the short put option order, but the user will pay a premium to the order creator. Later, when the user wants to exercise the option, since the cryptopunk does not support safetransferfrom, the user cannot exercise the option.

> Sorry, I did not consider this path. You are correct to say that a maker can create a short put option order with cryptopunks as a token and the holder of the long put option will not be able to exercise since cryptopunks cannot be transferred with `safeTransferFrom`. From that perspective, this is a valid issue. Thank you for bringing it up. I will defer to the judge for the final decision.

outdoteth (Putty Finance) acknowledged, but disagreed with severity and commented:

> We dont intend to support cryptopunks or cryptokitties. If users wish to use these tokens then they can get wrapped versions (ex: wrapped cryptopunks).

HickupHH3 (judge) decreased severity to Medium and commented:

> I thought cryptokitties are ERC721? I think they were the ones who popularized the standard actually :p Probably meant etherrocks.

> In general, non-compliant ERC-721 NFTs can be supported through wrappers, though some users might be unaware... Downgrading to med severity, similar to this issue from another contest.

hyh (warden) reviewed mitigation:

> Similar to M-01, M-02.

[M-08] Overlap Between `ERC721.transferFrom()` and

`ERC20.transferFrom()` Allows `order.erc20Assets` or `order.baseAsset` To Be ERC721 Rather Than ERC20

Submitted by kirk-baird, also found by reassor and sseefried

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L324

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L338

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L344

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L360

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L436

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L601

Impact

Both `ERC20.transferFrom(address to, address from, uint256 amount)` and `ERC721.transferFrom(address to, address from, uint256 id)` have the same function signature `0x23b872dd`. The impact of this is it's possible for `baseAsset` or `erc20Assets` to be ERC721 addresses rather than ERC20.

These functions will successfully transfer the NFT into the protocol however they will fail to transfer the NFT out of the contract. That is because the outgoing transfer is

`ERC20.safeTransfer()` which calls `transfer(to, amount)` which does not match up with any valid function signatures on ERC721.

Therefore any ERC721 tokens transferred into the contract in this manner via `fillOrder()` will be permanently stuck in the contract as neither `exercise()` nor `withdraw()` will successfully transfer the tokens out of the contract.

Proof of Concept

[ERC721.transferFrom()]

```
function transferFrom(
    address from,
    address to,
    uint256 id
) public virtual {
```

[ERC20.transferFrom()]

```
function transferFrom(
    address from,
    address to,
    uint256 amount
) public virtual returns (bool) {
```

Recommended Mitigation Steps

Consider whitelisting approved ERC721 and ERC20 token contracts. Furthermore, separate these two contracts into different whitelists for ERC20s and ERC721s then ensure each contract is in the right category.

[outdoteth (Putty Finance) acknowledged and commented]:

> Report: If an ERC721 token is used in places where ERC20 assets are supposed to be used then ERC721 tokens can get stuck in withdraw() and exercise().

hyh (warden) reviewed mitigation:

> Requires asset whitelisting, now addressed on UI/DB level.

[M-09] The contract serves as a flashloan pool without fee

Submitted by 0xc0ffEE, also found by horsefacts, pedroais, and unforgiven

The malicious user could leverage PuttyV2 contract to flashloan without paying fee the assets to make profit.

Consider a scenario that maker and taker is the same, and is a contract

1. The contract call PuttyV2. `fillOrder` with a Long Call order that has `order.baseAssets` references to a contract having custom logic other than standard ERC20. The order also specify `erc20Assets` to the `token` and `tokenAmount` that PuttyV2 contract is owing (similar to `erc721Assets`)
2. When the execution is at https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L324, the custom logic could execute on the contract address `order.baseAsset`.
3. The malicious contract then call `exercise` to exercise the short call position. This call will transfer out the assets specified in the order to the malicious contract by executing logics in `_transferERC20sOut, _transferERC721sOut`
4. The contract uses that assets to make profit on other platforms. After that, the execution continues at https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L324.
5. At the end of `fillOrder`, the contract just transfers enough assets back to PuttyV2 by executing logics in `_transferERC20sIn, _transferERC721sIn` to finish the execution.

Alex the Entreprenerd (warden) commented:

> Warden is saying that they can flashloan without fee, but any exercised option will pay a 3% fee, additionally the order of operations shown (gain control on base.asset.transfer when receiving premium), would mean that the order ERC20s

> and NFTs have yet to be transferred in, so a “mid-fillOrder” “exercise” would not only pay the fee, but also revert due to lack of the tokens.

Pedroais (warden) commented:

> The 3% will be paid in the fake asset since base asset is an attacker contract so there is no fee to perform the attack.

> This attack is done with assets that are already inside the contract so there is no revert in transfer out.

outdoteth (Putty Finance) acknowledged and commented:

> Acknowledging that technically this is true. Although no easy mitigation exists as far as I can see aside from adding nonReentrant to exercise and fillOrder - adding a non-negligible gas overhead.

Alex the Entreprenerd (warden) commented:

> I agree that the finding is valid, the fee can be paid in a mintable token to gain temporary ownership of a token underlying which is repaid at the end of `fillOrder`.

outdoteth (Putty Finance) commented:

> Report: It’s possible to flashloan all assets in the contract without paying a protocol fee.

HickupHH3 (judge) commented:

> Flash loans from the contract would be a feature, not a bug. However, being able to do so without paying a protocol fee (ie. paying in fake tokens) wouldn’t be great.

[M-10] Putty position tokens may be minted to non ERC721 receivers

Submitted by horsefacts, also found by 0xc0ffEE, 0xsanson, berndartmueller, BowTiedWardens, csanuragjain, defsec, IIIIIII, joestakey, Kenshin, Picodes, shenwilly, Sm4rty, unforgiven, and xiaoming90

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L302-L308

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2Nft.sol#L11-L18

Vulnerability Details

Putty uses ERC721 `safeTransfer` and `safeTransferFrom` throughout the codebase to ensure that ERC721 tokens are not transferred to non ERC721 receivers. However, the initial position mint in `fillOrder` uses `_mint` rather than `_safeMint` and does not check that the receiver accepts ERC721 token transfers:

PuttyV2#fillOrder

```
        // create long/short position for maker
        _mint(order.maker, uint256(orderHash));

        // create opposite long/short position for taker
        bytes32 oppositeOrderHash = hashOppositeOrder(order);
        positionId = uint256(oppositeOrderHash);
        _mint(msg.sender, positionId);
```

PuttyV2Nft#_mint

```
    function _mint(address to, uint256 id) internal override {
        require(to != address(0), "INVALID_RECIPIENT");
        require(_ownerOf[id] == address(0), "ALREADY_MINTED");

        _ownerOf[id] = to;

        emit Transfer(address(0), to, id);
```

```
}
```

Impact

If a maker or taker are a contract unable to receive ERC721 tokens, their options positions may be locked and nontransferable. If the receiving contract does not provide a mechanism for interacting with Putty, they will be unable to exercise their position or withdraw assets.

Recommendation

Consider implementing the `require` check in Solmate's `ERC721#_safeMint` in your own mint function:

```
function _safeMint(address to, uint256 id) internal virtual
    _mint(to, id);

    require(
        to.code.length == 0 ||
            ERC721TokenReceiver(to).onERC721Received(msg.ser
            ERC721TokenReceiver.onERC721Received.selector,
        "UNSAFE_RECIPIENT"
    );
}
```

However, note that calling `_safeMint` introduces a reentrancy opportunity! If you make this change, ensure that the mint is treated as an interaction rather than an effect, and consider adding a reentrancy guard:

```
    /*  ~~~ EFFECTS ~~~ */

    // create opposite long/short position for taker
    bytes32 oppositeOrderHash = hashOppositeOrder(order);
    positionId = uint256(oppositeOrderHash);

    // save floorAssetTokenIds if filling a long call order
    if (order.isLong && order.isCall) {
        positionFloorAssetTokenIds[uint256(orderHash)] = flo
    }
```

```
        // save the long position expiration
        positionExpirations[order.isLong ? uint256(orderHash) :

        emit FilledOrder(orderHash, floorAssetTokenIds, order);

        /* ~~~ INTERACTIONS ~~~ */

        _safeMint(order.maker, uint256(orderHash));
        _safeMint(msg.sender, positionId);
```

Alternatively, document the design decision to use `_mint` and the associated risk for end users.

outdoteth (Putty Finance) acknowledged, but disagreed with severity and commented:

> It's unlikely a contract will have all the setup required to interact with PuttyV2 but not be able to handle ERC721 tokens. Adding a check via safeMint adds a gas overhead as well as another re-entrancy attack vector so there is a tradeoff (as noted in the issue report^^).

> Report: Contracts that can't handle ERC721 tokens will lose their Putty ERC721 position tokens.

HickupHH3 (judge) commented:

> In addition, some contracts may have custom logic in their `onERC721Received()` implementation that is triggered only by the safe methods and not their "unsafe" counterparts.

