Coinspect

Source Code Review

Prepared for Augur • November 2017

v1.0

1. Table of Contents

2. Executive Summary

In September 2017, Augur engaged Coinspect to perform a security audit of the reference Solidity Compiler. The objective of the audit was to evaluate the security of the compiler.

During the assessment, Coinspect identified 0 high-risk issues, 0 medium-risk issues, and 10 low-risk issues. The issues identified during the assessment do not lead to the compilation of vulnerable code. Some of the low-risk issues were communicated to the Solidity team and fixed in newer releases, while some other issues remain unfixed.

3. Introduction

A whitebox security audit was conducted on the Solidity Compiler in order to detect detect compiler flaws that can result in:

- Reduction of the security of the deployed contracts.
- Result in non-deterministic behaviour.
- Malicious code execution or crashes when parsing specially crafted Solidity source code.
- Resource exhaustion during compilation, either CPU, memory or disk.
- Compiled code that consumes a non-constant amount of gas (e.g. depending on arguments), where the programmer would have expected constant cost.
- Facilitating underhanded code (trojans in open code)

Also common application security vulnerabilities were searched, including:

- Input validation.
- Denial of service prevention.
- Brute-forcing prevention.
- Information disclosure.
- Memory corruption vulnerabilities: buffer overflows, user supplied format strings.
- Integer overflows.
- Pointer management vulnerabilities: Double-free, use-after-free.

The audit was completed on October 2017, but the the report was completed on November 2017. This reports includes all the results from the audit.

3.1. Scope

The objective the audit was to review the implementation of the Solidity Compiler. The files audited belong to the following libraries:

- `libdevcore`
- `liblll`
- `Libsolidity`
- `solc`

4. Summary Of Findings

ID	Description	Risk
SOL-001	O(n^2) compiler output blow-up by forced warnings/errors	Low
SOL-002	O(n^3) compiler output blow-up by function name duplicates	Low
SOL-003	RAM Blow-up by constants cycles	Low
SOL-004	RAM Blow-up by exponential steps in constant cycle findings	Low
SOL-005	Unbounded gas cost when deleting dynamically sized arrays	Low
SOL-006	Duplicated super-constructor calls not reported	Low
SOL-007	Error-prone Multi-Assignment with empty LValues	Low
SOL-008	CPU blow-up using huge bignums literals	Low
SOL-009	Output messages size blow-up using huge bignums literals	Low
SOL-010	Easy underhanded code using false overrides	Low

5. Findings

SOL-001	O(n^2) compiler output blow-up by forced warnings/errors
Category	Compiler attacks
Total Risk	Low | Impact: Low | Likelihood: Medium | Effort to Fix: Low
Status	Fixed

Description

The compiler does not limit the number of errors or warnings reported (characters sent to stderr). Generally there is no benefit to the developer to receive more than 1000 errors/warnings at once.

An associated problem is that the compiler prints the full line for each error message. Therefore, if a single line contains multiple errors, the line is printed once every error found. A single line of char size n can create O(n^2) characters of error output.

This error might thwart the possibility of automated verification and matching of source code with bytecode, due to resource exhaustion by warning messages in the verification server.
The consequence is that a very small contract code when compiled may create a huge error file. For example, a 16 Kbyte source code can force the compiler to emit 130 Mbytes of errors, and compilation takes 5 minutes. A a 32 Kbyte source code generates 500 Gigabytes of output and takes 20 minutes to compile.

We've attached the file Exploit1.sol as an working example. The file structure is the following:

```
pragma solidity ^0.4.13;
 contract Exploit1 {

 function slow()
 {
 uint x;
 +x;+x;+x;+x;+x;+x;+x;+x;+x;+x;+x;+x;+x;+x;+x;+x;+x;+x;+x;….. continued..
 }
}
```

This code causes the online solidity compiler (transpiled to Javascript) to consume all browser memory (because of DOM filling) and hangs up the browser, and other nasty things, like blocking the ethereum online compiler forever until the browser local storage is manually cleared.

Recommendations

- Do not print the whole line for a warning/error in case the line is too long
- Set a maximum number of warnings/errors to report

SOL-002	O(n^3) compiler output blow-up by function name duplicates
Category	Compiler attacks
Total Risk	Low | Impact: Low | Likelihood: Medium | Effort to Fix: Low
Status	Fixed

Description

All contract methods that have the same name are reported as duplicates. Each duplicate pair is reported independently, so if a function has 8 duplicates, 28 error messages are created, each printing the two conflicting lines. When a single line contains n duplicates, the line is printed O(n^3) times.

