CHAINLINK

Chainlink Smart Contract Security Review

Version: 2.0

December, 2018

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the current implementation of the Chainlink platform. This review focuses on two library-like contracts (`Chainlinked` and `ChainlinkLib`) and an `Oracle` contract.

The combination of these three contracts allow users to deploy contracts which can request external data from oracles and pay for the service in LINK tokens.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review regarding, the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of Chainlink platform at the time of this review and of the contracts within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an open/closed status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as "informational". Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities found within the Chainlink contracts under review.

Overview

The Chainlink contracts can be logically separated into two parts. The Oracle entity (encapsulated in `Oracle.sol`) and the Chainlink "libraries" (contained in `ChainlinkLib.sol` and `Chainlinked.sol`).

An Oracle is an entity in the platform that fulfils requests for off-chain data. Contracts on the platform auction Link tokens to request this data. The tokens are transferred to the Oracle contract, which emits an event to alert the Oracle a new request has been made. The contract to which the Oracle should send its off-chain data when fulfilling a request may cancel the request after 5 minutes of the request being made if the Oracle has been unresponsive. The `owner` of the Oracle contract can fulfil the request and claim the Link tokens.

The Chainlink library contracts provide a set of Solidity helper functions which users of the platform can inherit in their contracts. These functions help construct and correctly encode requests to get sent to the user's chosen Oracle.

The platform is currently in the early stages of its overall design and development, as outlined in the Chainlink whitepaper [1].

Audit Summary

This review was initially conducted on commit cee3568, which contains the three contracts that are the focus of this review, specifically `ChainlinkLib.sol`, `Chainlinked.sol`, and `Oracle.sol`. There are a number of auxiliary contracts, which are outside the scope of this review. The complete list of contracts contained in the repository are:

```
contracts
├── Chainlinked.sol
├── ChainlinkLib.sol
├── Coordinator.sol
├── ENSResolver.sol
├── examples
│   ├── BasicConsumer.sol
│   ├── ConcreteChainlinked.sol
│   ├── ConcreteChainlinkLib.sol
│   ├── Consumer.sol
│   ├── GetterSetter.sol
│   ├── MaliciousChainlinked.sol
│   ├── MaliciousChainlinkLib.sol
│   ├── MaliciousConsumer.sol
│   ├── MaliciousRequester.sol
│   └── UpdatableConsumer.sol
├── interfaces
│   ├── ENSInterface.sol
│   ├── LinkTokenInterface.sol
│   └── OracleInterface.sol
├── Migrations.sol
└── Oracle.sol
```

The final version of this review targets commit 9adafeb.

To support this review, the testing team used the following automated testing tools:

- Rattle: `https://github.com/trailofbits/rattle`
- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output from these automated tools have been omitted from this report, but are available upon request.

All vulnerabilities identified during this assessment have been addressed by the development team.

Per-Contract Vulnerability Summary

Oracle (`Oracle.sol`)

A range of vulnerabilities with various severities were discovered in this contract. The vulnerabilities include stealing tokens from the Oracle contract, hijacking requests from legitimate Chainlink contracts, and the possibility of losing LINK tokens due to duplicate `internalId` entries. Some informational issues were also raised in relation to this contract.

Chainlinked (`Chainlinked.sol`)

Two low severity issues were discovered in this contract. Two modifiers of this contract will not function as expected if the callback contract is not the contract that made the request.

ChainlinkLib (`ChainlinkLib.sol`)

No potential vulnerabilities have been identified.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Chainlink contracts that were examined. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
LNK-01	Users Can Form Malicious Requests to Steal Tokens From Oracles	High	Resolved
LNK-02	Malicious Users Can DOS/Hijack Requests From Chainlinked Contracts	High	Resolved
LNK-03	Oracles Can Claim Token Payments Without Processing the Request Callback	Medium	Resolved
LNK-04	Oracles' `internalId` Can Be Duplicated, Leading to Unrecoverable Link Tokens	Medium	Resolved
LNK-05	Oracles Can Accept Invalid Requests, Constructed with Small `_data`	Low	Resolved
LNK-06	The `checkChainlinkFulfillment` Modifier Will Revert for All Requests Whose Callback is not the Calling Contract	Low	Resolved
LNK-07	`cancelChainlinkRequest` Reverts for All Requests whose Callback Address is not the Calling Contract	Low	Resolved
LNK-08	Token Transfer Function Call is not Checked For Success	Low	Resolved
LNK-09	Calldata Modifier is Only Valid if the `calldata` is not Given by a User	Informational	Resolved
LNK-10	Miscellaneous General Comments and Suggestions	Informational	Resolved

LNK-01	Users Can Form Malicious Requests to Steal Tokens From Oracles		
Asset	Oracle.sol		
Status	**Resolved:** In commit 4d968bf		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Malicious users can steal tokens from Oracles by crafting and submitting specific requests.

