chainlink

This smart contract audit was prepared by Quantstamp, the protocol for securing smart contracts.

Executive Summary

Category	Description
Type	Protocol Audit
Auditor(s)	Kacper Bąk, Senior Research Engineer John Bender, Senior Research Engineer Martin Derka, Senior Research Engineer Nadir Akhtar, Software Auditing Intern
Timeline	2018-10-16 through 2018-11-07
Language(s)	Solidity, Javascript
Method(s)	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification(s)	Whitepaper: https://link.smartcontract.com/whitepaper Chainlink Solidity Properties: https://docs.google.com/document/d/1M74I25gzw_9WZHHuOUb7IY8kOSbUuOk64X02hGUk0tY/edit?usp=sharing
Source code	Repository: chainlink Commit: bafa91c
Total Issues	13
High Risk Issues	0
Medium Risk Issues	3
Low Risk Issues	2
Informational Risk Issues	7
Undetermined Risk Issues	1

Severity Level	Explanation
High	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
Medium	The issue puts a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
Low	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate threat to continued operation or usage, but is relevant for security best practices, software engineering best practices, or defensive redundancy.
Undetermined	The impact of the issue is uncertain.

Goals

This report focused on evaluating security of smart contracts, as requested by the Chainlink team. Specific questions to answer:

- Can any race conditions occur?
- Can any participants lose their funds?
- Can any funds be locked up?

Changelog

- Date: 2018-11-01 - Initial report
- Date: 2018-11-05 - Excluded `Coordinator.sol` from the scope
- Date: 2018-11-07 - Reviewed report
- Date: 2019-02-28 - Reviewed report based on commits: `82b59d9`, `9073c9b`, `81398f1`, `f73e692`, `f9047cc`, `353e08c`, `f128921`, `5e1dc74`, `72df4cc`, `f37633c`, `ba9a8be`, and `bafa91c3`.

Overall Assessment

The contracts manage the decentralized oracles implementation by Chainlink. Quantstamp has found three important issues related to transaction ordering dependency. We also give a set of recommendations to ensure that the code conforms to the best practices and is less likely to break in the future. The Chainlink team addressed our feedback.

Quantstamp Audit Breakdown

Quantstamp's objective was to evaluate the chainlink repository for security-related issues, code quality, and adherence to specification and best practices. Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control

- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

The Quantstamp auditing process follows a routine series of steps:

1. Code review that includes the following:
    i. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
    ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
    iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
    i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
    ii. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Toolset

The below notes outline the setup and steps performed in the process of this audit.

Setup

Testing setup:

- Truffle v4.1.12
- Ganache v1.1.0
- solidity-coverage v0.5.8
- Oyente v1.2.5
- Mythril v0.2.7
- truffle-flattener v0.18.9
- MAIAN commit sha: ab387e1
- Securify

Assessment

Findings

Oracle value can change during fulfillment

Status: Fixed

Contract(s) affected: `Oracle.sol` , `Chainlinked.sol`

Severity: Medium

Description: If the oracle is updated between the calls of `chainlinkRequest()` and `cancelChainlinkRequest()`, the cancellation gets called on the updated oracle, resulting in a failed `require()` statement (line 123, `Oracle.sol`) and, consequently, the cancellation cannot take place.

Exploit Scenario:

1. Requester asks an oracle for external data (indirectly) via `Chainlinked.chainlinkRequest()`.
2. Chainlinked owner updates the oracle by calling (indirectly) `Chainlinked.setOracle()`.
3. Requester calls `Chainlinked.cancelChainlinkRequest()` passing their request Id.
4. The function `Chainlinked.cancelChainlinkRequest()` calls `cancel()` on the updated oracle.
5. The function `cancel()` fails because the updated oracle does not have the given request Id. Consequently, the requestor is unable to cancel the request and retrieve their funds.

Recommendation: We recommend to either: 1) freeze the value `oracle` while there are outstanding requests for it, or 2) add support for multiple oracles. The former requires a mechanism to refund/cancel outstanding requests upon the oracle change. A separate flag could also be included to prevent incoming requests to the oracle to allow the current oracle to fulfill its pending requests.