The effects of this vulnerability are similar to the previous one, but the consequences are worse. The vulnerability was fixed in version 0.4.17.

We've attached the file 16 Kbyte Exploit2.sol as an working example. The file structure is the following:

```
pragma solidity ^0.4.13;
contract s {

function s();function s();function s();function s();function s();function
s();function s();function s();function s();function s();..... continued..
 }
}
```

As an example, a 2 Kbyte solidity contract forces the compiler to generate a 135 Mbyte error file. A 4 Kbyte file, more than 500 Mbytes of error output.

Recommendations

- Report a single error for all duplicates found.

SOL-003	RAM Blow-up by constants cycles
Category	Compiler attacks
Total Risk	Low | Impact: Low | Likelihood: Medium | Effort to Fix: Low
Status	Not fixed in 0.4.19+commit.c4cbbb05.Linux.g++

Description

The solidity compiler has code to detect if a set of constant definitions posses a cycle by cross-reference. By forcing the compiler to recompute the cycles over and over, the compiler needs to allocate huge amounts of RAM, and compilation time can be made arbitrary long.

We've attached the file Exploit3.sol as an working example. The file defines a chain of constant references of length 5000. Constant names are encoded using an alphabet of 62 characters. The file structure is the following:

```
pragma solidity ^0.4.16;

contract a {}
contract XX {
a constant A=B;a constant B=C;a constant C=D;a constant D=E;a constant
E=F;a constant F=G;a constant G=H;a constant H=I;a constant I=J;a constant
J=K;
a constant K=L;a constant L=M;a constant M=N;a constant N=O;a constant
O=P;a constant P=Q;a constant Q=R;a constant R=S;a constant S=T;a constant
T=U;
a constant U=V;a constant V=W;a constant W=X;a constant X=Y;a constant
Y=Z;a constant Z=_a;a constant _a=_b;a constant _b=_c;a constant _c=_d;a
constant _d=_e;
```
….. continued….

```
a constant FMQ=FMR;a constant FMR=FMS;a constant FMS=FMT;a constant
FMT=FMU;a constant FMU=FMV;a constant FMV=FMW;a constant FMW=FMX;a
```

```
constant FMX=FMY;a constant FMY=FMZ;a constant FMZ=FMa;
a constant FMa=FMb;a constant FMb=FMc;a constant FMc=FMd;a constant
FMd=FMe;a constant FMe=FMf;a constant FMf=FMg;a constant FMg=FMh;a
constant FMh=FMi;a constant FMi=FMj;a constant FMj=FMk;

a constant FMk = a(0x00);

}
```

This example forces the compiler to allocate 10 GB of RAM very rapidly. Also it will take a lot of time to finish compiling (e.g. days). As memory allocation is quadratic, the attacker can force the target system to start trashing. The source code is about 372 Kbytes.

Recommendations

- Rewrite cycle-finding algorithm to avoid copying states while traversing or set a limit in the depth of constants references.

SOL-004	RAM Blow-up by exponential steps in constant cycle findings
Category	Compiler attacks
Total Risk	Low | Impact: Medium | Likelihood: Medium | Effort to Fix: Low
Status	Not fixed in 0.4.19+commit.c4cbbb05.Linux.g++

Description

This issue originates in the same code as the previous one. In this case, we force the compiler to performs an exponential number of steps to find a cycle in constants definitions.

We've attached the file Exploit4.sol as an working example. In the example there are only 160 constant definitions. There are two constant chains, A and B, each constant in the chain is numbered. Each constants refers to two other constants, one for the chain A and another from the chain B, forming a DAG. The compiler will try to explore every possible path from the first contant to the last, which results in 2^{80} steps. The solidity source code is tiny, only 4 Kbytes.