[M-11] `fee` can change without the consent of users

Submitted by Picodes, also found by 0xNineDec, 0xsanson, antonttc, berndartmueller, BowTiedWardens, catchup, dirky, Alex the Entreprenerd, horsefacts, Metatron, sseefried, and unforgiven_

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/Putty

V2.sol#L240

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L497

Impact

Fees are applied during `withdraw`, but can change between the time the order is filled and its terms are agreed upon and the withdrawal time, leading to a loss of the expected funds for the concerned users.

Proof of Concept

The scenario would be:

- Alice and Bob agrees to fill an order at a time fees are 0.1%
- During the duration of the option, fees are increased to 3%
- At withdrawal they'll pay 3% of the strike, although they wouldn't have created the order in the first place with such fees

Recommended Mitigation Steps

Mitigation could be:

- Store the fees in `Order` and verify that they are correct when the order is filled, so they are hardcoded in the struct
- Add a timestamp: this wouldn't fully mitigate but would still be better than the current setup
- Keep past fees and fee change timestamps in memory (for example in an array) to be able to retrieve the creation time fees at withdrawal

outdoteth (Putty Finance) confirmed and commented:

> Report: Admin can change fee at any time for existing orders.

outdoteth (Putty Finance) resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/4.

hyh (warden) reviewed mitigation:

> The same fix as in H-01.

[M-12] Options with a small strike price will round down to 0 and can prevent assets to be withdrawn

*Submitted by berndartmueller, also found by auditor0517, hansfriese, IIIIIII, Lambda, sashik*eth, shenwilly, and TrungOre_

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L499-L500

Impact

Certain ERC-20 tokens do not support zero-value token transfers and revert. Using such a token as a `order.baseAsset` for a rather small option strike and a low protocol fee rate can lead to rounding down to 0 and prevent asset withdrawals for those positions.

Proof of Concept

PuttyV2.sol#L499-L500

```
// send the fee to the admin/DAO if fee is greater than 0%
uint256 feeAmount = 0;
if (fee > 0) {
    feeAmount = (order.strike * fee) / 1000;
    ERC20(order.baseAsset).safeTransfer(owner(), feeAmount); //
}
```

Some ERC20 tokens revert for zero-value transfers (e.g. `LEND`). If used as a `order.baseAsset` and a small strike price, the fee token transfer will revert. Hence, assets and the strike can not be withdrawn and remain locked in the contract.

See Weird ERC20 Tokens - Revert on Zero Value Transfers

Example:

- `order.baseAsset` is one of those weird ERC-20 tokens
- `order.strike = 999` (depending on the token decimals, a very small option position)
- `fee = 1` (0.1%)

((999 * 1) / 1000 = 0.999) rounded down to 0 -> zero-value transfer reverting transaction

Recommended Mitigation Steps

Add a simple check for zero-value token transfers:

```
// send the fee to the admin/DAO if fee is greater than 0%
uint256 feeAmount = 0;
if (fee > 0) {
    feeAmount = (order.strike * fee) / 1000;

    if (feeAmount > 0) {
        ERC20(order.baseAsset).safeTransfer(owner(), feeAmount);
    }
}
```

outdoteth (Putty Finance) confirmed and commented:

> Report: withdraw() can be DOS'd for baseAsset ERC20s that prevent 0 transfers if the calculated feeAmount is 0 due to rounding.

outdoteth (Putty Finance) resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/4.

hyh (warden) reviewed mitigation:

> Fixed along with H-01 in `PR#4` .

[M-13] Order duration can be set to 0 by Malicious maker

Submitted by codexploder, also found by ACai, cccz, Critical, horsefacts, ignacio, shenwilly, unforgiven, and xiaoming90

A malicious maker can set a minimum order duration as 0 which means order will instantly expire after filling. Taker will get only the withdraw option and that too with fees on strike price, thus forcing the taker to lose money in this meaningless transaction

Proof of Concept

1. Maker creates an order with zero Order duration
2. Taker fills this order but the order instantly expires since duration was 0
3. Taker gets the only option to withdraw with fees on strike price

Recommended Mitigation Steps

Enforce at least x days of duration.

outdoteth (Putty Finance) confirmed and resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/7.

hyh (warden) reviewed mitigation:

> Fixed by requiring the minimal order duration of 15 minutes on filling.

[M-14] Order cancellation is prone to frontrunning and is dependent on a centralized database

Submitted by shung, also found by unforgiven

Order cancellation requires makers to call `cancel()`, inputting the order as a function parameter. This is the only cancellation method, and it can cause two issues.

This first issue is that it is an on-chain signal for MEV users to frontrun the cancellation and fill the order.

The second issue is the dependency to a centralized service for cancelling the order. As orders are signed off chain, they would be stored in a centralized database. It is unlikely that an end user would locally record all the orders they make. This means that when cancelling an order, maker needs to request the order parameters from the centralized service. If the centralized service goes offline, it could allow malicious parties who have a copy of the order database to fill orders that would have been cancelled otherwise.

Proof of Concept

1. Bob signs an order which gets recorded in Putty servers.
2. Alice mirrors all the orders using Putty APIs.
3. Putty servers go offline.
4. Bob wants to cancel his order because changing token prices makes his order less favourable to him.
5. Bob cannot cancel his order because Putty servers are down and he does not remember the exact amounts of tokens he used.
6. Alice goes through all the orders in her local mirror and fulfills the non-cancelled orders, including Bob's, with extremely favourable terms for herself.

Recommended Mitigation Steps

Aside from the standard order cancellation method, have an extra method to cancel all orders of a caller. This can be achieved using a "minimum valid nonce" state variable, as a mapping from user address to nonce.

```
mapping(address => uint256) minimumValidNonce;
```

Allow users to increment their `minimumValidNonce`. Make sure the incrementation function do not allow incrementing more than `2**64` such that callers cannot lock themselves out of creating orders by increasing `minimumValidNonce` to `2**256-1` by mistake. Then, prevent filling orders if `order.nonce < minimumValidNonce`.

Another method to achieve bulk cancelling is using counters. For example, Seaport uses counters, which is an extra order parameter that has to match the corresponding counter state variable. It allows maker to cancel all his orders by incrementing the counter state variable by one.

Either of these extra cancellation methods would enable cancelling orders without signalling to MEV bots, and without a dependency to a centralized database.

outdoteth (Putty Finance) confirmed and commented:

> Should this be tagged as Med or Low? Funds are not directly at risk unless the centralised order book server goes down and loses all the data. Perhaps there is a non-negligible chance that this *could* happen. But even then, orders have an "expiration" field attached to them which will render them useless after some set time period. There are also easy fixes on the frontend, such as allowing users to download a txt file with their order/orderHash so that they don't have to rely on the centralised DB for data availability.

> But will defer to judges.

> Report: Cannot cancel orders without reliance on centralised database.

HickupHH3 (judge) commented:

> The sponsor's point is valid: there is an expiration param that the maker signs as part of the order that marks its validity.

> However, the warden(s) concerns are valid too. While it is an edge case that is very unlikely to happen, there would arguably be a "loss" of assets of the maker because of the protocol's loss of functionality, as per the scenario described above. Hence, the medium severity rating is justified.

> I recommend implementing the warden's recommended fix; having a `minimumValidNonce` would be great in allowing easy on-chain cancellation of an order. It makes the system a little more trust-less and provides a "red button" option for makers to use if necessary.

outdoteth (Putty Finance) resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/10.

hyh (warden) reviewed mitigation:

> Fixed by the introduction of `setMinimumValidNonce` function and the corresponding control on order filling. See M.M-04 in the Mitigation Review below.

[M-15] Zero strike call options will avoid paying system fee

Submitted by hyh, also found by csanuragjain, minhquanym, and Treasure-Seeker

Zero and near zero strike calls are common derivative type. For such derivatives the system will not be receiving fees are the fee is now formulated as a fraction of order strike.

Also, it can be a problem for OTM call options, when the option itself is nearly worthless, while the fee will be substantial as strike will be big. Say 1k ETH BAYC call doesn't have much value, but the associated fee will be 10x of usual fee, i.e. substantial, while there is nothing to justify that.

Marking this as medium severity as that's a design specifics that can turn off or distort core system fee gathering.

