Zeppelin Solutions

augur

Serpent Compiler Audit

Version 1.0.0

Presented by
Manuel Araoz
CTO, Zeppelin Solutions

With services offered to
Augur

July 24, 2017

Introduction

This document includes the results of the audit performed by the Zeppelin Solutions team on the Serpent language compiler, at the request of the Augur team. The audited code can be found in the public ethereum/serpent Github repository, and the version used for this report is commit ad53fa2a8a496448d58ef9137959b4a1e86b14d7.

The goal of this audit is to perform a review of the language, comment on its design, tooling and ecosystem, and its implications towards smart contracts security, perform a code review of the compiler code itself, and uncover bugs that may potentially compromise the compiled code.

We include recommendations toward improving the compiler, as well as things to keep in mind when programming contracts in the language.

Disclaimer

Note that as of the date of publishing, the contents of this document reflect the current understanding of known security patterns and state of the art regarding EVM code generation and execution. Given the size of the project, the findings detailed here are not to be considered exhaustive, and further testing and audit is recommended after the issues covered are fixed.

Methodology

The Serpent compiler was approached from several directions for the purpose of the audit. Besides carefully analyzing the C++ code of the compiler, the team went through the documentation, examples, and recommended tools for working with Serpent contracts. Several minimal example contracts were written to verify and expose the issues found, and were analyzed based on both the LLL and EVM assembly code generated, as well as their behavior at runtime using pyethereum's ethereum.tools.tester module[1], as recommended by the Serpent documentation for testing. Selected issues were also double-checked in a geth testnet node, deploying the code generated by Serpent using web3.

Structure of the document

This report contains a list of issues and comments on different aspects of Serpent: the parser, the code generator, the tooling, and the design of the language itself. Each issue is assigned a severity level based on the potential impact of the issue, as well as a small example to reproduce it and recommendations to fix it, if applicable. For ease of navigation, an index by topic and another by severity are both provided at the beginning of the report.

[1] Version 8931e4945348ffa6e21cd77dba23846341056410

About Zeppelin Solutions

Zeppelin Solutions is a leading technology firm in the blockchain industry, providing consulting, security audits, development and escrow services for organizations. Zeppelin Solutions has developed industry security standards for designing and deploying smart contract systems.

Our goal is to help clients do a token sale, develop a blockchain application or build Dapps in a secure and timely fashion. We have a world-class team with technology, cybersecurity, and business development experience, and partners and investors with investment, legal and regulatory skills.

Zeppelin Solutions is the creator and main contributor of OpenZeppelin, the standard framework for secure smart contract development, maintained by a community of 750+ developers distributed around the globe.

Over $250 million have been raised with our audited smart contracts as of writing. Companies we've worked with include Golem, Brave, Augur, Blockchain Capital, Status, Cosmos, and Storj, among others.

More info at: https://zeppelin.solutions/

Severity level reference

Every issue in this report was assigned a severity level from the following:

critical Critical severity issues need to be fixed as soon as possible.

high High severity issues will probably bring problems to the users of this project and should be fixed.

medium Medium severity issues could potentially bring problems and should eventually be fixed.

low Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

List of issues by severity

critical

Very low quality tool and language 10
Development stalled 10
No tests 11
Language is untyped 15
Invalid syntax is accepted by the parser and compiler 25
Compiler does not fail on non-initialized variables 30
Can overwrite local memory location when accessing array out of bounds 30
Can overwrite storage location when accessing array out of bounds 32

high

Not widely used 10
No internal functions 15
Difference between short and long strings 16
Python reserved keywords remain unimplemented and are interpreted as uninitialized variables 18
Special any functions are chained, while shared are overwritten 19
Unexpected ABI generated from syntax error in method signature 26
Inline return statements are ignored 27
Return is not a reserved keyword 27
Medium sized strings are parsed as variables 28
Can overwrite local memory location when using setch with out of bounds index 31

medium

Lack of documentation 11
Consider using Viper 12
Broken and confusing examples 12
Serpent's sha3 function is not the standard SHA-3 14
Integer operations are signed 16
+ operator on short strings does not concatenate 16
Bad send API design 17
Variable names can begin with digits 17
Python boolean constants are unimplemented 18
Serpent's send behavior is different to Solidity's send 18
Constructor cannot accept parameters 20

No way to declare minimum compiler version 20
No error reporting on non-existent commands 22
Non-standard ABI names generated from CLI 24
Parser internal tokens clash with user-defined ones 26
Can overwrite non initialized variables 32
Can declare arrays of negative size 32
Array sizes can overflow 33
Compiler gives no warning when accessing uninitialized arrays 33

low

No releases or git tags 11
Several documentation errors 13
Use of magic constants 21
Code mixes tabs and spaces 21
Inconsistent casing 21
Bad CLI parameter parsing 22
Bad CLI error handling 23
Makefile does not specify dependencies correctly 24
Length function does not check argument types 34
Local variables defined in shared or any functions leak into user functions 34
Inconsistent usage of outitems vs outsz 35
Array arguments should be a pointer to first element 36

