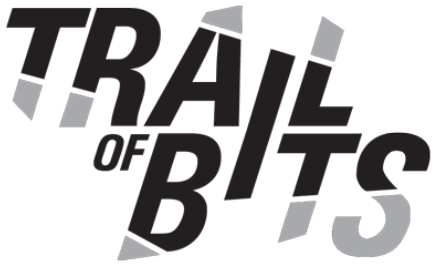

Golem

Security Assessment

Smart Contract and Token Protocol
April 5, 2018

Prepared For:
Julian Zawistowski | *Golem*
julian@golem.network

Prepared By:
Chris Evans | *Trail of Bits*
chris.evans@trailofbits.com

Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Changelog

March 23, 2018:	Initial report delivered
April 5, 2018:	Retest report delivered

Executive Summary

From March 12 through March 23, 2018, Golem engaged Trail of Bits to assess Golem's Solidity smart contracts. Trail of Bits conducted this assessment over the course of two person-weeks with two engineers.

The assessment focused on interactions between the user-facing token and the underlying proxy infrastructure that internally manages and uses the proxy tokens to perform deposits and mass-transaction operations. We directed static analysis and dynamic instrumentation to find interactions that could lead to unauthorized token manipulation in account balances or denial-of-service attacks against the smart contract protocols.

The code reviewed represents a work in progress. A large portion of the functionality is still in an experimental phase. ERC20 compatibility has been retrofitted onto the original token by implementing an intermediate token proxy class. As well as optimizing gas efficiency for batch transactions, it forms the basis of internal transactions used in the deposit process. It also duplicates the state of internal account balances and token supply tracking to maintain 1:1 parity with the original token contract.

A consequence of this complexity is a fragile interface between many moving parts. Most of the infrastructure is intended to be opaque to the average user and interacted with only in a client wrapper. The documentation is also opaque, which makes it difficult to understand the code.

Standardizing and consolidating contracts should be a top priority, followed by the migration of tokens to a single point of entry that adheres to a spec and implements all required functionality. The complexity of maintaining up to two separate balances in the original token and proxy contracts –- with no formal method of synchronization –- will lead to subtle bugs, especially as more functionality is added later.

As the Golem ecosystem adds features, the increased complexity will introduce bugs that will require changes and fixes. Immutable contracts are only good if they are simple. Currently there is no procedural method for updating deployed contracts for ancillary services. Golem should consider using an upgrade system that will allow them to introduce features and bugfixes into their new contracts.

Since the smart contracts only serve to provide tokens for economic use, minimal changes to established library templates for timelocks, deposits, and transfer mechanisms should be a goal.

This assessment focused on the TokenProxy, batch processing, and deposit smart contracts. Interactions with the Concent service and main Golem client application were

out of scope. Trail of Bits strongly recommends further studying the security of these features as they are integrated into the final environment.

Appendix C contains a short reference to the Slither static analyzer used in this engagement. Appendix D includes an overview of our dynamic analysis tool Manticore, as well as examples and scripts used to trigger TOB-Golem-06 and TOB-Golem-07.

Engagement Goals & Scope

The engagement was scoped to provide a security assessment of the risk factors related to the core Golem network smart contract ecosystem and token implementation.

In particular we sought to answer the following questions:

- Is it possible for an unauthorized third party to gain administrative access to deployed Golem contracts?
- Are tokens managed and stored securely within the contract?
- Can the token proxying infrastructure be manipulated to distort token balances?
- Is it possible to cause the contract services to enter an unrecoverable state?

The following components were out of scope for this assessment:

- Client application protocol and interface layer that interacts with on-chain contracts
- The external concent and verification/reimbursement implementation
- Message parsing and IPC libraries between the concent service and Golem
- Incomplete and in-progress smart-contract functionality (e.g., `GNTPayments.sol`)
- Initial crowdfunding and token genesis functionality
- The Golem website and installation scripts and dependencies

Trail of Bits conducted a detailed security analysis from the perspective of an attacker with access to the public Golem documentation and source code. We sought to identify risks, and scored their severity based on their likelihood and potential impact. We also sought to provide a mitigation strategy for each risk factor, whether it required a procedural change or a replacement of the solution, in whole or in part, with a more secure alternative.

Coverage

This audit focused on an in-depth analysis of the token implementation, in particular the contracts handling batch processing and proxy management processes.

GolemNetworkTokenBatching. Scenarios involving token ownership, transfer, and minting were assessed and tested. Usage of the OpenZeppelin base templates were analyzed for attack surface exposure. ERC20 compliance was also taken into consideration, as well as the handling and emission of events.

Token Proxy and Batch Processing. The current implementation relies on converting regular user tokens into an intermediary format controlled by a proxy contract on the backend in order to optimize for transfer costs and batch transactions. We explored the initial transfer mechanism using gate addresses for ways to block or intercept GolemNetworkToken (GNT) in holding. The invariants for internal account keeping which track token supply and user balances were also audited. The safety and semantics of the batch processing function was examined, in addition to the ERC677 `transferAndCall` mechanism.

Token deposit and timelocking. The ancillary deposit-escrow contract that interacts with the token proxy was analyzed for logic bugs that could be abused to trap balances or invalidate the 1:1 relationship of user account balances maintained between the proxy and the token contract. The modifier restrictions on ownership were verified and the semantics of the time-locking restriction were tested.

Retest Results

Trail of Bits performed a 3-day retest of Golem's smart contracts from March 28 to March 30, 2018 to verify the fixes of the issues reported during the two previous weeks. Each of the issues was re-examined and verified by the audit team.

Emphasis was placed on investigating the code that was patched, the efficacy of the patches on the reported issues, and the security ramifications that may arise on the rest of the contracts.

In total, Trail of Bits found that seven issues were fully addressed, one issue was partially addressed, and two issues were not addressed. Issues that were not addressed include two low-severity issues. The partial fix for the high severity issue reduces its impact to "low."