Proof of Concept

Currently fee is linked to the order strike which makes it vary heavily for different types of orders, for example deep ITM and OTM calls:

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L494-L506

```
        // transfer strike to owner if put is expired or call is
        if ((order.isCall && isExercised) || (!order.isCall && !
            // send the fee to the admin/DAO if fee is greater t
            uint256 feeAmount = 0;
            if (fee > 0) {
```

```
            feeAmount = (order.strike * fee) / 1000;
            ERC20(order.baseAsset).safeTransfer(owner(), fee
        }

        ERC20(order.baseAsset).safeTransfer(msg.sender, orde

        return;
    }
```

Recommended Mitigation Steps

Consider linking the fee to option premium as this is option value that cannot be easily manipulated and exactly corresponds to the trading volume of the system.

i.e. consider moving fee gathering to fillOrder:

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L322-L340

```
    // transfer premium to whoever is short from whomever is
    if (order.isLong) {
        ERC20(order.baseAsset).safeTransferFrom(order.maker,
    } else {
        // handle the case where the user uses native ETH ir
        if (weth == order.baseAsset && msg.value > 0) {
            // check enough ETH was sent to cover the premiu
            require(msg.value == order.premium, "Incorrect E

            // convert ETH to WETH and send premium to maker
            // converting to WETH instead of forwarding nati
            // 1) active market makers will mostly be using
            // 2) attack surface for re-entrancy is reduced
            IWETH(weth).deposit{value: msg.value}();
            IWETH(weth).transfer(order.maker, msg.value);
        } else {
            ERC20(order.baseAsset).safeTransferFrom(msg.send
        }
    }
```

Alex the Entreprenerd (warden) commented:

> Zero strike will indeed have a fee of 0.

outdoteth (Putty Finance) confirmed and commented:

> Report: Charging fees on the strike amount instead of the premium amount can lead to disproportionate fees.

outdoteth (Putty Finance) resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/4.

hyh (warden) reviewed mitigation:

> The same fix as in H-01.

[M-16] Use of Solidity version 0.8.13 which has two known issues applicable to PuttyV2

Submitted by hubble, also found by horsefacts

The solidity version 0.8.13 has below two issues applicable to PuttyV2

1. Vulnerability related to ABI-encoding.

   ref : https://blog.soliditylang.org/2022/05/18/solidity-0.8.14-release-announcement/
   This vulnerability can be misused since the function hashOrder() and hashOppositeOrder() has applicable conditions.
   "...pass a nested array directly to another external function call or use abi.encode on it."

2. Vulnerability related to 'Optimizer Bug Regarding Memory Side Effects of Inline Assembly'

   ref : https://blog.soliditylang.org/2022/06/15/solidity-0.8.15-release-announcement/
   PuttyV2 inherits solidity contracts from openzeppelin and solmate, and both these uses inline assembly, and optimization is enabled while compiling.

Recommended Mitigation Steps

Use recent Solidity version 0.8.15 which has the fix for these issues.

outdoteth (Putty Finance) confirmed and commented:

> Great catch.

> Report: Use of Solidity 0.8.13 with known issues in ABI encoding and memory side effects.

outdoteth (Putty Finance) resolved:

> PR with fix: https://github.com/outdoteth/putty-v2/pull/6.

hyh (warden) reviewed mitigation:

> Fixed by bumping the solidity version.

Low Risk and Non-Critical Issues

For this contest, 82 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **xiaoming90** received the top score from the judge.

The following wardens also submitted reports: reassor, lllllll, sseefried, BowTiedWardens, 0x1f8b, joestakey, zer0dot, Alex the Entreprenerd, 0xDjango, Chom, shung, __141345__, 0xNazgul, hubble, StErMi, 0xf15ers, defsec, zzzitron, Lambda, Metatron, TomJ, 0xsanson, danb, Funen, unforgiven, cryptphi, dirk_y, TrungOre, catchup, horsefacts, JohnSmith, Picodes, AmitN, samruna, antonttc, async, GimelSec, hake, Kaiziron, Nethermind, robee, rokinot, saneryee, sashik_eth, 0x29A, BnkeOx0, Limbooo, csanuragjain, ElKu, MadWookie, Waze, doddle0x, 0xNineDec, datapunk, gogo, Kenshin, MiloTruck, shenwilly, simon135, fatherOfBlocks, hansfriese, JC, oyc_109, delfin454000, 0xSolus, cccz, durianSausage, Hawkeye, itsmeSTYJ, pedroais, peritoflores, rajatbeladiya, ReyAdmirado, Yiko, 0x52, _Adam, aysha, David_, exd0tpy, Sneakyninja0129, *and* Treasure-Seeker.

Codebase Summary & Key Improvement Opportunities

Key Features

The in-scope system was found to be low in complexity, with the key features being as follows:

- fillOrder - Fills an offchain order
- exercise - Exercise a long order
- withdraw - Withdraws the assets from a short order
- cancel - Cancels an order which prevents it from being filled in the future

Code Quality and Test Coverage

In sumamry, the code quality of the PuttyV2 is high. The codes were also found to be well-documented and team took the efforts to document the NatSpec for all the functions within the `PuttyV2` contract. However, there are still some room of improvement as NatSpec was not documented for the `PuttyV2Nft` contract. For completeness and readability, it is recommended to document the NatSpec for all the functions in the contracts where feasible.

To futher improve readability of the codes, additional helper functions could be implemented. Refer to the “4.6 Code Can Be Refactored To Be More Readable” issue.

Test coverage was found to be high. All the key features have been covered in the test. However, periphery logic functions such as `batchFillOrder` and `acceptCounterOffer` that are exposed to the public users were not included in the tests. It is recommended to write proper tests for all functions.

Unexpected Token Behaviors

The system did not implement a whitelisting mechanism to ensure that only approved tokens are allowed to be traded within Putty. Thus, users could specify any tokens within an order/option. This permissionless approach increases the attack surface of the system and exposes the system to various attack vectors. Some tokens might be malicious, some tokens might contain hooks, and some tokens might not work as intended. Thus, it is challenging for Putty to guard against all kinds of

vulnerabilites that arise due to the need to support large number of tokens. If possible, it is recommended to only allow approved tokens that have been vetted and reviewed by Putty to be traded within the system to minimize the risks during the initial stage. Once the system is more stable, the team can progressively open up to more tokens.

Re-entrancy Risks

The key features (e.g. fillOrder, exercise) were found to be following the “Checks Effects Interactions” pattern rigorously, which help to prevent any possible re-entrancy attack. However, further improvement can be made to guard against future re-entrancy attack. As the key features make many external contract calls via token transfers, the risks of re-entrancy attack is significantly higher for Putty compared to other protocols. Thus, it is prudent to implement additional reentrancy prevention wherever possible by utilizing the nonReentrant modifier from **Openzeppelin Library** to block possible re-entrancy as a defense-in-depth measure.

Out-of-gas/Revert Risks

The order contains many arrays such as `whitelist`, `floorTokens`, `erc20Assets`, and `erc721Assets`. It was found that the contract did not place an upper limit on the number of elements that can be stored within the array, and proceed to loop through all the elements within an array in many parts of the codes. This might cause out-of-gas error to happen and cause a revert, thus causing the feature to be unusable in certain scenarios. It is recommended for the team to review each of the loop structures involving an array within the contract to determine if it is possible to place an upper limit.

Authorisation Controls

Robust authorisation controls have been implemented for all the key features to ensure that only authorised and correct actors could call the functions. For instance, only owner of long position could call the `exercise` function and only owner of short position could call the `withdraw` function. No authorisation issues were observed during the contest.

Summary Of Findings

The following is a summary of the low and non-critical findings observed during the contest.

No.	Title	Risk Rating
L-01	Lack Of Reentrancy Guards On External Functions	Low
L-02	Discontinuity in Exercise Period	Low
L-03	Insufficient Input Validation	Low
L-04	Order Cannot Be Filled Due To Unbounded Whitelist Within An Order	Low
L-05	Order Cannot Be Filled Due To Unbounded floorTokens, ERC20Asset Or ERC721Asset Within An Order	Low
L-06	Order Can Be Cancelled Even After Being Filled	Low
L-07	No Check if onERC721Received Is Implemented	Low
N-01	Omissions in events	Non-Critical
N-02	Draft OpenZeppelin Dependencies	Non-Critical
N-03	Insufficient Tests	Non-Critical
N-04	Owner Can Renounce Ownership	Non-Critical
N-05	Consider two-phase ownership transfer	Non-Critical
N-06	Code Can Be Refactored To Be More Readable	Non-Critical
N-07	Inconsistent use of named return variables	Non-Critical
N-08	Unused imports	Non-Critical
N-09	Incorrect functions visibility	Non-Critical

[L-01] Lack Of Reentrancy Guards On External Functions

The following external functions within the `PuttyV2` contract contain function calls (e.g. `safeTransferFrom`, `safeTransfer`) that pass control to external contracts. Additionally, if ERC777 tokens are being used within an order, it contains various hooks that will pass the execution control to the external party.

Thus, it might allow an malicious external contract to re-enter to the contract.

- `PuttyV2.fillorder`
- `PuttyV2.exercise`
- `PuttyV2.withdraw`
- `PuttyV2.batchFillOrder`
- `Putty.acceptCounterOffer`

No re-entrancy attacks that could lead to loss of assets were observed during the assessment. Thus, this issue is marked as Low.

The following shows examples of function call being made to an external contract

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L324

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L436

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L451

Recommendation

It is recommended to follow the good security practices and apply necessary reentrancy prevention by utilizing the nonReentrant modifier from Openzeppelin Library to block possible re-entrancy.