List of issues by topic

General Observations **10**

Very low quality tool and language 10

Not widely used 10

Development stalled 10

No tests 11

No releases or git tags 11

Lack of documentation 11

Consider using Viper 12

Broken and confusing examples 12

Several documentation errors 13

Serpent’s sha3 function is not the standard SHA-3 14

Language Design **15**

Language is untyped 15

No internal functions 15

Integer operations are signed 16

Difference between short and long strings 16

+ operator on short strings does not concatenate 16

Bad send API design 17

Variable names can begin with digits 17

Python reserved keywords remain unimplemented and are interpreted as uninitialized variables 18

Python boolean constants are unimplemented 18

Serpent’s send behavior is different to Solidity’s send 18

Special any functions are chained, while shared are overwritten 19

Constructor cannot accept parameters 20

No way to declare minimum compiler version 20

Compiler Code Review **21**

Use of magic constants 21

Code mixes tabs and spaces 21

Inconsistent casing 21

Tooling **22**

No error reporting on non-existent commands 22

Bad CLI parameter parsing 22

Bad CLI error handling 23

Makefile does not specify dependencies correctly 24

Non-standard ABI names generated from CLI 24

Parser **24**

Invalid syntax is accepted by the parser and compiler 25
Unexpected ABI generated from syntax error in method signature 26
Parser internal tokens clash with user-defined ones 26
Inline return statements are ignored 27
Return is not a reserved keyword 27
Medium sized strings are parsed as variables 28

Code Generation **30**

Compiler does not fail on non-initialized variables 30
Can overwrite local memory location when accessing array out of bounds 30
Can overwrite local memory location when using setch with out of bounds index 31
Can overwrite storage location when accessing array out of bounds 32
Can overwrite non initialized variables 32
Can declare arrays of negative size 32
Array sizes can overflow 33
Compiler gives no warning when accessing uninitialized arrays 33
Length function does not check argument types 34
Local variables defined in shared or any functions leak into user functions 34
Inconsistent usage of outitems vs outsz 35

Pyethereum issues **36**

Array arguments should be a pointer to first element 36

Issue Descriptions and Recommendations

General Observations

Very low quality tool and language

critical

The Serpent compiler is a very low quality piece of software. It is untested, there's very little documentation, and has very poor error handling. Additionally, very few developers use it, and the open source repository and community is very inactive. The compiler also gives almost no assurances to developers, and things that should be errors are considered valid Serpent programs: redefining language keywords, using undeclared variables, and accessing arrays out of bounds. All this poses a security risk for developers using the tool. Consider using a more tested, documented, and widely-used programming language, like Solidity.

Not widely used

high

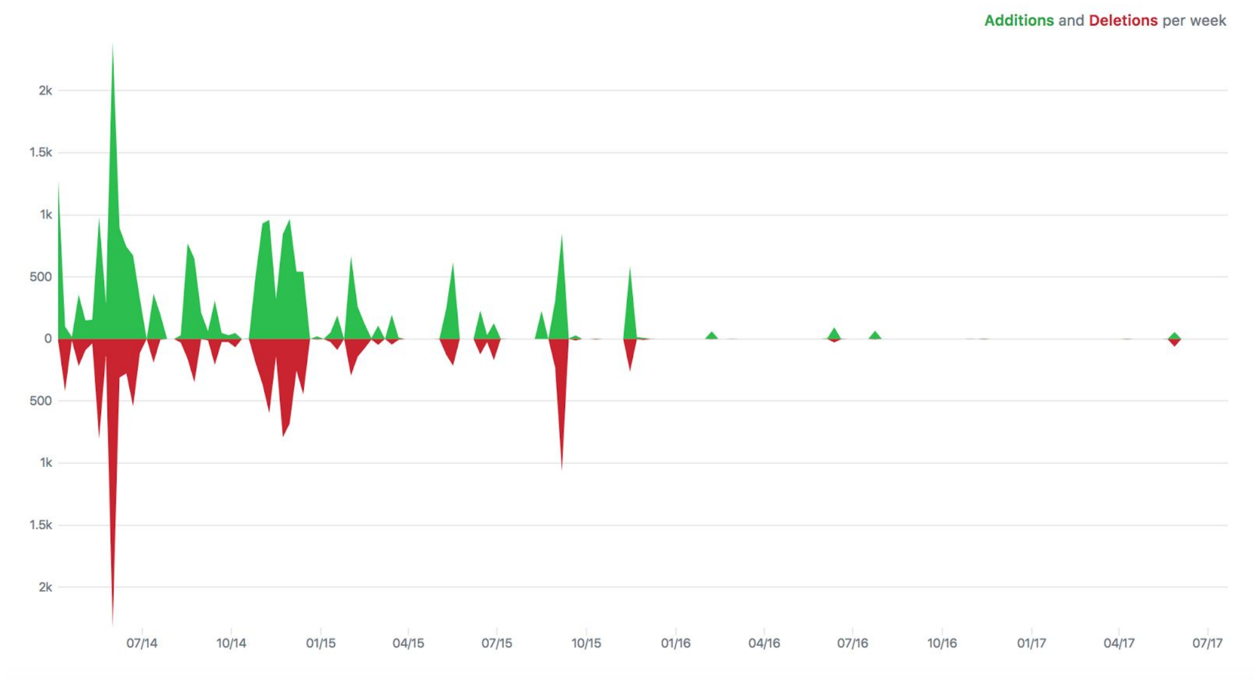
Since nobody uses the language there is very little possibility of people finding its problems, and their being fixed. There are only 185 Github repositories containing Serpent .se files, according to Github's public data as of this writing, compared to 2497 .sol from Solidity[2]. Furthermore, it seems to have been developed by one person alone: less eyes on the code means less bugs being noticed.