When the `owner` of the Oracle fulfils a request, it performs an external call on line [118] which is intended to supply the off-chain data to the calling contract. The current implementation is such that a user can manufacture any arbitrary address that the Oracle will call when fulfilling a request, along with any function signature and initial parameter. Thus a user can craft a malicious request to the Link token contract, which when fulfilled, will transfer Link tokens from the Oracle to the attacker.

To detail this further, the call on line [118] is:

```
return callback.addr.call(callback.functionId, callback.externalId, _data);
```

A malicious user can create a request, which sets `callback.addr` to the address of the Link token contract address. They can set `callback.functionId` to the function signature of `transfer(address,uint)` and they can set `callback.externalId` to be a beneficiary address of their choosing. These variables are set when creating a request. The request's `data` can be set to ask for a specific value (which is less than or equal to the available Link token balance of the Oracle, i.e. the current price of ether in wei). If an Oracle fulfils such a request and calls `fulfillData()` (line [101]) with `_data` being a value, i.e. `200e18` (and the Oracle has more than 200 Link tokens), the call on line [118] will send these tokens to the attacker's beneficiary address.

An example of this attack is given in the tests that accompany this report, specifically test `test_attack_can_steal_oracle_tokens`.

Note: This attack is not localised to just Link tokens but to all contracts and functions that do not require more than two parameters. See LNK-02 for a further example.

Recommendations

There are a number of ways to address this issue, some more restrictive (in platform functionality) than others. The least restrictive is to blacklist addresses that `callback.addr` can take. For example, adding a require statement that ensures `callback.addr` is not the Link token address would be one approach. Although this resolves the immediate issue, it is typically dangerous to allow external users to set arbitrary addresses and function calls for a low level call as LNK-02 illustrates.

A more restrictive approach would be to only allow callback addresses from the contracts that created the request. This solution would also address LNK-02, LNK-07, and LNK-06. This however may be too restrictive for the purposes of the Chainlink platform.

Resolution

The callback address is now restricted from being the `LINK` address.

LNK-02	Malicious Users Can DOS/Hijack Requests From Chainlinked Contracts		
Asset	Chainlinked.sol and Oracle.sol		
Status	**Resolved:** In commit a741f6b		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Malicious users can hijack or perform Denial of Service (DOS) attacks on requests of `Chainlinked` contracts by replicating or front-running [2] legitimate requests.

The `Chainlinked` (`Chainlinked.sol`) contract contains the `checkChainlinkFulfillment()` modifier on line [145]. This modifier is demonstrated in the examples that come with the repository. In these examples this modifier is used within the functions which contracts implement that will be called by the `Oracle` when fulfilling requests. It requires that the caller of the function be the `Oracle` that corresponds to the request that is being fulfilled. Thus, requests from `Chainlinked` contracts are expected to only be fulfilled by the `Oracle` that they have requested. However, because a request can specify an arbitrary callback address, a malicious user can also place a request where the callback address is a target `Chainlinked` contract. If this malicious request gets fulfilled first (which can ask for incorrect or malicious results), the `Oracle` will call the legitimate contract and fulfil it with incorrect or malicious results. Because the known requests of a `Chainlinked` contract gets deleted (see line [147]), the legitimate request will fail.

It could be such that the `Oracle` fulfils requests in the order in which they are received. In such cases, the malicious user could simply front-run the requests to be higher in the queue.

We further highlight this issue with a trivial example. Consider a simple contract that is using Chainlink to estimate the price of Ether relative to a random token, which users then purchase based off this price. A malicious user could front-run the price request, putting their own request in with a malicious price source that is significantly lower than the actual price. The callback address of the malicious request would be the simple contract, and once fulfilled, would set the price of the simple contract to the malicious one given in the source.

An example of this attack is given in the test: `test_attack_can_hijack_request` that accompanies this report.

