DappHub

Security Assessment

DappHub Smart Contract Libraries
December 8, 2017

Prepared For:
Andy Milenius | *DappHub*
andy@dapphub.com

Prepared By:
Josselin Feist | *Trail of Bits*
josselin@trailofbits.com

Evan Sultanik | *Trail of Bits*
evan.sultanik@trailofbits.com

Executive Summary

From November 13 through December 8, 2017, DappHub engaged with Trail of Bits to conduct an assessment of the DappHub libraries. The libraries reviewed were ds-auth, ds-chief, ds-exec, ds-guard, ds-group, ds-proxy, ds-roles, ds-stop, ds-vault and ds-weth. All assessed code was written in Solidity. Trail of Bits conducted these assessments over the course of 8 person-weeks with two engineers.

The two first weeks focused on the highest priority libraries: ds-auth, ds-exec, ds-guard, ds-proxy, ds-roles, ds-stop, and ds-vault. The two final weeks focused on analyzing the remaining libraries: ds-cache, ds-value, ds-chief, ds-clock, ds-group, ds-prism and ds-weth.

Though many of the libraries had few or no issues and were well-written, the assessment did identify two issues of high severity. The first high severity issue could lead to the loss of tokens as well as breaking the scheduling mechanism in DSClock. The second high severity issue could prevent a legitimate finalist from being elected through DSPrism. Other reported issues involved a variety of errors, such as missing contract existence checks, incorrect implementation of a scheduling feature, incorrect function parameter ordering, and issues related to the ERC20 standard.

Overall, we found that most of the contracts were written with a clear awareness of current smart contract development best practices. The design of small smart contracts with a clear purpose is an excellent practice. The clear separation of logic allows for targeted testing. However, not all the contracts benefit from the same code quality and some would benefit from better tests, such as DSClock and DSPrism.

After this review, one possible way to improve trust in the contracts' behavior would be to write a formal specification for each contract. The DappHub libraries are a good fit for such a specification, and it would further ensure that the contracts behave as they are intended.

Engagement Goals

The goal of the engagement was to evaluate the security of the DappHub libraries. These libraries are designed to perform only a small set of actions and any unexpected behavior would be an issue. Specifically, we sought out answers to the following questions:

- Are there any inconsistencies in the documentation for the libraries?
- Are any unintended actions possible by using the libraries?

Project Dashboard

Application Summary

Name	DappHub librairies
Version	ds-auth, ds-exec, ds-chief, ds-guard, ds-group, ds-proxy, ds-roles, ds-stop, ds-vault and ds-weth
Type	Ethereum Smart Contract
Platform	Solidity

Engagement Summary

Dates	November 13–December 8, 2017
Method	Whitebox
Consultants Engaged	2
Level of Effort	8 person-weeks

Vulnerability Summary

Total High Severity Issues	3	🟦🟩🟥
Total Medium Severity Issues	5	🟦🟦🟩🟨🟫
Total Low Severity Issues	0	
Total Informational Severity Issues	3	🟦🟥🟪
Total	11	

Category Breakdown

Data Validation	8	🟦🟦🟦🟩🟩🟨🟫🟥
Timing	1	🟥
Auditing and Logging	1	🟦
Undefined Behavior	1	🟪
Total	11	

Legend: 🟦 DSClock, 🟩 DSPrism, 🟨 DSProxy, 🟫 (DSProxy+DSExec), 🟥 DSWeth, 🟪 DSChief

Recommendations Summary

Short Term

Ensure that all the high severity issues are remediated. Fix the `wireId` assignment in DSclock, and add the last finalist to the election in DSPrism.

Integrate continuous integration with commits on Github. This will ensure that erroneous tests are not added.

Ensure that the documentation matches the implementation. We found many corner cases where the expected behavior of a library was not mentioned in its documentation.

Long Term

Improve unit test coverage. Many of the vulnerabilities found may have been detected with better test coverage.

Thoroughly test temporal features. The DappHub testing framework does not handle time-related conditions well. Extra effort should be made to test features related to time.