[L-02] Discontinuity in Exercise Period

The position can be exercised if current block timestamp is less than the position's expiration.

The position can be withdrawed if current block timestamp is greater than the position's expiration

However, when current block timestamp is equal to the position's expiration (`block.timestamp == positionExpirations`), the state is unknown (cannot be exercised or withdraw)

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L401

```
    function exercise(Order memory order, uint256[] calldata floorAs
        ..SNIP..
        // check position has not expired
        require(block.timestamp < positionExpirations[uint256(orderH
        ..SNIP..
    }
```

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L481

```
    function withdraw(Order memory order) public {
        ..SNIP..
        // check long position has either been exercised or is expir
        require(block.timestamp > positionExpirations[longPositionId
        ..SNIP..
    }
```

Recommendation

Allow the user to withdraw the position upon expiration.

```
function withdraw(Order memory order) public {
    ..SNIP..
    // check long position has either been exercised or is expir
    require(block.timestamp >= positionExpirations[longPosition]
    ..SNIP..
}
```

[L-03] Insufficient Input Validation

The `PuttyV2.fillOrder` function does not validate that the `msg.sender` (order taker) is the same as the order maker, which might potentially lead to unwanted behaviour within the system. Order taker should not be the same as order maker under any circumstances.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L268

```
    function fillOrder(
        Order memory order,
        bytes calldata signature,
        uint256[] memory floorAssetTokenIds
    ) public payable returns (uint256 positionId) {
        /* ~~~ CHECKS ~~~ */

        bytes32 orderHash = hashOrder(order);

        // check signature is valid using EIP-712
        require(SignatureChecker.isValidSignatureNow(order.maker

        // check order is not cancelled
        require(!cancelledOrders[orderHash], "Order has been can

        // check msg.sender is allowed to fill the order
        require(order.whitelist.length == 0 || isWhitelisted(ord

        // check duration is valid
        require(order.duration < 10_000 days, "Duration too long

        // check order has not expired
        require(block.timestamp < order.expiration, "Order has e
```

```
        // check base asset exists
        require(order.baseAsset.code.length > 0, "baseAsset is r
```

Recommendation

Implement the necessary check to ensure that order taker is not the same as order maker.

```
    require(msg.sender != order.maker, "Invalid order taker");
```

[L-04] Order Cannot Be Filled Due To Unbounded Whitelist Within An Order

An order can contain large number of addresses within the `whitelist` array of an order.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L78

```
    struct Order {
        address maker;
        bool isCall;
        bool isLong;
        address baseAsset;
        uint256 strike;
        uint256 premium;
        uint256 duration;
        uint256 expiration;
        uint256 nonce;
        address[] whitelist;
        address[] floorTokens;
        ERC20Asset[] erc20Assets;
        ERC721Asset[] erc721Assets;
    }
```

When the `PuttyV2.fillOrder` function is called, it will attempt to check if the caller is whitelisted by looping through the `order.whitelist` array. However, if `order.whitelist` array contains large number of addresses, it will result in out-of-gas error and cause a revert. Thus, this order can never be filled.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L284

```
    function fillOrder(
        Order memory order,
        bytes calldata signature,
        uint256[] memory floorAssetTokenIds
    ) public payable returns (uint256 positionId) { // @audit-issue
            ..SNIP..
        // check msg.sender is allowed to fill the order
        require(order.whitelist.length == 0 || isWhitelisted(order.w
            ..SNIP..
    }
```

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L669

```
    function isWhitelisted(address[] memory whitelist, address targe
        for (uint256 i = 0; i < whitelist.length; i++) {
            if (target == whitelist[i]) return true;
        }

        return false;
    }
```

Recommendation

It is recommended to restrict the number of whitelisted addresses within an order to a upper limit (e.g. 30).

Although client-side or off-chain might have already verified that the number of whitelisted addresses do not exceed a certain limit within an order, simply relying on

client-side and off-chain validations are not sufficient. It is possible for an attacker to bypass the client-side and off-chain validations and interact directly with the contract. Thus, such validation must also be implemented on the on-chain contracts.

[L-05] Order Cannot Be Filled Due To Unbounded floorTokens, ERC20Asset Or ERC721Asset Within An Order

An order can contain large number of tokens within the `floorTokens`, `ERC20Asset` or `ERC721Asset` arrays of an order.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L78

```
    struct Order {
        address maker;
        bool isCall;
        bool isLong;
        address baseAsset;
        uint256 strike;
        uint256 premium;
        uint256 duration;
        uint256 expiration;
        uint256 nonce;
        address[] whitelist;
        address[] floorTokens;
        ERC20Asset[] erc20Assets;
        ERC721Asset[] erc721Assets;
    }
```

When the `PuttyV2.fillOrder` function is called, it will attempts to loop through all the `floorTokens`, `ERC20Asset` or `ERC721Asset` arrays of an order to transfer the required assets to `PuttyV2` contract from the order maker or taker.

The `_transferERC20sIn`, `_transferERC721sIn`, `_transferFloorsIn` attempt to loop through all the tokens within the array. However, if array contains large number of tokens, it will result in out-of-gas error and cause a revert. Thus, this order can never be filled.

Following is an example of the vulnerable function.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L593

```
    function _transferERC20sIn(ERC20Asset[] memory assets, address f
        for (uint256 i = 0; i < assets.length; i++) {
            address token = assets[i].token;
            uint256 tokenAmount = assets[i].tokenAmount;

            require(token.code.length > 0, "ERC20: Token is not cont
            require(tokenAmount > 0, "ERC20: Amount too small");

            ERC20(token).safeTransferFrom(from, address(this), toker
        }
    }
```

Recommendation

It is recommended to restrict the number of tokens within the `floorTokens`, `ERC20Asset` or `ERC721Asset` arrays of an order. (e.g. Maximum of 10 tokens)

Although client-side or off-chain might have already verified that the number of tokens do not exceed a certain limit within an order, simply relying on client-side and off-chain validations are not sufficient. It is possible for an attacker to bypass the client-side and off-chain validations and interact directly with the contract. Thus, such validation must also be implemented on the on-chain contracts.

[L-06] Order Can Be Cancelled Even After Being Filled

Once an order has been filled, no one should be able to cancel the order or mark the order as `Cancelled`.

The following code shows that the order maker can change the status of the order to `Cancelled` at any point of time.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/Putty

V2.sol#L526

```
    /**
        @notice Cancels an order which prevents it from being fillec
        @param order The order to cancel.
     */
    function cancel(Order memory order) public {
        require(msg.sender == order.maker, "Not your order");

        bytes32 orderHash = hashOrder(order);

        // mark the order as cancelled
        cancelledOrders[orderHash] = true;

        emit CancelledOrder(orderHash, order);
    }
```

Although changing the status of an order to `Cancelled` after it has been filled does not cause any lost of funds at the later stages (e.g. when exercising or withdrawing), it might cause unnecessary confusion to the users as it does not accurately reflect the status of an order on-chain.

Users might fetch the status of an order directly from the `cancelledOrders` mapping or poll the on-chain for emitted event, and come to a wrong conclusion that since the order has been cancelled, it has not been filled.

Recommendation

It is recommended to update the `cancel` function to only allow order maker to call this function only if an order has not been filled.

```
    function cancel(Order memory order) public {
        require(msg.sender == order.maker, "Not your order");

        bytes32 orderHash = hashOrder(order);

        // If an order has been filled, the positionExpirations[orde
        require(positionExpirations[orderHash] == 0, "Order has alre

        // mark the order as cancelled
```

```
        cancelledOrders[orderHash] = true;

        emit CancelledOrder(orderHash, order);
    }
```

[L-07] No Check if onERC721Received Is Implemented

The `PuttyV2.fillOrder` will mint a long position NFT and short position NFT to the order maker and taker. When minting a NFT, the function does not check if a receiving contract implements onERC721Received().

The intention behind this function is to check if the address receiving the NFT, if it is a contract, implements onERC721Received(). Thus, there is no check whether the receiving address supports ERC-721 tokens and position could be not transferrable in some cases.

Following shows that `_mint` is used instead of `_safeMint`.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L303

```
    function fillOrder(
            Order memory order,
            bytes calldata signature,
            uint256[] memory floorAssetTokenIds
    ) public payable returns (uint256 positionId) {
            ..SNIP..
            // create long/short position for maker
            _mint(order.maker, uint256(orderHash));

            // create opposite long/short position for taker
            bytes32 oppositeOrderHash = hashOppositeOrder(order);
            positionId = uint256(oppositeOrderHash);
            _mint(msg.sender, positionId);
            ..SNIP..
```

Recommendation

Consider using `_safeMint` instead of `_mint`.