Recommendations

This issue is related to LNK-01, LNK-07, and LNK-06 in that it arises due to the fact that any request can specify its own arbitrary callback address. A restrictive solution would be the same as given in LNK-01, where callback addresses are localised to the requester themselves.

A less restrictive approach may be to include `msg.sender` in the `callbacks` mapping in `Oracle.sol`. Then, when fulfilling the request, the `Oracle` could send the `msg.sender` as an extra parameter. The `checkChainlinkFulfillment()` modifier can then accept or reject the fulfilment based on the original requester, preventing malicious requests from being fulfilled.

Resolution

The `internalId` within an `Oracle` has been modified to `requestId` which is a Keccak hash of the sender with the sender's `nonce`. This same id is used within the `Chainlinked` contracts which is required to fulfil a request. Thus a malicious user can no longer hijack requests, as doing so would require the malicious request to be sent from the `Chainlinked` contract in order to generate an equivalent `requestId`.

LNK-03	Oracles Can Claim Token Payments Without Processing the Request Callback		
Asset	Oracle.sol		
Status	**Resolved:** In commit 27e126f		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

An Oracle can "fulfil" a request without updating state or performing useful calls to the requester. This is due to the lack of gas stipend in the external call made on line [118].

This means an `Oracle` when submitting a fulfilment request can choose a gas amount which is only enough to perform the operations of changing its own state (that is, lines [110] through [114] and approx `52000` gas) and leave little for the external call (the `delete` on line [113] provides a gas refund meaning some gas will be left for the call). In such a scenario, the external call will fail, however the current transaction will pass, which allows Oracles to withdraw the tokens for the given request.

See the test `test_attack_oracle_fulfill_no_callback` for an example.

It should be noted, that even if the `Oracle` is assumed to not be malicious, it could still be possible that `estimateGas` may incorrectly determine the gas to be sent in the external call. Thus an `Oracle` can mistakenly be paid without fulfilling the request and as the `callbacks` mapping is deleted, cannot resend the request.

If an `Oracle`, blindly follows `estimateGas`, then this leaves an `Oracle` vulnerable to malicious requests which consume large amounts of gas, such as creating Gas Tokens [3].

Recommendations

One solution would be to require Oracles to dedicate at least some fixed amount of gas for updating the callback contract. This amount is dependent on how much state/operations are expected for implementations of such callback functions in the Chainlink platform.

There are a number of ways this could be implemented in the current code. One such way would be to implement a lower bound on the gas required to send for a callback. This could be done by adding a require statement that ensures the `Oracle` has dedicated at least some constant (here `gasConstant`) amount of gas for the callback contract. For example, adding the following just above line [118]:

```
require(gasleft() >= gasConstant);
```

Alternatively, an upper-bound on gas sent could be set by specifying a fixed amount in the external call itself. This solution would also prevent malicious callback requests from consuming large amounts of gas.

Resolution

A minimum gas limit of `400000` was introduced. This forces `Oracles` to supply at least `400000` gas when fulfilling a request. For malicious contracts trying to exploit excess Oracle gas, it is up to the Oracle to decide whether or not to pay for any extra gas beyond the `400000`, which they can do when performing the transaction.

LNK-04	Oracles' `internalId` Can Be Duplicated, Leading to Unrecoverable Link Tokens		
Asset	Oracle.sol		
Status	**Resolved:** In commit 48dde7e		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `internalId` is constructed from the keccak hash of `msg.sender` and `_externalId`[1] (which is an arbitrary value sent by the caller). Although the `Chainlinked` contracts increment this value, it is possible for contracts to implement their own versions or call the Oracle directly (via the Link token). In such cases, a repeated `_externalId` from a contract/address will lead to a duplication of `internalId` in the Oracle contract (one example would be an implementation that accidentally sends 0 as the `externalId` many times, or repeats an id).

In such cases, the Link tokens that were sent in the previous request (of the same `internalId`) become unrecoverable as the `callbacks` mapping entry gets overridden. This means a user cannot cancel the original request, nor can the Oracle fulfil the original request.

As the `withdraw()` function can only withdraw `withdrawableWei` amount of tokens, the original tokens sent to the contract are also non-withdrawable.

Please refer to the `test_duplicate_id_lost_tokens` test that accompanies this report for an exploitation example.