Consider writing formal specifications of the contracts. The DappHub libraries are a good fit for formal specification. It would ensure that the contracts behave as expected.

Test ERC20 tokens against the ERC20-K specification. ERC20-K is a recent project that aims to formalize the ERC20 standard. Consider testing ERC20 tokens against this specification.

Findings Summary

#	Title	Type	Severity
1	Missing contract existence check can cause lost ethers	Data Validation	Medium
2	Cached destructible contracts may lead to corrupted execution	Data Validation	Medium
3	Wrong operator leads to unexecuted operations and lost tokens	Data Validation	High
4	Missing loop iteration prevents the last finalist from being elected	Data Validation	High
5	Race condition in the ERC20 approve function may lead to token theft	Timing	High
6	Actions without expiration times are not executable	Data Validation	Medium
7	Wrong parameter order leads to unusable function	Data Validation	Medium
8	Calling ERC20.transferFrom to itself may lead to unexpected behavior	Data Validation	Low
9	DSClock test hangs forever	Auditing and Logging	Undetermined
10	Tie Breaking in DS-Prism	Data Validation	Medium
11	Mismatches between the DSChief documentation and code may lead to unexpected behavior	Undefined Behavior	Undetermined

Contracts Summary

Library	Security Issues	Type	Severity
DSAuth	None identified		
DSCache	None identified		
DSChief	TOB-DappHub-011	Undefined Behavior	Undetermined
DSClock	TOB-DappHub-003	Data Validation	High
	TOB-DappHub-006	Data Validation	Medium
	TOB-DappHub-007	Data Validation	Medium
	TOB-DappHub-009	Auditing and Logging	Undetermined
DSExec	TOB-DappHub-001	Data Validation	Medium
DSGuard	None identified		
DSGroup	None identified		
DSPrism	TOB-DappHub-004	Data Validation	High
	TOB-DappHub-010	Data Validation	Medium
DSProxy	TOB-DappHub-001	Data Validation	Medium
	TOB-DappHub-002	Data Validation	Medium
DSRoles	None identified		
DSStop	None identified		
DSToken	TOB-DappHub-005	Timing	High
	TOB-DappHub-008	Data Validation	Undetermined
DSValue	None identified		
DSVault	None identified		
DSWeth	TOB-DappHub-005	Timing	High
	TOB-DappHub-008	Data Validation	Undetermined

1. Missing contract existence check can cause lost ethers

Severity: Medium   Difficulty: Low
Type: Data Validation   Finding ID: TOB-DappHub-001
Target: DSExec, DSProxy

Description

DSExec and DSProxy provide a contract calling mechanism with more features than the original EVM calling mechanism. Due to the lack of a code contract existence check, a call to a non-contract address or to a self-destructed contract will be considered successful, while in fact no code is actually executed. Library users will not expect this behavior.

The function `tryExec` of DSExec (exec.sol#L21-L26) calls `target` and returns the success of the execution:

```solidity
function tryExec( address target, bytes calldata, uint value)
        internal
        returns (bool call_ret)
{
    return target.call.value(value)(calldata);
}
```

The function `execute` of DSProxy (proxy.sol#L63-L70) calls the target through `delegatecall` and ensures that the call was a success:

```solidity
assembly {
    let succeeded := delegatecall(sub(gas, 5000), _target, add(_data,
0x20), mload(_data), 0, 32)
    response := mload(0)      // load delegatecall output
    switch iszero(succeeded)
  case 1 {
       // throw if delegatecall failed
     revert(0, 0)
}
```

The Solidity documentation mentions this warning:

> The low-level `call`, `delegatecall`, and `callcode` will return success if the calling account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.

As a result, if the target of the call in `tryExec` or in `delegatecall` does not possess code, then DSExec and DSProxy consider the call a success. This can occur if the target is a non-contract address or if the contract was destructed.

Exploit Scenario

Bob creates a contract which can receive ethers and can be self-destructed. Bob uses DSExec to call the contract. The contract is destroyed. Bob continues to send ethers to the contract without realizing that the contract was destroyed. As a result, all the ethers sent by Bob are lost.

