Urbit Constitution JS Library

Security Audit

July 15th, 2018

Version 1.0.0

Prepared by

Bloctrax

Introduction

This document includes the results of the security audit for Urbit's Constitution JS library as found in the section titled 'Source Code'. The security audit was performed by the Bloctrax team from July 8th 2018 to July 15th 2018.

The purpose of this audit is to review Urbit's Constitution JS library source code, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

Overall Assessment

The Urbit Consitution JS library is low quality and rife with issues. We recommend that the Urbit team bring in a new developer to rewrite the whole thing from scratch using the existing source code as a guide.

Specification

Our understanding of the specification was based on the following sources:

- Our understanding of the desired behavior based on our previous audit of the Urbit Constitution Solidity code.
- Discussions with the Urbit team (in person and via email).

Source Code

The following source code was reviewed during the audit:

Repository	Commit
index.js	b2fe461f400d716df8561b89e344b6c59e7fea76

Note: This document contains an audit only of the code contained in the files listed above.

Severity Level Reference

Level	Description
High	The issue poses existential risk to the project, and the issue identified could lead to massive financial or reputational repercussions.
Medium	The potential risk is large, but there is some ambiguity surrounding whether or not the issue would practically manifest.
Low	The risk is small, unlikely, or not relevant to the project in a meaningful way.
Code Quality	The issue identified does not pose any obvious risk, but fixing it would improve overall code quality, conform to recommended best practices, and perhaps lead to fewer development issues in the future.

Issues Descriptions and Recommendations

Use of `For In` to Iterate Through Arrays

HIGH

The `buildOwnedShips` function uses a `for in` loop to iterate through an array, however, this is a bad idea because a `for in` loop will enumerate all properties of the array, not just the array indices. In addition, there is no guarantee that the indices will be enumerated in order. Instead, a standard for (var i = 0; i < nameOfArray.length; i++) loop should be used instead.

```
for (var i in x) {
```

```
    if (i == x.length - 1) {
      // transfer shiplist once built
      buildShipData(x[i], true, callback);
    } else {
      // transfer shiplist once built
      buildShipData(x[i], false, callback);
    }
  }
```

The `readOwnedShips` function also exhibits the same issue.

Validation Functions Continue To Execute Even After Failing Validation

HIGH

The validation functions are written using a middleware-like pattern where they accept the argument to be validated, a `callback` function (to be called in the case of an error), and a `next` function (to be called if the validation succeeds). If the validation succeeds, then the functions behave as expected, however, due to a missing early return statement, if the validation fails then they will execute BOTH the `callback` and the `next` function, which means both the error case logic AND the success case logic will run.

```
var validateGalaxy = function(galaxy, callback, next) {
 if (galaxy < 0 || galaxy > 255)
   callback({ error: { msg: "Ship " + galaxy + " not a galaxy."
}, data: '' });
 return next();
}
```

The `doDeposit` function also contains a similar issue where its nested function `checkHasNotBeenBooted` checks if the ship has been booted already, and if so, calls the callback with an error, but it will still call `transact()` after as well because there is no early return.

Non Error Code Continues to Execute After Encountering Error

HIGH

In the `generateTx` function, the following line checks if an error occurred in an asynchronous request, and if it did, notifies the caller of the error via a `callback` function. However, once the `callback` function completes, the rest of the code will continue to execute, but that code is only supposed to execute if there was no error. In addition, this happy path code also calls the same `callback` function so it will be called twice.

```
ajaxReq.getTransactionData(wallet.getAddressString(),
function(data) {
     if (data.error) callback({ error: { msg: data.msg }, data:
'' });
     data = data.data;
     tx.to = tx.to == '' ? '0xCONTRACT' : tx.to;
...
```

Mutation of global variables in-between function calls

HIGH

In some cases, the code mutates global variables between function calls. An example of this is the `doTransaction` function which mutates the `tx` global:

```
tx.data = data;
tx.value = value || 0;
tx.unit = "wei";
```

Reasoning about whether this will cause bugs or not is difficult because while the standard JavaScript runtime environment is not multi-threaded, it does have an

event-loop which can still lead to concurrency bugs even though there is no true parallelism. After the lines above which mutate the global variable, the following lines of code may be executed:

```
ajaxReq.getTransactionData(wallet.getAddressString(),
function(data) {
          if (data.error) {
            callback({ error: { msg: data.msg }, data: '' });
          } else {
            data = data.data;
            tx.to = address;
            tx.contractAddr = tx.to;
            showRaw = false;
            generateTx(callback);
          }
        });
```

