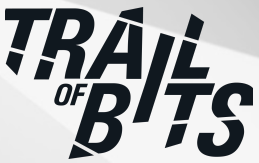

Solang Parser and Semantic Analysis

Security Assessment (Summary Report)

March 20, 2023

Prepared for:
Sean Young
Solana Labs

Prepared by: **Samuel Moelius and Anders Helsing**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Solana Labs engaged Trail of Bits to review the security of the Solang parser and semantics. From March 13 to March 17, 2023, a team of two consultants conducted a security review of the client-provided source code, with two person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered only issues of informational or undetermined severity that could impact system confidentiality, integrity, or availability.

Summary of Recommendations

In addition to the findings in this report, Trail of Bits recommends the following steps be taken.

- Reduce the amount of code duplication. There appears to be considerable code duplication within this codebase. Duplicated code can result in incomplete fixes or inconsistent behavior (e.g., if the code is modified in one location but not in all). (TOB-SOLANG-6)
- Increase the use of Clippy. Currently, Clippy appears to be run with only its default set of lints (`clippy:all`) enabled. However, several pedantic lints appear to produce valid warnings when run on the code. (TOB-SOLANG-3)
- Review all casts and eliminate those that are unnecessary. There are at least 63 casts within the source code, which seems excessive. Consider using `#[deny(clippy::as_conversions)]` and selectively allowing the lint where a cast is necessary and known to be safe. (TOB-SOLANG-4)
- Incorporate fuzzing into your software development process. Nearly all of the data a compiler operates on is user controlled. For this reason, compilers often benefit considerably from fuzzing. (TOB-SOLANG-7)

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	0
Low	0
Informational	6
Undetermined	3

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	2
Error Reporting	1
Patching	3
Testing	3

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineers were associated with this project:

Anders Helsing, Consultant
anders.helsing@trailofbits.com

Samuel Moelius, Consultant
samuel.moelius@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
March 13, 2023	Pre-project kickoff call
March 20, 2023	Delivery of report draft
March 20, 2023	Report readout meeting

Project Goals

The engagement was scoped to provide a security assessment of the Solang compiler parsing and semantic analysis phases. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are there unintended parser differentials between Solang and the Solidity compiler?
- Does the parser/lexer interpret Solidity files correctly?
- Are operations involving types implemented correctly?
- Are the existing testing strategies sufficient, or can they be extended?

Project Targets

The engagement involved a review and testing of the following target.

Solang parser

Repository	https://github.com/hyperledger/solang/tree/main/solang-parser
Version	`52879197d1cbc0e83c1b44de70639256d73de105`
Type	Rust
Platform	Solana

Solang semantics

Repository	https://github.com/hyperledger/solang/tree/main/src/sema
Version	`52879197d1cbc0e83c1b44de70639256d73de105`
Type	Rust
Platform	Solana

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Dependency review.** We ran `cargo upgrade --incompatible` over the codebase and reviewed the changes that the command applied to the `solana-ledger` crate.
- **Static analysis.** We ran static analysis tools and linters over the codebase and triaged the results.
- **Code review.** We reviewed the `lalrpop` definitions and the lexer from the `solang-parser` crate. We also began reviewing the semantic definitions focusing on problems stemming from type conversions and overflow conditions.
- **Fuzzing.** We changed cases of silent integer truncation to result in a panic on invalid conversions, such as expressions on the form "`x as u8`" into "`x.try_into()`", and compiled the contracts in the `fuzzy-sol` repository. We also fuzzed the lexer.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

1. Completing the review of semantic definitions with regard to type conversions and overflow conditions
2. Handling of data types that could differ between `solc` and Solang, including addresses, messages, and blocks
3. ABI encoding (`encodeWithSelector`, etc.)

We think the above could be completed in four person-weeks of effort. Note that this does not include the review of YUL.

We recommend that each of the following areas be the subject of their own audit, independent of the above.

1. YUL
   As the YUL implementation becomes more complete, this audit should focus on ensuring the security implications resulting from the use of Solidity inline assembly operating in a Solana context.