Recommendation

Check for the existence and validity of the contract address before all calls, or explicitly mention this limitation in the documentation of DSExec and DSProxy.

Carefully review the Solidity documentation. In particular, any section that contains a warning must be carefully understood since it may lead to unexpected or unintentional behavior.

2. Cached destructible contracts may lead to corrupted execution

Severity: Medium   Difficulty: High
Type: Data Validation   Finding ID: TOB-DappHub-002
Target: DSProxy

Description
DSProxy provides a contract calling mechanism with more features than the original EVM calling mechanism. DSProxy contains a cache of previously known contracts. An attacker can take advantage of the cache system to fool a user into calling a destructed contract.

DSProxyCache stores the addresses of known contracts in `cache` through `write`:

```
function write(bytes _code) public returns (address target) {
        assembly {
            target := create(0, add(_code, 0x20), mload(_code))
            switch iszero(extcodesize(target))
            case 1 {
                // throw if contract failed to deploy
                revert(0, 0)
            }
        }
        bytes32 hash = keccak256(_code);
        cache[hash] = target;
    }
```

Anyone can at any time replace the contract stored by a new instance of it. For destructible contracts, an attacker can replace the stored instance with a new one he or she controls. As a result, the attacker can destruct the stored contract. A user can then be fooled into using a destructed contract instead of a live one.

This situation can occur for contracts calling `selfdestruct`, `suicide`, or `delegatecall`.

Exploit Scenario
Bob creates a contract which can be destructed by its owner. Bob uses this contract through `DSProxy`. Alice deploys a new instance of this contract. Alice destructs the contract. As a result, any of Bob's future calls through `DSProxy` will be delegated to the destructed contract. As a result, no code is executed, and the Bob's execution is compromised.

Recommendation
We suspect that DSProxy should be callable only for non-destructible contracts. Clarify this in the documentation.

3. Wrong operator leads to unexecuted operations and lost tokens

Severity: High   Difficulty: Low
Type: Data Validation   Finding ID: TOB-DappHub-003
Target: DSClock

Description

DSClock allows scheduling of actions that may transfer ethers. Due to the use of the equality operator instead of assignment, only one scheduled action can be stored. As a result, any action previously stored is not taking into consideration. Thus, the actions are not triggered, and the related ethers are lost.

DSClock stores each new item using the wireID index. wireID is never assigned; instead the equality operator is used (clock.sol#L68-L80):

```
wireID == ++next;
wires[wireID] = Wire({
    owner: msg.sender,
    [...]
    done: false
});
```

As a result, wireID is always 0 and a new item is always stored in wires[0].

Exploit Scenario

Bob schedules an event in DSClock which is triggered with 10 ethers. Alice schedules another event in DSClock. As a result, the event of Bob is removed and the 10 ethers are lost.

Recommendation

Fix the assignment.

Improve the unit tests coverage. This vulnerability would have been found by any test containing more than one event scheduled.

4. Missing loop iteration prevents the last finalist from being elected

Severity: High   Difficulty: Low
Type: Data Validation   Finding ID: TOB-DappHub-004
Target: DSPrism

Description
DSPrism provides a multi-step voting system. Due to incorrect loop bounds, the last finalist can never be elected.

Once a list of finalists is established, the function `snap` can elect the winners. The loop iterating over all the finalists does not iterate over the last finalist (prism.sol#L162):

```
for( uint i = 0; i < finalists.length - 1; i++ ) {
```

As a result, the last finalist cannot be elected.

Appendix C contains a test case to trigger this issue.

Exploit Scenario
Alice and Bob are the two finalists of the election. Both have the same weight. Alice and Bob should be both elected, but only Alice is elected.

Recommendation
Add the last finalist to the election.

Improve the unit tests coverage. This vulnerability would have been found by any test checking the election of the last finalist.

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

Severity: High   Difficulty: High
Type: Timing   Finding ID: TOB-DappHub-005
Target: DSWeth, DSToken and DSTokenBase

Description
There is a [known race condition](#) in the ERC20 standard, on the approve function, leading to the possible theft of tokens.