Transaction ordering dependence between fulfilling and cancelling the request

Status: Fixed

Contract(s) affected: `Chainlinked.sol`, `Oracle.sol`

Severity: Medium

Description: If the requester sees a data transaction submitted to the blockchain by the oracle after the cancel embargo, they can race to call `cancel()`. If the requester wins (or the miner intentionally includes the `cancel()` transaction first), the cancellation happens although the requester has seen the data. We classified this vulnerability severity level as medium, since the Chainlink team indicated that: 1) data itself is not the main service that Chainlink provides -- the attestation is; 2) oracles that consistently fulfill the data around or after the embargo are considered faulty, and 3) before fulfilling the data, offchain oracle software queries the onchain contract to check if the request has not been canceled. The severity level, however, partly depends on sensitivity of the data to be stored onchain. The more sensitive the data, the more severe the impact. The current assumption is that oracles shall not store sensitive data onchain.

Exploit Scenario:

1. Requester asks an oracle for external data (indirectly) via `Chainlink.chainlinkRequest()`.
2. Oracle owner fulfills the request by submitting an Ethereum transaction using `Oracle.fulfillData()` right before the cancel embargo passes.
3. The transaction makes it to the transaction pool and the requester sees the submitted data.
4. After the cancel embargo has passed, requester races to call `Oracle.cancel()` to cancel their request.
5. For the given request Id, both `Oracle.fulfillData()` and `Oracle.cancel()` end up in the same pool of transactions to be mined.
6. Due to race condition, `Oracle.cancel()` is included before `Oracle.fulfillData()`.
7. This allows the requester to see the data without paying for it.

Recommendation: Given the context provided by the Chainlink team, we have no further recommendations. However, if the oracle data were sensitive, a possible solution should disallow fulfilling the data after the cancel embargo has passed, since such a request may be considered expired. Furthermore, an oracle should not submit the data too close to the cancel embargo expiry, since it could happen that the fulfillment transaction gets mined after the cancel embargo, and the requester could see the data without paying for it. A proper (likely impractical) solution that would be fork-resistant, would allow cancellation requests only after a number of block confirmations on top of the expired cancel embargo. It is a common practice among exchanges to wait for 24 block confirmations (~6min).

Transaction ordering dependence upon oracle update

Status: Fixed

Contract(s) affected: `Chainlinked.sol`

Severity: Medium

Description: The system is architected so that the oracle can be updated in the contract `Chainlinked`. If the contract implementing `Chainlinked` does not guarantee the stability of the oracle, then an oracle update might race with a new `chainlinkRequest()` call. Consequently, the wrong oracle might get the query and the transfer.

Exploit Scenario:

1. Requester asks an oracle for external data (indirectly) via `Chainlinked.chainlinkRequest()`.
2. Chainlinked owner updates the oracle by calling (indirectly) `Chainlinked.setOracle()`.
3. Both transactions make it to the transaction pool.
4. Due to race condition, `Chainlinked.setOracle()` is included before `Chainlinked.chainlinkRequest()`.
5. This causes the requester to submit a request and payment to the wrong oracle.

Recommendation: There are multiple solutions to this issue. First, have a lock flag that is set before each oracle update, and unset afterward. When the flag is set, do not allow for any new requests (any pending requests must still be taken care of). Another solution is to make augment requests with an oracle address. If the address does not match the current value, the request can be rejected. Alternatively, provide support for multiple oracles.

`OracleInterface` is missing the function `requestData()`

Status: Fixed

Contract(s) affected: `OracleInterface.sol`, `Chainlinked.sol`, `ChainlinkLib.sol`

Severity: Low

Description: The interface `OracleInterface` has no function `requestData()`. `Chainlinked.sol` (line 49), however, relies on `encodeForOracle()` from `ChainlinkLib.sol`, hence requiring `OracleInterface` to implement `requestData()`.

Recommendation: Implement the function `requestData()` in `OracleInterface`.

`ChainlinkLib` implicitly depends on `Oracle`

Status: Fixed

Contract(s) affected: `Oracle.sol`, `ChainlinkLib.sol`

Severity: Low

Description: The function `encodeForOracle()` in the `ChainlinkLib` contract implicitly depends on the contract `Oracle` by assuming the existence of function `requestData()` (lines 6 and 37), although `Oracle` is not mentioned anywhere in `ChainlinkLib` dependencies. Consequently, evolution of `Oracle` may break `ChainlinkLib` if it does not get updated accordingly.

Recommendation: Extract `encodeForOracle()` from `ChainlinkLib` into `Chainlinked.sol` or into a separate contract. Furthermore, implement the function `requestData()` in `OracleInterface` if every oracle is supposed to have this function.