	High	Medium	Low	Info	TOTAL
Fixed	■■	■■	■	■■	7 issues
Partially Fixed	■				1 issue
Unfixed			■■		2 issues

Figure 1: Remediation status since the initial security assessment, as of March 30, 2018

In the process of the retest, Trail of Bits discovered three new issues produced by the recent patches, including a medium-severity issue related to the amount of tokens available in the already-deployed `GolemNetworkToken`. These issues are described in TOB-Golem-12, TOB-Golem-13, and TOB-Golem-14.

Further information about the patching status of the findings is in Appendix F.

Project Dashboard

Application Summary

Name	Golem Smart Contracts
Version	62a1e0dab3baf8e9bff79b653dffa7df5f2d10a0
Type	Smart contracts
Platform	Ethereum / Solidity

Engagement Summary

Dates	March 12 to March 30, 2018
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks + 3 person-day retest

Vulnerability Summary

Total High Severity Issues	3	■■■
Total Medium Severity Issues	3	■■■
Total Low Severity Issues	5	■■■■■
Total Informational Severity Issues	2	■■
Total	13	

Category Breakdown

Access Controls	1	■
Auditing and Logging	1	■
Authentication	1	■
Configuration	1	■
Data Validation	5	■■■■■
Denial of Service	1	■
Patching	2	■■
Timing	1	■
Total	13	

Recommendations Summary

Short Term

❑ **Update Solidity to the latest version.** The solc compiler is under active development. Downstreaming security changes will help increase overall contract security and catch outdated practices.

❑ **Consider including the suggested race condition fixes for `approve` and `transferFrom`.** If not required by the Golem ecosystem, locking down these methods may help address an outstanding vulnerability in the ERC20 spec.

❑ **Track upstream changes to OpenZeppelin and other dependencies.** Hardcoded library versions leave Golem vulnerable to attacks that are later fixed, and are easily identified by malicious actors.

❑ **Match GolemNetworkToken `transfer` function with library standard.** Allow 0 value transfers and remove the require condition that causes this to revert the transaction. Do not return false due to an invalid transfer. Instead, propagate error conditions to cause the transfer to fail.

❑ **Disable transfers to 0 address in `batchTransfer`.** Regular users should not be able to burn tokens. Revert the entire transaction or handle it appropriately within batch processing with a consistent response.

❑ **Do not emit `Mint` or `Burn` events for mundane user operations.** The `withdraw` and `transferToGate` functions in the `TokenProxy` functions should not emit these events for 0-value amounts. Consider removing these events entirely (or renaming them if required for internal tracking).

❑ **Do not allow self transfers inside `batchTransfer`.** This action will delete all of the user's tokens included in the transaction.

❑ **Remove the hard-coded `burn` address in GNTDeposit.** The current target address is active and will receive all burnt tokens. Burn tokens by subtracting from the internal balance manually, outside of the transfer function.

❑ **Handle additional deposits into GNTDeposit during an existing timelock .** Either restrict additional deposits during this time, or adjust the lockout window upon receiving additional deposits. Consider granular locking for individual deposits.

❑ **Distinguish timelock deposit events with task identification**. Otherwise these events can be used to trick users into believing a deposit was made on their behalf.

Long Term

❑ **Standardize contract versions and dependency management.** Implement testing frameworks and deployment harnesses that will allow for systematic code coverage. Use a package manager to keep both Golem code and external code up to date

❑ **Resolve parity differences between separate token implementations and combine them.** Having multiple implementations with different behaviors substantially increases complexity and likelihood of future vulnerabilities.

❑ **Introduce contract capabilities for managing token economics.** Golem should outline economic considerations of various token markets and consider ways to adjust inflation, supply, and allocation of tokens in response to live market changes.

❑ **Reduce the amount of shared state and independent variable tracking amongst contracts.** Consolidate token logic and management to a central core contract. Don't mimic balances elsewhere. Any functions that modify supply and ownership of tokens should all call into the same contract. Don't allow external sources to arbitrarily adjust these values.

❑ **Improve the test to include corner cases and unexpected behaviors.** The code includes a good amount of unit tests, but they only cover expected interactions. Create integration tests to cover all of the intricacies, edge cases, and action sequences that may occur out of order.

❑ **Standardize error propagation handling in token transfers between individuals and batch operations**. Inconsistent behavior between similar operations may lead to client confusion and loss of tokens.

❑ **Solidify and explicitly enforce the penalties and mechanics of the Concent system.** Building around the limitations and requirements of an external verification system will ensure that the infrastructure is there to address the evolving concerns of a live deployment.

❑ **Document large-picture interactions and scenarios of complicated functions.** Similar to the `TokenProxy::Gate` docs, a high-level summary of external interactions of a contract helps contextualize the code for readers and developers.

Findings summary

#	Title	Type	Severity
1	Contracts specify outdated compiler version	Patching	Informational
2	Race condition in the ERC20 approve function may lead to token theft	Timing	Medium
3	OpenZeppelin dependencies are not integrated to track upstream changes	Patching	Low
4	User can silently burn tokens in `batchTransfer` functions	Data Validation	Low
5	Empty accounts can fire `Mint` and `Burn` events	Data Validation	Informational
6	Deletion of user tokens in `batchTransfer` function	Data Validation	High
7	Hardcoded non-zero burn address is active	Configuration	High
8	User can silently burn tokens in the GNTDeposit withdraw function	Data Validation	Medium
9	Depositing tokens in GNTDeposit does not reset the timelock	Access Controls	High
10	Timelock events can be re-used	Auditing and Logging	Low
11	Users can burn their own tokens	Authentication	Low
12	Burning tokens do not update the corresponding total supply	Data Validation	Medium
13	A single user can stop a batch payment providing 0x0 as an address	Denial of Service	Low

1. Contracts specify outdated compiler version

Severity: Informational
Type: Patching
Target: All

Difficulty: Undetermined
Finding ID: TOB-Golem-01

Description
Golem contracts specify various outdated versions of the Solidity compiler in their pragma declarations.

The Solidity compiler is under active development. Each new version contains new checks and warnings for suspect code.

- SMT Checker: Take if-else branch conditions into account in the SMT encoding of the program variables.
- Syntax Checker: Deprecate the `var` keyword (and mark it an error as experimental 0.5.0 feature).
- Type Checker: Allow `this.f.selector` to be a pure expression.
- Type Checker: Issue warning for using `public` visibility for interface functions.
- Type Checker: Limit the number of warnings raised for creating abstract contracts.

