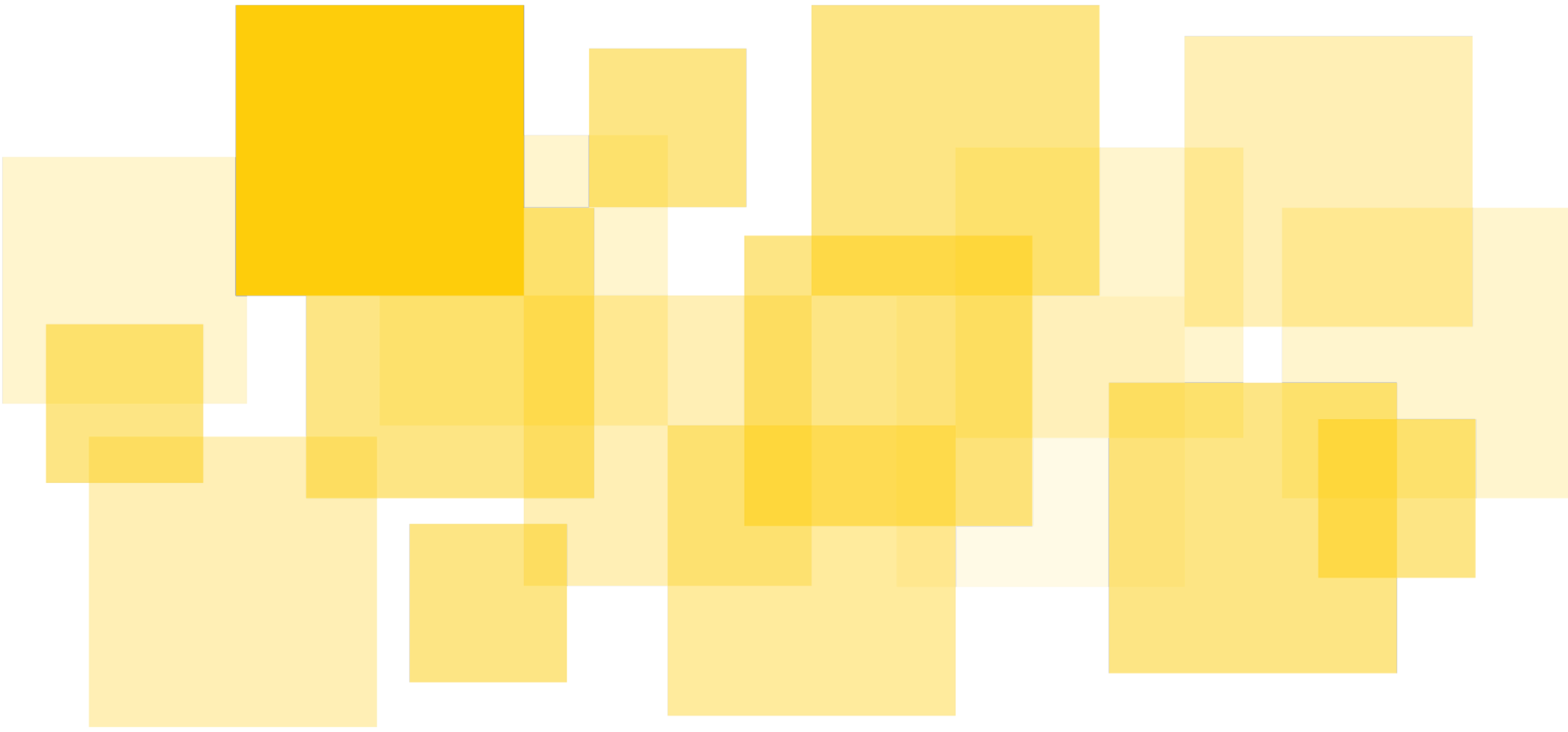
Security Audit Report

Stakefish Ethereum Staking 2.0

Delivered: October 22, 2021



Prepared for Stakefish by

runtime verification

Summary

Runtime Verification, Inc. conducted a security audit on the Stakefish Ethereum Staking 2.0 contract. The audit was conducted from October 4, 2021 to October 22, 2021.