Ignoring the result of `transfer()`

Status: Fixed

Contract(s) affected: `Oracle.sol`

Severity: Informational

Description: The function `withdraw()` (line 116 of the contract `Oracle.sol`) ignores the result of `transfer()`. By the ERC20 standard, as well as the Link Token interface, the method returns a boolean and does not necessarily revert the transaction upon failure (See: EIP20). Yet, the balance is always subtracted from `withdrawableWei` (line 115). We note, however, that as long as the LINK token contract is used (as defined in the dependencies), `revert()` will get called by `SafeMath.sol` functions upon integer overflow/underflow.

Recommendation: The Quantstamp team verified that the implementation of the LINK token does not return false on transfer without actively reverting the transaction. Nevertheless, we recommend checking the result of `transfer()` in `Oracle.sol` using `require(LINK.transfer(_recipient, _amount));` as a best practice.

Deleting requests inside a modifier violates *checks-effects-interactions* pattern

Status: Fixed

Contract(s) affected: `Chainlinked.sol`

Severity: Informational

Description: The code of the contract `Chainlinked` deletes requests inside the modifier `checkChainlinkFulfillment` after they have been completed. It is not customary for function modifiers to have side effects. Furthermore, it violates the checks-effects-interactions pattern, which states that interactions with other contracts should happen at the very end of the function.

Recommendation: We recommend establishing a common interface for callback functions, and performing any side effects within a function that would invoke the callback. Also, we recommend performing the deletion before invoking any callback. If necessary, pass the request as a parameter to the callback.

Unlocked Pragma

Status: Fixed

Severity: Informational

Description: Every Solidity file specifies in the header the required Solidity compiler version number following the format `pragma solidity (^)0.4.*`. The caret (`^`) before the version number implies an unlocked pragma, meaning Solidity compiler in the specified version and above, hence the term "unlocked."

Recommendation: For consistency and to prevent unexpected behavior in the future, it is recommended to remove the caret to lock the file onto a specific Solidity version for all the contracts that are not meant to be reused.

Magic constants

Status: Fixed

Contract(s) affected: `Oracle.sol`, `ChainlinkLib.sol`

Severity: Informational

Description: Number constants (line 24 of `ChainlinkLib.sol`, and lines 55 and 57 of `Oracle.sol`) are present in the code without any further explanation.

Recommendation: Define named constants and use them throughout the code.

Code clone

Status: Fixed

Contract(s) affected: `Oracle.sol`, `ChainlinkLib.sol`

Severity: Informational

Description: `permittedFunc` in `Oracle.sol` is cloned from `ChainlinkLib.sol`.

Recommendation: Extract any code clones into separate contract(s) and reuse by composition or inheritance, as appropriate.

`Oracle` does not explicitly implement `OracleInterface`

Status: Fixed

Contract(s) affected: `Oracle.sol`, `OracleInterface.sol`

Severity: Informational

Description: The contract `Oracle` does not explicitly implement the interface `OracleInterface` even though the name, interface function signatures, and tests imply that it should.

Recommendation: Make the contract `Oracle` implement the interface `OracleInterface`.

Establish an interface for callbacks

Status: Fixed

Contract(s) affected: `Chainlinked.sol`, `Oracle.sol`

Severity: Informational

Description: On the one hand, the contract `Chainlinked` allows a downstream implementation to pass an arbitrary callback signature to `newRun()`. On the other hand, part of the callback signature is determined by the arguments passed to the callback in line 107 of `Oracle.sol`. Although arbitrary callback name, seemingly, allows for more flexibility, it: 1) makes it easier for authors of downstream code to introduce bugs by not conforming to interfaces, and 2) introduces implicit dependencies.

Recommendation: Define an interface for `Chainlinked.sol` with, at least, a callback function with the following signature: `fulfill(bytes32 _requestId, bytes32 _price)`. Make the contract `Chainlinked` implement that interface, but not the function. Furthermore, update the `Callback` struct in `Oracle.sol` so that instead of having `address addr`, it is of type of the newly defined Chainlinked interface. Finally, invoke the callback function `fulfill()` in line 107 of `Oracle.sol`. **Update**: The Chainlink team decided not to establish a standardized interface for callbacks in order to provide more flexibility for the developers.

`onTokenTransfer()` calls `requestData()` via `delegatecall()` instead of making an explicit call

Contract(s) affected: `Oracle.sol`

Severity: Undetermined

Description: The function `onTokenTransfer()` calls `requestData()` via `delegatecall()` instead of making an explicit call (line 60). The call to `requestData()` is enforced by the modifier `permittedFunctionsForLINK()`. The code appears overly complicated and indirect in expressing the intent, which may harm code's readability and may be a source of future security vulnerabilities.