Figure 1: Solidity releases new checks and warnings for suspect code in each new version

Running the latest available compiler (`0.4.21` as of this writing) on the Golem contracts codebase emits warnings that should be fixed.

There are also inconsistencies in the Solidity compiler version requirements between contracts and their dependencies. For example, GNTDeposit.sol requires Solidity `0.4.16` while TokenProxy.sol uses Solidity `0.4.19`, but imports OpenZeppelin templates that use `0.4.18`.

Recommendations
Ensure that the latest version of Solidity compiles all code without warnings. Compiler warnings are often indicators of bugs that may only manifest at runtime or under specific conditions. Newer versions of Solidity emit warnings for a broader set of error-prone programming practices.

Standardize the version of Solidity required by contracts and their dependencies.

2. Race condition in the ERC20 approve function may lead to token theft

Severity: Medium
Type: Timing
Target: `StandardToken`

Difficulty: High
Finding ID: TOB-Golem-02

Description
A known race condition in the ERC20 standard, on the `approve` function, could lead to the theft of tokens.

The ERC20 standard describes how to create generic token contracts. Among others, a ERC20 contract defines these two functions:

- `transferFrom(from, to, value)`
- `approve(spender, value)`

These functions give permission to a third party to spend tokens. Once the function `approve(spender, value)` has been called by a user, `spender` can spend up to `value` tokens of the user's by calling `transferFrom(user, to, value).`

This schema is vulnerable to a race condition when the user calls `approve` a second time on a `spender` that has already been allowed. If the spender sees the transaction containing the call before it has been mined, then the spender can call `transferFrom` to transfer the previous value and still receive the authorization to transfer the new value.

Exploit Scenario
1. Alice calls `approve(Bob, 1000)`. This allows Bob to spend 1,000 tokens.
2. Alice changes her mind and calls `approve(Bob, 500).` Once mined, this will decrease the number of tokens that Bob can spend to 500.
3. Bob sees the transaction and calls `transferFrom(Alice, X, 1000)` before `approve(Bob, 500)` has been mined.
4. If Bob's transaction is mined before Alice's, 1000 tokens will be transferred by Bob. But once Alice's transaction is mined, Bob can call `transferFrom(Alice, X, 500)`.Bob has transferred 1500 tokens even though this was not Alice's intention.

Recommendations
While this issue is known and can have a severe impact, there is no straightforward solution.

One mitigation is to forbid a call to `approve` if all the previous tokens are not spent, by adding a requirement to approve. This solution prevents the race condition but it may cause unexpected behavior for a third party.

```
require(allowed[msg.sender][_spender] == 0)
```

Another mitigation is the use of a temporal mutex. Once `transferFrom` has been called for a user, it needs to prevent a call to `approve` during the window in which the transaction is occuring. The user can then verify if someone transferred the tokens. This solution adds complexity and may also result in unexpected behavior for a third party.

This issue is a flaw in the ERC20 design. It cannot be easily fixed without modifying the standard and it must to be considered by developers while writing code.

3. OpenZeppelin dependencies do not track upstream changes

Severity: Low
Type: Patching
Target: `open_zeppelin` folder

Difficulty: Low
Finding ID: TOB-Golem-03

Description
The `BasicToken`, `ERC20`, `ERC20Basic`, `SafeMath` and `StandardToken` implementations from OpenZeppelin are copy-pasted into the repository. This makes receiving updates and security fixes on these dependencies unreliable as they must be updated manually.

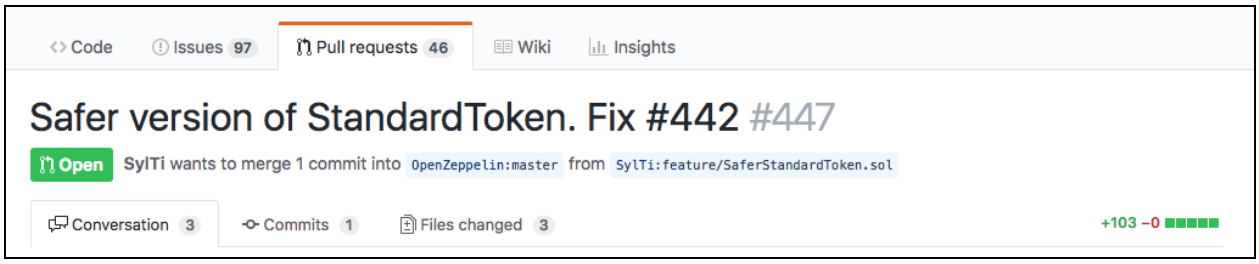

Figure 2: OpenZeppelin receives ongoing testing and updates regularly

Exploit Scenario
OpenZeppelin releases a critical fix for a vulnerability in the underlying token implementations that allow unauthorized withdrawal. An attacker could scan for token repositories that use hardcoded and outdated copies of the OpenZeppelin base templates and use the vulnerability against Golem.

Recommendations
Include the OpenZeppelin sources as a submodule in your Git repository so that internal path consistency can be maintained and updated periodically.

In the long term use an Ethereum development environment and NPM to manage the package as part of your project. A quick start for OpenZeppelin and Truffle can be found at https://github.com/OpenZeppelin/zeppelin-solidity#getting-started. This will ensure that the Golem smart contracts and their dependencies are cohesively managed.

4. User can silently burn tokens in batchTransfer function

Severity: Low
Difficulty: High
Type: Data Validation
Finding ID: TOB-Golem-05
Target: `GolemNetworkTokenBatching`

Description
The amount of minted tokens is tracked in the GNTB contract by the `totalSupply` function (returning an underlying `totalToken` variable). Burning tokens is disabled by default in `transfer.` ERC20 enforces an explicit call and event to trigger a token burn. However, the `batchTransfer` method does not restrict the address destination of `address(0)`, allowing tokens to effectively be burned without firing a `Burn` event or decreasing the `totalSupply` variable.