2. Code generation
   Apart from the problem of assuring correct interpretation of the Solidity contract code, this audit should focus on ensuring that the generated Solana binary does not exhibit problems common to those outlined in the Sealevel Attacks.

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Reliance on deprecated/unmaintained dependencies	Patching	Informational
2	Reliance on outdated dependencies	Patching	Informational
3	Inefficient linter use	Testing	Informational
4	Over-reliance on casts	Data Validation	Undetermined
5	Rational evaluation code always errors	Error Reporting	Undetermined
6	Code duplication	Patching	Informational
7	Arithmetic underflow in lexer	Data Validation	Undetermined
8	Excessive use of allow for lalrpop generated code	Testing	Informational
9	No explicit tests for operator precedence	Testing	Informational

Detailed Findings

1. Reliance on deprecated/unmaintained dependencies

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-SOLANG-1
Target: `Cargo.lock`	

Description

The Solang compiler relies on the crate `parity-wasm`, which has been declared "deprecated" by its author. Deprecated software is less likely to receive security updates. Hence, alternatives should be sought.

```
Crate:     parity-wasm
Version:   0.42.2
Warning:   unmaintained
Title:     Crate `parity-wasm` deprecated by the author
Date:      2022-10-01
ID:        RUSTSEC-2022-0061
URL:       https://rustsec.org/advisories/RUSTSEC-2022-0061
Dependency tree:
parity-wasm 0.42.2
├── wasmi-validation 0.4.1
│   └── wasmi 0.11.0
│       └── solang 0.2.2
└── wasmi 0.11.0

Crate:     parity-wasm
Version:   0.45.0
Warning:   unmaintained
Title:     Crate `parity-wasm` deprecated by the author
Date:      2022-10-01
ID:        RUSTSEC-2022-0061
URL:       https://rustsec.org/advisories/RUSTSEC-2022-0061
Dependency tree:
parity-wasm 0.45.0
└── solang 0.2.2
```

Figure 1.1: Results of running `cargo-audit`

Additionally, the Solang compiler relies on code generated by the `lalrpop` crate. In 2018, the crate's author stated that he did not "have the time to devote to LALRPOP that it really deserves," and sought to "form a core team." However, it is unclear whether such a team was formed. See for example issue #712 (Fork and add maintainers?) from March 14, 2023.

At present, we do not know of a good alternative to recommend over `lalrpop`, however.

Exploit Scenario
Eve learns of a vulnerability in the code generated by `lalrpop`. Knowing that the Solang compiler relies on code generated by `lalrpop`, Eve exploits the compiler's users.

Recommendations
Short term, switch from `parity-wasm` to `wasm-tools`, are recommended in the RUSTSEC advisory. Additionally, keep an eye out for a fork of `lalrpop`, or the announcement of a suitable alternative. Software that is actively maintained is more likely to receive security updates.

Long term, regularly run `cargo-audit` over your code base. Doing so will help to identify vulnerable or unmaintained dependencies.

References
- form a core team #290
- Fork and add maintainers? #712

2. Reliance on outdated dependencies

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-SOLANG-2
Target: `Cargo.toml`	

Description
Updated versions of many of the Solang compiler's dependencies are available. Because silent bug fixes are common, all dependencies should be periodically reviewed and updated wherever possible.

Note that some of these outdated dependencies have updated versions that are considered incompatible by Cargo; because of this, simply running `cargo update` will not cause them to be updated in the project's `Cargo.lock` file. Dependencies for which incompatible upgrades are available appear in table 2.1.

Dependency	Version currently in use	Latest version available
`contract-metadata`	1.5.1	2.1.0
`tower-lsp`	0.18	0.19.0
`anchor-syn`	0.26	0.27.0
`wasmi`	0.11	0.28.0
`rand (rand_07)`	0.7	0.8.5

Table 2.1: Dependencies for which incompatible upgrades are available

Exploit Scenario
Eve learns of a vulnerability in an outdated version of a Solang parser dependency. Knowing that Solana still relies on this outdated version, Eve exploits the vulnerability.