Scope

The target of the audit is the smart contract source files at git-commit-id d91928f3a270f6115831fe3a21a69eb98bf57b26. We also made a best effort to audit the changes from commit-id 0e964a40f69cfa7d56f124efe694b5e55410ed5b. Best effort means that it was performed beyond the agreed-upon scope, and it may not necessarily have the same quality.

The audit focused on the following core contracts, reviewing their functional correctness and integration with external contracts:

- `StakefishERC20Wrapper.sol`
- `StakefishERC721Wrapper.sol`
- `StakefishServicesContract.sol`
- `StakefishServicesContractFactory.sol`
- `interfaces/IStakefishServicesContract.sol`
- `interfaces/IStakefishServicesContractFactory.sol`

The library contracts (`libraries/`) and interfaces (`interfaces/`) were given only lightweight review.

The audit is limited in scope within the boundary of the Solidity contract only. Off-chain and client-side portions of the codebase, as well as deployment and upgrade scripts are *not* in the scope of this engagement.

Assumptions

The audit is based on the following assumptions and trust model.

- The operator of a service contract is trusted to be responsive and behave honestly throughout the contract's lifetime. This assumption is critical to rule out the deposit front-running vulnerability. The client has confirmed that this assumption will be satisfied because of their operating model.
- The intended exit date of a validator is set far enough in the future. In particular, we assume that the Eth2.0 merge will happen before the exit date.
- The probability of address collisions, in particular in the presence of deterministic addresses, is sufficiently small. Specifically, we assume that the addresses of the

deployed contracts cannot be owned simultaneously by any user other than the contract owner.

- Cryptographic primitives (e.g., keccak256) possess their intended security properties.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have followed the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Then, we carefully checked if the code is vulnerable to known security issues and attack vectors. Finally, we symbolically executed part of the compiled bytecode to systematically search for unexpected, possibly exploitable, behaviors at the bytecode level, that are due to EVM quirks or Solidity compiler bugs.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Findings

A01: Transaction Order Dependence

[Severity: Medium | Difficulty: Medium | Category: Security]

The `approve` and `approveWithdrawal` methods of `StakefishServicesContract.sol` suffer from the same race-condition as ERC20 (https://swcregistry.io/docs/SWC-114).

Scenario

1. Alice allows Bob to withdraw 16Eth.
2. She later decides that she doesn't fully trust Bob and wants to reduce his allowance to 8Eth.
3. Bob sees Alice's transaction and front runs her to withdraw 16Eth.
4. If Bob's transaction is included before Alice's, he can withdraw another 8Eth.
5. Bob has now withdrawn 24Eth from Alice.

Recommendation

A straightforward mitigation technique is to implement `incrementApproval` and `decrementApproval` methods instead of `approve`. If Alice used `decreaseAllowance` in the above scenario, then Bob could still front-run Alice and withdraw his allowance before it is decremented, but not more than that.

`decrementApproval` could either revert when the new amount would underflow or set the new amount to zero instead. The first option is still susceptible to a front-running scenario:

1. Alice approves Bob 16Eth
2. Alice decreases the allowance by 8Eth
3. Bob sees Alice's tx on the mempool and withdraws 9Eth
4. If Bob's tx is included before Alice's, Alice's tx will revert due to underflow
5. Bob can later withdraw another 7Eth.

The second option avoids this scenario by enforcing a decrementation. Remark, however, that Bob could still withdraw 16Eth in step 3 as mentioned above.

Status

Fixed in 3df5f54b0b11bfe1b6c71fa92cd301e0cc637e01 and added forced decrease variants in f3fc10c4f85a536a9cb21b01913cfa532cd7022e

A02: Possibility of disproportional payments after services end

[Severity: High | Difficulty: High | Category: Security]

Once the StakefishServicesContract enters the Withdrawn phase, depositors can withdraw their share from the contract. If ServiceContract hasn't been paid its collateral and staking rewards, depositors won't get their fair proportional share if they withdraw too early.

