Solang Code Generation

Security Assessment (Summary Report)

July 12, 2023

Prepared for:
Sean Young
Solana Labs

Prepared by: **Samuel Moelius and Vara Prasad Bandaru**

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

About Trail of Bits **1**

Notices and Remarks **2**

Table of Contents **3**

Executive Summary **4**

Project Summary **5**

Project Goals **6**

Project Targets **7**

Summary of Findings **8**

Detailed Findings **10**

1. Dependency with open RUSTSEC advisory 10
2. Outdated dependencies 12
3. Insufficient test coverage 14
4. Tests do not pass with latest stable Rust 16
5. Strength reduction does not properly handle undefined variables 17
6. Solang fails to compile struct containing dynamic-sized arrays of its own type 20
7. Monolithic test 22
8. Optimizations hide errors contracts 24
9. Solang compiled contracts can have multiple storage accounts 25
10. An attacker can reinitialize a Solang contract 27
11. Compiler does not verify the developer specified size for the data account 29
12. The bump is not guaranteed to be at the end of seeds array 31
13. Appending state variables to Solang contracts affects their storage layout 33

A. Vulnerability Categories **35**

B. Non-Security-Related Findings **37**

Executive Summary

Engagement Overview

Solana Labs engaged Trail of Bits to review the security of Solang's `codegen` module, specifically in how it generates Solana code.

A team of two consultants conducted the review from June 23 to July 12, 2023, for a total of four engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

As discussed under TOB-SOLCG-3, there are no tests to verify that unoptimized and optimized code behave the same. During the project kickoff call, the Solang team described an improperly applied optimization as a "worst case scenario." Having tests to help verify the optimization passes' correctness is the best way to defend against such possibilities. Hence, we highly recommend that such tests be added.

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	1
Low	3
Informational	6
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	5
Patching	2
Testing	3
Undefined Behavior	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:

Samuel Moelius, Consultant
samuel.moelius@trailofbits.com

Vara Prasad Bandaru, Consultant
vara.bandaru@trailofbits.com

Project Timeline

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

Date	Event
June 15, 2023	Technical Onboarding call
June 23, 2023	Pre-project kickoff call
June 30, 2023	Status update meeting #1
July 12, 2023	Delivery of report draft
July 12, 2023	Report readout meeting

Project Goals

The engagement was scoped to provide a security assessment of the Solang's `codegen` module, specifically in how it generates Solana code. We sought to answer the following non-exhaustive list of questions:

- Does code emitted by the codegen module preserve the semantics of the original source code?
- Are optimizations applied under appropriate circumstances?
- Do optimizations preserve the semantics of the unoptimized code?
- Does Solang's codegen strategy introduce behavior that would be surprising to Solidity or Solana developers?

Project Targets

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

Solang codegen module

Repository	https://github.com/hyperledger/solang/tree/main/src/codegen
Version	`a84b0ad3b67a17b524ef6b7437fd4c5376833807`
Type	Rust/Solidity
Platform	Solana

Summary of Findings

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

ID	Title	Type	Severity
1	Dependency with open RUSTSEC advisory	Patching	Informational
2	Outdated dependencies	Patching	Informational
3	Insufficient test coverage	Testing	Informational
4	Tests do not pass with latest stable Rust	Testing	Informational
5	Strength reduction does not properly handle undefined variables	Data Validation	Low
6	Solang fails to compile struct containing dynamic-sized arrays of its own type	Data Validation	Low
7	Monolithic test	Testing	Informational
8	Optimizations hide errors contracts	Undefined Behavior	Informational
9	Solang compiled contracts can have multiple storage accounts	Data Validation	High
10	An attacker can reinitialize a Solang contract	Data Validation	High
11	Compiler does not verify the developer specified size for the data account	Data Validation	Medium
12	The bump is not guaranteed to be at the end of seeds array	Undefined Behavior	Low

13	Appending state variables to Solang contracts affects their storage layout	Undefined Behavior	**High**

Detailed Findings

1. Dependency with open RUSTSEC advisory	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-SOLCG-1
Target: `Cargo.lock`	

Description
The `borsh` dependency (which the `codegen` module relies upon) has an outstanding RUSTSEC advisory. A fix has been merged, but apparently not released. Solang should use an updated version of `borsh` as soon as one is released with the fix.