Exploit Scenario
Scenario 1: Alice programmatically interacts with the Golem token network as a legitimate member. A calculation results in a transfer to the null or empty address of 0. As a result, Alice loses her tokens.

Scenario 2: Bob is a malicious third party intent on destabilizing the Golem network. He burns a significant amount of tokens in the `GolemNetworkTokenBatching` contract to cause an internal consistency between the amount of tokens in circulation and tracked token supply count. He can use this information by either manipulating the economics of additional token minting, or by causing an invariant failure in token supply conditions for a contract migration.

Recommendation
Add a `require` condition in `batchTransfer` that explicitly forbids burning tokens.

In the future, outline the exact circumstances of how token economies are impacted by concurrent supply. Ensure unit tests verify all ways in which a transfer can affect the tracked token supply. An inaccurate token count can lead to loss of faith in the the Golem ledger's accounting and may reduce trust in the system as a whole.

5. Empty accounts can trigger Mint and Burn events

Severity: Informational
Type: Data Validation
Target: `TokenProxy`

Difficulty: Easy
Finding ID: TOB-Golem-06

Description
Both the `withdraw` and `transferFromGate` functions do not require the callers to have non-zero amounts they wish to withdraw or deposit. This allows third parties with no tokens to call into these functions and trigger arbitrary `Mint` and `Burn` events.

If these events are used outside the blockchain to trigger external code, it could produce unexpected results (e.g., division by zero).

See Appendix D for example Manticore scripts that trigger these findings.

Exploit Scenario
A client application for the Golem network listens to events to calculate remaining token costs for pending processing jobs. The UI uses the `Mint` event to calculate how many tokens are being consumed per job node as a percentage. The 0 value is used in calculating that, causing an error in the display which breaks the display or causes an unhandled exception at runtime.

Recommendation
In `withdraw` add a check for `require(balance > 0)`. For `transferToGate`, add a check for `require(value > 0)`. Alternatively, re-evaluate the need for `Mint` and `Burn` events with the `TokenProxy` altogether.

Long term, it's important to reduce the amount of state required by proxy contracts as much as possible. Consolidating events and contract semantics in a single area will reduce the attack surface. By distributing logic and state in both the proxy and token interface, the complexity of maintaining both systems will grow exponentially.

6. Deletion of user tokens in batchTransfer function

Severity: High
Type: Data Validation
Target: `GolemNetworkTokenBatching`

Difficulty: Easy
Finding ID: TOB-Golem-07

Description

When tracking the user's initial balance during a a batch-transfer request, the value is stored in a local variable and decremented within the `for` loop. The transfer completes successfully if there is enough initial balance to make all the payments, otherwise the entire transaction is reverted. After successful completion, the user's balance is set to the remaining value left in the local variable after having subtracted all payment requests.

The issue occurs when a payment element contains the address of the initiating user (`msg.sender`). Despite being incremented in the loop correctly as the recipient, `balances[msg.sender]` will be reset and any tokens sent to the initiating user will be transparently lost.

```
function batchTransfer(bytes32[] payments, uint64 closureTime) external {
        require(block.timestamp >= closureTime);
        uint balance = balances[msg.sender];
        for (uint i = 0; i < payments.length; ++i) {
            // A payment contains compressed data:
            // first 96 bits (12 bytes) is a value,
            // following 160 bits (20 bytes) is an address.
            bytes32 payment = payments[i];
            address addr = address(payment);
            uint v = uint(payment) / 2**160;
            require(v <= balance);
            balances[addr] += v;
            balance -= v;
            BatchTransfer(msg.sender, addr, v, closureTime);
        }
        balances[msg.sender] = balance;
}
```

Figure 3: `batchTransfer` sets the user balance only once after the loop has completed

Exploit Scenario
Alice submits a computationally intensive job to the Golem network. She collaborates alongside the others working on this job on the Golem network. When time for payment arrives, Alice's address is included in the client as a designated payment address. Tokens sent to her are lost, irrecoverably and silently.

Recommendation
Adding `require(addr != msg.sender)` inside the loop will be a quick workaround to mitigate this issue.

The larger concern remains with handling error propagation and recovery in `batchTransfer` edge cases (i.e sending to oneself, sending to a null address, insufficient transfer amount to complete payments). The semantics surrounding multiple batch transactions should conform to expected behavior of single transaction. Maintaining this relationship should be an API and security priority.

7. Hardcoded non-zero burn address is active

Severity: High
Type: Configuration
Target: `GNTDeposit`

Difficulty: High
Finding ID: TOB-Golem-08

Description

The `0xdeadbeef` was designated as a special address that the Concent service can use to burn tokens in `GNTDeposit`.

```
function burn(address _whom, uint256 _burn) onlyConcent external {
        _reimburse(_whom, 0xdeadbeef, _burn);
        Burn(_whom, _burn);
    }
```

Figure 4: The burn implementation in `GNTDeposit`

A hardcoded burn address is insecure because it is impossible to know whether the private key is known and held by an arbitrary address. In this case, the `0xdeadbeef` address shows evidence of activity and recent transactions -- suggesting it is not a safe target for token burning. If the account ever decided to interact with the Golem network, its balance would not be unaccounted for in the total token supply metric, and would likely be abnormally large.