Recommendations
Short term, update the dependencies to their latest versions wherever possible. Verify that all unit tests pass following such updates. Document any reasons for not updating a dependency. Using out-of-date dependencies could mean critical bug fixes are missed.

Long term, regularly run `cargo upgrade --incompatible`. This will help ensure the project stays up to date with its dependencies.

3. Inefficient linter use

Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-SOLANG-3
Target: Various source files	

Description

The Solang parser appears to run Clippy with only the default set of lints (`clippy::all`) enabled. The maintainers should consider running additional lints, as many are triggered when enabled for the project.

Running Clippy with `-W clippy::pedantic` produces several hundred warnings. A selection that could have a positive performance impact on the project appears in figures 3.1 through 3.4.

```
warning: used `cloned` where `copied` could be used instead
   --> src/sema/contracts.rs:384:55
    |
384 | ...                   override_specified.iter().cloned().collect();
    |                                                 ^^^^^^ help: try: `copied`
    |
    = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#cloned_instead_of_copied
```

Figure 3.1: Sample warning produced by the `clippy::cloned_instead_of_copied` lint

```
warning: you seem to be using a `LinkedList`! Perhaps you meant some other data
structure?
  --> src/sema/symtable.rs:85:12
   |
85 |     names: LinkedList<VarScope>,
   |            ^^^^^^^^^^^^^^^^^^^^
   |
   = help: a `VecDeque` might work
   = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#linkedlist
```

Figure 3.2: Sample warning produced by the `clippy::linkedlist` lint

```
warning: this argument is passed by value, but not consumed in the function body
   --> src/sema/expression/function_call.rs:238:19
    |
```

```
238 |     function_nos: Vec<usize>,
    |                   ^^^^^^^^^^ help: consider changing the type to: `&[usize]`
    |
    = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#needless_pass_by_value
```

Figure 3.3: Sample warning produced by the `clippy::needless_pass_by_value` lint

```
warning: this argument (1 byte) is passed by reference, but would be more efficient
if passed by value (limit: 8 byte)
   --> src/sema/yul/expression.rs:100:12
    |
100 |     value: &bool,
    |            ^^^^^ help: consider passing by value instead: `bool`
    |
    = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#trivially_copy_pass_by_ref
```

Figure 3.4: Sample warning produced by the `clippy::trivially_copy_pass_by_ref` lint

Exploit Scenario

Eve uncovers a bug in the Solang parser. The bug might have been caught by the Solang compiler developers had additional lints been enabled.

Recommendations

Short term, consider running Clippy with `-W clippy::pedantic` regularly. Addressing Clippy warnings generally tends to produce cleaner code, which in turn reduces the likelihood of the code containing bugs. (See also TOB-SOLANG-4 below.)

Long term, regularly review Clippy lints that have been allowed, to see whether they should still be given such an exemption. Allowing a Clippy lint unnecessarily could cause bugs to be missed.

4. Over-reliance on casts

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SOLANG-4
Target: `sema/expression/literals.rs`	

Description

Clippy lints `cast_sign_loss` and `cast_possible_wrap` warn about casts that could fail. When run on the Solan parser, the lints produce 63 warnings. Moreover, user controlled data can cause at least one of those casts to fail.

The cast in question, and its associated Clippy warning, appear in figure 4.1. When the Solang parser is run on file `38837.sol` from the `fuzzy-sol` repository, the code tries to cast the value 1508 to a u8, which is invalid.

```
warning: casting `usize` to `u8` may truncate the value
  --> src/sema/expression/literals.rs:70:29
   |
70 |             ty: Type::Bytes(length as u8),
   |                             ^^^^^^^^^^^^
   |
   = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#cast_possible_truncation
```