Development stalled

critical

The latest version of the language was released 2 years ago, and since October 2015 there have been very few commits to the repository.

[2] This is of course an approximation, as .sol files could be used in other contexts.



Source: Github

No tests

critical

Critical code such as that of a compiler for a smart contract language should have automated tests. Even though writing tests requires a significant workload, it greatly helps preventing regression problems and catching issues early. A regression bug appears when a previously correct component gets broken based on a recent change. The audited project has no tests, and as such is open to regression problems. Consider adding tests to have a more solid codebase.

No releases or git tags

low

The main GitHub repo has 0 releases and 0 tags. Consider publishing the current develop branch as a 1.0.0 release and organize a release schedule and methodology.

Lack of documentation

medium

The project's documentation is scarce and sparse. The README file is very short and seems to be only intended for Serpent project's developers, not users.
Sources we used as documentation for this audit are:

- https://github.com/ethereum/wiki/wiki/Serpent

- https://www.cs.umd.edu/~elaine/smartcontract/guide.pdf

Consider improving the compiler's documentation, especially the repository's README.md file.

Consider using Viper

medium

There's a successor to the Serpent language and compiler called Viper. Vitalik, creator of the Ethereum platform, said the following on this topic: "I definitely recommend starting to look at Viper over Serpent." (Source: https://ethereumclassic.github.io/blog/2017-03-13-viper.) Even though we haven't audited the Viper language, it looks superior to Serpent in many ways. It builds upon the two years of experience in the Ethereum ecosystem since Serpent development stopped. Consider deprecating Serpent in favor of Viper, and for projects using Serpent, migrating them to Viper. Consider, as well, contributing to Viper development and ensuring it follows better software engineering practices than its predecessor.

Broken and confusing examples

medium

Examples found in the repo are mostly broken and very confusing to new developers trying to learn the Serpent language. Take subcurrency.se, for instance:

```
def init():
    self.storage[msg.sender] = 1000000

def balance_query(k):
    return(self.storage[addr])

def send(to, value):
    fromvalue = self.storage[msg.sender]
    if fromvalue >= value:
        self.storage[from] = fromvalue - value
        self.storage[to] += value
```

Both `balance_query` and `send` functions are completely broken, even though they compile without any problems. In `balance_query`, the `addr` variable is not defined so it's equivalent to 0, and the value returned will always be `self.storage[0]`. In `send`, the `from` variable is not defined, resulting in the function only modifying `self.storage[0]`.
This is just one case, many other examples in the repository are broken and/or confusing for beginners.
Consider fixing existing examples, and writing simpler and more didactic ones.

Several documentation errors

low

Throughout the documentation in the official language wiki, as well as in the project's README, several instructions are incorrect or simply do not work. Given these resources are official and are most of the available documentation on the language, user onboarding becomes particularly difficult and frustrating. Consider going through the documentation and fixing all instructions to reflect the current state of the language.

README test instructions do not work

low

As mentioned in README.md line 11, tests are to be run via:

```
$ pip install tox -r requirements-dev.txt
```

This yields:

```
Collecting py_ecc (from ethereum==2.0.4->-r requirements-dev.txt (line 3))
  Could not find a version that satisfies the requirement py_ecc (from
ethereum==2.0.4->-r requirements-dev.txt (line 3)) (from versions: )
No matching distribution found for py_ecc (from ethereum==2.0.4->-r
requirements-dev.txt (line 3))
```

Error in example for running contracts

low

The instructions to run a sample contract in https://github.com/ethereum/wiki/wiki/Serpent are incorrect:

```
>>> from ethereum import tester as t
>>> c = t.Chain()
>>> x = s.contract('mul2.se')
>>> x.double(42)
```

1. Import fails
2. Variable s is undefined
3. Contract requires the language name as a named param

The code should be:

```
>>> from ethereum.tools import tester as t
>>> chain = t.Chain()
>>> c = chain.contract("examples/mul2.se", language='serpent')
>>> c.double(42)
```

Contract creation examples do not work

low

The wiki example for invoking CREATE does not work in pyethereum, the recommended environment, as it fails with a `TransactionFailed` error. The following contracts are to be created:

`mul2.se:`

```
def double(x):
    return(x * 2)
```

`returnten.se:`

```
extern mul2.se: [double:[int256]:int256]

MUL2 = create('mul2.se')
def returnten():
    return(MUL2.double(5))
```

And running in Python:

```
>>> from ethereum import tester as t
>>> c = t.Chain()
>>> x = c.contract('returnten.se')
>>> x.returnten()
10
```

Fails with an `ethereum.tools.tester.TransactionFailed` error.

Serpent's `sha3` function is not the standard SHA-3

medium

Ethereum uses KECCAK-256 as its hash function. The function was a candidate to be standardized as SHA-3, but different parameters were chosen for the standard in the end. Because of this, Serpent's `sha3` function is not really SHA-3 but KECCAK-256. This may cause confusion if comparing a Serpent-produced hash with one calculated externally using the standard function. The function was renamed `keccak256` in Solidity to prevent this; consider doing the same. More info at https://ethereum.stackexchange.com/a/554/7256.