Recommendations

This vulnerability can be mitigated by ensuring that an `internalId` does not exist before entering it into the `callbacks` mapping. More specifically, adding a require after line [76] which ensures that the `internalId` is not currently in use, will prevent overwriting the `callbacks` mapping.

Alternatively, a nonce could be introduced in the generation of `internalId`, however this would require modifying the logic of the `cancel()` function to accommodate duplicate `externalId` entries.

Resolution

A require statement has been added on line [86] which ensures there is no current entry in the `callbacks` mapping, preventing duplication.

[1]See lines [76] and [122].

LNK-05	Oracles Can Accept Invalid Requests, Constructed with Small `_data`		
Asset	Oracle.sol		
Status	**Resolved:** In commit afe8509		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The current `Oracle.sol` implementation does not check the length of `_data` in `onTokenTransfer` on line [51].

This function uses assembly to add the `address` and `amount` to the `_data` variable before calling `delegatecall` on `_data`. If the length of data is less than 64 bytes, the assembly still succeeds and the variables to be inserted into `_data` can be truncated (depending on the length of `_data`). For incorrect lengths of `_data` greater than 24 bytes, the `delegatecall` will revert due to incorrect encoding of the `requestData()` parameters. However, the `onTokenTransfer()` function succeeds for `_data` values of lengths 24 bytes and under.

In these cases, Link tokens successfully get sent to the contract and the `sender` and `amount` variables are set to 0 (regardless of how many tokens are actually sent). This scenario leads to locking of Link tokens, as even if the `Oracle` fulfils the request, the `amount` is 0 and so `withdrawableWei` does not get incremented. This means the `Oracle` cannot claim these locked tokens.

An example of this scenario is given in the test: `test_on_token_transfer_data_length` that accompanies this report.

Recommendations

The `_data` variable should be checked to ensure that its length is at least as long as expected. Short lengths can lead to unintended manipulations of the `sender` and `amount` variables as they are injected through assembly under the assumption that `_data` is long enough to contain them.

Resolution

A modifier was created, `validRequestLength()` which is used on line [58]. This modifier ensures the length of `_data` is at least as long as the minmum length of the expected arguments it requires in ensuing `delegatecall` on line [68].

LNK-06	The `checkChainlinkFulfillment` Modifier Will Revert for All Requests Whose Callback is not the Calling Contract		
Asset	Chainlinked.sol		
Status	**Resolved:** In commit 0e8c523		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A `Chainlinked` contract stores a mapping of `unfulfilledRequests`. The modifier `cancelChainlinkRequest()` checks this mapping to ensure the `msg.sender` is the `Oracle` of the request. This only functions correctly if the request was made from the current contract.

Consider two `Chainlinked` contracts, one as the requester and one as the callback contract to receive fulfilled requests. The requester's `unfulfilledRequests` mapping will store the requests made, however the callback contract's mapping will not. If the callback contract were to utilise the `checkChainlinkFulfillment` modifier, requests will always revert due to the mapping being empty. (We assume the callback contract is not also sending requests of its own.)

Recommendations

This issue is related to LNK-01, LNK-02, and LNK-07 in that it arises from the general ability to specify arbitrary contracts to fulfil the data.

A restrictive approach would be to enforce that the contract making the request be the callback contract, as discussed in LNK-01.

As long as this modifier is not used in this scenario (which is fundamentally an implementation issue), there are no immediate related security risks. Thus, this issue could be acknowledged and we recommend adding comments that indicate that the `checkChainlinkFulfillment` modifier is only to be used for contracts that are both the requester and the callback contract.

Resolution

A new function, `addExternalRequest()` on line [94] has been added which allows `Chainlinked` contracts to add external requests, allowing for other contracts to make requests on their behalf.

LNK-07	`cancelChainlinkRequest` Reverts for All Requests whose Callback Address is not the Calling Contract		
Asset	Chainlinked.sol		
Status	**Resolved:** In commit 0e8c523		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `Chainlinked` contract provides a number of helper functions designed to be inherited by contracts using the Chainlink platform. One of these functions, `cancelChainlinkRequest()` (see line [54]) will only operate as expected if the `_callbackAddress` of the request is the current contract. This is due to the `require` on line [123] of `Oracle.sol`.

Thus, if requests are made with callback addresses that have not explicitly implemented a `cancelChainLinkRequest()` function, these requests will be impossible to cancel.