The following is an excerpt from the RUSTSEC advisory:

> *Affected versions of borsh cause undefined behavior when zero-sized-types (ZST) are parsed and the Copy/Clone traits are not implemented/derived. For instance if 1000 instances of a ZST are deserialized, and the ZST is not copy (this can be achieved through a singleton), then accessing/writing to deserialized data will cause a segmentation fault.*
>
> *There is currently no way for borsh to read data without also providing a Rust type. Therefore, if not [sic] ZST are used for serialization, then you are not affected by this issue.*

A fix was merged on June 7, 2023. However, as of this writing, the fix does not appear in any release.

Note: `cargo-audit` warns about dependencies besides `borsh`. However, none of those dependencies are used by the `codegen` module.

Exploit Scenario
Alice, a Solang developer, writes a test that uses zero sum types. Eve learns of this fact, and exploits the bug on Alice's machine.

Recommendations
Short term, watch the `borsh` repository, and switch to a new version of `borsh` as soon as one is released with the fix. Doing so will help ensure that Solang developers and users do not use vulnerable dependencies.

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

References

- RUSTSEC-2023-0033: Parsing borsh messages with ZST which are not-copy/clone is unsound
- BorshDeserialize can cause UB by copying zero sized objects with no safe Copy impl
- Forbid Zero-sized types from deserialization

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

Description
Updated versions of many of the `codegen` module'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
`itertools`	0.10.5 (Sep 18, 2022)	0.11.0 (Jun 22, 2023)
`indexmap`	1.9.3 (Mar 24, 2023)	2.0.0 (Jun 23, 2023)
`anchor-syn`	0.27.0 (Mar 8, 2023)	0.28 (Jun 9, 2023)

Table 2.1: Dependencies for which incompatible upgrades are available

Note: Dependencies besides those of table 2.1 can be upgraded. However, none of those dependencies are used by the `codegen` module.

Exploit Scenario
Eve learns of a vulnerability in an outdated version of a `codegen` dependency. Knowing that the `codegen` module 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 that the project stays up to date with its dependencies.

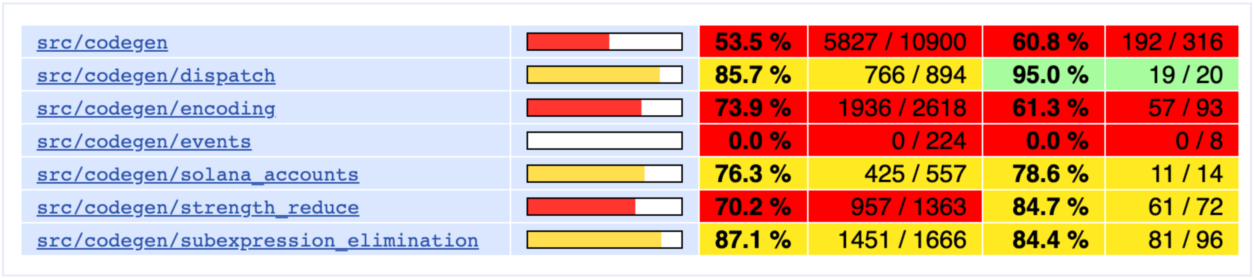
3. Insufficient test coverage

Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-SOLCG-3
Target: `tests` subdirectory	

Description

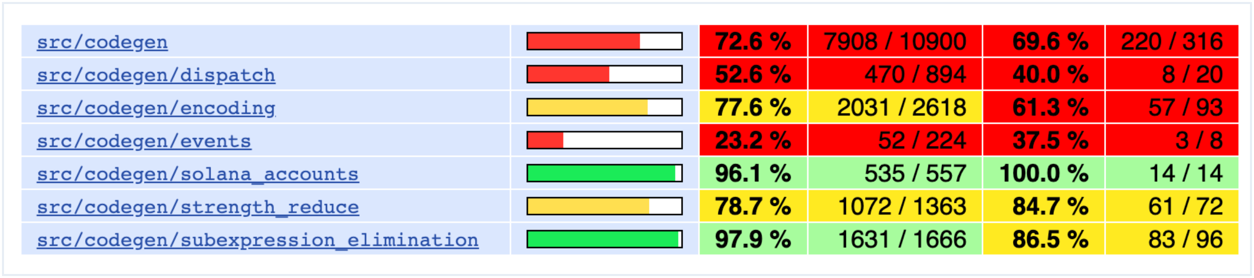
Much of the `codegen` module is not covered by any test. Most notably, code related to optimizations is inadequately tested.