Language Design

Language design issues are not bugs per-se, but rather design decisions or unimplemented features that make it easier for a user of the language to inadvertently introduce errors or security holes in their contracts.

Language is untyped

critical

Serpent is an untyped language: it allows any operation to be performed on any data. Every value is a 256 bit sequence which can be used as an address, a contract, an integer, or an array. Types have proven a useful feature to prevent programming errors, facilitate refactoring, and in general enable more robust software engineering. Serpent is far from robust. It is actually very close to an assembly language in the features and guarantees provided to its users.

Even more problematic is the fact that the language syntax *appears* to have type annotations on function arguments, but these are *not* used for type checking, they are only used externally: to construct the ABI of a contract, and to decide how to pass arguments to a function. Array arguments and return values have to be treated differently in the stack, so the language provides facilities to choose this special treatment for a certain value, irrespectively of whether it is correct or not to do so. The programmer might be led to believe that they're programming in a typed language, but will not be warned of errors they might introduce.

This is a hard issue to overcome, but we recommend adding a type-checking phase to the compiler, to reject incorrect programs.

Here is an example from the wiki page to illustrate this: *"Putting the :arr after a function argument means it is an array, and putting it inside a return statement returns the value as an array (just doing return([x,y,z]) would return the integer which is the memory location of the array)."*

No internal functions

high

All Serpent functions are external (public) and all function calls are compiled to EVM calls. This is a security problem if functions meant to be internal aren't properly guarded from being called by an external agent. It's also an issue of efficiency and gas costs, since an internal function call could be compiled as a jump rather than a call. These issues inhibit modularization. In fact, one of the Serpent tutorials proposes as a fix to not separate functions into smaller internal ones,

which is exactly opposite to good software development practice, and goes against our general security recommendations. Consider adding support for internal functions.
Here is an example of this problem, taken from a tutorial: *"Note that any function can be called directly by a user. For example, let's say we have a function A and a function B. If B has the logic that sends ether and A just does the check, and A calls B to send the ether, an adversary could simply call function B and get the ether without ever going through the checks."*

Integer operations are signed

medium

Integer operations are signed by default, i.e. operators will interpret the 256-bit values they operate on as representations of signed integers (using two's complement). In the case of addition and subtraction it makes no difference, and one uses the same operator regardless of sign. Inequality comparisons (less-than, greater-than, etc.) and division, however, behave differently if their operands represent signed or unsigned numbers. For example, the maximum representable 256-bit unsigned integer actually represents a negative integer when taken as signed, and therefore it's not the maximum.

The *operators* (`<`, `>`, `/`, ...) work on signed integers, and a different set of *functions* must be used to work on unsigned integers (`lt`, `gt`, `div`, ...). The programmer must be aware of this fact and make a conscious choice. The compiler will offer no help or indication of error. Since most operations on the Ethereum blockchain are usually on unsigned integers, consider adding a compiler flag to warn when a signed operator is used.

Difference between short and long strings

high

The Serpent language makes a difference between "short" and "long" strings. These are represented in two different ways: the former as integers, the latter as arrays of bytes. Short strings will lead to unexpected results when working with them, since they are actually integers (see the following entry). As a consequence of the language being untyped, one might use a short string where a long string is expected, and the compiler will not produce any warning. Consider removing short strings from the language, or requiring that they are not used in your codebase.

+ operator on short strings does not concatenate

medium

Since short strings are integers, `"foo" + "bar"` does not concatenate the strings as a user would expect, but actually add their integer values. We recommend either removing short strings altogether, or enforcing a difference in the compiler between short strings and integers for operations purposes, even if their internal representation is equivalent, to avoid this confusion by language users.

Example

```
def sum():
  concat = "foo" + "bar"
  return concat == "foobar" #=> Returns 0 (false)
```

Bad send API design

medium

The Serpent's equivalent to Solidity's send built-in function has a confusing API. The gas amount sent with the call can be set as an optional parameter *at beginning of a the send call*. This is very non-standard (optional parameters tend to be at the end of the parameter list) and can lead to confusion, especially in combination with the lack of type checking, as can be seen in the following examples.
As an example, the following three functions are valid Serpent code, though it is difficult to identify the behavior of each of them.

```
def foo1(x):
  send(x, 100)

def foo2(x):
  send(50, x, 100)

def foo3(x):
  send(50, x)
```

Consider changing the `send` API to be more robust: for example, optional parameters tend to be at the end of the parameter list.

Variable names can begin with digits

medium

Most high-level languages restrict valid variable names to be symbols that begin with a non-numeric character, to disambiguate with expressions such as scientific notation. For instance, the following code inadvertently returns 0, because 1e18 is an uninitialized variable.

```
def foo():
  return 1e18
```

The following code is also valid, and returns 3.

```
def foo():
  1e18 = 3
  return 1e18
```

Consider making variable names more strict.