We acknowledge that the purpose of this may be to make the requests generic, i.e. the callback address may be another contract which inherits the `Chainlinked` contract, however due to LNK-06 this kind of setup will also revert.

A noteworthy point is that this structure gives a non-intuitive return of tokens. By this we mean, the contract that paid for the request, does not necessarily get refunded if the request gets cancelled. Only the callback contract can initiate the cancel and is the one that will receive the cancelled tokens.

We raise this issue primarily to ensure that it is known by the authors.

Recommendations

This issue is related to LNK-01, LNK-02, and LNK-06 in that it arises from the general ability to specify arbitrary contracts to fulfil the data.

A restrictive approach would be to enforce that the contract making the request be the contract that can cancel the request.

This may be too restrictive, in which case an `or` statement could be added to the require on line [54] of `Oracle.sol` which would allow the caller to also cancel the request. This would also involve adding another field to the `Callback` struct, which would also store the `msg.sender` of the request.

Alternatively this may be a known issue and no modification may be desired.

Resolution

As in LNK-06, a new function, `addExternalRequest()` on line [94] has been added which allows `Chainlinked` contracts to add external requests, allowing for other contracts to make requests on their behalf.

LNK-08	Token Transfer Function Call is not Checked For Success		
Asset	Oracle.sol		
Status	**Resolved:** In commit 251e5a8		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `withdraw()` function on line [129] does not check for the success of the Link transfer on line [135]. Although the current implementation of Link tokens revert instead of returning `false`, other ERC20 implementations / future implementations of Link tokens may return false. In such cases, Link tokens will become unrecoverable as, `withdrawableWei` is reduced in this function without verifying that the tokens were in fact transferred.

We raise this issue as we assume the author is anticipating this scenario as the other Link transfer call is checked within this contract (see line [148]).

Recommendations

Wrap the transfer on line [135] in a require.

Resolution

The transfer call has been wrapped in a require statement.

LNK-09	Calldata Modifier is Only Valid if the `calldata` is not Given by a User
Asset	Oracle.sol
Status	**Resolved:** In commit bafa91c
Rating	Informational

Description

The functionality in the `permittedFunctionsForLINK()` modifier (on line [175]) only restricts the function signature if the function it protects is called by a contract that doesn't allow a user to enter the `calldata`. The current Link token is an example. If a function with this modifier is allowed to be called by any external actor, the modifier can be bypassed, simply by modifying the `calldata`.

If this modifier is not used in conjunction with a modifier such as `onlyLINK()` (and the `LINK` contract does not allow users to modify `calldata`) it will be ineffective and thus can potentially introduce unexpected behaviours in future versions, or if the functionality of the Link token changes.

The modifier is vulnerable as a user can construct `calldata` in such a way that satisfies the `require` in `permittedFunctionsForLink()` whilst giving arbitrary values to `_data` in the `onTokenTransfer()` function. The modifier checks that the 4 bytes at position 132 in the calldata are the required function selector. For typical calldata, this position corresponds to the first 4 bytes of the dynamic `_data` variable. This can be seen by looking at `calldata` for a conventional call to the `onTokenTransfer()` function, which will be of the form:

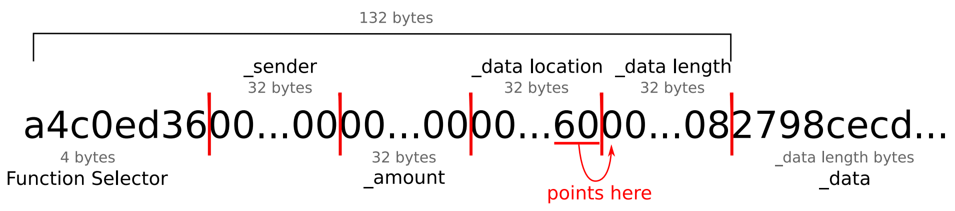

Figure 1: Standard encoded calldata

With the standard `calldata`, the modifier checks the byte position at 132 to ensure that `_data` has the correct function selector. Notice here, that `2798cecd` represents the `requestData()` function selector. The `a4c0ed36` represents the `onTokenTransfer()` function selector. In this case, the modifier would function as expected.