Scenario

1. Let's assume that each of Alice and Bob has deposited 16Eth.
2. The validator has performed well over its lifetime and accumulated 1Eth of staking rewards.
3. If `endOperatorServices` is called before the collateral + rewards are paid back to the service contract, two things can happen:
    a. The operator will not receive his commission fee because the contract's balance is less than 32Eth.
    b. If Alice withdraws her shares too early, she will be paid 0 Eth. Bob is more observant and waits until the service contract has been paid. He can then withdraw 33Eth.

Discussion

Stakefish has opted to allow depositors to enter the withdrawal phase to protect their stakes if the operator isn't working correctly or responding. In this case, depositors should still be able to withdraw their funds. As mitigation against the above scenario, the operator chooses a long waiting time (`_exitDate + 1 year`) before depositors can call `endOperatorServices` successfully. The residual risk remains - especially before the Eth 2.0 merge has happened - that this waiting period is too short, and the scenario could be exploited. Alternative mitigations, like a committee-based approach to exit, were considered but rejected due to their complexity and additional risks.

Recommendation

Since the operator is trusted, we recommend that the operator initializes service contracts with an `exitDate` that lives far in the future to minimize the residual risk of the above scenario. Additionally, depositors and approved withdrawers should carefully check that the collateral and staking rewards have been paid back to the service contract before withdrawing. Users should be informed transparently about this scenario.

Furthermore, as an orthogonal measure, add a slippage protection for withdrawals. Users are asked to provide the minimum amount of proceeds for their withdrawal requests, and the minimum amount is compared to the return value of `_executeWithdrawal()`. If the return value is smaller than the minimum, then the withdrawal request should revert. This can help prevent users from requesting withdrawals too early and losing their funds accidentally. Note that the minimum amount can also be automatically and conservatively calculated by the UI front-end by retrieving the eth2 state off-chain.

Status

Addressed in f3fc10c4f85a536a9cb21b01913cfa532cd7022e by adding the slippage protection.

A03: StakefishERC721Wrapper non-conformance with ERC721

[Severity: High | Difficulty: Low | Category: Functional Correctness]

ERC721 requires that the following methods throw, if `_to` is the zero-address:

- `safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes data)`
- `safeTransferFrom(address _from, address _to, uint256 _tokenId)`
- `transferFrom(address _from, address _to, uint256 _tokenId)`

`StakefishERC721Wrapper` doesn't check these conditions and doesn't throw.

Tokens owned by the zero-address are interpreted as non-existent tokens. Therefore the transfer methods can lead to an unintentional burning of tokens.

Scenario

1. Alice mints some StakefishERC721 tokens
2. Alice interacts with the contract through an external tool that expects that the StakefishERC721Wrapper conforms to the ERC721 specification.
3. The external tool calls the `safeTransferFrom` function on Alice's behalf with the _to-address set to the zero-address.
4. The external tool does not check that the _to-address is zero because it is built on the assumption that the StakefishERC721Wrapper will revert anyway.
5. The call succeeds, and Alice loses her tokens.

Status

Fixed in 3df5f54b0b11bfe1b6c71fa92cd301e0cc637e01.

A04: Vulnerability in StakefishServicesContract.receive()

[Severity: Medium to High | Difficulty: Medium to High | Category: Security]

The receive() function implements a feature that allows users to deposit via a plain Ether transfer transaction instead of calling the deposit() function. This can lead to the loss of user funds.

Scenario 1

Suppose the balance of the service contract *S* has been 31 ether for a while. Suppose the following two things happen at a similar time:

1. Alice notices the opportunity to deposit 1 ether to *S*, and sends her 1 ether to the service contract, via a plain Ether transfer rather than submitting a deposit() transaction.
2. The operator of *S* doesn't want to wait for more investors and submit a *single* transaction that deposits 1 ether, followed by calling createValidator().

If the operator's transaction (the second one) is somehow processed before Alice's by miners, then Alice loses her funds and cannot claim it back. Her funds will be just distributed by other deposits later when they withdraw.

Scenario 2

Suppose a service contract is deployed but not yet initialized for some reason. Without noticing the uninitialized status, Bob sends his funds to the service contract via a plain Ether transfer, and he will lose his funds.

Recommendation

For the short term, clearly document this behavior and strongly recommend users to call the deposit() function rather than a plain Ether transfer when they deposit.