Figure 4.1: Example warning produced by the `clippy::cast_possible_truncation` lint

```
contract TokenResolver {
    bytes4 constant INTERFACE_META_ID = 0x01ffc9a7;
    bytes4 constant ADDR_INTERFACE_ID = 0x3b3b57de;
    bytes4 constant ABI_INTERFACE_ID = 0x2203ab56;
    bytes32 constant ROOT_NODE =
0x637f12e7cd6bed65eeceee34d35868279778fc56c3e5e951f46b801fb78a2d26;
    bytes TOKEN_JSON_ABI = '[{"constant":true,"inputs":[],"name":"name","output...';
```

Figure 4.2: fuzzy-sol/Messi-Q_Smart-Contract-Dataset/38837.sol#L20–L25 (The highlighted constant is 1508 characters long.)

Exploit Scenario

Eve discovers the invalid cast in 4.1. Eve writes a Solana program that appears safe, in part, because of a large constant it contains. However, Eve's contract actually uses only the first 256 bytes of that constant. Eve performs a rugpull and uses the bug to claim plausible deniability.

Recommendations

Short term, address the bug in figure 4.1 by adding an appropriate check. Doing so will address a bug that causes parts of large constants to be ignored.

Long term, review all casts. For each, either add an appropriate check, or document why they are guaranteed to be safe. Doing so will reduce the likelihood of bugs similar to the one in figure 4.1.

5. Rational evaluation code always errors

Severity: **Undetermined**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-SOLANG-5
Target: `sema/expression/arithmetic.rs, sema/eval.rs`	

Description

The code in figure 6.1 always produces an error. Unnecessary error messages cause user fatigue, reducing their effectiveness.

More specifically, the code in figure 6.1 constructs an `Expression::Equal`. That expression is passed to `eval_const_rational`, which switches on the expression's kind. `eval_const_rational` does not have a case to handle `Expression::Equal`, hence, an error is returned.

```
let expr = Expression::Equal {
    loc: *loc,
    left: Box::new(left.cast(&left.loc(), &ty, true, ns, diagnostics)?),
    right: Box::new(right.cast(&right.loc(), &ty, true, ns, diagnostics)?),
};

if ty.is_rational() {
    if let Err(diag) = eval_const_rational(&expr, ns) {
        diagnostics.push(diag);
    }
}
```

Figure 5.1: sema/expression/arithmetic.rs#L612–L622

```
/// Resolve an expression where a compile-time constant(rational) is expected
pub fn eval_const_rational(
    expr: &Expression,
    ns: &Namespace,
) -> Result<(pt::Loc, BigRational), Diagnostic> {
    match expr {
        ...
        _ => Err(Diagnostic::error(
            expr.loc(),
            "expression not allowed in constant rational number
expression".to_string(),
        )),
    }
}
```

Figure 5.2: sema/eval.rs#L167–L252

Recommendations

Short term, correct the code in figure 5.1 so that an error is returned only when necessary. Unnecessary error messages cause user fatigue, reducing their effectiveness.

Long term, develop additional tests for the `sema` module. Try to develop tests that exercise both “happy” (successful) and “sad” (failing) paths. Doing so could help to expose bugs like this one.

6. Code duplication

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-SOLANG-6
Target: Various source files	

Description

The Solang parser contains significant code duplication. Duplicated code can result in incomplete fixes or inconsistent behavior (e.g., if the code is modified in one location but not in all).

For example, the buggy code from figure 5.1 appears nearly verbatim in the following locations:

- sema/expression/resolve_expression.rs#L161–L173
- sema/expression/resolve_expression.rs#L202–L214
- sema/expression/resolve_expression.rs#L242–L254
- sema/expression/resolve_expression.rs#L282–L294