Recommendation: Call `requestData()` explicitly and pass the explicit arguments from `_data`.

Test Results

Test Suite Results

```
Contract: BasicConsumer
  ✓ has a predictable gas price (95ms)
```

```
  #requestEthereumPrice
    without LINK
      ✓ reverts (89ms)
    with LINK
      ✓ triggers a log event in the Oracle contract (109ms)
      ✓ has a reasonable gas cost (82ms)
  #fulfillData
    ✓ records the data given to it by the oracle (47ms)
    ✓ logs the data given to it by the oracle (40ms)
    when the consumer does not recognize the request ID
      ✓ does not accept the data provided (39ms)
    when called by anyone other than the oracle contract
      ✓ does not accept the data provided
  #cancelRequest
    before 5 minutes
      ✓ cant cancel the request
    after 5 minutes
      ✓ can cancel the request (138ms)
  #withdrawLink
    ✓ transfers LINK out of the contract (53ms)

Contract: Chainlinked
  ✓ has a limited public interface

Contract: ConcreteChainlinkLib
  ✓ has a limited public interface (40ms)
  #close
    ✓ handles empty payloads (38ms)
  #add
    ✓ stores and logs keys and values (79ms)
    ✓ handles two entries (94ms)
  #addBytes
    ✓ stores and logs keys and values (67ms)
    ✓ handles two entries (96ms)
    ✓ handles strings (65ms)
  #addInt
    ✓ stores and logs keys and values (63ms)
    ✓ handles two entries (100ms)
  #addUint
    ✓ stores and logs keys and values (66ms)
    ✓ handles two entries (90ms)
  #addStringArray
    ✓ stores and logs keys and values (130ms)

Contract: ConcreteChainlinked
  #newRun
    ✓ forwards the information to the oracle contract through the link token (45ms)
  #chainlinkRequest(Run)
    ✓ emits an event from the contract showing the run ID (86ms)
  #chainlinkRequestFrom(Run)
    ✓ emits an event from the contract showing the run ID (78ms)
    ✓ emits an event on the target oracle contract (98ms)
    ✓ does not modify the stored oracle address (81ms)
  #cancelChainlinkRequest
    ✓ emits an event from the contract showing the run was cancelled
    ✓ throws if given a bogus event ID
  #checkChainlinkFulfillment(modifier)
    ✓ emits an event marking the request fulfilled (43ms)
  #completeChainlinkFulfillment(function)
    ✓ emits an event marking the request fulfilled (43ms)
  #chainlinkToken
    ✓ returns the Link Token address
  #LINK
    ✓ multiplies the value by a trillion (38ms)
    ✓ throws an error if overflowing
  #addExternalRequest
    ✓ allows the external request to be fulfilled (40ms)
    ✓ does not allow the same requestId to be used
```

```
Contract: Coordinator
  ✓ has a limited public interface (41ms)
  #getPackedArguments
    ✓ returns the following value, given these arguments
  #getId
    ✓ matches the ID generated by the oracle off-chain
  #initiateServiceAgreement
    with valid oracle signatures
      ✓ saves a service agreement struct from the parameters (59ms)
      ✓ returns the SAID (41ms)
      ✓ logs an event (57ms)
    with an invalid oracle signatures
      ✓ saves no service agreement struct, if signatures invalid
    Validation of service agreement deadlines
      ✓ Rejects a service agreement with an endAt date in the past
  #executeServiceAgreement
    when called through the LINK token with enough payment
      ✓ logs an event
    when called through the LINK token with not enough payment
      ✓ throws an error (41ms)
    when not called through the LINK token
      ✓ reverts
  #fulfillData
    cooperative consumer
      when called by a non-owner
        - raises an error
      when called by an owner
        ✓ raises an error if the request ID does not exist
        ✓ sets the value on the requested contract (53ms)
        ✓ does not allow a request to be fulfilled twice (49ms)
    with a malicious requester
      ✓ cannot cancel before the expiration (293ms)
      ✓ cannot call functions on the LINK token through callbacks
    with a malicious consumer
      fails during fulfillment
        - allows the oracle node to receive their payment
        ✓ can't fulfill the data again (43ms)
      calls selfdestruct
        - allows the oracle node to receive their payment
      request is canceled during fulfillment
        - allows the oracle node to receive their payment
        ✓ can't fulfill the data again (47ms)
      requester lies about amount of LINK sent
        ✓ the oracle uses the amount of LINK actually paid (70ms)

Contract: Oracle
  ✓ has a limited public interface
  #setFulfillmentPermission
    when called by the owner
      ✓ adds an authorized node
      ✓ removes an authorized node
    when called by a non-owner
      ✓ cannot add an authorized node
  #onTokenTransfer
    ✓ does not allow recursive calls of onTokenTransfer
    when called from any address but the LINK token
      ✓ triggers the intended method
    when called from the LINK token
      ✓ triggers the intended method (64ms)
      with no data
        ✓ reverts
    malicious requester
      ✓ cannot withdraw from oracle (92ms)