For the long term, redesign the architecture to receive the eth2 withdrawal payment in a separate contract so that the service contract can reject users' ether transfer after createValidator() is executed or before the contract is initialized.

Status

Addressed in 3df5f54b0b11bfe1b6c71fa92cd301e0cc637e01 by removing the plain deposit feature.

A05: Initialization of the logic contract of StakefishServicesContract

[Severity: Unknown | Difficulty: Low | Category: Security]

In the constructor() of the StakefishServicesContractFactory contract, an instance of the StakefishServicesContract contract (which is to be used as the implementation contract of proxies) is deployed but *never* initialized. Although the uninitialized implementation contract will *not* directly harm proxies associated with it, this may open other attack surfaces and lead to undesired situations.

Scenario

Suppose the implementation contract is deployed without being initialized. Several undesired scenarios are possible.

- Users may accidentally send their funds into the implementation contract, possibly due to misunderstanding the proxy pattern. Since it is uninitialized, the `_state` value is `NotInitialized`. Thus any plain Ether transfer to it will *not* be recorded in the `_deposits` mapping, which cannot be withdrawn.
- On the other hand, adversaries may initialize the implementation contract with their own address, becoming the operator for it. They may pretend to be a legitimate operator and phish users (e.g., phishing websites that look identical to the legitimate one). Poor users may deposit funds into it, and adversaries may steal users' funds (e.g., using the exploit described in A02, or convincing users to approve withdrawals).

Recommendation

For the short term, consider initializing the `_servicesContractImpl` contract, e.g., `initialize(0, 0, 0)`, in the StakefishServicesContractFactory.constructor(). This way, no one can take control of the implementation contract because `_operatorAddress` is set to zero. Moreover, the `_state` is set to PreDeposit, and thus any accidental deposits (made by either deposit() calls or plain Ether transfers) into the implementation contract can be withdrawn later, preventing users from losing funds by mistake.

For the long term, consider adding a pause feature to the service contract and pause the implementation contract at the time of deployment so that any (accidental or malicious) interactions to the implementation contract can be rejected in the first place. Note that such a pause feature can also be helpful for actual service contract instances in case of any unexpected (internal or external) situations.

Status

Fixed in 3df5f54b0b11bfe1b6c71fa92cd301e0cc637e01.

A06: DoS with failed calls for endOperatorServices()

Accidentally or maliciously, operators could become incapable of receiving ethers, which could lock user funds indefinitely.

Scenario

Suppose the operator becomes non-functioning and incapable of receiving ethers. Suppose that the eth2 exit payment is received and the balance is now bigger than 32 ether. Then, endOperatorServices() will fail because of the failure of the operator fee payment. That means the funds will be locked, and the depositors cannot withdraw their funds.

Recommendation

Consider implementing the pull pattern for the operator fee payment. That is, endOperatorServices() puts aside the fee, and the operator claims it later.

Status

Fixed in 3df5f54b0b11bfe1b6c71fa92cd301e0cc637e01.

Informative Findings

B01: Unused Constant

[Severity: N/A | Difficulty: N/A | Category: Best Practice]

`StakefishServicesContract` defines a constant `MAX_TIME_TO_WITHDRAW` that is never used.

Status

Fixed in 3df5f54b0b11bfe1b6c71fa92cd301e0cc637e01.

B02: Missing check for exit date during endOperatorServices

[Severity: N/A | Difficulty: N/A | Category: Usability]

When the operator creates a validator, he provides an intended exit date. It should not be possible for the operator to accidentally signal a voluntary exit message before the date has passed. However, the implementation of the `endOperatorServices` function lacks a proper check for that condition. Note that it is still possible for the operator to exit earlier intentionally by first changing the intended exit date.

Status

Fixed in 3df5f54b0b11bfe1b6c71fa92cd301e0cc637e01.