The ERC20 standard describes how to create generic token contracts. Among other things, an ERC20 contract must define these two functions:

1. `transferFrom(from, to, value)`
2. `approve(spender, value)`

The goal of these functions is to 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 by calling `transferFrom(user, to, value).`

This schema is vulnerable to a race condition when the user calls `approve` a second time on an already allowed spender. If the spender sees the transaction containing the call before it has been mined, they 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, 500)`. This allows Bob to spend 500 tokens.
2. Alice changes her mind and calls `approve(Bob, 1000).` This changes the number of tokens that Bob can spend to 1000.
3. Bob sees the transaction and calls `transferFrom(Alice, X, 500)` before it has been mined.
4. If the transaction of Bob is mined before the one of Alice, `500` tokens have been transferred by Bob. But, once the transaction of Alice is mined, Bob can call `transferFrom(Alice, X, 1000)`.

Bob has transferred 1500 tokens even though this was not Alice's intention.

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

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

```
require(_approvals[msg.sender][guy] == 0)
```

Another solution 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 limited time. The user can then verify if someone transferred the tokens. However, 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 modifications to the standard and it must to be considered by developers while writing code.

6. Actions without expiration times are not executable

Severity: Medium	Difficulty: Low
Type: Data Validation	Finding ID: TOB-DappHub-006
Target: DSClock	

Description

DSClock allows executing actions during an interval of time. The documentation states that it is possible to schedule an action without an expiration time. Due to an implementation error, an action scheduled without expiration time cannot be executed.

Among others, an action has a shut field. The [documentation](#) states that:

> shut: *The last timestamp when this action is fireable. 0 means no expiration.*

wire ([clock.sol#L55-L68](#)) schedules an action:

```
function wire( address target, uint256 value, bytes data, uint256
reward, uint256 start, uint256 end)
        public
        payable
        returns (uint256 wireID)
    {
        require( msg.value >= value + reward );
        require( start <= end );
        wireID == ++next;
```

wire requires that start <= end. As a result, an action can have no expiration time only if start is 0.

fire ([clock.sol#L101-L111](#)) executes an action:

```
function fire(uint256 wireID)
        public
        payable
    {
        Wire storage W = wires[wireID];
        require( W.open <= now && now <= W.shut );
        require( !W.done );
        W.done = true;
        require( W.target.call.value(W.value)(msg.data) );
```

```
            require( msg.sender.call.value(W.reward)() );
    }
```

fire requires that now <= W.shut. As a result, any action with 0 as shut cannot be executed.

Exploit Scenario
Bob schedules an action without an expiration time. Alice wants to trigger the action, but is not able to.

Recommendation
Implement the execution of actions without expiration time.

Improve the unit tests' coverage. When using ds-test, consider testing more thoughtfully any feature related to the time, as now returns 0 by default during the dapphub testing.

7. Wrong parameter order leads to unusable function

Severity: Medium   Difficulty: Low
Type: Data Validation   Finding ID: TOB-DappHub-007
Target: DSClock

Description
`DSClock` allows for scheduling and executing actions. Due to an incorrect parameter order, the `wire` function, which schedules an action, cannot be executed.

`wire(target, value, data)` (clock.sol#L89-L98) is meant to be a wrapper around `wire(target, value, data, reward, start, stop)`:

```
function wire( address target, uint256 value, bytes data) public
payable
        returns (uint256 wireID)
    {
        return wire(target, value, data, now, 0, msg.value-value);
    }
```

`wire(target, value, data)` does not provide `reward`, `start`, and `stop` in the correct order. As a result, the action is incorrectly scheduled, and the storing operation is likely to fail.

Exploit Scenario
Bob develops a smart contract using `DSClock`. To schedule an action, the contract uses only `wire(target, value, data)`. As a result, the smart contract of Bob is not working properly and no action is scheduled.

Recommendation
Correct the parameters order.

Improve the unit tests' coverage. This vulnerability would have been found by testing the `wire` wrapper.