      if the requester tries to create a requestId for another contract
        ✓ the requesters ID will not match with the oracle contract (83ms)
        ✓ the target requester can still create valid requests (462ms)
  #requestData
```

```
    when called through the LINK token
      ✓ logs an event
      ✓ uses the expected event signature
      ✓ does not allow the same requestId to be used twice (39ms)
    when not called through the LINK token
      ✓ reverts
  #fulfillData
    cooperative consumer
      when called by an unauthorized node
        ✓ raises an error
      when called by an authorized node
        ✓ raises an error if the request ID does not exist
        ✓ sets the value on the requested contract (52ms)
        ✓ does not allow a request to be fulfilled twice (53ms)
      when the oracle does not provide enough gas
        ✓ does not allow the oracle to withdraw the payment (61ms)
        ✓ 500000 is enough to pass the gas requirement (53ms)
    with a malicious requester
      ✓ cannot cancel before the expiration (306ms)
      ✓ cannot call functions on the LINK token through callbacks
    with a malicious consumer
      fails during fulfillment
        ✓ allows the oracle node to receive their payment (111ms)
        ✓ can't fulfill the data again (47ms)
      calls selfdestruct
        ✓ allows the oracle node to receive their payment (78ms)
      request is canceled during fulfillment
        ✓ allows the oracle node to receive their payment (123ms)
        ✓ can't fulfill the data again (50ms)
      requester lies about amount of LINK sent
        ✓ the oracle uses the amount of LINK actually paid (82ms)
      tries to steal funds from node
        ✓ is not successful with call (230ms)
        ✓ is not successful with send (205ms)
        ✓ is not successful with transfer (255ms)
  #withdraw
    without reserving funds via requestData
      ✓ does nothing (43ms)
    reserving funds via requestData
      but not freeing funds w fulfillData
        ✓ does not transfer funds
      and freeing funds
        ✓ does not allow input greater than the balance (67ms)
        ✓ allows transfer of partial balance by owner to specified address (72ms)
        ✓ allows transfer of entire balance by owner to specified address (47ms)
        ✓ does not allow a transfer of funds by non-owner
  #withdrawable
    ✓ returns the correct value
  #cancel
    with no pending requests
      ✓ fails (109ms)
    with a pending request
      ✓ has correct initial balances
      from a stranger
        ✓ fails
      from the requester
        ✓ refunds the correct amount (144ms)
        ✓ triggers a cancellation event (127ms)
        canceling twice
          ✓ fails (153ms)

Contract: GetterSetter
  #setBytes32Val
    ✓ updates the bytes32 value (41ms)
    ✓ logs an event
  #requestedBytes32
    ✓ updates the request ID and value (52ms)
  #setUint256
```

```
    ✓ updates uint256 value (44ms)
    ✓ logs an event
  #requestedUint256
    ✓ updates the request ID and value (55ms)

Contract: ServiceAgreementConsumer
  ✓ gas price of contract deployment is predictable (93ms)
  #requestEthereumPrice
    without LINK
      ✓ reverts (57ms)
    with LINK
      ✓ triggers a log event in the Coordinator contract (88ms)
      ✓ has a reasonable gas cost (100ms)
  #fulfillData
    ✓ records the data given to it by the oracle (43ms)
    when the consumer does not recognize the request ID
      ✓ does not accept the data provided
    when called by anyone other than the oracle contract
      ✓ does not accept the data provided

Contract: UpdatableConsumer
  constructor
    ✓ pulls the token contract address from the resolver
    ✓ pulls the oracle contract address from the resolver
  #updateOracle
    when the ENS resolver has been updated
      ✓ updates the contract's oracle address (41ms)
    when the ENS resolver has not been updated
      ✓ keeps the same oracle address
  #fulfillData
    ✓ records the data given to it by the oracle (56ms)
    when the oracle address is updated before a request is fulfilled
      ✓ records the data given to it by the old oracle contract (69ms)
      ✓ does not accept responses from the new oracle for the old requests
      ✓ still allows funds to be withdrawn from the oracle (183ms)


123 passing (2m)
4 pending

Contract: UptimeSLA
  before updates
    ✓ does not release money to anyone (341ms)
  #updateUptime
    ✓ triggers a log event in the Oracle contract (210ms)
  #fulfillData
    when the value is below 9999
      ✓ sends the deposit to the client (341ms)
    when the value is 9999 or above
      ✓ does not move the money (370ms)
      and a month has passed
        ✓ gives the money back to the service provider (323ms)
    when the consumer does not recognize the request ID
      ✓ does not accept the data provided (51ms)
    when called by anyone other than the oracle contract
      ✓ does not accept the data provided


7 passing (5s)
```