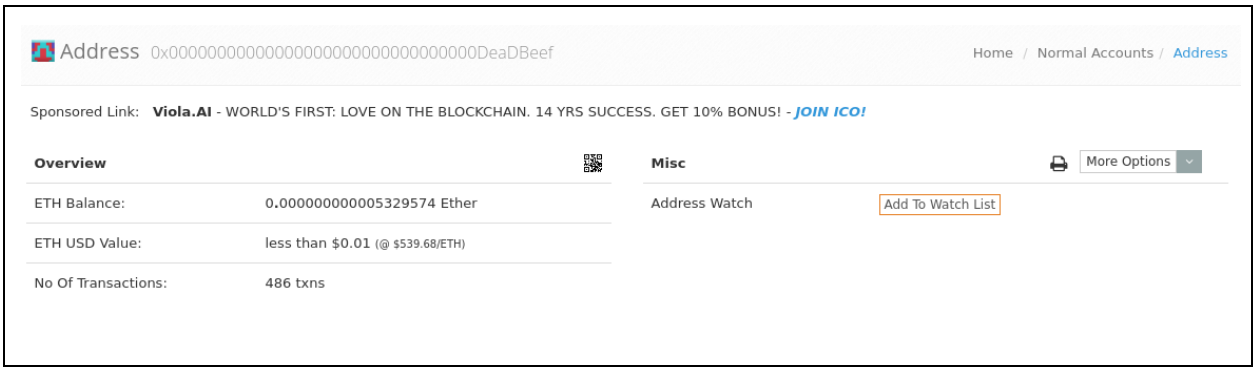

Figure 5: Address `0xdeadbeef` is active on mainnet

Exploit Scenario

As the Concent service burns tokens over the course of time, the holder of the `0xdeadbeef` private key notices token activity occurring with their address. They decide to withdraw the entire burned token supply out of the contract, potentially causing a complete economic collapse of the Golem ecosystem and a loss of massive amounts of ether.

Recommendation
Do not use a token transfer to manage the bookkeeping of burned tokens. Token burning should be handled internally by decreasing the existing token supply and user balances. Refer to OpenZeppelin's BurnableToken contract on how to implement a compliant `burn` operation.

In the future, minting and burning actions that must operate on the total supply of tokens in a predictable and secure manner will necessitate that these variables are tracked in a single token location, rather than being distributed amongst various internal representations.

8. User can silently burn tokens in the GNTDeposit withdraw function

Severity: Medium
Type: Data Validation
Target: `GNTDeposit`

Difficulty: Low
Finding ID: TOB-Golem-09

Description

Only the Concent user should be able to burn tokens, but normal users can work around this restriction using the `withdraw` function to transfer tokens to the special address `0xdeadbeef`. This effectively allows tokens to be burned without firing a `Burn` event.

```
function withdraw(address _to) onlyUnlocked external {
    var _amount = balances[msg.sender];
    balances[msg.sender] = 0;
    locked_until[msg.sender] = 0;
    require(token.transfer(_to, _amount));
    Withdraw(msg.sender, _to, _amount);
}
```

Figure 6: The `withdraw` function allows to transfer/burn tokens to `0xdeadbeef` by allowing any address as a parameter

Exploit Scenario

Bob is a malicious third party intent on destabilizing the Golem network. He burns a significant amount of tokens in the `GNTDeposit` contract to cause an internal inconsistency between the amount of tokens in circulation and tracked token supply count. He can use this discrepancy either to manipulate the economics of additional token minting, or to cause an invariant failure in token supply conditions for a contract migration.

Recommendation

Implementing the recommended fix in TOB-Golem-08 will prevent regular users from being able to burn since the 0 address will be reverted by the token transfer.

Since it is possible (and valid) to withdraw a deposit to an account that has not yet been registered via the gate proxy, ensure that this edge case is handled appropriately in tests when adding additional functionality.

9. Depositing tokens in GNTDeposit does not reset the timelock

Severity: High
Type: Access Controls
Target: `GNTDeposit`

Difficulty: Medium
Finding ID: TOB-Golem-10

Description
The GNTDeposit contract implements a timelock on user accounts to restrict the initial window that tokens may be withdrawn. However, a user is able to withdraw tokens earlier than expected since the individual deposit of new tokens is not tracked. After an initial deposit from a user, the withdrawal period window is not extended for any subsequent deposits. By manipulating low-cost jobs, it may be possible to pre-empt the waiting period for a later, higher-cost job.

```
function onTokenReceived(address _from, uint _amount, bytes /* _data */) public onlyToken {
        balances[_from] += _amount;
        Deposit(_from, _amount);
}
```

Figure 7: onTokenReceived does not check or reset the timelock.

Exploit Scenario
1. Bob unlocks his account in the `GNTDeposit` with no tokens and waits until the unlock time is about to elapse.
2. Bob submits a computationally intensive job to the Golem network.
3. Alice takes the job but ask for the confirmation of transaction of the payment into the `GNTDeposit`.
4. Bob transfers the tokens into his `GNTDeposit` balance.
5. Alice verifies that deposit and checks that the tokens are effectively time-locked.
6. Alice finishes the job and send the results to Bob.
7. Bob withdraws his tokens since they are no longer timelocked.

Recommendation
One mitigation is to forbid a deposit when the balance is time-locked. Another possible mitigation is to adjust the corresponding timelock if a user increases his balance during the window that an existing lock is active, or to have granular locking control over each individual deposit.

For the future, ensure that penalty actions and the infrastructure supporting token moderation is appropriately restrictive and exhaustive in scope.

10. Timelock events can be reused

Severity: Low
Type: Auditing and Logging
Target: `GNTDeposit`

Difficulty: High
Finding ID: TOB-Golem-11

Description

The `GNTDeposit` contract implements timelocks in order to prevent users from withdrawing their tokens for a certain period of time. However, `Deposit` events do not include corresponding task information and are indistinguishable from one another. A malicious user can cite a prior event to deceive a participant into believing that a deposit has been made into the Concent service.

```
function onTokenReceived(address _from, uint _amount, bytes /* _data */) public onlyToken {
        balances[_from] += _amount;
        Deposit(_from, _amount);
}
```

Figure 8: onTokenReceived does not record any identification string to each Deposit

Exploit Scenario

1. Bob submits a computationally intensive job to the Golem network.
2. Alice accepts that job.
3. Bob deposits tokens in `GNTDeposit` to satisfy Alice's prerequisite to use the Concent service.
4. Bob submits a second computationally intensive job to the Golem network similar to the first one.
5. Carol accepts the job, also requiring participation in the Concent service.
6. Bob points to his deposit to Alice in GNTDeposit to convince Carol that her tokens are safe until she finish computing the second job.
7. Bob cancels his job with no penalty and Carol must eat the cost of having done work for no reason.

Recommendation

This issue can be mitigated by identifying every timelock with its corresponding task and allowing users to query this information to avoid deposit forgery. There is a bytestring in the parameter of `transferAndCall` that can be used to pass additional arguments.