B03: Dependency between salt and operatorDataCommitment

[Severity: N/A | Difficulty: N/A | Category: Usability]

The parameters of `StakefishServicesContractFactory.deployContract` are not independent: The `operatorDataCommitment` depends on the `saltValue`, because the `salt` is used to compute the deterministic address of the about-to-be deployed contract, and the `operatorDataCommitment` depends on the address. Consequently, the function caller must ensure that the `operatorDataCommitment` he is passing in is actually computed from the `salt` he is passing in. If he fails to meet this requirement, he will deploy a dysfunctional contract that will never be able to create a validator because the validation of the `operatorDataCommitment` always fails.

Scenario

1. Alice wants to deploy a new service contract with the intention to create a validator once the necessary 32Eth has been deposited into the service contract.
2. Alice calls the deployContract function with an operatorDataCommitment that is not computed correctly from the given salt.
3. The service contract deposits reached the threshold of 32Eth.
4. Alice wants to create the validator node, but the createValidator method keeps failing because the commitment-hash cannot be validated.
5. Alice notices her mistake in step 2 and notifies the stakeholders about the unfortunate situation.
6. Depositors can withdraw their deposits. No funds are permanently lost; only gas costs cannot be recovered.

Recommendation

Document the dependency between the parameters and provide users with the necessary information and instructions on computing the operatorDataCommitment from the salt.

Status

Acknowledged by the client.

B04: Exit date change after user's verification

[Severity: N/A | Difficulty: N/A | Category: Informational]

The service contract uses a commitment scheme to maintain parameters that the operator uses during validator creation. The intention of the commitment scheme is to save gas costs: Instead of storing the parameter on-chain, the operator provides a data commitment on initialization and only reveals the actual parameters on validator creation. The service contract validates the revealed parameters against the commitment right before the validator is created. The parameters are made available via the frontend to potential users to validate the commitment on their own before investing in the contract. One particular parameter used in this commitment scheme is the intended exit date of the validator. However, the operator can change the exit date to an earlier date even after the validator has been created and users have verified it. This is indeed intentional: The exitDate is pessimistically set to a date far in the future to lower the risk of A02. Once the operator confirms that the service contract has been paid its collateral and staking rewards, he can opt to enter the withdrawal phase earlier than the originally provided pessimistic exit date. This way, deposits are not locked in longer than strictly necessary.

Scenario

1. Alice considers investing in a service contract.
2. She discovers a service contract to her liking and validates that the operator correctly committed the exit date.
3. Alice deposits some Eth in the service contract.
4. Once the service contract has reached the threshold of 32, the operator creates a validator node.
5. The validator signals its voluntary exit.
6. The operator notices that the collateral and staking rewards have been paid back to the service contract.
7. The operator intentionally sets the exit date to an earlier date and advances to the withdrawal phase so that depositors can withdraw their fair shares.
8. Alice is surprised that the exit date is earlier than the exit, which she validated.
9. Although Alice didn't lose any money, she is confused by the early exit and loses trust in the StakefishServiceContract.

Recommendation

We recommend that the above behavior be adequately documented to avoid users being surprised or confused when the exit date changes even after validating the original exit date.

Status

Acknowledged by the client.

B05: Input validation for commission rate

[Severity: N/A | Difficulty: N/A | Category: Input Validation]

It is recommended to check the validity of `_commissionRate` in the StakefishServicesContract.initialize() function as follows:

```
require(_commissionRate <= COMMISSION_RATE_SCALE);
```

Discussion

Although it is checked in the factory contract, it is still better practice to add the same check in the service contract in the case that a service contract instance is deployed without going through the factory.

Status

Fixed in 3df5f54b0b11bfe1b6c71fa92cd301e0cc637e01.

B06: Recycling unused storage slots in ERC721 wrappers

[Severity: N/A | Difficulty: N/A | Category: Best Practice]

Both `_servicesContracts[tokenId]` and `_deposits[tokenId]` are *not* deleted in StakefishERC721Wrapper.redeemTo(), while both `_owners[tokenId]` and `_tokenApprovals[tokenId]` are cleared. It is a better defensive programming practice to recycle unused storage slots, to make storage corruption attacks harder.

Scenario

Suppose a token X had been redeemed in the ERC721 wrapper. Suppose Alice somehow manages to corrupt `_owners[X]` to be set to an account in her control. Then, she can redeem the token X again because both `_servicesContracts[X]` and `_deposits[X]` still hold the previous data.