Many Python reserved keywords remain unimplemented and are interpreted as uninitialized variables

high

Python keywords that affect control flow, such as break or continue, are not supported by the language, and are instead interpreted as uninitialized variables. Since Serpent aims to have the syntax of Python, probably to make it easier for beginners that already know the latter, this inconsistency will create unexpected results for such a user of the language.

Example

```
def foo():
  while i < 5:
    if i == 2: break
    i += 1
  return i
```

This example returns 5 instead of 2. Consider reviewing Python's list of reserved keywords and either implement them or raise an error when a user attempts to use them, to avoid confusion.

Python boolean constants are unimplemented

medium

Python's True and False constants are not implemented; as such, they are handled as uninitialized variables with value zero, so the token `True` is always falsey. The following code, for example, returns `2`.

Examples

```
def bar():
  if True:
    return 1
  else:
    return 2
```

Consider adding True and False as reserved keywords, and adding a rewrite rule for the compiler to output `1` and `0` for True and False respectively.

Serpent's send behavior is different to Solidity's send

medium

Explanation: The `send` function is the recommended way to send ether to an address, since it forwards a reduced amount of ether so as to mitigate the risk of a reentrant attack. In Solidity, send forwards a "stipend" of 2300 gas, that allows a receiving contract to perform some validation, log an event, etcetera. Serpent's `send` forwards 0 gas, which doesn't allow the receiving contract to execute any code at all. This mismatch could be a source of problems if not taken into account, for example, if a Serpent contract wants to send ether to a Solidity contract that logs an event in its fallback function.

Special any functions are chained, while shared are overwritten

high

Serpent provides two special kinds of functions: `any` and `shared`. These are executed before user functions (and `shared` is also executed before the `init` constructor). However, defining multiple `any` functions will cause all of them to be executed before each call to a user function, while multiple `shared` functions will cause only the last one to be executed. This can be confusing to Serpent users, and is not documented in the wiki or tutorial.

We recommend keeping the same semantics for both special functions, whether it is chaining or overwriting, and document them appropriately.

Example 1: any

In this example, `any` is defined twice, and executed twice before calling `foo()`.

```
data x

def init():
  self.x = 10

def any():
  self.x += 1

def any():
  self.x += 1

def foo():
  return self.x # => Returns 12
```

Example 2: shared

In this example, `shared` is defined twice, but is executed only once before calling `foo()`.

```
data x

def init():
  self.x = 10

def shared():
  self.x += 1

def shared():
  self.x += 1

def foo():
  return self.x # => Returns 11
```

Constructor cannot accept parameters

medium

The `init` function is purposely checked to have no arguments, thus making it impossible to inject values on construction. As such, additional methods are needed to set any initial state for the contract after creation, with additional checks to ensure that the same method is not used afterwards to freely alter its state. This is burdensome to the language user, and having to write more code typically makes room for more potential bugs.

We recommend removing this restriction from the `init` function, and allowing values to be passed in when creating an instance of the contract, as is supported by most OO languages (including Python) when creating a new instance of a class.

No way to declare minimum compiler version

medium

The Serpent compiler has no way of indicating minimum compiler version required by the source code. This makes it difficult to upgrade code if new features are added to the language. Consider adding a language version pragma (and Serpent versioning) similar to Solidity's version pragma.

Compiler Code Review

The following findings do not directly affect the language's end users, but are highlights of the compiler code, which affect code readability and maintainability.

Use of magic constants

low

There are several magic constants in the contract code. Some examples are:

- https://github.com/ethereum/serpent/blob/develop/compiler.cpp#L67
- https://github.com/ethereum/serpent/blob/develop/compiler.cpp#L188
- https://github.com/ethereum/serpent/blob/develop/compiler.cpp#L280
- https://github.com/ethereum/serpent/blob/develop/compiler.cpp#L295
- https://github.com/ethereum/serpent/blob/develop/compiler.cpp#L416

Use of magic constants reduces code readability, makes it harder to understand code intention, and hinders maintainability. We recommend extracting magic constants into constants.

Code mixes tabs and spaces

low

The Serpent compiler uses a mix of tabs and spaces in its codebase. Consider standardizing the style by using only tabs or only spaces. Our recommendation is to use spaces.

Inconsistent casing

low

The Serpent compiler code uses different style with regards to name casing. Some parts use camelCase and others use underscore_case. Consider unifying the casing style for better readability.

Example

array_lit_transform in line line 422 of rewriter.cpp vs **logTransform** line 436 of ewriter.cpp.

Tooling

Serpent's command line interface offers a number of commands, which are handled in cmdline.cpp, besides actual compilation of Serpent programs. However, most of the tasks listed are undocumented, and error handling for user input is very poor.

No error reporting on non-existent commands

medium

Running a non existent command does not raise any errors, or even return a non-zero exit code. This may cause an error on a batch script difficult to detect given the lack of feedback. We recommend displaying a warning to the user in standard error, and return a non-zero code as most command line utils do.

Example

```
./serpent fakecommand examples/short_namecoin.se
# No output
```

Bad CLI parameter parsing

low

The Serpent compiler CLI interface has error-prone and inconsistent parameter parsing. Flags that are usually expected from a CLI interface don't work or are wrongly interpreted by the tool. This makes the experience of using the compiler very frustrating. Consider implementing more standard CLI parameter parsing by using tools like Boost.Program_options and GNU getopt.