However, one could equally send the following calldata (if the user were able to send or modify the calldata):

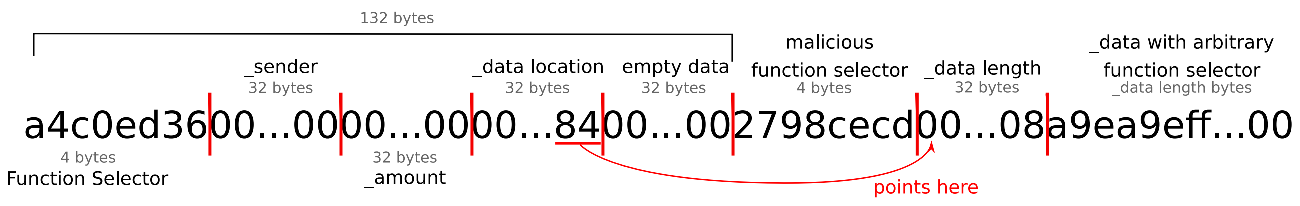

Figure 2: Maliciously encoded calldata

Notice that in this example, that the encoding for the dynamic sized `_data` variable has been abused. We have modified the location to point further away in the calldata, and left the `2798cecd` function selector at position 132, thus satisfying the `require` of the modifier. With this `calldata`, the `onTokenTransfer()` function will read `_data` as `a9ea9eff...` and thus allow us to call any function we choose, bypassing the intended modifier.

This issue appears to not be exploitable in the current context, assuming the Link token does not allow users to modify the `calldata` when calling this function. We raise this issue as informational as something to be aware of in other contracts or in future upgrades.

Recommendations

This issue can be resolved by checking the first 4 bytes of the `_data` variable itself, rather than checking a specific location in calldata (which can be externally manipulated).

More concretely, we suggest using a modifier of the form:

```
modifier permittedFunctionsForLINK(bytes _data) {
    bytes4 funcSelector;
    assembly {
        funcSelector := mload(add(_data, 32))
    }
    require(funcSelector == permittedFunc, "Must use whitelisted functions");
}
```

Resolution

The `permittedFunctionsForLINK` modifier was adjusted to that of the recommendation.

LNK-10	Miscellaneous General Comments and Suggestions
Asset	Oracle.sol
Status	**Resolved:** See inline comments
Rating	Informational

Description

This section outlines miscellaneous comments and suggestions that were found as a by-product of this review. We include this section as it may be useful for the authors to improve readability of the code.

- line [23] - `withdrawableWei` represents the amount of Link tokens that can be withdrawn by the Oracle. Something akin to `withdrawableTokens` may be a better suited name for this variable.
  ✓ Resolved in commit [f975f16]
- line [82] - There is a constant, hard-coded expiry time in the `Oracle` contract. It may benefit users to make this a public constant variable, such that users can lookup the contract to know when they are able to cancel their requests for this oracle. This will also aid readability of the code. For example, `uint constant public EXPIRY_TIME = 5 minutes`.
  ✓ Resolved in commit [51c063d]
- Constants could be highlighted by naming with capitals. Specifically, on line [22] in `Oracle.sol`, `oneForConsistentGasCost` could be set to `ONE_FOR_CONSISTENT_GAS_COST`. Similarly, the `permittedFunc` constant on line [148] could be moved to the top of the contract and given capitals, such as `PERMITTED_FUNC`. There are also four (4) constants in `Chainlinked.sol` which could be changed to comply with this naming convention.
  ✓ Resolved in commit [65c28a5]
- The `public` functions in `Oracle.sol` are better suited to the `external` visibility modifier (gas saving), with the exception of the `onTokenTransfer()`.
  ✓ Resolved in commit [ba9a8be] and [c06456b]
- line [94] - It was noticed that Oracles are limited to 32 bytes in data that can be returned. It appears multiple requests would be required if a Chainlinked contract needed more than 32 bytes to express the result.
  ✓ Acknowledged and extensions are planned in future versions.

Recommendations

Ensure these are as expected.