8. Calling ERC20.transferFrom to itself may lead to unexpected behavior

Severity: Undetermined   Difficulty: Low
Type: Data Validation   Finding ID: TOB-DappHub-008
Target: DSToken and DSWeth

Description

The `ERC20.transferFrom(src, dst, value)` allows transferring tokens on behalf of another user, using a token allowance mechanism. DSToken and DSWeth have a different behavior than most of the other ERC20 tokens when the source parameter of `transferFrom` is the caller.

In `DSToken` and `DSWeth`, if the `src` of `transferFrom(src, dst, value)` is the sender, the allowance is not checked and decreased. As a result, using `transferFrom` to itself does not use the allowance mechanism.

This case is not specified in the ERC20 standard, but it differs from most of the other ERC20 tokens implementation. As a result, a client could have an unexpected behavior using `DSToken` or `DSWeth`.

Exploit Scenario

Bob develops a smart contract using ERC20 tokens. The function `spend` of the contract iterates over a set of source and destination addresses using `transferFrom` until all the allowance is spent. The contract uses DSToken. The allowance is never decreased. As a result, `spend` runs out of gas.

Recommendation

Use the allowance in `transferFrom` for all the transfers, or ensure that it does not use the allowance for a transfer to itself.

ERC20-K is a recent project that aims to formalize the ERC20 standard. This corner case is discussed in the section 3.6.2 . Consider testing the ERC20 tokens on this specification.

9. DSClock test hangs forever

Severity: Informational   Difficulty: Undetermined
Type: Auditing and Logging   Finding ID: TOB-DappHub-009
Target: DSClock

Description
The tests provided with DSClock (clock.t.sol) run indefinitely without termination.

We did not investigate this issue, as it may be related to a problem in the DappHub testing framework itself.

Note: this bug has been triggered using ethrun (`dapp test-ethrun`).

Recommendation
Fix the test.
Add automatic tests tied to github to prevent erroneous tests.

10. Tie Breaking in DS-Prism

Severity: Medium   Difficulty: Low
Type: Data Validation   Finding ID: TOB-DappHub-010
Target: DSPrism

Description

The assertions in swap and drop ensuring the list of finalists is sorted by weight use strict inequalities for comparison. This is presumably to break ties according to the candidate addition order. However, the fact that DSPrism allows token weights to be manipulated arbitrarily through `lock` and `free` allows this tie breaking rule to be circumvented.

Exploit Scenario 1

Alice currently has a finalist at index `i` with `approvals[i]` in vote weight. Bob wants his candidate at address `b` to be higher than Alice's, but he does not want to spend more than `approvals[i]` voting tokens. Bob calls the following:

  1. `lock(amt)`, where `amt` is equal to `approvals[i] + 1`
  2. drop(i, b) to replace Alice's candidate with his
  3. free(1) to retrieve the extra token.

This results in Alice's candidate being replaced by Bob's, even though Alice's candidate was a finalist first and Bob's candidate ultimately has the same number of votes. Alice could of course turn around and do the same thing back to Bob, but this requires that Alice have at least `approvals[i]+1` voting tokens.

Exploit Scenario 2

Alice has a finalist at index `i` and Bob has a finalist at index `j`, where `i < j`. Let `v = approvals[i] - approvals[j]` be the voting token weight difference between the two finalists. Bob can call `lock(v+1)`, `swap(i, j)`, `free(1)`. This will rank Bob's finalist higher than Alice's, even though their final weights are tied and Alice's finalist was added first. Alice will be unable to have her finalist regain its position unless she has at least one additional voting token.

It is unclear from the documentation whether the order of the finalist list is significant, and we expect it is likely dependent on how the DSPrism library is implemented by the user. It may very well be the case that this scenario is innocuous for most, if not all, implementations.

Recommendation

One way to prevent this issue is to only allow calling `free` *after* the vote is complete. Another is to separately track when finalists are added and use that for tie breaking. If such

a mitigation is undesirable, then the documentation should be updated to note the behavior described in these scenarios.