Example 1

```
./serpent compile -
```

Prints:

```
Error (file "main", line 1, char 0): Line malformed, not enough args for -
terminate called after throwing an instance of 'std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >'
Aborted (core dumped)
```

Example 2

```
 ./serpent compile --help
```

Prints:

“6100098061000e6000396100175660205160000360000035b6000f3”

Bad CLI error handling

low

CLI has very bad error handling. It is very easy to make the compiler break by sending the wrong parameters, since inputs to most calls are not validated, and the lack of documentation makes it even more difficult to work with the CLI. Consider improving error handling and documentation for the compiler's CLI.

Example 1

```
$ ./serpent pretty_compile_lll examples/short_namecoin.se
```

Prints: "examples"

Problem is that the command compiles LLL-lang inputs and it was given a .se serpent file. Error reporting is non-existent, the program just plainly prints "examples" for no apparent reason.

Example 2

```
$ ./serpent compile examples
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted (core dumped)
```

Example 3

```
$ ./serpent examples/subcurrency.se
Not enough arguments for serpent cmdline
terminate called after throwing an instance of 'int'
Aborted (core dumped)
```

Example 4

```
./serpent assemble examples/mul2.se
```

This yields a question mark on the console and returns with exit code 0.

Example 5

Compiler crashes if cycles exist between macros or if macros are self-referential.

foo.se:

```
macro A: B
macro B: A
A
```

```
$ ./serpent compile examples/foo.se
Segmentation fault (core dumped)
```

Makefile does not specify dependencies correctly

low

Using the Makefile to compile the project will fail under certain circumstances, such as compiling in parallel. The reason is that the `lib` target is missing the dependency `$(COMMON_OBJS)`. Add this missing dependency to fix the problem.

Non-standard ABI names generated from CLI

medium

Running `serpent mk_contract_info_decl foo.se`, on a contract with a function `foo(a,b,c)`, yields the following ABI definition:

```
[{
  "name": "foo(int256,int256,int256)",
  "type": "function",
  "constant": false,
  "inputs": [{ "name": "a", "type": "int256" }, { "name": "b", "type":
"int256" }, { "name": "c", "type": "int256" }],
  "outputs": [{ "name": "out", "type": "int256" }]
}]
```

According to the ABI spec, the ABI “name” field should include the name of the function, without parenthesis and the input types, which are used later when computing the function identifier.

Note that this is handled in web3, so everything works anyway when invoking functions with this ABI, but it is non-standard. Consider fixing `mk_contract_info_decl` to produce standard ABIs.

Parser

The Serpent parser has been manually implemented, as can be found in file parser.cpp. No automated tests for this parser could be found in the repository, and there is no specification for the syntax of the language that can be used as documentation or for formal validation.

Several issues were found regarding the parser itself, some of which become apparent on compilation, but others can silently produce results entirely unexpected to the user, yielding potentially critical security issues. Issues vary from accepting clearly invalid syntax to misinterpreting language tokens, and only some of the more representative are listed here.

We strongly recommend writing a formal grammar specification (see Python's BNF for an example) and using it to automatically generate a parser using existing tools, and/or a comprehensive test suite to validate the correctness and output of the parser, be it generated or written by hand. Alternatively, if the language syntax is largely to match Python's, working with a modified version of the Python parser could also be viable.

Invalid syntax is accepted by the parser and compiler

critical

As was mentioned, clearly invalid syntax is accepted by the parser, and the compiler generates bytecode from an invalid structure. Accepting invalid syntax and generating bytecode that does not correspond to the user's writings may open the door for any kind of issues, as the code to be executed does not match the user's intention.

The following code, for example, defines a function that accepts no arguments, and does not return any values.

```
def foo a, ()b:
  return a + b
```

Note that a Python interpreter correctly detects the error and warns the user:

```
  File "contract.se", line 1
    def foo a, ()b:
            ^
SyntaxError: invalid syntax
```

A complete rewrite of the parser is needed to handle this issues, preferably starting from a formal specification of the grammar, and backed by a strong suite of unit tests.

Unexpected ABI generated from syntax error in method signature

high

Similar to the previous issue, the parser does not recognise syntax errors and may generate an unexpected AST as a result, inducing errors in the ABI generation as well.

In the following example, the parser understands the argument name to be ":", as can be seen in the ABI, and also causes that the input value is not loaded into the "bar" symbol.

```
def foo(bar : str : str):
  return bar

contract.foo(1) #=> 0
```

The ABI specification generated by `mk_full_signature` is:

```
"name": "foo(bytes)",
"type": "function",
"constant": false,
"inputs": [{ "name": ":", "type": "bytes" }],
"outputs": [{ "name": "out", "type": "int256" }]
```

As with similar issues, the parser must be strict in rejecting invalid syntax, to prevent both invalid code and ABI to be generated.

Parser internal tokens clash with user-defined ones

medium

Given the internal implementation of the parser, it relies on a generating a fake "id" function as an auxiliary token, which is then erased in a successive pass. This causes that any user functions named "id" are erased as well, causing a compilation error if this function is invoked.