[N-01] Omissions in events

Throughout the codebase, events are generally emitted when sensitive changes are made to the contracts. However, some events are missing important parameters

Instance #1 - Missing Old Value

When setting a new `baseURI` and `fee`, only the new value is emitted within the event.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L228

```
function setBaseURI(string memory _baseURI) public payable onlyO
    baseURI = _baseURI;
    emit NewBaseURI(_baseURI);
}
```

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L240

```
function setFee(uint256 _fee) public payable onlyOwner {
    require(_fee < 30, "fee must be less than 3%");
    fee = _fee;
    emit NewFee(_fee);
}
```

The events should include the new value and old value where possible.

[N-02] Draft OpenZeppelin Dependencies

The `PuttyV2` contract utilised `draft-EIP712`, an OpenZeppelin contract. This contract is still a draft and is not considered ready for mainnet use. OpenZeppelin

contracts may be considered draft contracts if they have not received adequate security auditing or are liable to change with future development.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L40

```
import "openzeppelin/utils/cryptography/SignatureChecker.sol";
import "openzeppelin/utils/cryptography/draft-EIP712.sol";
import "openzeppelin/utils/Strings.sol";
```

Recommendation

Ensure the development team is aware of the risks of using a draft contract or consider waiting until the contract is finalised.

Otherwise, make sure that development team are aware of the risks of using a draft OpenZeppelin contract and accept the risk-benefit trade-off.

[N-03] Insufficient Tests

It is crucial to write tests with possibly 100% coverage for smart contract systems.

The following functions were found to be not included in the test cases:

- `PuttyV2.batchFillOrder` - https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L546
- `PuttyV2.acceptCounterOffer` - https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L573

Recommendation

It is recommended to write proper tests for all possible code flows and specially edge cases

[N-04] Owner Can Renounce Ownership

Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities.

The Openzeppelin's `Ownable` used in `PuttyV2` contract implements `renounceOwnership`. This can represent a certain risk if the ownership is renounced for any other reason than by design. Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L53

```
/**
    @title PuttyV2
    @author out.eth
    @notice An otc erc721 and erc20 option market.
 */
contract PuttyV2 is PuttyV2Nft, EIP712("Putty", "2.0"), ERC721Tc
```

Recommendation

We recommend to either reimplement the function to disable it or to clearly specify if it is part of the contract design

[N-05] Consider two-phase ownership transfer

Admin calls `Ownable.transferOwnership` function to transfers the ownership to the new address directly. As such, there is a risk that the ownership is transferred to an invalid address, thus causing the contract to be without a owner.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L53

```
contract PuttyV2 is PuttyV2Nft, EIP712("Putty", "2.0"), ERC721Tc
```

Recommendation

Consider implementing a two step process where the admin nominates an account and the nominated account needs to call an acceptOwnership() function for the transfer of admin to fully succeed. This ensures the nominated EOA account is a valid and active account.

[N-06] Code Can Be Refactored To Be More Readable

In many parts of the `PuttyV2` contract, it uses the following conditions to check the type of the order being passed into the function:

- order.isLong && order.isCall (equal to long call)
- order.isLong && !order.isCall (equal to long put)
- order.!isLong && order.isCall (equal to short call)
- order.!isLong && order.!isCall (equal to short put)

These affect the readability of the codes as the readers have to interpret the condition to determine if it is a “long call”, “long put”, “short call” or “short put”. This might increase the risk of mistakes in the future if new developer works on the contracts.

Recommendation

Consider implementing the following functions to improve readability:

- isLongCall(Order order) public view returns (bool)
- isLongPut(Order order) public view returns (bool)
- isShortCall(Order order) public view returns (bool)
- isShortPut(Order order) public view returns (bool)

[N-07] Inconsistent use of named return variables

There is an inconsistent use of named return variables in the `PuttyV2` contract

Some functions return named variables, others return explicit values.

Following function return explicit value

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L669

```
function isWhitelisted(address[] memory whitelist, address targe
    for (uint256 i = 0; i < whitelist.length; i++) {
        if (target == whitelist[i]) return true;
    }

    return false;
}
```

Following function return return a named variable

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L683

```
function hashOppositeOrder(Order memory order) public view retur
    // use decode/encode to get a copy instead of reference
    Order memory oppositeOrder = abi.decode(abi.encode(order),

    // get the opposite side of the order (short/long)
    oppositeOrder.isLong = !order.isLong;
    orderHash = hashOrder(oppositeOrder);
}
```

Recommendation

Consider adopting a consistent approach to return values throughout the codebase by removing all named return variables, explicitly declaring them as local variables, and adding the necessary return statements where appropriate. This would improve both the explicitness and readability of the code, and it may also help reduce regressions during future code refactors.

[N-08] Unused imports

To improve readability and avoid confusion, consider removing the following unused imports:

In the `PuttyV2Nft` contract:

- openzeppelin/utils/Strings.sol

Note that the `Strings.sol` has already been imported in `PuttyV2` contract. Thus, this import can be safely removed.

Within the `PuttyV2Nft` contract, it does not use any of the functions from `Strings.sol`.

Recommendation

Consider removing the unused import if it is not required.

[N-09] Incorrect functions visibility

Whenever a function is not being called internally in the code, it can be easily declared as `external`, saving also gas. While the entire code base have explicit visibilities for every function, some of them can be changed to be `external`.

Following are some the functions that can be changed to be `external`

- `PuttyV2.fillorder`
- `PuttyV2.exercise`
- `PuttyV2.withdraw`
- `PuttyV2.batchFillOrder`
- `Putty.acceptCounterOffer`

Recommendation

Review the visibility of the affected functions and change visibility of these functions to `external`.

[N-10] NatSpec Is Missing

NatSpec if missing for the following function

- `PuttyV2Nft._mint` - https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2Nft.sol#L11
- `PuttyV2Nft.transferFrom` - https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2Nft.sol#L21
- `PuttyV2Nft.balanceOf` - https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2Nft.sol#L40

Recommendation

Implement NatSpec for all functions.

outdoteth (Putty Finance) commented:

> High quality report.

HickupHH3 commented:

> L-03 Insufficient Input Validation
>
> Disagree, I remember seeing there was a use case for having taker == maker discussed in 1 of the issues somewhere.
>
> L-04 Order Cannot Be Filled Due To Unbounded Whitelist Within An Order
>
> Sort a duplicate of #290.
>
> Agree that overall it is a very good report!

Gas Optimizations

For this contest, 94 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Alex the Entreprenerd** received the top score from the judge.

The following wardens also submitted reports: 0xA5DF, lllllll, 0x1f8b, defsec, TerrierLover, 0xf15ers, BnkeOx0, 0xkatana, ElKu, joestakey, MiloTruck, 0xNazgul, 0xsanson, gogo, grrwahrr, JohnSmith, TomJ, simon135, _Adam, 0xKitsune, Metatron, RedOneN, sashik_eth, JC, rokinot, UnusualTurtle, __141345__, Aymen0909, PwnedNoMore, saian, Tomio, rfa, zer0dot, 0xc0ffEE, BowTiedWardens, c3phas, hansfriese, jayfromthe13th, Limbooo, Picodes, ReyAdmirado, swit, Waze, z3s, durianSausage, fatherOfBlocks, oyc_109, reassor, 0v3rf10w, 0xNineDec, ajtra, ak1, catchup, delfin454000, Fitraldys, Funen, horsefacts, Kenshin, m_Rassska, mektigboy, mrpathfindr, natzuu, Randyyy, slywaters, Sm4rty, StErMi, Kaiziron, MadWookie, sach1r0, ACai, Chom, cRat1st0s, ladboy233, Lambda, rajatbeladiya, zeesaw, cryptphi, datapunk, Hawkeye, ignacio, minhquanym, 0xDjango, 0xHarry, apostle0x01, asutorufos, codetilda, exd0tpy, hake, Haruxe, robee, Ruhum, StyxRave, *and* Yiko.

Executive Summary

The codebase is already pretty well thought out, by extending the idea of using ERC721 IDs as identifiers, we can save massive amount of gas for day to day operations, we can also apply a few basic logic transformations as well as basic gas saving tips to reduce the total gas for the average operation by over 20k gas

The first few refactoring will offer strong gas savings with minimal work (20k+ gas), the remaining findings are minor and I expect most other wardens to also be suggesting them

Massive Gas Savings in `exercise` by removing `exercisedPositions` - Around 20k gas on every `exercise` (17k to 23k from foundry tests)

It seems like `exercisedPositions` is used exclusively for `withdraw`, however through the cleverly designed "send to 0xdead" system, we can replace the `exercisedPositions` function with the following

```
function exercisedPositions(uint256 id) external view returr
    return ownerOf(id) == address(0xdead);
}
```

In fact, a position was exercised if it was transfered to 0xdead, as such there's no need to store a mapping in storage.