A second example concerns the handling of `base58` constants. Code for converting such constants appears in both `sema/expression/literals.rs` (figure 6.1, left) and `sema/types.rs` (figure 6.1, right). Since this is an area where the Solang parser behaves differently than `solc`, it warrants special attention. Consolidating the code into a single function will help to ensure that correct behavior is exhibited everywhere necessary.

```
match address.from_base58() {
    Ok(v) => {
        if v.len() != ns.address_length {

diagnostics.push(Diagnostic::error(
                *loc,
                format!(
                    "address literal {}
incorrect length of {}",
                    address,
                    v.len()
                ),
            ));
            Err(())
        } else {
            Ok(Expression::NumberLiteral
{
```

```
match string.from_base58() {
    Ok(v) => {
        if v.len() != ns.address_length {

ns.diagnostics.push(Diagnostic::error(
                loc,
                format!(
                    "address literal {}
incorrect length of {}",
                    string,
                    v.len()
                ),
            ));
        } else {
            seen_program_id =
Some(note.loc);
```

```
                loc: *loc,
                ty: Type::Address(false),
                value:
BigInt::from_bytes_be(Sign::Plus, &v),
            })
        }
    }

Err(FromBase58Error::InvalidBase58Length)
=> {

diagnostics.push(Diagnostic::error(
            *loc,
            format!("address literal
{address} invalid base58 length"),
        ));
        Err(())
    }

Err(FromBase58Error::InvalidBase58Charact
er(ch, pos)) => {
        let mut loc = *loc;
        if let pt::Loc::File(_, start,
end) = &mut loc {
            *start += pos;
            *end = *start;
        }

diagnostics.push(Diagnostic::error(
            loc,
            format!("address literal
{address} invalid character '{ch}'"),
        ));
        Err(())
    }
}
```

```
ns.contracts[contract_no].program_id =
Some(v);
        }
    }


Err(FromBase58Error::InvalidBase58Length)
=> {

ns.diagnostics.push(Diagnostic::error(
            loc,
            format!("address literal
{string} invalid base58 length"),
        ));
    }


Err(FromBase58Error::InvalidBase58Charact
er(ch, pos)) => {
        if let pt::Loc::File(_, start,
end) = &mut loc {
            *start += pos + 1; //
location includes quotes
            *end = *start;
        }

ns.diagnostics.push(Diagnostic::error(
            loc,
            format!("address literal
{string} invalid character '{ch}'"),
        ));
    }
}
```

Figure 6.1: sema/expression/literals.rs#L270–L309 (left); sema/types.rs#L433–L466 (right, with some whitespace added)

Exploit Scenario

Alice, a Solang compiler developer, is asked to fix a bug in the handling of base58 constants. Alice does not realize that the bug also applies to more than just `sema/expression/literals.rs`. Eve discovers that the bug is not fixed in `sema/types.rs` and exploits it.

Recommendations

Short term, take the following steps:

- Ensure the bug from figure 4.1 is fixed in the bulleted locations above. Unnecessary error messages cause user fatigue, reducing their effectiveness.

- Consolidate the code for handing `base58` constants into a single function. Taking these steps will reduce the likelihood of an incomplete fix for a bug affecting both implementations.

Long term, adopt code practices that discourage code duplication. Doing so will help to prevent this problem from recurring.

7. Arithmetic underflow in lexer

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SOLANG-7
Target: `solang-parser/src/lexer.rs`	

Description

The Solang compiler's lexer contains an arithmetic underflow. This can cause a panic in debug builds of the compiler.

```
fn parse_number(
    &mut self,
    start: usize,
    end: usize,
    ch: char,
) -> Result<(usize, Token<'input>, usize), LexicalError> {
    let mut is_rational = false;
    ...
    if ch == '.' {
        is_rational = true;
        start -= 1;
    }
    ...
    if is_rational {
        end_before_rational = start - 1;
        rational_start = start + 1;
    }
```

Figure 7.1: solang-parser/src/lexer.rs#L560–L622

```
#[test]
fn parse_number() {
    let mut comments = Vec::new();
    let mut errors = Vec::new();

    let _ = Lexer::new(".9", 0, &mut comments, &mut errors)
        .collect::<Vec<Result<(usize, Token, usize), LexicalError>>>();
}
```

Figure 7.2: A test that triggers the arithmetic underflow in the code in figure 7.1

Recommendations

Short term, eliminate the arithmetic underflow bug that now occurs in the lexer. At a minimum, this will eliminate a panic that could occur in debug builds of the compiler.