As an example, the following code crashes during compilation with the error "Invalid call: id"; renaming id to any other valid identifier compiles successfully.

```
def id(x):
  return x + 1

def foo():
  return self.id(42)
```

The same happens when parsing array accesses, as they are converted to an “access” function with “id” as a first parameter. As such, attempting to invoke a user defined function named “access” causes the compiler to crash, if its first parameter is named “id”. As an example:

```
def access(id):
  return id

def foo():
  return self.access(1)
```

Again, a rewrite of the parser is required to avoid the usage of tokens that may clash with user-defined ones, or better error reporting and documentation needs to be added on reserved keywords of the language.

Inline return statements are ignored

high

Explanation: A return statement is ignored if it is found in the same line as an if-statement; it is compiled as a variable instead. In order to be properly recognised as a return action, it must be placed in a newline. This leads to entirely unexpected behaviours of the program, as can be seen in the following examples. Note that both programs are equivalent in Python.

```
def foo():
  if 1: return 1
  return 2
  # returns 1

def foo():
  if 1:
    return 1
  return 2
  # returns 2
```

This kind of errors can be easily caught by a comprehensive test suite for the parser, and can be avoided by a careful grammar definition and using a parser generator rather than a manually written one. As such, we again recommend a full rewrite of the parser using automatic parser generation tools.

Return is not a reserved keyword

high

Explanation: The keyword `return` is a valid variable name, and as such can be used as one, potentially leading to confusion, as can be seen in the following example.

```
def bar():
  return = 42
  return return # => returns 42
```

We recommend defining a set of keywords for the language, and restricting their usage as variables, arguments or user defined functions, as most languages do. A reasonable candidate for such as list is Python's, given Serpent's similarity with it.

Medium sized strings are parsed as variables

high

While short strings are parsed as a number, and very long strings throw a "value too large" compilation error, medium sized strings are parsed as *variables*. This, combined with the fact that uninitialized variables have a value zero by default, may lead to inadvertently returning a zero value.

Example

The following code returns 0, since the string is parsed as a variable, and being uninitialized, has a value of zero.

```
def bar():
  return "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
```

Example 2

The following code returns 10, as the string is considered a variable, and assigned that value.

```
def bar():
  "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" = 10
  return "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
```

Example 3

The following correctly returns the numerical representation of "xxxx", which is 54490394922520587650080162453656890713529578349178079819725652483204479713280 L, but also assigns an unreachable variable named "xxxx" with the value 10.

```
def bar():
  "xxxx" = 10
  return "xxxx"
```

It is unclear whether such medium sized strings should be understood as integers like short strings, or should raise a compilation error like a very long string (without the `text` modifier) does; either way, failing silently and altering the semantics of a literal is not acceptable. We recommend either having a consistent and clearly documented maximum length for short strings, or removing them altogether from the language as they are a major source of confusion for users.

Code Generation

Code generation involves a first step of generating LLL (Low-level Lisp-like Language) from the parsed AST, then apply a set of rewriting rules found in rewriter.cpp, and finally compile it into EVM opcodes. Most sanity checks in the generated code can be found in compiler.cpp, which handles the opcode generation stage; however, most issues listed in this section are caused by missing checks, and should be fixed at a compiling stage.

Compiler does not fail on non-initialized variables

critical

Referencing a non initialized variable does not raise a compile error, and just returns 0. This leads to any misspelling causing a reference to the variable to silently fail, and yield potentially invalid values; it even affects some of the basic examples provided in the language, such as subcurrency.se (see "Broken and confusing examples" section). Furthermore, several other major issues we have detected in the language depend on this one, since the compiler usually generates references to zero when failing to resolve a symbol.

```
def foo():
  return x # => 0

def foo():
  return x + 2 # => 2

def foo():
  return getch(x, 2) # => Runtime error

def foo():
  stopped = true
  if (!stoped) do_dangerous_operation() # => performs dangerous operation
```

We strongly recommend to follow the lead of most of other major languages, including Python, and loudly fail (preferably at compile time, or at runtime at the very least) if a previously undefined token is referenced. Moreover, implicit declarations through variable assignment (which are valid in Python) should be disallowed, to further prevent potential issues caused by misspellings.

Can overwrite local memory location when accessing array out of bounds

critical

Arrays are unchecked for out-of-bounds accesses, both for reads and writes. This causes potential data leaks or overwrites by reading or modifying unexpected sections of memory. It is worth mentioning that Serpent does store the size of an array, which can be checked through the `len` function, but it is doesn't make any checks on accesses, even for statically sized arrays.

```
def foo():
  a = array(5)
  b = array(5)
  a[7] = 10
  return b[0] # => returns 10
```

Accesses are not even checked to be positive integers, and as such, previous memory positions can be read or altered by using negative indices. This is further aggravated by the fact that negative indices are legal in Python, and are used to access elements indexed from the back of the array; so a user trying to write pythonic code might inadvertently step into this issue.

```
def foo():
  a = array(5)
  b = array(5)
  b[-3] = 10
  return a[4] # => returns 10
```

Even though skipping checks on arrays access produce minor performance improvements, as in low level languages like C, which are translated to reduced gas cost in the EVM, the security implications of skipping such checks are so critical that basic checks for accessing an array within its bounds are absolutely required, even more when the user is presented with Python-like high level language, which traditionally protect the user against such issues.