The file structure is the following:

```
pragma solidity ^0.4.16;

contract XX {
int constant v0a=v1a+v1b;int constant v0b=v1a+v1b;
int constant v1a=v2a+v2b;int constant v1b=v2a+v2b;
int constant v2a=v3a+v3b;int constant v2b=v3a+v3b;
int constant v3a=v4a+v4b;int constant v3b=v4a+v4b;
int constant v4a=v5a+v5b;int constant v4b=v5a+v5b;
int constant v5a=v6a+v6b;int constant v5b=v6a+v6b;
```
….. continued….
```
int constant v76a=v77a+v77b;int constant v76b=v77a+v77b;
int constant v77a=v78a+v78b;int constant v77b=v78a+v78b;
int constant v78a=v79a+v79b;int constant v78b=v79a+v79b;
int constant v79a=v80a+v80b;int constant v79b=v80a+v80b;
int constant v80a=0;
int constant v80b=0;
}
```

Recommendations

- Rewrite cycle-finding algorithm to avoid exponential blow-up or set a limit in the depth of constants references.

SOL-005	Unbounded gas cost when deleting dynamically sized arrays
Category	Gas cost
Total Risk	Medium | Impact: Medium | Likelihood: Medium | Effort to Fix: Low
Status	Not fixed in 0.4.19+commit.c4cbbb05.Linux.g++

Description

The “delete” operation when applied to a dynamically sized array in Solidity generates code to delete each of the elements contained. If the array is large, this operation can surpass the block gas limit and raise an OOG exception. Also nested dynamically sized objects can produce the same results.

Recommendations

Warn the user of the implications of deleting dynamically sized arrays.

SOL-006	Duplicated super-constructor calls not reported
Category	Parsing
Total Risk	Low | Impact: Low | Likelihood: Low | Effort to Fix: Low
Status	Not fixed in 0.4.19+commit.c4cbbb05.Linux.g++

Description

Solidity has two methods to define super constructor calls: the first one is embedding the arguments in super contract definition and the second is by declaring a new constructor and adding the name and arguments for the super contract. If both methods are used concurrently, the constructor definition is used. However, the compiler does not warn the user that the first set of arguments has been overridden by the second.
The following example shows the problem. The parent constract is constructed with the argument 40, not 20.

```
pragma solidity ^0.4.16;

contract P1  {
     function P1(uint v)  {}
}

contract P2 is P1(20) {
      function  P2(uint v) P1(40) {
      }
}
```

Recommendations

Either prevent the user from defining two super-constructors or warn if one super-constructor overrides the other.

SOL-007	Error-prone Multi-Assignment with empty LValues
Category	Solidity Language Design
Total Risk	Low | Impact: Medium | Likelihood: Low | Effort to Fix: Low
Status	Not fixed in 0.4.19+commit.c4cbbb05.Linux.g++

Description

Solidity has a multi-assignment feature. This works for lists (or tuples) created explicitly or from function return values. A function can return a list of values and the caller can assign each returned value to a different variable (LValue).

Some examples are:

```
function return2E() returns(uint,uint) {
         return (1,2);
     }
...
var (a1,a2,a3) = (V0,V1,V2);
var (h1,) = return2E();
```

Solidity has three special assignment modes for these tuples: head empty LValue, tail empty LValue, and positional empty LValue. The tail empty LValue applies when the number of elements of the destination list is smaller than the number of elements of the source list, and the rightmost destination element is empty. This means that additional elements to the right (the tail) will be ignored after being computed. The head empty LValue applies when the number of elements of the destination list is smaller than the number of elements of the source list, and the leftmost destination element is empty. This means that the additional leftmost elements (the head) will be ignored after being computed. The positional empty LValue is the one normally expected: if the number of elements in the destination list is equal to the number of elements in the source list, then any empty source element means to ignore the element at that particular position.

Example of positional empty LValue:

```
var (,y,) = (V0,V1,V2);
```

Example of tail empty LValue:

```
var (g1,) = (return1(),return2());
```

Example of head empty LValue:

```
var (,d1, d2) = (V0,V1,V2,V3,V4,V5,V6);
```

However, if two or more positions are empty at one side, but the number of elements in source and destination list differs, it still means a head/tail empty LValue, as this example shows. This assignment corresponds to e2 = v3 and e3 = v4, even if it suggests e2=v2 and e3=v3:

```
var ( , ,e2, e3) = (V0,V1,V2,V3,V4);
```

Recommendations

We recommend that in case of head/tail empty LValues, no other side empty LValue can be specified. We also recommend that head/tail empty LValues are marked with three dots to differentiate from positional empty LValues, as in here:

```
var ( ... , e3, e4) = (V0,V1,V2,V3,V4);
```