Long term, incorporate fuzzing into your CI process. Doing so could help to uncover similar bugs.

8. Excessive use of allow for lalrpop generated code

Severity: **Informational**	Difficulty: **Low**
Type: Testing	Finding ID: TOB-SOLANG-8
Target: `solang-parser/src/lib.rs`	

Description

The Solang parser hides all Clippy warnings in the code generated from the `lalrpop` definitions. This can result in serious problems caused by future updates to `lalrpop` or the definitions being undetected.

```
#[allow(clippy::all)]
mod solidity {
    include!(concat!(env!("OUT_DIR"), "/solidity.rs"));
}
```

Figure 8.1: solang-parser/src/lib.rs#L19-L22

Recommendations

Short term, selectively allow the Clippy warning in the generated code. This ensures that new problems are still flagged by running Clippy.

Long term, use the allow-all directive sparingly, as can hide problems that otherwise would be detected by Clippy.

9. No explicit tests for operator precedence

Severity: **Informational**	Difficulty: **Low**
Type: Testing	Finding ID: TOB-SOLANG-9
Target: The `solang-parser` crate	

Description

The Solang parser does not have tests to verify the correct implementation of operator precedence by the `lalrpop` definitions. Expressing operator precedence using the `lalrpop` syntax is somewhat convoluted and, therefore, error-prone. While we did not uncover any problem with this during the engagement, future updates to the definitions could cause precedence issues.

Note, during testing, each instance of changes made to operator precedence caused tests to fail. However, because the tests failing where testing other properties than operator precedence, the resulting error messages did not hint that this was the underlying problem. I.e., the lack of explicit tests makes it hard to root-cause the cause of the test failing to be the result of improper operator precedence.

```
Precedence4: Expression = {
    <a:@L> <l:Precedence4> "*" <r:Precedence3> <b:@R> =>
Expression::Multiply(Loc::File(file_no, a, b), Box::new(l), Box::new(r)),
    <a:@L> <l:Precedence4> "/" <r:Precedence3> <b:@R> =>
Expression::Divide(Loc::File(file_no, a, b), Box::new(l), Box::new(r)),
    <a:@L> <l:Precedence4> "%" <r:Precedence3> <b:@R> =>
Expression::Modulo(Loc::File(file_no, a, b), Box::new(l), Box::new(r)),
    Precedence3,
}

Precedence3: Expression = {
    <a:@L> <l:Precedence2> "**" <r:Precedence3> <b:@R> =>
Expression::Power(Loc::File(file_no, a, b), Box::new(l), Box::new(r)),
    Precedence2,
}
```

*Figure 9.1: Example of how operator precedence for '**' is reversed from '*/%'.*

Exploit Scenario

The definition for the power operator (**), which, when reading the code, appears backward to the other operators. This is because of the breaking change in Solidity v0.8.0, making exponentiation right-associative. Alice, an inexperienced Solang developer, notices this difference and changes the definition to match the other operators.

Recommendations

Short term, add tests verifying the correct implementation of the operator precedences. This would add a safeguard to any future changes to the precedence definitions.

Long term, strive to cover as much of the parser and semantic definitions with tests as possible.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Non-Security-Related Findings

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

- Adopt an API that produces diagnostics consistently. In some places, diagnostics are pushed by the callee:

```
if let Some(loc) = call_args_loc {
    diagnostics.push(Diagnostic::error(
        loc,
        "call arguments not permitted for internal calls".to_string(),
    ));
}
```

  And in some places, diagnostics are returned and pushed by the caller:

```
if let Err(diag) = eval_const_rational(&expr, ns) {
    diagnostics.push(diag);
}
```

  The mix of styles can be jarring for readers of the code.

- Separate `lexertest` into multiple tests:

```
#[test]
fn lexertest() {
    ... // Nearly 500 lines
}
```

  Using a single, monolithic test can make it difficult to determine the cause of errors and can hamper development.