The tests most applicable to generating Solana code are the codegen and `solana` tests. Figures 3.1 and 3.2 summarize the code covered by these tests, respectively.



src/codegen	53.5 %	5827 / 10900	60.8 %	192 / 316
src/codegen/dispatch	85.7 %	766 / 894	95.0 %	19 / 20
src/codegen/encoding	73.9 %	1936 / 2618	61.3 %	57 / 93
src/codegen/events	0.0 %	0 / 224	0.0 %	0 / 8
src/codegen/solana_accounts	76.3 %	425 / 557	78.6 %	11 / 14
src/codegen/strength_reduce	70.2 %	957 / 1363	84.7 %	61 / 72
src/codegen/subexpression_elimination	87.1 %	1451 / 1666	84.4 %	81 / 96

Figure 3.1: Code covered by the `codegen` test. The four rightmost columns are: percentage of lines covered, number of lines covered, percentage of functions covered, number of functions covered.



src/codegen	72.6 %	7908 / 10900	69.6 %	220 / 316
src/codegen/dispatch	52.6 %	470 / 894	40.0 %	8 / 20
src/codegen/encoding	77.6 %	2031 / 2618	61.3 %	57 / 93
src/codegen/events	23.2 %	52 / 224	37.5 %	3 / 8
src/codegen/solana_accounts	96.1 %	535 / 557	100.0 %	14 / 14
src/codegen/strength_reduce	78.7 %	1072 / 1363	84.7 %	61 / 72
src/codegen/subexpression_elimination	97.9 %	1631 / 1666	86.5 %	83 / 96

Figure 3.2: Code covered by the `solana` test. The four rightmost columns are: percentage of lines covered, number of lines covered, percentage of functions covered, number of functions covered.

Note that none of the tests in the `solana` test are specific to code generation. In particular, there appears to be no test that does the following:

- Compile a Solidity program with optimizations disabled.
- Run the resuling binary on one or more test vectors.

- Compile the same program with optimizations enabled.
- Run the resulting binary on the same set of test vectors.
- Verify that the two binaries' outputs are equal.

Ideally, this test would operate on a large number of Solidity programs, and would have many test vectors for each.

Exploit Scenario
A bug is found in an optimization pass. The bug could have been exposed by more thorough unit or integration tests.

Recommendations
Short term, add tests to compile code with and without optimizations, and verify that the resulting binaries behave similarly. Doing so will help increase confidence in the code that performs optimizations.

Long term, regularly compute and review test coverage using a tool such as `cargo-llvm-cov`. Doing so will help ensure that the tests are relevant and that all important conditions are tested.

4. Tests do not pass with latest stable Rust

Severity: **Informational**	Difficulty: **Undetermined**
Type: Testing	Finding ID: TOB-SOLCG-4
Target: `tests` subdirectory	

Description

The tests do not pass when built with the latest version of the Rust compiler (1.70.0). To ensure the code can benefit from compiler bug fixes, the code should be kept up-to-date with the latest stable Rust.

An error message produced by running the `solana` test compiled with Rust 1.70.0 appears in figure 4.1.

```
thread 'solana_tests::abi_decode::decode_address' panicked at 'misaligned pointer
dereference: address must be a multiple of 0x8 but is 0x7f724841a82c',
.../solana_rbpf-0.2.38/src/interpreter.rs:270:26
```

Figure 4.1: Error message produced by running the `solana` test compiled with Rust 1.70.0

Exploit Scenario

Rust version 1.70.1 fixes a critical bug in the compiler. Because Solang cannot be compiled with Rust 1.70.0, Solang does not benefit from the bug fix. Eve notices this and exploits the Solang instance running on Alice's machine.

Recommendations

Short term, diagnose and fix all tests that do not pass when compiled with Rust 1.70.0. Doing so will allow the code to benefit from fixes to the current stable version of Rust, and will ease the transition to the next version.

Long term, regularly test the code with the latest stable Rust. Doing so will help the code to benefit from compiler bug fixes.

5. Strength reduction does not properly handle undefined variables

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SOLCG-5
Target: `codegen/cfg.rs`	

Description

The strength reduction optimization runs even when undefined variables are present. This can result in an assertion violation and a panic.