The implementation details of the Concent service must be rigorously applied and standardized before the Golem network scales. Without a strong and consistent deterrent against misbehaving clients, nodes will be susceptible to abuse en-masse by freeloaders.

11. Users can burn their own tokens

Severity: Low
Type: Authentication
Target: `GolemTokenNetworkBatching`

Difficulty: Low
Finding ID: TOB-Golem-12

Description
Only the Concent user should be allowed to burn tokens from users. However, a user could work around this restriction using the `burn` function in `GolemTokenNetworkBatching`. This only allows an attacker to burn their own tokens.

Exploit Scenario
Bob is a malicious third party intent on destabilizing the Golem network. He burns a significant amount of tokens in the `TokenProxy` contract to cause an internal inconsistency between the amount of tokens in circulation and tracked token supply count. He can use this information by either manipulating the economics of additional token minting, or by causing an invariant failure in token supply conditions for a contract migration.

Recommendations
Specify the Concent user as an additional parameter in the `GolemTokenNetworkBatch` contract to validate the user calling the `burn` in that contract.

Long term, it is strongly recommended to consolidate token logic and management to a central core contract that allows token creation, burning, and locking.

12. Burning tokens does not update the corresponding total supply

Severity: Medium
Difficulty: Low
Type: Data Validation
Finding ID: TOB-Golem-13
Target: `GolemTokenNetwork`, `GolemTokenNetworkBatching`

Description
The burn function in `GolemTokenNetworkBatching` does not update the `totalSupply` in the `GolemTokenNetwork`. Since the burned tokens are deleted and no longer associated with one particular address (e.g. 0x0), the `GolemTokenNetwork` reports more tokens than it should. This issue may cause code or logic that depends on the value of `totalSupply` (for instance, code that calculates the value of a Golem token) to report an incorrect value.

Exploit Scenario
Bob is a malicious third party intent on destabilizing the Golem network. He burns a significant amount of tokens in the `TokenProxy` contract to cause an internal inconsistency between the amount of tokens in circulation and tracked token supply count. He can use this information by either manipulating the economics of additional token minting, or by causing an invariant failure in token supply conditions for a contract migration.

Recommendations
One possible mitigation is to implement a similar function to burn tokens in the `GolemTokenNetwork` contract and call it using the token infrastructure from `GolemTokenNetworkBatching`. Nevertheless, a naive implementation is not recommended to avoid other security issues such as TOB-Golem-12.

In the long term, it is strongly recommended to consolidate token logic and management to a central core contract that allows token creation, burning, and locking. Rather than implementing proxy classes that manage internal state independently, migrate users to a single token instance that is interoperable with all Golem smart contracts. This will ensure that the consistency is maintained and accurately reflects the tokens in circulation.

13. A user can stop a batch payment by providing `0x0` as an address

Severity: Low
Difficulty: High
Type: Denial of Service
Finding ID: TOB-Golem-14
Target: `GolemTokenNetworkBatching`

Description
An attacker can provide the address `0x0` as their own [what?] to cause the revert of payments performed using `batchTransfer`. This will cause a temporary denial of service since it stops the payments for all the other users in the same batch.

Exploit Scenario
1. Alice submits several computationally intensive jobs to the Golem network
2. Bob takes one of the jobs providing the address `0x0` as his own to get his payment.
3. Bob performs the requested computation.
4. Alice waits until there are a large number of payments and uses `batchTransfer` to perform them in order to save some gas.

Bob has now blocked the payments for all the users in the batch.

Recommendations
Make sure the code that performs the call to `batchTransfer` discards any payment to 0x0. This issue can be mitigated by preventing users from controlling their payment address used in the `batchTransfer` function.

A. Vulnerability classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Arithmetic	Related to arithmetic calculations
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Code quality recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance readability and may prevent the introduction of vulnerabilities in the future.

Use `SafeMath` for arithmetic operations

- The OpenZeppelin framework already requires `SafeMath` as a dependency. Any operations involving user-supplied arithmetic with respect to balance accounting and transaction management should be performed with this library.
- Handling and reverting runtime overflows where they occur is superior to relying on conditional logic in specific function entry points.

Use `require` instead of `revert` for verifying single-line conditionals

- In the GolemNetworkToken.sol `transfer` function, for example.

Do not `require` token `transfer` calls

- While returning `false` is valid per the spec, the OpenZeppelin framework tokens rely on reverting and throwing on errors -- this should be kept consistent throughout the entire codebase.

Update use of deprecated keywords with suitable replacements

- `Var`, `throw`, `constant`, etc. are all emitted as compiler warnings (see TOB-Golem-01).

Minimize the code and functions available in the contracts

- A large number of possible operations allows potential attackers to explore the code for vulnerabilities.

Do not repeat or shadow variable names in different contracts

- Variables such as `_token` refer to both `GolemNetworkToken` and `GolemNetworkTokenBatching` classes in different contracts.
- GNTDeposit, GolemNetworkToken and TokenProxy all maintain separate instances of the `balances` address mapping. In some cases this is inherited from a common token interface, but sometimes it is declared as a standalone contract member.

C. Slither static analysis

Trail of Bits has included our Solidity static analyzer, Slither, with this report. Slither works on the Abstract Syntax Tree (AST) generated by the Solidity compiler and detects some of the most common smart contract security issues, including:

- The absence of a constructor
- The presence of unprotected functions
- Uninitialized variables
- Unused variables
- Functions declared as constant that change the state
- Deletion of a structure containing a mapping

Slither is an unsound static analyzer and may report false positives. The lack of proper support for inheritance and some object types (such as arrays) may lead to false positives.

In order to use Slither, simply launch the analysis on the Solidity file:

```
$ python /path/to/slither.py file.sol
```

Ensure that import dependencies and libraries, such as OpenZeppelin, can be found by the solc compiler in the same directory.

D. Manticore formal verification

We reviewed the feasibility of formally verifying the contract with Manticore, a simple, open-source dynamic EVM analysis tool that takes advantage of symbolic execution.

Symbolic execution allows us to explore program behavior in a broader way than classical testing methods, such as fuzzing. The central part of our work involved defining plausible scenarios to analyze the Golem contracts. Such scenarios let Manticore explore how attackers could manipulate the code and how those manipulations would affect the contracts.