We can delete every mention of `exercisedPositions` as well as the storage mapping

This single change will reduce almost 20k gas from `exercise`

To make `withdraw` work we can write the following

```
            // Inlined is even cheaper (8 gas for the JUMP)
            bool isExercised = ownerOf(uint256(longPositionId)) == a
```

And we can delete the `exercisedPositions` mapping from the contract

You can use the function above so the test still passes, in case you need a way to verify if a position was exercised, otherwise you can just use the `isExercised` line and delete very other mention of `exercisedPositions`

Gas Math

BEFORE exercise ¦ 5759 ¦ 55920 ¦ 68002 ¦ 133534 ¦ 18
withdraw ¦ 3075 ¦ 27840 ¦ 24545 ¦ 71168 ¦ 10

AFTER exercise ¦ 5759 ¦ 42356 ¦ 45806 ¦ 115777 ¦ 18 withdraw ¦ 3075 ¦ 26968 ¦ 23807 ¦ 70836 ¦ 10

Diff

```
155c155,157
<     mapping(uint256 => bool) public exercisedPositions;
---
>     function exercisedPositions(uint256 id) external view retu
>         return ownerOf(id) == address(0xdead);
>     }
415,417d416
<         // mark the position as exercised
<         exercisedPositions[uint256(orderHash)] = true;
<
478c477
<         bool isExercised = exercisedPositions[longPositionId];
```

```
---
>          bool isExercised = ownerOf(longPositionId) == address(
```

Avoid a SLOAD optimistically - 2.1k gas saved

Because you already computed `isExercised`, you can save an SLOAD (2.1k gas) if you check for `isExercised` first.

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L481-L482

```
            require(block.timestamp > positionExpirations[longPositi
```

You could also refactor to the reverse check (if you expect the majority of positions to expire worthless), either way the short circuit can be used to save a SLOAD, whicever priority you give it

Change to

```
            require(isExercised || block.timestamp > positionExpirat
```

To save 2.1k gas if the option was exercised

Double SLOAD - 94 gas saved

Everytime a Storage Variable is loaded twice, (SLOAD), it would be cheaper to use a supporting memory variable The cost of a Second SLOAD is 100 gas, the cost of an MSTORE is 3 and and MLOAD another 3 gas.

Using a supporting variable will save 94 gas on the second read, and 97 gas for each subsquent MSTORE

In the case of `fee` this can save 94 gas

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L498-L499

```
                if (fee > 0) {
                    feeAmount = (order.strike * fee) / 1000;
```

Storage Pointer to `floorTokenIds` will save gas (avoid copying whole storage to memory) - Between 400 and 2k gas saved on Withdraw and Exercise - 400 / 2k * 2 gas saved

If you pass a `storage` pointer as argument to a `memory` typed function like `_transferFloorsOut`

```
        function _transferFloorsOut(address[] memory floorTokens, ui
```

You are copying the storage values to memory and then reading them and looping through that copy, this incurs an overhead and is equivalent to looping over storage, copying into memory and then passing the memory variable.

This is one of the few cases where a storage declaration (passing the storage pointer) will save gas

Refactor to

```
        function _transferFloorsOut(address[] memory floorTokens, ui
```

Gas Math

Before

exercise ¦ 5759 ¦ 55920 ¦ 68002 ¦ 133534 ¦ 18 withdraw ¦ 3075 ¦ 27840 ¦ 24545 ¦ 71168 ¦ 10

After exercise ¦ 5759 ¦ 55176 ¦ 65795 ¦ 133534 ¦ 18
withdraw ¦ 3075 ¦ 27365 ¦ 24545 ¦ 71003 ¦ 10

Save gas by avoiding NOT - Saves 6 gas sometimes

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L403-L406

```
            // check floor asset token ids length is 0 unless the pc
            !order.isCall
                ? require(floorAssetTokenIds.length == order.floorTc
                : require(floorAssetTokenIds.length == 0, "Invalid f
```

Because the logic is fully known, it would be cheaper to swap the if else to:

```
            order.isCall
                ? require(floorAssetTokenIds.length == 0, "Invalid f
                : require(floorAssetTokenIds.length == order.floorTc
```

As that would avoid the extra `NOT`

Check msg.value first - 1 gas per instance - 3 gas total

Reading msg.value costs 2 gas, while reading from memory costs 3, this will save 1 gas with no downside

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L327

```
    if (weth == order.baseAsset && msg.value > 0) {
```

Change to

```
    if (msg.value > 0 && weth == order.baseAsset) {
```

Common Gas Savings

Below are a bunch of basic gas saving tips you probably will receive in most competent submissions added below for the sake of completeness

Save 3 gas by not reading `orders.length` - 3 gas per instance - 27 gas saved

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L551-L552

```
            require(orders.length == signatures.length, "Length misn
```

`orders.length` is read multiple times, because of that, especially for the loop, you should cache it in a memory variable to save 3 gas per instance

Refactor to:

```
            ordersLength = orders.length;
            require(orders.length == signatures.length, "Length misn
```

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L670-L671

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L658

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L647

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L637

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L627

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L611

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L594

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L728

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L742

Avoid default assignment - 3 gas per instance - 27 gas saved

Declaring `uint256 i = 0;` means doing an MSTORE of the value 0 Instead you could just declare `uint256 i` to declare the variable without assigning it's default value, saving 3 gas per declaration

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L556-L557

```
for (uint256 i = 0; i < orders.length; i++) {
```

Instances: https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/Putty

V2.sol#L670-L671

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L658

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L647

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L637

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L627

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L611

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L594

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L728

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L742

Cheaper For Loops - 25 to 80 gas per instance - 9 instances

You can get cheaper for loops (at least 25 gas, however can be up to 80 gas under certain conditions), by rewriting:

```
for (uint256 i = 0; i < orders.length; /** NOTE: Removec
        // Do the thing

        // Unchecked pre-increment is cheapest
        unchecked { ++i; }
}
```

Instances: https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L670-L671

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L658

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L647

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L637

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L627

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L611

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L594

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L728

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L742

Simplify Value Comparison - Saves some bytecode and makes logic leaner

https://github.com/code-423n4/2022-06-putty/blob/3b6b844bc39e897bd0bbb69897f2deff12dc3893/contracts/src/PuttyV2.sol#L495-L496

```
if ((order.isCall && isExercised) || (!order.isCall && !
```

Can be refactored with

```
// transfer strike to owner if put is expired or call is
if (order.isCall == isExercised) {
```

As in both case they were both true or both false

Consequence of Smart Comparison Above - Saves 20 to 80 gas (avg is 60)

Because of the gas save above we can refactor the entire block to an if / else

```
if ((order.isCall && isExercised) || (!order.isCall && !
    // send the fee to the admin/DAO if fee is greater t
    uint256 feeAmount = 0;
    if (fee > 0) {
        feeAmount = (order.strike * fee) / 1000;
        ERC20(order.baseAsset).safeTransfer(owner(), fee
    }

    ERC20(order.baseAsset).safeTransfer(msg.sender, orde

    return;
}

// transfer assets from putty to owner if put is exercis
if ((order.isCall && !isExercised) || (!order.isCall &&
    _transferERC20sOut(order.erc20Assets);
```

```
        _transferERC721sOut(order.erc721Assets);

        // for call options the floor token ids are saved ir
        // and for put options the floor tokens ids are save
        uint256 floorPositionId = order.isCall ? longPositic
        _transferFloorsOut(order.floorTokens, positionFloorA

        return;
    }
```

Becomes

```
    if (order.isCall == isExercised)) {
            // transfer assets from putty to owner if put is

            // send the fee to the admin/DAO if fee is great
            uint256 feeAmount = 0;
            if (fee > 0) {
                    feeAmount = (order.strike * fee) / 1000;
                    ERC20(order.baseAsset).safeTransfer(owne
            }

            ERC20(order.baseAsset).safeTransfer(msg.sender,
    } else {
            // transfer assets from putty to owner if put is

            _transferERC20sOut(order.erc20Assets);
            _transferERC721sOut(order.erc721Assets);

            // for call options the floor token ids are save
            // and for put options the floor tokens ids are
            uint256 floorPositionId = order.isCall ? longPos
            _transferFloorsOut(order.floorTokens, positionFl
    }
```

Which reduces bytecode size, and saves gas in most situations as you're avoiding up to 3 extra comparisons

[outdoteth commented]():

> Cool idea with relying on 0xdead to see if an option is exercised or not. Unfortunately this doesnt work because a user can intentionally “burn” their NFT to 0xdead which then messes up the logic in withdraw. This can be mitigated, but requires too many changes.

> High quality report though.

Mitigation Review

Mitigation review by **hyh**

Project repository: **https://github.com/outdoteth/putty-v2/tree/master/src**

Review commit: 6df33b2f20e8ddc1bf3dadde4feb834333b2c218

Intro

The following is a review of mitigations originated from Code4rena (C4) audit contest that took place between June 29 and July 4, 2022.

Disclaimer

This mitigation review does not guarantee the absence of any further vulnerabilities, being, however, the result of exercising reviewer’s best efforts. Also, it focuses on the resolved issues and the resulting code base, leaving out of scope all the acknowledged issues from the original report.

Mitigation Overview

The following is a high-level overview of the core changes introduced as the mitigation `PR#1-10`, arranged per original report findings.