The panic can be observed by making the change depicted in figure 5.1. The panic occurs in the code in figure 5.2. Two other parts of the call chain appear in figures 5.3 and 5.4. (Several call frames that would appear between figures 5.3 and 5.4 are omitted.) Note the comments in figure 5.4, which appears to not accurately reflect the current code.

```
contract MyTest {
    // BEGIN-CHECK: MyTest::MyTest::function::test_this__uint32_address
    function test_this(uint32 i, address addr) public view returns (uint32) {
        AccountInfo info = tx.accounts[i];
        if (info.key == addr) {
            // CHECK: branchcond ((load (load (struct %info field 0))) == (arg #1)),
block3, block4
            return 0;
        } else if (info.lamports == 90) {
```

Figure 5.1: *`tests/codegen_testcases/solidity/load_account_info_members.sol#L5–L12`*
Changing the highlighted = to `;` makes `info` undefined and causes a panic.

```
impl Type {
    /// Default value for a type, e.g. an empty string. Some types cannot have a
default value,
    /// for example a reference to a variable in storage.
    pub fn default(&self, ns: &Namespace) -> Option<Expression> {
        match self {
            ...
            Type::Ref(ty) => {
                assert!(matches!(ty.as_ref(), Type::Address(_)));
                ...
            }
```

Figure 5.2: *`codegen/statements.rs#L1440–L1488`*

```
impl Type {
    /// Default value for a type, e.g. an empty string. Some types cannot have a
pub(super) fn expression_values(
    expr: &Expression,
    vars: &Variables,
    ns: &Namespace,
) -> HashSet<Value> {
    match expr {
        ...
        Expression::Undefined { ty } => {
            // If the variable is undefined, we can return the default value to
optimize operations
            if let Some(default_expr) = ty.default(ns) {
                return expression_values(&default_expr, vars, ns);
            }

            HashSet::new()
        }
```

Figure 5.3: `codegen/strength_reduce/expression_values.rs#L13-L84`

```
/// Detect undefined variables and run codegen optimizer passess
pub fn optimize_and_check_cfg(
    cfg: &mut ControlFlowGraph,
    ns: &mut Namespace,
    func_no: ASTFunction,
    opt: &Options,
) {
    reaching_definitions::find(cfg);
    if func_no != ASTFunction::None {
        // If there are undefined variables, we raise an error and don't run
optimizations
        if undefined_variable::find_undefined_variables(cfg, ns, func_no) {
            return;
        }
    }
    if opt.constant_folding {
        constant_folding::constant_folding(cfg, ns);
    }
    if opt.vector_to_slice {
        vector_to_slice::vector_to_slice(cfg, ns);
    }
    if opt.strength_reduce {
        strength_reduce::strength_reduce(cfg, ns);
    }
```

Figure 5.4: `codegen/cfg.rs#L1539-L1561`

Exploit Scenario

Alice tries to compile her code using the Solang compiler. The compiler crashes without producing any useful diagnostics.

Recommendations

Short term, eliminate the assertion failure that can occur in the code in figure 5.1. Doing so will eliminate a panic that could occur in the `codegen` module.

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

6. Solang fails to compile struct containing dynamic-sized arrays of its own type

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SOLCG-6
Target: `sema` module	

Description

Solang considers structs containing multidimensional dynamic-sized arrays of its own type with fixed size innermost arrays to have infinite size, as a result, fails to compile them.

The structs containing a member of its own type or a fixed-size array of its own type are considered to have infinite size and the compilation of them is not possible. The structs containing dynamic-size arrays of its own type, irrespective of dimensions, should be considered to have finite size and compilation should be possible.

Figure 6.1 contains an example struct definition which contains a dynamic-sized array of its own type with a dynamic-size innermost array.

```
struct A {
    A[][1][2] b;
}
```

Figure 6.1: Example struct containing dynamic-sized array of its type with a dynamic-sized innermost array.

Solang correctly considers the struct to have finite size and successfully compiles them.

Figure 6.2 contains an example of struct definition which contains a dynamic-sized array of its own type but with a fixed-size innermost array. Solang fails to compile them with the error *"struct 'A' has infinite size"*.

```
struct A {
    A[2][1][] b;
}
```

Figure 6.2: Example struct containing dynamic-sized array of its type with fixed-sized innermost array.

Exploit Scenario

A contract contains a struct definition containing a dynamic-size array of its own type with a fixed-size innermost array similar to definition in figure 6.2. The compiler fails with the error “struct has infinite size”.