Automated Analyses

Oyente

Repository: https://github.com/melonproject/oyente

Oyente is a symbolic execution tool that analyzes the bytecode of Ethereum smart contracts. It checks if a contract features any of the predefined vulnerabilities before the contract gets deployed on the blockchain.

Oyente Findings

Oyente reported the following issues:

- Integer overflow in the function `transfer()`. Upon closer inspection, we classified it as a false positive.
- Integer overflow in the function `onTokenTransfer()` in the contract `Oracle.sol` due to pointer arithmetic. Upon closer inspection, we classified it as a false positive as long as `_data` is created with the function `ChainlinkLib.Run.newRun()`.
- Callstack Depth Attack Vulnerability in line 107 of the contract `Oracle.sol`. This attack, however, no longer applies since it was eliminated by EIP150.

Mythril

Repository: https://github.com/ConsenSys/mythril

Mythril is a security analysis tool for Ethereum smart contracts. It uses concolic analysis, taint analysis and control flow checking to detect a variety of security vulnerabilities.

Mythril Findings

Mythril reported the following issues:

- the use of `assert()` in place of `require()` in `SafeMath.sol` functions. It is a known and benign issue with former Open Zeppelin implementations. It could be mitigated by using the most recent version of the library.
- calls to non-trusted contracts, such as `LinkToken` from `Consumer.withdrawLink()` and `Oracle.cancel()`. Upon closer inspection, we classified them as false positives since LINK is assumed to be a trusted token contract.
- multiple sends in one transaction in `Consumer.withdrawLink()`. Upon closer inspection, we classified it as a false positive.
- multiple calls to external contract in one transaction within `Consumer.withdrawLink()`. Upon closer inspection, we classified it as a benign issue.
- the function `fulfillData()` in the contract `Oracle.sol` uses address as an argument in `callback.addr.call`. We discussed this issue earlier in the report.
- the function `fulfillData()` in the contract `Oracle.sol` ignores the return value of `call()`. We recommend checking the return value.
- in the function `cancel()` in the contract `Oracle.sol` the state is modified after `Link.transfer()`.
- the function `withdraw()` in the contract `Oracle.sol` makes an external call.

MAIAN

Repository: https://github.com/MAIAN-tool/MAIAN

MAIAN is a tool for automatic detection of trace vulnerabilities in Ethereum smart contracts. It processes a contract's bytecode to build a trace of transactions to find and confirm bugs.

MAIAN Findings

MAIAN has not detected any issues.

Securify

Repository: https://github.com/eth-sri/securify

Securify Findings

Securify has not completed the analysis -- it timed out. Securify reported, however, the following issues:

- Unrestricted writes to storage. Upon closer inspection, we classified them as false positives.
- Unsafe call to untrusted contract `ERC677Token.sol`. Given the context, we classified it as a false positive.

Adherence to Specification

The code only partially conforms to the specification. We found the following discrepancies between the first audited revision and the specification:

- `Chainlinked.sol`
  - *cancelChainlinkRequest cancels the request on the Oracle.sol contract, and withdraws any payment deposited* -> No withdrawal happens in `cancelChainlinkRequest()`.
  - *cancelChainlinkRequest prevents a request from subsequently being fulfilled by the oracle(prevented on the Chainlinked side)* -> It appears to be prevented on the Oracle side.
- `Oracle.sol`
  - *requestData locks up tokens paid until a response is received, or the request is cancelled* -> `requestData()` does not appear to lock up any tokens.

Code Documentation

The code is missing documentation. We recommend providing inline comments in the code.