We defined three scenarios to explore with Manticore. The first and the second ones requires two users: Alice and Bob. Alice is benevolent. Bob is the attacker. Initially, Alice holds some GNT tokens and starts to migrate some tokens to the GNTB network. The last scenario requires only one user holding some GNTB tokens.

- Scenario 1: Alice executes `openGate` to open a gate and transfer some tokens to it. Bob will try to steal or interfere with the token migration. We allowed Manticore to perform two fully symbolic transactions using the GNT contract and check if the attacker could perform some transactions that altered Alice's balance.

- Scenario 2: Alice executes `openGate` to open a gate, transfers some tokens to it and finishes the migration executing `transferFromGate`. Bob will try to steal or block Alice's tokens. We allowed Manticore to perform fully two symbolic transactions using the GNTB contract and check if the attacker could perform some transactions that altered Alice's balance.

- Scenario 3: The last scenario requires only one user holding some GNTB tokens. Bob will try to increase, burn or block his own tokens to subvert the balances in the GNTB contract. We allowed Manticore to perform two fully symbolic transactions using the methods in the GNTB contracts and check if the attacker could perform some transactions that altered his balance.

The first scenario can be symbolically explored using the `Golem_openGate.py` script, the second one, using the `Golem_transferFromGate.py` script and the third one using the `Golem_own_tokens.py` script. While not demonstrative of vulnerabilities, they provide a reference for interacting with Golem smart contracts using the Manticore tool. They can also be useful for re-testing when there is new code to fix these issues.

TOB-Golem-06: Empty accounts can trigger Mint and Burn events

TOB-Golem-06 can be reproduced using the `Golem_zero_mint.py` and `Golem_zero_burn.py` scripts. The process of minting tokens in the Golem network comprises a 3-step procedure. A user should (1) open a gate using the openGate, (2) transfer some ERC20 tokens from a her account the associated gate and (3) call `transferFromGate` to finish the procedure. This last transaction will fire the `Mint` event announcing the amount of tokens minted. The `Golem_zero_mint.py` script reproduces the issue of firing `Mint` event with no tokens. After running the script, we can observe this issue in this list of ethereum transactions:

```
…
Transactions Nr. 4
From: 0xf522dfdc3f12cc0c75ffbff51e5876bf982e52b2
To: 0x5074d85b9194e696cc596130ffe95f02eaa1c3df
Function call: openGate() -> STOP

Transactions Nr. 5
From: 0xf522dfdc3f12cc0c75ffbff51e5876bf982e52b2
To: 0x5074d85b9194e696cc596130ffe95f02eaa1c3df
Function call: transferFromGate() -> STOP
```

It is also possible to burn Golem tokens using the `withdraw` function indicating the amount of tokens to burn. This transaction will fire the `Burn` event announcing the amount of tokens minted. The `Golem_zero_burn.py` script reproduces the issue of firing a `Burn` event with no tokens. After running the script, we can observe this issue in this list of ethereum transactions:

```
...
Transactions Nr. 4
From: 0xf522dfdc3f12cc0c75ffbff51e5876bf982e52b2
To: 0x5074d85b9194e696cc596130ffe95f02eaa1c3df
Function call: withdraw(0) -> STOP
```

TOB-Golem-07: Deletion of user tokens in batchTransfer function

TOB-Golem-07 can be reproduced using the `Golem_batchTransfer_burn.py` script. Using it, Manticore initializes the contracts and creates an account (with the address `0x75ffbff51e5876bf982e524e5a695365d51f264a`) that holds 1000 tokens. Then, the account performs a call to `batchTransfer` to pay 1000 tokens (encoded as "`\x03\xe8u`") to itself (encoded as "`\xff\xbf\xf5\x1eXv\xbf\x98.RNZiSe\xd5\x1f&J`"). Finally, Manticore checks the balance of the account to verify that it has no tokens. After running the script, we can observe this issue in this list of ethereum transactions:

```
…
Transactions Nr. 9
From: 0x75ffbff51e5876bf982e524e5a695365d51f264a
To: 0x5074d85b9194e696cc596130ffe95f02eaa1c3df
Function call: balanceOf(0x75ffbff51e5876bf982e524e5a695365d51f264a) ->
RETURN
return: 1000

Transactions Nr. 10
From: 0x75ffbff51e5876bf982e524e5a695365d51f264a
To: 0x5074d85b9194e696cc596130ffe95f02eaa1c3df
Function call:
batchTransfer('\x00\x00\x00\x00\x00\x00\x00\x00\x00\x03\xe8u\xff\xb
f\xf5\x1eXv\xbf\x98.RNZiSe\xd5\x1f&J',0) -> STOP

Transactions Nr. 11
From: 0x75ffbff51e5876bf982e524e5a695365d51f264a
To: 0x5074d85b9194e696cc596130ffe95f02eaa1c3df
Function call: balanceOf(0x75ffbff51e5876bf982e524e5a695365d51f264a) ->
RETURN
return: 0
```

E. Issues discovered in GolemNetworkToken

In the process of the security review, Trail of Bits discovered two issues related to the transfer function of the already deployed GolemNetworkToken (GNT) contract. These issues may affect the internal consistency of the remaining contracts, since GNT is directly or indirectly called by them. Additionally, the transfer function is an integral part of any token system, so we decided to include these additional issues in this special section.

Token transfer not ERC20 compliant

The GNT implementation mimics ERC20 behavior and maintains partial parity with the underlying GNTBatch token proxy which aims for ERC20 compliance. As a result, `Transfer` events are expected from both classes every time a transaction occurs. However the GNT implementation disallows transfer values of 0, which is explicitly required by the ERC20 spec, and so does not fire a `Transfer` event when this scenario occurs. In addition, the return of a false value is not required due to revert conditions on transfer failure.

```
function transfer(address _to, uint256 _value) returns (bool) {
        // Abort if not in Operational state.
        if (funding) revert();

        var senderBalance = balances[msg.sender];
        if (senderBalance >= _value && _value > 0) {
            senderBalance -= _value;
            balances[msg.sender] = senderBalance;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        }
        return false;
}
```

Figure 9: Function returns false and does not fire the `Transfer` event if `_value == 0`

Exploit Scenario

Bob uses a third-party exchange client to attempt purchase of Golem network tokens. The client relies on an initial 0 value transfer to establish a handshake and verify successful interaction with the token contract. It cannot complete the transaction. Depending on implementation, Bob may lose ether as a result of the unexpected behavior.