Recommendations

Short term, correct the handling of recursive structures, including allowing the code in figure 6.1. As the code is valid Solidity, it should be accepted.

Long term, improve tests for compilation of recursive structs. Doing so will help to identify problems like the one described here.

7. Monolithic test

Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-SOLCG-7
Target: `codegen/strength_reduce/tests.rs`	

Description

The `expresson_known_bits` test is approximately 1200 lines (figure 7.1). Large tests can prevent errors from being caught and can hamper future development.

```
#[test]
fn expresson_known_bits() {
    use crate::Target;
    use solang_parser::pt::Loc;
    ...
    ... // just under 1200 lines
    ...
    assert!(v.known_bits[0]);
    assert!(v.value[0]);
}
```

Figure 7.1: `codegen/strength_reduce/tests.rs#L29–L1230`

There are good reasons to break a large test up into multiple, smaller tests.

First, if a large test fails, it could be difficult for a developer to determine the cause. More specifically, if the test fails on the *n*th statement, it could be difficult for the developer to determine which of the *n-1* preceding statements contributed to the failure.

Second, an oft overlooked benefit of tests is that they serve as documentation. However, a monolithic test detracts from this benefit. Suppose a developer wants to know how to use statement X, which happens to be on line *n* of the test. If *n* is large, it could be difficult for the developer to determine which of the preceding *n-1* statements were necessary to use X.

Exploit Scenario

Alice, a Solang developer, makes a change to the code that causes the `expresson_known_bits` test to fail. The amount of time that Alice spends trying to determine the cause of the failure is more than it would have been had a smaller test failed.

Recommendations

Short term, break the `expresson_known_bits` test up into smaller tests. This will make determining the cause of failures easier and will help streamline future development.

Long term, consider enabling Clippy's `too-many-lines` lint and setting its lint level to `deny`. Doing so will help limit the size of future tests.

8. Optimizations hide errors contracts

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-SOLCG-8
Target: codegen module	

Description

The compiler does not raise an error for contracts containing undefined variables when optimizations are enabled. As a result, the developer might not be aware of incorrectness in their contracts.

The compiler runs the remove unused variables optimization before undefined variable detection. If the undefined variables are not used then the remove unused variables optimization will remove them and the undefined variable detection cannot find the error. However, when the optimizations are disabled, the undefined variable will not be removed and the compiler will raise the undefined variable error.

```
contract Test {
    struct A {
        uint256 b;
    }

    function test() public {
        A storage share;
        share.b = uint(10);
    }
}
```

Figure 8.1: Example contract containing undefined variable.

The `share` variable is undefined in the above contract. The compiler would raise the undefined variable error when the contract is compiled without optimizations. However, with the optimizations, the contract is compiled without any warnings or errors.

Recommendations

Short term, update the implementation to run undefined variables detection before performing any optimizations.

Long term, write tests to verify the equivalence of the code compiled with and without optimizations.

9. Solang compiled contracts can have multiple storage accounts	
Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SOLCG-9
Target: `codegen/solana_deploy.rs`	

Description

The compiler generated constructor code does not ensure the uniqueness of the contract's data account which might lead to account confusion issues where a data account different from the intended data account can be used.

The contract storage is represented using a data account. All the state variables are stored in that account. The constructor initializes the data account by writing the magic value in the first eight bytes of the account data. This magic value is used by the contract functions to verify that the correct data account is passed, ensuring that the correct account is used for storage.

The constructor does not prevent a user from creating multiple data accounts. Any user can call the constructor with a new account and the constructor will write the same magic value to the account. The new account can be used as the storage for the contract. This allows for use cases where a single deployment of the contract can be used for multiple instances of it, each with its own storage. All instances will have the same program id but different data accounts.

The disadvantage of this is that the users and protocols interacting with the contract have to ensure that the intended data account is being used by the contract, i.e., they are interacting with the intended instance of the contract.

This approach becomes an issue when a part of the contract's state is independent of the storage. For example, if the contract uses a PDA to interact with external contracts then that PDA can be considered to be part of the contract's state. The PDA address depends on the program id and a list of seeds. If the seeds are static and are fixed at the compile time, the derived PDA address will be independent of the contract's storage.