Can overwrite local memory location when using `setch` with out of bounds index

high

The length of a long string is not checked when invoking `getch` or `setch`, so it is possible to set a value in another memory location by using a sufficiently large index in `setch.` This opens the door for several vulnerabilities that can exploit reading from or writing to arbitrary memory locations.

```
def foo():
  a = text("aaa")
  z = text("zzz")
  setch(a, 96, "a")
  return getch(z, 0) #=> returns 97 (“a”)
```

Basic checks should be implemented in the language to restrict out of bounds accesses, especially bearing such a similarity to Python, where the runtime ensures that such accesses are not performed.

Can overwrite storage location when accessing array out of bounds

critical

Finite arrays in the store are unchecked for out-of-bounds accesses, both for reads and writes. This opens the door to critical security vulnerabilities, since the very contract state could be altered by an incorrect or malicious access, potentially leaving the contract in an invalid and irrecoverable state.

```
data a[5]
data b[5]

def foo():
  self.a[5] = 10
  return self.b[0] # => returns 10
```

Checks for bounds when accessing an array in the contract storage should be generated by the compiler, to prevent unrestricted access to the contract state.

Can overwrite non initialized variables

medium

Since uninitialized variables refer to the start of the local memory heap, that position can be written using unbounded negative access to a string. Then, all references to an uninitialized variable would yield such a value. This may cause a misspelling to yield not zero but entirely unexpected values.

```
def foo():
  t = text("a")
  setch(t, -33, "b")
  return x # => returns 98 (“b”)
```

Since this issue is enabled by allowing uninitialized variables to be referenced, and by allowing unchecked access to an array or string, fixing such issues should resolve this one as well.

Can declare arrays of negative size

medium

Serpent allows arrays of negative size to be declared and used. This is a problem because negative-sized arrays make no sense. Consider adding a check to disable this option.

```
def foo():
  a = array(-1)
  a[0] = 10
  return a[0] # => returns 10
```

Array sizes can overflow

medium

Declaring an array of size[3] `2 ** 256 / 32 - 1` causes the LLL code generator to allocate a zero-size array, since array sizes are multiplied by 32 and added an extra memory position to store the array size, in order to allocate the required memory. Note that the compiler does enforce that all integers are less than 2 ** 256, but it fails to check for overflows when calculating memory sizes for allocating arrays. This causes any writes to an array that is expected to accommodate large values to end up silently overwriting other memory positions.

```
def foo():
  a = array(2 ** 256 / 32 - 1)
  b = array(1)
  b[0] = 2
  a[1] = 3
  return b[0] # => returns 3 instead of 2
```

The compiler should check that the requested size for an array can actually be requested, and does not generate an overflow when calculating the actual amount of memory required.

Compiler gives no warning when accessing uninitialized arrays

medium

Since uninitialized variables have value zero, accessing a position of an uninitialized array is equivalent to accessing that position of an array in memory position 0. This makes accessing an uninitialized array a way to read and write memory close to such position.

```
def foo():
```

[3] 3618502788666131106986593281521497120414687020801267626233049500247285301247

```
  a[5] = 10
  b[5] = 3
  return a[5] # => returns 3

def foo():
  b = 3
  a[2] = 10
  return b #=> returns 10
```

Accesses to uninitialized arrays, in both memory and storage, should fail at compile time, or at runtime at the very least, as should all references to uninitialized variables.

Length function does not check argument types

low

Since arrays are implemented as a pointer to the first element, with the size of the array stored in the previous position, len is implemented as accessing the previous position in memory. However, since len does not check for types, any variable or even literal can be used. This is a bug in the implementation and can have unexpected results for users of the compiler.

```
def foo():
  a = 10
  f = 96
  return len(f) # => returns 10

def foo():
  a = 10
  return len(96) # => returns 10
```

Consider making `len` only work on arrays.

Local variables defined in shared or any functions leak into user functions

low

A local variable defined in an `any` or `shared` function is visible from a user function; local variables should not be visible outside their scope. Consider making variables declared in `any` or `shared` functions only visible inside those functions.

Example (shared)

```
def shared():
  z = 2
```

```
def foo():
  return z # => Returns 2
```

Example (any)

```
def any():
  z = 2

def foo():
  return z # => Returns 2
```

Inconsistent usage of `outitems` vs `outsz`

low

`outitems` and `outsz` are synonyms in the language, though the documentation refers exclusively to `outsz`, while the compiler errors refer to `outitems`, thus being confusing to the user. The examples in the codebase use either of them, even within the same method. Always favour one way of doing things, especially in a Pythonic environment.

Pyethereum issues

These are not actual issues with the Serpent compiler, but are problems that arise from its usage via the pyethereum ethereum.tools.tester, which the Serpent documentation recommends using.

Array arguments should be a pointer to first element

low

In-memory arrays in Serpent are represented by the size plus the elements, and the actual array variable points to the first element. When passing an array as an argument from the pyethereum console, it points to the length instead to the first element. This does not happen when deploying the binary via other tools like truffle.

Example

Given the contract:

```
def foo(x : arr):
  return x[0]
```

Running:

```
contract.foo([10,11,12,13])
```

Returns 4 from pyethereum tester, but 10 from the truffle console connected to a testnet node.