SOL-008	CPU blow-up using huge bignums literals
Category	Compiler attacks
Total Risk	Low | Impact: Medium | Likelihood: Low | Effort to Fix: Low
Status	Not fixed in 0.4.19+commit.c4cbbb05.Linux.g++

Description

The Solidity compiler accepts numeric literals of arbitrary precision. But computing those literals can take arbitrary high time.
When the following sample contract is compiled, the compiler runs for several hours (we've not waited until it finishes):

```
contract BIGNUMTEST {
        function bignum() {
                uint c;
                c=1E10000000;
        }
}
```

Recommendations

Limit the size of numeric literals.

SOL-009	Output messages size blow-up using huge bignums literals
Category	Compiler attacks
Total Risk	Low | Impact: Medium | Likelihood: Low | Effort to Fix: Low
Status	Not fixed in 0.4.19+commit.c4cbbb05.Linux.g++

Description

The Solidity compiler accepts numeric literals of arbitrary precision. When a literal is assigned to a variable of shorter size, an error will be written to stderr containing the expanded representation of the literal.

For example, the following code, generates an error output of approximate size 600 KBytes:

```
contract BIGNUMTEST {
	function bignum() {
		uint c;
		c=1E600000;
	}
}
```

The huge error line is:
filename.sol:4:5: Error: Type int_const 1000000000000000000……….000000000
is not implicitly convertible to expected type uint256.

Since computing the 1E600000 integer takes approximately 30 seconds, the same output blow-up can be obtained by forcing multiple conversion errors.

The following example generates a 6 Mbyte output in only 6 seconds:

```
contract BIGNUMTEST {
	function bignum() {
		uint c;
		c=1E10000;c=1E10000;c=1E10000;
		c=1E10000;c=1E10000;c=1E10000;
		c=1E10000;c=1E10000;c=1E10000;
		...last line repeated 200 times …
		}
}
```

Recommendations

- Shorten literal constants by replacing intermediate digits by "..." when printing errors to stderr (e.g. 1000...000)
- Reduce the number of warnings/errors written to stderr

SOL-010	Easy underhanded code using false overrides
Category	Solidity Language Design
Total Risk	Low | Impact: Medium | Likelihood: Low | Effort to Fix: Low
Status	Not fixed in 0.4.19+commit.c4cbbb05.Linux.g++

Description

Solidity smart contracts often override a function defined in a parent contract to restrict it access, for example, by adding an onlyOwner() modifier. However function overrides only work when the exact function signature is used (same argument types and same return parameters). It's easy to make small changes in the function signature that may go unnoticed by a reviewer and that will expose the parent function without restrictions.
The following is an example contains a hidden backdoor by not providing the correct override:

```
pragma solidity ^0.4.13;

// String Helper contract library
library String {
    function equals(string memory _a, string memory _b) internal returns (bool) {
                bytes memory a = bytes(_a);
                bytes memory b = bytes(_b);
                if (a.length != b.length)
                       return false;

                for (uint i = 0; i < a.length; i ++)
                       if (a[i] != b[i])
                              return false;
                return true;
       }
}

contract MyEtherWallet {
  using String for string;
  event onWithdrawal(string logmsg,address a);
  function MyEtherWallet() {
  }
     // Basic function to withdraw an amount of funds from the Token contract
     function  withdrawFunds(string logmsg) returns(bool) {
         if (logmsg.equals("OWNER"))
            onWithdrawal(logmsg,msg.sender);
         return msg.sender.send(this.balance);
    }
}

contract MyProtectedWallet is MyEtherWallet {
  address owner;
  function MyProtectedWallet()  {
      owner = msg.sender;
  }
  // This function restricts withdrawals to the contract owner and
  // fixes log msg.
  function  withdrawFunds(String logmsg) returns(bool) {
   if (msg.sender!=owner)
     return false;

   return super.withdrawFunds("OWNER");
  }
}
```

The widrawalFunds() method that is supposed to override the parent, uses the String type instead of the string type, and therefore does not override the parent.
The same can be accomplished by replacing an int argument with an uint argument.

Recommendations

Modify the solidity language to require the keyword “override” as modifier to function definitions in these cases. The compiler should generate an error when attempting to compile a method with the override modifier which does not override a parent method.