Resolution

These recommendations have been implemented, see inline comments above.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `pytest` framework was used to perform these tests and the output is given below.

```
tests/test_attacks.py::test_attack_can_steal_oracle_tokens                        FAILED   [2%]
tests/test_attacks.py::test_attack_can_hijack_request                             FAILED   [4%]
tests/test_attacks.py::test_attack_oracle_fulfill_no_callback                     FAILED   [6%]
tests/test_chainlinked.py::test_new_run[A9E-self-fulfillRequest(bytes32,bytes32)] PASSED   [9%]
tests/test_chainlinked.py::test_new_run[A9E-self-invalid_callback_selector]       PASSED  [11%]
tests/test_chainlinked.py::test_new_run[A9E-self-]                                PASSED  [13%]
tests/test_chainlinked.py::test_new_run[A9E-other-fulfillRequest(bytes32,bytes32)]PASSED  [16%]
tests/test_chainlinked.py::test_new_run[N/A-self-fulfillRequest(bytes32,bytes32)] PASSED  [18%]
tests/test_chainlinked.py::test_make_request[128-500-True]                        PASSED  [20%]
tests/test_chainlinked.py::test_make_request[128-0-True]                          PASSED  [23%]
tests/test_chainlinked.py::test_make_request[0-500-True]                          PASSED  [25%]
tests/test_chainlinked.py::test_make_request[128-10001-False]                     PASSED  [27%]
tests/test_chainlinked.py::test_consumer_can_make_multiple_requests               PASSED  [30%]
tests/test_chainlinked.py::test_can_cancel_request                                PASSED  [32%]
tests/test_chainlinked.py::test_cannot_cancel_if_different_callback_addr          PASSED  [34%]
tests/test_chainlinked.py::test_can_fulfill_request                               PASSED  [37%]
tests/test_chainlinked.py::test_cannot_fulfill_request_if_different_callback_addr PASSED  [39%]
tests/test_chainlinklib.py::test_adding_data_sizes[1]                             PASSED  [41%]
tests/test_chainlinklib.py::test_adding_data_sizes[2]                             PASSED  [44%]
tests/test_chainlinklib.py::test_adding_data_sizes[10]                            PASSED  [46%]
tests/test_chainlinklib.py::test_adding_data_sizes[127]                           PASSED  [48%]
tests/test_chainlinklib.py::test_adding_data_sizes[128]                           PASSED  [51%]
tests/test_chainlinklib.py::test_adding_data_sizes[129]                           PASSED  [53%]
tests/test_chainlinklib.py::test_adding_data_sizes[500]                           PASSED  [55%]
tests/test_chainlinklib.py::test_adding_data_sizes[1000]                          PASSED  [58%]
tests/test_chainlinklib.py::test_adding_string_array                              PASSED  [60%]
tests/test_chainlinklib.py::test_add_bytes                                        PASSED  [62%]
tests/test_chainlinklib.py::test_close_buffer                                     PASSED  [65%]
tests/test_deploy.py::test_deploy                                                 PASSED  [67%]
tests/test_oracle.py::test_on_token_transfer_data_length[1-False]                 PASSED  [69%]
tests/test_oracle.py::test_on_token_transfer_data_length[2-False]                 PASSED  [72%]
tests/test_oracle.py::test_on_token_transfer_data_length[12-False]                PASSED  [74%]
tests/test_oracle.py::test_on_token_transfer_data_length[13-False]                PASSED  [76%]
tests/test_oracle.py::test_on_token_transfer_data_length[256-False]               PASSED  [79%]
tests/test_oracle.py::test_on_token_transfer_data_length[288-True]                PASSED  [81%]
tests/test_oracle.py::test_on_token_transfer_data_length[320-True]                PASSED  [83%]
tests/test_oracle.py::test_on_token_transfer_data_length[352-True]                PASSED  [86%]
tests/test_oracle.py::test_can_duplicate_id                                       FAILED  [88%]
tests/test_oracle.py::test_cannot_DOS_requests[450000.0]                          PASSED  [90%]
tests/test_oracle.py::test_cannot_DOS_requests[500000.0]                          PASSED  [93%]
tests/test_oracle.py::test_cannot_DOS_requests[1000000.0]                         PASSED  [95%]
tests/test_oracle.py::test_cannot_DOS_requests[2000000.0]                         PASSED  [97%]
tests/test_oracle.py::test_cannot_DOS_requests[5000000.0]                         PASSED [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Ari Juels Steve Ellis and Sergey Nazarov. Chainlink: A Decentralized Oracle Network, September 2017, Available: https://link.smartcontract.com/whitepaper.

[2] Sigma Prime. Solidity Security - Front Running. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html#race-conditions. [Accessed 2018].

[3] Gas Token. Cheaper Ethereum Transactions Today, Available: https://gastoken.io/. [Accessed 2018].