Adherence to Best Practices

In our opinion, the code does not adhere to best practices, especially with regards to defining and using interfaces as well as maintaining thorough code documentation. On the other hand, it does a good job at managing dependencies via package managers. **Update**: The Chainlink team updated the code to align it with best practices. They also explained some design choices that had an impact on code complexity.

Appendix

File Signatures

The following SHA-256 signatures were calulated using the following comnand: `shasum -a 256 ./contract_name.sol`

Contracts

```
contracts/Chainlinked.sol: 8fa8df49f5b8a94c3a58f86f8912fdfe06f161154e798dc95531076550027fa5
contracts/Coordinator.sol: da9ab0ed5af479a31ae9cb8ba6fda210f4f4b5c73642217d1d63fdb7be454659
contracts/Migrations.sol: 7f774a6cfb3af15c7af6936984622ca8f66b3b3d8aa3a564f4a6c9dc3a4d7f63
contracts/Oracle.sol: b7321084b4eb3543e256730b1624dc2846d33da9701623f4d4fd52539119db6f
contracts/ChainlinkLib.sol: 9c1f8bf225b013c355eede1bbb8642845ed69e31922b604e72bc7ae403f73b1c
contracts/examples/Consumer.sol: 1bcdc8396d398430d07904a8ffb0146925899a286e323b475e1bd54a26af1a3c
contracts/examples/ConcreteChainlinked.sol:
2d15cb4a419201de82ed01b58dcc2f8c3d22fc5f68c5126bb3e0ee240c228568
contracts/examples/ConcreteChainlinkLib.sol:
b62b77a927561080bec2b93499143d797fe06c23d5c6accbe0787be9a73ddac2
contracts/examples/MaliciousRequester.sol:
e0d86837533e9744d021e485087905de568e939363f24164ec0ab91e44e9c18f
contracts/examples/EmptyOracle.sol: a92bb854f718b1e3eb8c71cac1be9b71072fdd62f036d75c38c5001c13f4eac8
contracts/examples/MaliciousConsumer.sol:
b141d2a50ba9f0903aca9dab0bb43bb74c9a8a8e826aff28384fb4e9791feb22
contracts/examples/BasicConsumer.sol: c94ea74a3422f7d51bf4e68926af8f8108728f4b7cd580daa0c422a66ce43d41
contracts/examples/ServiceAgreementConsumer.sol:
ef10a31d3722a88c87f9117523feae701230a5f0ec486b2ab1b513655ae54085
```

```
contracts/examples/MaliciousServiceAgreementConsumer.sol:
604fe33faf17f900b099dffdf88d09adbb9352c579e92d288665447d48e75d22
contracts/examples/MaliciousChainlinkLib.sol:
58c57e42d5a8a74237dc0e4bb34065b86e650542550562e25b3d596e84619b74
contracts/examples/GetterSetter.sol: a717667821174150f205b4f309bccfcb1cb9985ea5ef0d9bfcc62dc81fd913e0
contracts/examples/UpdatableConsumer.sol:
791de34238af7a9fa54dca5bb26a8c74ea5f307c512d0ab9373538c0051ee07b
contracts/examples/MaliciousChainlinked.sol:
61784df5d73b24e118575ee049f96fb5e3d6b268974950fa40f2f0fdbd86a513
contracts/ENSResolver.sol: 620eb198df1570c7e6a5a5013a102a78f6d970a3f2df5170176a8ce989b40dfc
contracts/interfaces/LinkTokenInterface.sol:
ca75babc19f017050bda6f158365ac2717fb9f6c1951322a8f4492b064873dd1
contracts/interfaces/ENSInterface.sol: d10d3578c67bc4848ab048e2b016d0f6576a13028eb78d63c0bff114499d8f3c
contracts/interfaces/OracleInterface.sol:
e2463bacd133848dd1bbac10ddf3df9273a545a9c794d0939f595548abfca2c5
contracts/interfaces/CoordinatorInterface.sol:
b22369744f67386367e3d4bc8cac508c80c8447350e7f22dc430fe9543a43c4b
```