- [H-01] Fixed by changing the fee base to be `order.premium` `PR#4`, which is now paid uniformly for all option types on order filling. Utilizing `order.strike` as the fee base was the root cause for [M-04], [M-06], [M-11], [M-15], so the change to `order.premium` was a shared mitigation for all of them.

- [H-02] Fixed by requiring that order can't be in the filled state on cancel. This fully adheres to the original logic, but wasn't controlled for before.
- [H-03] Fixed by prohibiting non-empty `order.floorTokens` for short calls. Other option types do need `floorTokens`: long calls' taker provides floor tokens on filling, while long put owner brings in the floor tokens on exercise, taking the strike. Short put owner can thereafter retrieve the tokens on withdraw.
- [H-04] Fixed by conditioning call's logic on `order.strike > 0`. There is no use case for zero strike puts and so this case remains unconditioned, i.e. still always require successful `order.strike` transfer.
- [M-01, M-02] Acknowledged, now addressed on UI/DB level.
- [M-03] Fixed zero amount part by introducing the noop for zero amount transfers in both `_transferERC20sIn` and `_transferERC20sOut` ERC20 transfer functions. The second part of the issue, fake tokens, is similar to [M-01], [M-02].
- [M-04] The same fix as in [H-01].
- [M-05] Fixed with native funds amount control added to strike transfer logic of `fillOrder`. Zero strike and zero premium corner cases are yet unhandled as described below in M.M-01 and M.M-02.
- [M-06] The same fix as in [H-01]: as the platform fee is now transferred on order filling, any owner griefing can only yield a denial of service. There will be no loss of funds as this way position is only about to be created when the fee is transferred.
- [M-07] Acknowledged, similar to [M-01], [M-02].
- [M-08] Acknowledged, requires asset whitelisting, now addressed on UI/DB level.
- [M-09] Acknowledged.
- [M-10] Acknowledged.
- [M-11] The same fix as in [H-01].
- [M-12] Fixed along with [H-01] in `PR#4`.
- [M-13] Fixed by requiring the minimal order duration of 15 minutes on filling.
- [M-14] Fixed by the introduction of `setMinimumValidNonce` function and the corresponding control on order filling. See M.M-04 below.

- [M-15] The same fix as in [H-01].
- [M-16] Fixed by bumping the solidity version.

Findings

Findings are referenced as `M.S-N`, `Mitigation.Severity-Number`.

Source means the source of the latest relevant change, not necessary the issue itself, as some of them existed in the snapshot used in the original audit.

M.M-01 WETH zero strike call fillOrder's msg.value will be lost

If there is any `msg.value` mistakenly attached to fillOrder() for a WETH based zero strike call, it will be lost for a caller. The same happens in exercise().

That's asset freezing due to user's mistake impact in both cases.

Proof of Concept

Namely, when `order.baseAsset == address(weth)`, `order.isCall` and `order.strike == 0`, the native funds attached to the fillOrder() call aren't controlled and can be lost:

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L504-L505

```
            // check native eth is only used if baseAsset is weth
            require(msg.value == 0 || order.baseAsset == address(wet
```

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L527-L553

```
            if (order.isCall) {
                // -- exercising a call option

                // transfer strike from exerciser to putty
```

```
            // handle the case where the taker uses native ETH i
            if (order.strike > 0) {
                if (msg.value > 0) {
                    // check enough ETH was sent to cover the st
                    require(msg.value == order.strike, "Incorrec

                    // convert ETH to WETH
                    // we convert the strike ETH to WETH so that
                    // - because withdraw() assumes an ERC20 int
                    IWETH(weth).deposit{value: msg.value}();
                } else {
                    ERC20(order.baseAsset).safeTransferFrom(msg.
                }
            }

            // transfer assets from putty to exerciser
            _transferERC20sOut(order.erc20Assets);
            _transferERC721sOut(order.erc721Assets);
            _transferFloorsOut(order.floorTokens, positionFloor
        } else {
            // -- exercising a put option
            // exercising a put never needs native ETH
            require(msg.value == 0, "Puts can't use native ETH")
```

The same approach is used in exercise(), and the same WETH zero strike call's `msg.value` isn't controlled and can be lost:

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L504-L505

```
        // check native eth is only used if baseAsset is weth
        require(msg.value == 0 || order.baseAsset == address(wet
```

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L527-L544

```
        if (order.isCall) {
            // -- exercising a call option
```

```
// transfer strike from exerciser to putty
// handle the case where the taker uses native ETH i
if (order.strike > 0) {
    if (msg.value > 0) {
        // check enough ETH was sent to cover the st
        require(msg.value == order.strike, "Incorrec

        // convert ETH to WETH
        // we convert the strike ETH to WETH so that
        // - because withdraw() assumes an ERC20 int
        IWETH(weth).deposit{value: msg.value}();
    } else {
        ERC20(order.baseAsset).safeTransferFrom(msg.
    }
}
```

Such ETH funds will be permanently frozen on the contract balance as there is no mechanics to retrieve them.

Source

PR#3

Recommended Mitigation Steps

Consider checking that nothing is attached to the zero strike call:

fillOrder()

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L530-L544


```
// transfer strike from exerciser to putty
// handle the case where the taker uses native ETH i
if (order.strike > 0) {
    if (msg.value > 0) {
        // check enough ETH was sent to cover the st
        require(msg.value == order.strike, "Incorrec

        // convert ETH to WETH
```

```
                    // we convert the strike ETH to WETH so that
                    // - because withdraw() assumes an ERC20 int
                    IWETH(weth).deposit{value: msg.value}();
                } else {
                    ERC20(order.baseAsset).safeTransferFrom(msg.
                }
-           }
+           } else {
+               require(msg.value == 0, "0 strike calls can't us
+           }
```

exercise()

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L530-L544

```
            // transfer strike from exerciser to putty
            // handle the case where the taker uses native ETH i
            if (order.strike > 0) {
                if (msg.value > 0) {
                    // check enough ETH was sent to cover the st
                    require(msg.value == order.strike, "Incorrec

                    // convert ETH to WETH
                    // we convert the strike ETH to WETH so that
                    // - because withdraw() assumes an ERC20 int
                    IWETH(weth).deposit{value: msg.value}();
                } else {
                    ERC20(order.baseAsset).safeTransferFrom(msg.
                }
-           }
+           } else {
+               require(msg.value == 0, "0 strike calls can't us
+           }
```

M.M-02 Zero premium short fillOrder's msg.value will be lost

If there is any native funds mistakenly attached to fillOrder() for a zero premium short option, `msg.value` will be lost for a caller. As an example, zero premiums can

correspond to the case when option is settled off-chain and access is managed with `order.whitelist`.

The impact is asset freezing due to user's mistake. The freeze is permanent as there is no mechanics to retrieve native funds from the contract.

Proof of Concept

Similarly, when `!order.isLong` and `order.premium == 0`, the native funds attached to the fillOrder() call aren't controlled and can be lost as the checks reside in the `if (order.premium > 0) {` block:

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L408-L442

```
        // transfer premium to whoever is short from whomever is
        if (order.premium > 0) {
            if (order.isLong) {
                // transfer premium to taker
                ERC20(order.baseAsset).safeTransferFrom(order.ma

                // collect fees
                if (feeAmount > 0) {
                    ERC20(order.baseAsset).safeTransferFrom(orde
                }
            } else {
                // handle the case where the user uses native ET
                if (msg.value > 0) {
                    // check enough ETH was sent to cover the pr
                    require(msg.value == order.premium, "Incorre

                    // convert ETH to WETH and send premium to m
                    // converting to WETH instead of forwarding
                    // 1) active market makers will mostly be us
                    // 2) attack surface for re-entrancy is redu
                    IWETH(weth).deposit{value: order.premium}();

                    // collect fees and transfer to premium to m
                    IWETH(weth).transfer(order.maker, order.prem
                } else {
                    // transfer premium to maker
```

```
                ERC20(order.baseAsset).safeTransferFrom(msg.

                // collect fees
                if (feeAmount > 0) {
                    ERC20(order.baseAsset).safeTransferFrom(
                }
            }
        }
    }
```

After that short option cases only transfer the assets (`!order.isLong && !order.isCall` and `!order.isLong && order.isCall`):

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L444-L464

```
        if (!order.isLong && !order.isCall) {
            // filling short put: transfer strike from maker to
            ERC20(order.baseAsset).safeTransferFrom(order.maker,
        } else if (order.isLong && !order.isCall) {
            // filling long put: transfer strike from taker to c
            // handle the case where the taker uses native ETH i
            if (msg.value > 0) {
                // check enough ETH was sent to cover the strike
                require(msg.value == order.strike, "Incorrect ET

                // convert ETH to WETH
                // we convert the strike ETH to WETH so that the
                // - because exercise() assumes an ERC20 interfa
                IWETH(weth).deposit{value: msg.value}();
            } else {
                ERC20(order.baseAsset).safeTransferFrom(msg.senc
            }
        } else if (!order.isLong && order.isCall) {
            // filling short call: transfer assets from maker tc
            _transferERC20sIn(order.erc20Assets, order.maker);
            _transferERC721sIn(order.erc721Assets, order.maker);
```