Recommendations
Remove the check condition that enforces `_value > 0`. Do not return false as a valid boolean result. Instead, rely on the revert conditions to cause transfer to fail.

In the long term, thought must be given to the viability of maintaining two separate tokens with implementation differences that must maintain 1-to-1 parity. At the very least, sharing a single base template will help in maintaining development standards and ensure uniform application of best practices.

User can silently burn tokens in GNT transfer functions

The amount of minted tokens is tracked in the GNT contract by the `totalSupply` function (returning an underlying `totalToken` variable). This value is used during the migration contract and is tracked by the `TokenProxy` class as well. Burning tokens is disabled by default in `transfer.` ERC20 enforces an explicit call and event to trigger a token burn. However, this method in GolemNetworkToken does not restrict the address destination of `address(0)`, allowing tokens to effectively be burned without firing a `Burn` event or decreasing the `totalSupply` variable.

```
function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender]);
```

Figure 10: ERC20 BasicToken does not allow transfers/burns to address(0)

Exploit Scenario

Scenario 1: Alice programmatically interacts with the Golem token network as a legitimate member. A calculation results in a transfer to the null or empty address of 0. As a result, Alice loses her tokens.

Scenario 2: Bob is a malicious third party intent on destabilizing the Golem network. He burns a significant amount of tokens in the `TokenProxy` contract to cause an internal consistency between the amount of tokens in circulation and tracked token supply count. He can use this information by either manipulating the economics of additional token minting, or by causing an invariant failure in token supply conditions for a contract migration.

Recommendation

Add a `require` condition in `transfer` that explicitly forbids burning tokens.

In the future, outline the exact circumstances of how token economies are impacted by concurrent supply. Ensure unit tests verify all ways in which a transfer can affect the tracked token supply. An inaccurate token count can lead to loss of faith in the the Golem ledger's accounting and may reduce trust in the system as a whole.

F. Fix Log

Golem made the following modifications to their codebase as a result of the assessment. Each of the fixes was verified by the audit team. The reviewed code is available in git shortcode: 4e50ca2c.

ID	Title	Severity	Status
1	Contracts specify outdated compiler version	Informational	Fixed
2	Race condition in the ERC20 approve function may lead to token theft	Medium	Fixed
3	OpenZeppelin dependencies do not track upstream changes	Low	Not Fixed
4	User can silently burn tokens in batchTransfer function	Low	Fixed
5	Empty accounts can trigger Mint and Burn events	Informational	Fixed
6	Deletion of user tokens in batchTransfer function	High	Fixed
7	Hardcoded non-zero burn address is active	High	Fixed
8	User can silently burn tokens in the GNTDeposit withdraw function	Medium	Fixed
9	Depositing tokens in GNTDeposit does not reset the timelock	High	Partial
10	Timelock events can be re-used	Low	Not Fixed

Issue 9 has a partial fix that reduces the severity from high to low. Note that issues 11, 12, and 13 were discovered during the retest and therefore fixes for them were not reviewed.

Detailed Fix Log

Finding 1: Contracts specify outdated compiler version
Fixed by updating all the contracts to Solidity version 0.4.21.
https://github.com/golemfactory/golem-contracts/commit/1af6431b214cb75d7bec0604d5ff3e7a2d5f55c0

Finding 2: Race condition in the ERC20 approve function may lead to token theft
Fixed. User cannot make subsequent calls to `approve` until the previously approved tokens have all been transferred.
https://github.com/golemfactory/golem-contracts/commit/ec5e6ab223eff61523e47cae7f59dcb024c73369

Finding 3: OpenZeppelin dependencies do not track upstream changes
Not fixed.

Finding 4: User can silently burn tokens in `batchTransfer` functions
Fixed. The token transfer functions in `GolemTokenNetworkBatch` and `GNTDeposit` no longer allow transfers to the 0x0 address. Note, users can still burn their tokens using `transfer` in `GolemNetworkToken`.
https://github.com/golemfactory/golem-contracts/commit/84df986928dc69efcf23902f164a2070dbda725f

Finding 5: Empty accounts can trigger Mint and Burn events
Fixed. Burning tokens now requires a strictly positive balance. Minting of zero tokens will trigger a revert.
https://github.com/golemfactory/golem-contracts/commit/817973c0b060182084bdc11b684838e1cbc32148

Finding 6: Deletion of user tokens in `batchTransfer` function
Fixed by avoiding a transfer to yourself or to address 0x0.
https://github.com/golemfactory/golem-contracts/commit/b90a2912253aa14f9f1466612a0e895622dd34f9

Finding 7: Hardcoded non-zero burn address is active
Fixed by using BurnableTokens from OpenZeppelin, which defines a new method to burn ERC20 tokens using the `burn` function. It does not implement this functionality using a particular address to transfer burned tokens.
https://github.com/golemfactory/golem-contracts/commit/3a95a27953234a0b767353cc1cbd8f5d41140d04

Finding 8: User can silently burn tokens in the `GNTDeposit` withdraw function.

Fixed by avoiding the use of 0x0 as an address for withdrawals.
https://github.com/golemfactory/golem-contracts/commit/84df986928dc69efcf23902f164a2070dbda725f

Finding 9: Depositing tokens in `GNTDeposit` does not reset the timelock
Partially fixed by resetting a timelock after a deposit. Nevertheless, users need to check that a timelock deposit is not near expiration. This reduces the severity of the issue from High to Low.
https://github.com/golemfactory/golem-contracts/commit/4e50ca2cba13d99219f5b71dc58e9fe3c0efabfc

Finding 10: Timelock events can be re-used
Not fixed. Golem said that they are aware of the issue and it can be treated as by design since deposits were never meant to be per task. Golem also indicated they applied mitigations for this issue off-chain.