Tests

```
test/Oracle_test.js: 44bc7fec012261099d76e9961406380623bd9a48828102541b757d533699f1ed
test/ConcreteChainlinked_test.js: 86f9e0d0258d0c406bb2f56978bc08ca7bf0803570196dd7a6db2349da52e8b4
test/Coordinator_test.js: a0cb26ff239d97a1578d9ae6f55fd6315354f9b6074442c8b5f71c8ed09bba57
test/ServiceAgreementConsumer_test.js: 25d7fd537843530d1c212967904d03ae5708d59a712b26f20a525e31df5ed0fe
test/Chainlinked_test.js: 92263106cef1436d7c32ed339a2825ad3b4acca199ef757aa1bf3f891b53c161
test/GetterSetter_test.js: df4010e3d1c1675baa55f58586ea1da9c8299ae294580d8cfb4a44bfe1575d3d
test/support/matchers.js: 419a66b8629f5e4571858e82d796862b11f7edf6b0a814c55b16587ed5f1c4fe
test/support/helpers.js: 3dcd78792468359eb2c01702a4b3a75f954d019bd943b03e6623c2ba41405cf4
test/UpdatableConsumer_test.js: ad5663d48e0e4b24049d22a8243fc7c0453b93c151223b063567d3c8d03606a6
test/ConcreteChainlinkLib_test.js: 1339733950c7eec7331c4ba87428203351b1233f80620b5ecd576845be6251d1
test/BasicConsumer_test.js: 5400af1d887a4f8eef974cbf82b2bb2b320d4d648ee98eec420ba2b19985bf4a
```

Steps

Steps taken to run the full test suite:

- Installed Truffle: `npm install -g truffle`
- Installed Ganache: `npm install -g ganache-cli`
- Installed the solidity-coverage tool (within the project's root directory): `npm install --save-dev solidity-coverage`
- Ran the coverage tool from the project's root directory: `./node_modules/.bin/solidity-coverage`
- Flattened the source code using `truffle-flattener` to accommodate the auditing tools.
- Installed the Mythril tool from Pypi: `pip3 install mythril`
- Ran the Mythril tool on each contract: `myth -x path/to/contract`
- Installed the Oyente tool from Docker: `docker pull luongnguyen/oyente`
- Migrated files into Oyente (root directory): `docker run -v $(pwd):/tmp -it luongnguyen/oyente`
- Ran the Oyente tool on each contract: `cd /oyente/oyente && python oyente.py /tmp/path/to/contract`
- Ran the MAIAN tool on each contract: `cd maian/tool/ && python3 maian.py -s path/to/contract contract`

About Quantstamp

Quantstamp is a Y Combinator-backed company that helps to secure smart contracts at scale using computer-aided reasoning tools, with a mission to help boost adoption of this exponentially growing technology.

Quantstamp's team boasts decades of combined experience in formal verification, static analysis, and software verification. Collectively, our individuals have over 500 Google scholar citations and numerous published papers. In its mission to proliferate development and adoption of blockchain applications, Quantstamp is also developing a new protocol for smart contract verification to help smart contract developers and projects worldwide to perform cost-effective smart contract security audits. To date, Quantstamp has helped to secure hundreds of millions of dollars of transaction value in smart contracts and has assisted dozens of blockchain projects globally with its white glove security auditing services. As an evangelist of the blockchain ecosystem, Quantstamp assists core infrastructure projects and leading community initiatives such as the Ethereum Community Fund to expedite the adoption of blockchain technology.

Finally, Quantstamp's dedication to research and development in the form of collaborations with leading academic institutions such as National University of Singapore and MIT (Massachusetts Institute of Technology) reflects Quantstamp's commitment to enable world-class smart contract innovation.

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by Quantstamp; however, Quantstamp does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication. publication.

Notice of Confidentiality

This report, including the content, data, and underlying methodologies, are subject to the confidentiality and feedback provisions in your agreement with Quantstamp. These materials arenot to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Quantstamp.

Links to other websites

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Quantstamp, Inc. (Quantstamp). Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that Quantstamp are not responsible for the content or operation of such web sites, and that Quantstamp shall have no liability to you or any other person or entity for the use of third-party web sites. Except as described below, a hyperlink from this web site to another web site does not imply or mean that Quantstamp endorses the content on that web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the report. Quantstamp assumes no responsibility for the use of third-party software on the website and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such.

Disclaimer

This report is based on the scope of materials and documentation provided for a limited review at the time provided. Results may not be complete nor inclusive of all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The Solidity language itself and other smart contract languages remain under development and are subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity or the smart contract programming language, or other programming aspects that could present security risks. You may risk loss of tokens, Ether, and/or other loss. A report is not an endorsement (or other opinion) of any particular project or team, and the report does not guarantee the security of any particular project. A report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. To the fullest extent permitted by law, we disclaim all warranties, express or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked website, or any website or mobile application featured in any banner or other advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. You may risk loss of QSP tokens or other loss. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.