11. Mismatches between the DSChief documentation and code may lead to unexpected behavior

Severity: Undetermined   Difficulty: Undetermined
Type: Undefined Behavior   Finding ID: TOB-DappHub-011
Target: DSChief

Description
There are several mismatches between the DSChief documentation and the implementation. This difference may lead to unexpected behavior for a user of the library or a third-party client.

These mismatches are:
- `vote(address[] yays, address lift_whom)` fails if `lift_whom` is not elected. As a result, the vote is not taken into account. It is not the expected behavior, according to the documentation.
- Similarly `vote(bytes32 slate, address lift_whom)` fails if `lift_whom` is not elected. The documentation is not clear on the expected outcome.
- The events `LogLockFree`, `LogEtch`, `LogVote` and `LogLift` are not implemented. A client based on these events will not work.
- `DSChief.getUserRoles(address who)` does not return the maximum bytes32 if the address is the current chief.
- `DSChief.setUserRole` does not throw if the role is 0.
- `DSChief.isUserRoot`, `DSChief.setUserRole` and `DSChief.setRootUser` do not call up `DSRoles`.

Recommendation
Clarify the correct behavior of DSChief.

A. 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
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.

DSVault
- Add a constructor to `DSVault` and initialize the `token`.
- Clarify the usage of `DSVault`. The documentation of `DSVault` mentions that it is "*bounded to a single token*." `DSVault` inherits `DSMultiVault` and, as a result, the functions handling ERC20 tokens such as `push(ERC20, address,uint)` and `pull(ERC20, address,uint)` are present in `DSVault`.
- Change `assert` with `require` (multivault.sol#25,28). `assert` is meant to be used only for statements that should never be reached (see the <u>solidity documentation</u>).

DSRoles
- Add the argument `enable` in the documentation of `setRoleCapability.`
- Replace the type `var` in ds-roles.sol#70,80,100,120 with an explicit type

DSGuard
- Replace the type `var` in ds-guard.sol#44,45 with an explicit type

DSValue
- Change `assert` to `require` (value.sol#30).

DSCache
- Change `assert` to `require` (cache.sol#33,34).
- Specify that the time is considered expired before the expiration time (*strictly less than*).

DSChief
- Add the `auth` modifier to `DSChief.setRootUser`. Note that it is not a vulnerability here, as this function will always revert, but it could become a vulnerability in a future code refactoring.
- Replace `constant` by `view` in `DSChief.isUserRoot`.
- Remove the ability of the owner to influence the roles (note: this was fixed by the DappHub team in commit <u>6843e13b</u>).
- Note: the documentation uses a different variable name than the code in:

- ○ `vote(address[] yays)` in the documentation versus `vote(address[] guys)` in the code
- ○ `vote(address[] yays, address lift_whom)` in the documentation versus `vote(address[] guys, address lift_whom)` in the code

DSWeth

- Note: `totalSupply` uses `this.balance` instead of the correct amount of tokens emitted. The DappHub team reported they were aware of the potential difference.

DSClock

- Remove the value parameter in `DSClock.wire`. Value can be inferred from `msg.value` and `reward`. This would prevent an user to accidentally sent more ethers than necessary when scheduling an action.
- Keep the same naming convention between `Wire.open/start` and `Wire.shut/end`.
- Remove `payable` from `fire`. There is no purpose of keeping this function payable, the ethers sent will be trapped.
- Note: DSClock may be vulnerable to the issue TOB-DappHub-001. The documentation is not clear if the reward should be given if the contract does not exist.

DSGroup

- Change all the `assert` with `require`.
- Update the compiler version to a more recent one, and use `view` instead of `constant`.
- Add `quorum<=member.length` in the constructor to avoid an initialization error.
- Change `++actionCount` with `actionCount++` in group.sol#L96 to start `id` at 0.

C. Test cases

TOB-DappHub-004

This function can be directly added in prism.t.sol:

```solidity
function testAuditTob004() public{
        var slateID = initial_vote(); // initials vote puts c1, c2 and
c3 as finalist with the same weight
        prism.snap();
        assert(prism.isElected(c1));
        assert(prism.isElected(c2));
        assert(prism.isElected(c3)); // it fails
}
```