It's not clear exactly what `ajaxReq` is supposed to be because the provided code leaves `ajaxReq` as an uninitialized variable (which is a separate issue), but assuming the intended behavior is that it trigger some kind of asynchronous network request, then the current use of the global `tx` object is likely racey as a different call to `doTransaction` could occur while the network request is still outstanding, modifying the `tx` object, and then when the network request returned and the anonymous callback above was fired, the call to `generateTx` would be using a tx object that has state from the second call to `doTransaction` instead of the first.

Instead of mutating global variables, the Urbit team should just allocate a `tx` object per-request, and pass it explicitly between functions as an argument instead of having multiple top-level functions referring to the same global variable.

Use of Out Of Date ABI

HIGH

The `getVotesAddress` function tries to read a public variable of type `address` out of the constitution contract, however, the name of that variable has changed to `polls`, so the function as written will not work with the latest version of the Constitution contract.

Similarly, the `getKeys` function calls the `getKeys` function on the ships contract, however, the `getKeys` function on the Ships contract returns 4 different values, whereas the JavaScript code is expecting only one return value.

Also, the `configureKeys` function is now called `setKeys` and has a different function signature as well.

Incorrect Validation Logic

HIGH

Some of the helper functions have incorrect validation logic that is not congruent with the Solidity contracts themselves. For example, the `doReject` function verifies that the caller is the owner of the escapee ship when it should verify that the caller is the owner of the sponsoring ship.

Improper Wallet Initialization Checks

LOW

doTransaction performs the following check at the beginning:

```
if (wallet.getAddressString() == null) {
    return;
}
```

However, later in the function it performs the following check:

```
if (wallet == null)
```

If wallet was null then the first check would have thrown a null pointer exception. Its possible that wallet could have been set to null in-between the two sections of code, but it's unlikely. Its difficult to provide more structured feedback on how to improve this code, however, because the existing code-base just leaves wallet as an uninitialized global variable.

Unlinted Source Code

CODE QUALITY

The source code appears to be unlinted as it contains numerous issues that even the most lax linter would complain about. Ex:

```
if (wallet.getAddressString() == null) {
return;
}
```

Consider picking any standard JavaScript linter, running it on the code base, and fixing any issues.

Frequent Use of Global Values

CODE QUALITY

The source code makes heavy use of sharing data between function calls by mutating global values which is a nightmare scenario for trying to identify bugs / issues. For example, the following:

```
var contract = {
 address: '',
 abi: '',
 functions: [],
 selectedFunc: null
```

```
};
```

is defined at the top of index.js, and modified / access via various different functions at different times. The `addr` variable (as well as several other variables) are manipulated the same way.

Another example of this is the `buildOwnedShips` function which omits the `var` keyword when initializing the `tmp` variable, which implicitly creates a globally scoped variable called `tmp` which is then accessed from the global scope by the `buildShipData` function. Code structured this way is extremely difficult to reason about.

There are several other examples of global variables in the code base, but they are too many to enumerate the code base should just be rewritten to not share state in this way at all.

Silent Failures Instead of Explicit Errors

CODE QUALITY

The source code contains several functions which fail silently instead of providing explicit errors. For example:

- `doTransaction` checks if there is an available wallet, and if there is not, it just silently fails. Instead, it should probably provide a structured error to the callback function so that the caller can decide whether or not they want to handle that case.
- `doTransaction` just prints an error (that the user won't see) if anything goes wrong during the gas estimation phase.

Unresolved TODOs

CODE QUALITY

The source code contains several unresolved TODOs:

- `doTransaction` contains the following TODO: "TODO 1.8 is a bit much though. consult experts on why this can be so unpredictable, and how to fix it."

- `readBalance` contains a comment that says: “throw an error here” but it does not throw an error.
- `doDeposit` contains a comment that says: “will this bork if you enter a new pool address on the deposit screen?”
- `doSafeTransferFrom` contains a comment that says: “add check to validate that the caller has been approved to initiate transfer”

Referring to Error Messages by Index Instead of Name

CODE QUALITY

The doTransaction function refers to error messages by their index in an array exported by the `globalFuncs` variable. Consider exporting explicit variable names instead of an array so they can be referenced by human-friendly names instead of indices, otherwise it’s hard to tell from reading the code whether the correct error is being thrown.