Recommended Mitigation Steps

It's enough to add the similar check for `!order.isLong` case when main logic is avoided:

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L408-L442

```
            // transfer premium to whoever is short from whomever is
            if (order.premium > 0) {
                if (order.isLong) {
                    // transfer premium to taker
                    ERC20(order.baseAsset).safeTransferFrom(order.ma

                    // collect fees
                    if (feeAmount > 0) {
                        ERC20(order.baseAsset).safeTransferFrom(orde
                    }
                } else {
                    // handle the case where the user uses native ET
                    if (msg.value > 0) {
                        // check enough ETH was sent to cover the pr
                        require(msg.value == order.premium, "Incorre

                        // convert ETH to WETH and send premium to m
                        // converting to WETH instead of forwarding
                        // 1) active market makers will mostly be us
                        // 2) attack surface for re-entrancy is redu
                        IWETH(weth).deposit{value: order.premium}();

                        // collect fees and transfer to premium to m
                        IWETH(weth).transfer(order.maker, order.prem
                    } else {
                        // transfer premium to maker
                        ERC20(order.baseAsset).safeTransferFrom(msg.

                        // collect fees
                        if (feeAmount > 0) {
                            ERC20(order.baseAsset).safeTransferFrom(
                        }
                    }
                }
-           }
+           } else {
+               if (!order.isLong) require(msg.value == 0, "0 premiu
```

```
+         }
```

M.M-03 Unsafe transfer of an arbitrary token is used by withdrawFees() for the cumulative fee retrieval

withdrawFees() will not revert on the transfer call failure and will be inaccessible for tokens not fully compliant with Solmate's ERC20 (say, USDT).

One issue is that when a token do not revert, but instead returns false, this will go unnoticed. As `unclaimedFees` will be reset to `0` without actual transfer, this means that the cumulative `asset` denominated fee that PuttyV2 instance had at that moment will become frozen within the contract.

Another is that ERC20 compliance is expected by using `ERC20(asset).transfer`, while it is not always the case and such `assets` will not be retrievable by withdrawFees() as the call will fail due to function signature mismatch. The corresponding cumulative fee will also be frozen on the balance.

Proof of Concept

withdrawFees() uses plain `transfer` call while the `asset`, being `order.baseAsset`, can be arbitrary:

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L301-L311

```
    function withdrawFees(address asset, address recipient) publ
        uint256 fees = unclaimedFees[asset];

        // reset the fees
        unclaimedFees[asset] = 0;

        emit WithdrewFees(asset, fees, recipient);

        // send the fees to the recipient
        ERC20(asset).transfer(recipient, fees);
    }
```

https://github.com/Rari-Capital/solmate/blob/25015a1d18e75921f8cb1bacd4beb9f364e788a9/src/tokens/ERC20.sol#L76

```
    function transfer(address to, uint256 amount) public virtual
```

Source

PR#4

Recommended Mitigation Steps

Consider using Solmate's safeTransfer that performs lower level call with result check:

https://github.com/Rari-Capital/solmate/blob/ca96d493e0b061cccccf02b2fd9f5c6fe08826df/src/utils/SafeTransferLib.sol#L63-L92

```
    function safeTransfer(
    ...
```

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L301-L311

```
    function withdrawFees(address asset, address recipient) publ
        uint256 fees = unclaimedFees[asset];

        // reset the fees
        unclaimedFees[asset] = 0;

        emit WithdrewFees(asset, fees, recipient);

        // send the fees to the recipient
-       ERC20(asset).transfer(recipient, fees);
+       ERC20(asset).safeTransfer(recipient, fees);
    }
```

M.M-04 minimumValidNonce can be reduced due to an operational mistake, enabling old orders

`setMinimumValidNonce()` , introduced as the mitigation for `M-14` , allows user to both decrease and increase personal nonce. The regular operation here is nonce increase, which invalidates outdated orders.

Nonce decrease operation, on the other hand, has no regular use case and is a subject to an operational mistakes that can easily lead to loss of funds as outdated orders were set to inactive state by nonce increased most likely in result of becoming non-market and so enabling them will allow an attacker, who can track the setMinimumValidNonce() calls, to immediately fill such orders in the case they indeed are now off the market and were enabled by mistake.

Maker's allowances for the assets in question can remain open as other similar, but currently marked to market, same maker's orders can be live on Putty or elsewhere.

This way, an ability to decrease the nonce provides a surface for user mistakes with highly probable loss of funds: most of orders are invalidated as out of the market, while some allowances can realistically remain, which gives medium overall probability of a loss given erroneous nonce decrease by a user. As user has to track the nonces and so such a decrease is conditional only on user failing to do so, which isn't a negligible event.

Impact this way is loss of funds conditional on user operational mistake with both events having medium probability.

Proof of Concept

`setMinimumValidNonce()` allows setting any nonce, including one that's significantly lower than current:

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L700-L709

```
        /**
            @notice Sets the minimum valid nonce for a user. Any uni
                    smaller than this minimum will no longer be vali
```

```
        @param _minimumValidNonce The new minimum valid nonce.
     */
    function setMinimumValidNonce(uint256 _minimumValidNonce) pu
        minimumValidNonce[msg.sender] = _minimumValidNonce;

        emit SetMinimumValidNonce(_minimumValidNonce);
    }
```

Source

PR#10

Recommended Mitigation Steps

Consider requiring new nonce to be strictly higher than the old one or, more preferrably, restrict the function to exactly one step increase:

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L700-L709

```
    /**
-       @notice Sets the minimum valid nonce for a user. Any unf
+       @notice Increments the minimum valid nonce for a user. A
                smaller than this minimum will no longer be vali
-       @param _minimumValidNonce The new minimum valid nonce.
     */
-   function setMinimumValidNonce(uint256 _minimumValidNonce) pu
+   function incrementMinimumValidNonce() public {
-       minimumValidNonce[msg.sender] = _minimumValidNonce;

+       emit SetMinimumValidNonce(++minimumValidNonce[msg.sender
    }
```

M.L-01 Fee limit remain the same after fee mechanics base has changed

The 3% limit stayed the same after fee mechanics change. 3% of a premium and 3% of a strike can be vastly different as average strike is about 1-2 magnitudes higher than average premium (i.e. the crude evaluation that most options are priced at 1-10% of their strikes isn't too incorrect).

This way, while 3% of the strike was rather high, being more than the whole premium in some cases, 3% of the premium can be too restrictive as a limit. Anyway, lesser fees are good for protocol adoption, so this is just a matter of reevaluation, not necessary change.

Proof of Concept

`setFee()` uses `30 / 1000` limit as it was in the `order.strike` based fee mechanics:

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L288-L294

```
        function setFee(uint256 _fee) public payable onlyOwner {
            require(_fee < 30, "fee must be less than 3%");

            fee = _fee;

            emit NewFee(_fee);
        }
```

Now it's `order.premium` based:

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L401-L406

```
            // calculate the fee amount
            uint256 feeAmount = 0;
            if (fee > 0) {
                feeAmount = (order.premium * fee) / 1000;
                unclaimedFees[order.baseAsset] += feeAmount;
            }
```

Recommended Mitigation Steps

Review the fee limit taking the account changed magnitude of the average base amount

M.N-01 Unnecessary payable functions

Administrative functions have payable modifiers that aren't used as no fund transfers are involved, while enabling Owner's operational mistakes surface.

There is only a small gas cost for checking zero `msg.value`, while any native funds sent over with such functions will be frozen permanently.

Proof of Concept

setBaseURI():

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L276-L280

```
    function setBaseURI(string memory _baseURI) public payable o
        baseURI = _baseURI;

        emit NewBaseURI(_baseURI);
    }
```

setFee():

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L288-L294

```
    function setFee(uint256 _fee) public payable onlyOwner {
        require(_fee < 30, "fee must be less than 3%");

        fee = _fee;

        emit NewFee(_fee);
    }
```

withdrawFees():

https://github.com/outdoteth/putty-v2/blob/6df33b2f20e8ddc1bf3dadde4feb834333b2c218/src/PuttyV2.sol#L301-L311

```
        function withdrawFees(address asset, address recipient) publ
            uint256 fees = unclaimedFees[asset];

            // reset the fees
            unclaimedFees[asset] = 0;

            emit WithdrewFees(asset, fees, recipient);

            // send the fees to the recipient
            ERC20(asset).transfer(recipient, fees);
        }
```

Reference

`msg.value` check costs about 25 gas:

https://coinsbench.com/solidity-payable-vs-regular-functions-a-gas-usage-comparison-b4a387fe860d

Recommended Mitigation Steps

As there operations are rare and gas increase is low consider removing `payable` modifiers

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct

formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top