Discussion

We admit that storage slots corruption is very hard, but note that the recycling practice will make such attacks much harder because it will require adversaries to corrupt *three* storage slots rather than a *single* slot otherwise.

Recommendation

Delete both `_servicesContracts[tokenId]` and `_deposits[tokenId]` after the token transfer in StakefishERC721Wrapper.redeemTo().

Status

Fixed in 3df5f54b0b11bfe1b6c71fa92cd301e0cc637e01.

B07: Input validation for StakefishERC721Wrapper.approve()

[Severity: N/A | Difficulty: N/A | Category: Input Validation]

It can be considered to have an extra check for the existence of `_owners[tokenId]` in StakefishERC721Wrapper.approve().

Discussion

Although there exist *no* logical paths that can pass the following second require clause in case of `_owners[tokenId] == 0`, provided that the `_operatorApprovals` mapping is *never* compromised:[1]

```
	require(
		msg.sender == owner || isApprovedForAll(owner, msg.sender),
		"Not owner nor approved for all"
	);
```

It is still a better defensive programming practice to have an explicit check and revert earlier in case of unexpected catastrophic failures due to hidden bugs.

Recommendation

Add “`require(_owners[tokenId] != address(0))`” in StakefishERC721Wrapper.approve().

Status

Acknowledged by the client.

[1] For example, if the `_operatorApprovals` mapping is somehow corrupted, where `_operatorApprovals[0][msg.sender]` is set *non-zero*, then the second require clause could *not* detect the non-existence.

B08: Missing setters or immutable annotations in StakefishServicesContractFactory

[Severity: N/A | Difficulty: N/A | Category: Best Practice]

In the StakefishServicesContractFactory contract, the `_servicesContractImpl` variable cannot be updated later after initialization, while it is *not* declared `immutable`.

Recommendation

Add a setter, ***or*** the `immutable` annotation, for `_servicesContractImpl`.

NOTE: the `_servicesContractImpl` variable *cannot* be declared as `immutable` if it is read in the constructor() to adopt the recommendation of A05.

Status

The client confirmed that the variable is immutable.

B09: Griefing attacks for contract creations in StakefishServicesContractFactory

[Severity: N/A | Difficulty: N/A | Category: Security]

Salt values are revealed in the createContract() or createMultipleContracts() calls, which can be exploited for griefing attacks.

Scenario

1. Alice submits a transaction that calls createContract() or createMultipleContracts().
2. Seeing Alice's transaction in the mempool, Bob submits another transaction with the same salt but dummy invalid operatorDataCommitment(s) and zero ether, at a higher gas price.
3. If Bob's transaction is processed before Alice's, then Alice's transaction will fail.
4. However, Alice may *not* notice the failure because a ContractCreated event is emitted with the given salt value, although it comes from Bob's transaction.
5. Without noticing the failure of her legitimate transaction, Alice may proceed with receiving deposits from users, and later she will fail to create a validator because the commitment is invalid. At this point, Alice has to ask users to withdraw their funds and deposit again into another service contract, which can be annoying, and users may leave.

Recommendation

Consider adding the `onlyOperator` modifier for both createContract() and createMultipleContracts().

Status

Acknowledged by the client.

B10: Potential reentrancy vulnerability in ERC721 wrappers

[Severity: N/A | Difficulty: N/A | Category: Security]

In the mintTo() function of the ERC721 wrapper, the `_safeMint()` call is made before the transferDepositFrom() call, where a user-provided callback is executed inside `_safeMint()`, which may introduce a potential attack surface of reentrancy. It is a better defensive programming practice to take collateral before minting tokens to avoid potential exploits.

Recommendation

In StakefishERC721Wrapper.mintTo(), move the `_safeMint()` call after the transferDepositFrom() call, to enforce the deposit transfer to be made before minting tokens. (Similarly, consider moving down the _mint() call in StakefishERC20Wrapper.mintTo(), to follow the same pattern, although it is not directly related to this finding.)

Also, clearly document the security implications of ERC721 callbacks to inform potential users of the ERC721 wrappers.

Status

Fixed in 3df5f54b0b11bfe1b6c71fa92cd301e0cc637e01.