Unnecessary use of try/catch

CODE QUALITY

The code uses try/catch blocks (which are generally to be avoided in JavaScript when possible) unnecessarily. For example, in `doTransaction` a try/catch block is created, however, all the errors being thrown are being thrown explicitly within the try block. Instead of creating a try/catch block, it would be better to use guards statements along with early callbacks. The `generateTx` function has the same problem.

Use of Angular Specific Variables

CODE QUALITY

The `generateTxOffline` function makes frequent use of the `$scope` variable which is not defined anywhere in the file. We assume that this is a reference to the Angular `$scope` variable and is the result of copy-pasting this code out of the MyEtherWallet codebase.

Regardless of whether this is intentional or not, we think its bad practice, and the function should be modified to accept arguments and return values instead of accepting its arguments as global variables and then mutating global state (also via global variables).

Repetitive / Copy-Pasted Code

CODE QUALITY

Some portionS of the code are repetitive / copy-pasted. For example, the following block of error handling code is duplicated in the `doTransaction` and `generateTx` functions:

```
if (wallet == null) {
     throw globalFuncs.errorMsgs[3];
   } else if (!ethFuncs.validateHexString(tx.data)) {
      throw globalFuncs.errorMsgs[9];
   } else if (!globalFuncs.isNumeric(tx.gasLimit) ||
parseFloat(tx.gasLimit) <= 0) {
     throw globalFuncs.errorMsgs[8];
   }
```

Frequent Use of Magic Numbers Instead of Named Constants

CODE QUALITY

The code makes frequent use of hard-coded magic numbers like 256 (number of galaxies) and 4294967295 (maximum ship value) which would be cleaner and easier to read if they were replaced with constants like `numGalaxies` and `maxShipNum` or something like that.

Ship Validation Logic Does Not Take Floats Into Account

CODE QUALITY

All JavaScript numbers are 64-bit floating point, but ships can only be integers. The ship validation logic is filled with operations like this:

```
(ship < 0 || ship > 4294967295)
```

Which will pass for an invalid value like 0.5.

Redundant Validation Logic

CODE QUALITY

There are two types of validation functions: validation middleware which is used within the package itself and not exported, and exported validation helpers (which return booleans instead of executing callbacks like the middleware functions). While the structure of the functions are different, justifying the existence of both, they contain too much redundant logic. Instead, the middleware functions should be written to reuse the helper functions.

```
var validateShip = function(ship, callback, next) {
 if (ship < 0 || ship > 4294967295)
   callback({ error: { msg: "Ship " + ship + " not a galaxy,
star or planet." }, data: '' });
 return next();
}

var valShip = function(ship) {
 if (ship < 0 || ship > 4294967295 || typeof ship !== 'number')
{
   return true;
 } else {
   return false;
 }
}
```

Appendix

bloctrax

Exhibit A -Disclaimer

Bloctrax makes no warranties, either express, implied, statutory, or otherwise, with respect to the services or deliverables provided in this report, and Bloctrax specifically disclaims all implied warranties or merchantability, fitness for a particular purpose, noninfringement and those arising from a course of dealing, usage or trade with respect thereto, and all such warranties are hereby excluded to the fullest extent permitted by law.

Bloctrax will not be liable for any lost profits, business, contracts, revenue, goodwill, production, anticipated savings, loss of data, or costs of procurement of substitute goods or services or for any claim or demand against company by any other party. If no event will Bloctrax be liable for consequential, incidental, special, indirect, or exemplary damages arising out of this agreement or any work statement, however caused and (to the fullest extent permitted by law) under any theory of liability (including negligence), even if Bloctrax has been advised of the possibility of such damages.

The scope of this report and review is limited to a review of only the code presented by the Urbit team and only the source code Bloctrax notes as being within the scope of Bloctrax's review within this report. This report does not include an audit of the deployment scripts used to deploy the Solidity contracts in the repository corresponding to this audit. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. In this report you may through hypertext or other computer links, gain access to websites operated by persons other than Bloctrax. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such websites' owners. You agree that Bloctrax is not responsible for the content or operation of such websites, and that Bloctrax shall have no liability to your or any other person or entity for the use of third party websites. Bloctrax assumes no responsibility for the use of third party software and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.