When the PDA address is independent of the contract storage and only depends on the code, all instances of the contract with different storage accounts will use the same PDA account. This creates an overlap between states of different instances of the contract. An attacker can exploit this by creating a new data account with storage favorable to them and using the PDA of existing instances to perform operations and profit from them.

Exploit Scenario

Consider the contract with the following description:

- The constructor sets the `owner` state variable to the caller given account.
- The contract owns tokens using the PDA derived from seeds `[“token owner”]`.
- The contract contains the `withdraw` function which when called by the `owner`, with the `owner` is a signer, transfers tokens owned by the PDA to the `owner` account.

Bob, the developer, deploys the contract and calls the contract with data account `A`. The `owner` value in account `A` is owned by Bob. After some time, with the normal usage of the contract, the PDA derived from `[“token owner”]` seed owns 1 million worth of tokens.

Eve, an attacker, calls the constructor with data account `B`. The `owner` value in account `B` is owned by Eve. Eve calls the `withdraw` function using the data account `B` and Bob’s PDA. Because the PDA does not depend on the storage, it will be the same for Eve’s instance as well. The withdraw function succeeds and Eve steals the tokens owned by Bob.

Recommendations

Short term, consider updating the compiler to ensure uniqueness of the data account for a given program id and the contract. This can be achieved by ensuring that the data account is a PDA derived using static seeds. If the feature is needed, add warnings to the developer documentation explaining the risks with the current approach. Also add the documentation for external protocols and users interacting with the Solang contract to verify the data account’s address.

Long term, document the design choices along with the assumptions made and perform a review to ensure that the selected design choices does not break the system invariants.

10. An attacker can reinitialize a Solang contract

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SOLCG-10
Target: `codegen/solana_deploy.rs`	

Description

The compiler generated constructor code does not check that a data account is already initialized. As a result, an attacker can call the constructor using the initialized data account and update important state variables.

The contract storage is represented using a data account. All the state variables are stored in that account. The constructor initializes the data account by writing the magic value in the first eight bytes of the account data.

Before running the initialization routines, the constructor does not check the account's magic value and proceeds with initialization. As a result, the state variables initialized in the constructor will be updated with the initial values and the caller provided arguments.

Exploit Scenario

```
contract Test {
    address owner;

    constructor(address admin) {
        owner = admin;
    }
    [...]

    function withdraw() public {
        // verify owner is signer and transfer all assets.
    }
}
```

Figure 10.1: Example contract vulnerable to this issue.

Bob, the developer, deploys the `Test` contract. He calls the constructor and sets the `owner` to his address. After some time, with continuous usage of the contract, the contract owns assets worth of 10 million USD.

Eve, an attacker, calls the constructor with her address as `admin`. The constructor updates the `owner` variable. Eve calls the `withdraw` function and steals 10 million USD worth of assets.

Recommendations

Short term, update the `solana_deploy` function to add initialization checks in the constructor code.

Long term, write a reference implementation in a high level language for every instance of compiler generated code written using low level `codegen` instructions. Review the high level reference implementation and ensure that the low level implementation is equivalent to the reference implementation.

11. Compiler does not verify the developer specified size for the data account

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SOLCG-11
Target: `codegen/solana_deploy.rs`	

Description

The constructor does not ensure the minimum size requirement for the data account while creating the account using the developer provided value. As a result, the data account could become unusable during the usage of the contract.

The constructor creates the data account if it is not given by the caller. The data account is required to have a certain minimum size. The developer can specify the data account size using the space annotation. The space value could be static, known during the compilation, or it could be dynamic, given as an argument. The compiler neither performs compile time checks nor adds run time checks for the space value. If the developer incorrectly calculates the required size or mistakenly provides the wrong value, the created data account could have smaller space than required.

The minimum size is referred to as the `contract.fixed_layout_size`. It represents the size required to store the contract's fixed size storage variables. If the data account has size less than `fixed_layout_size`, then only the first few variables can be read or written. All operations which require reading or writing the fixed size variable stored at the end will fail with out of bounds error.

Because only some of the operations might fail, the issue may not be caught during the early usage of the contract and the contract could become unusable in an intermediate state.

Exploit Scenario

```
contract Test {
    address owner;
    [...]
    bool withdrawn;

    @payer(...)
    @space(2000)
    constructor(address admin) {
        owner = admin;
    }
    function deposit() public { [...] }
    function withdraw() public {
        // verify owner is signer and transfer all assets.
        // The function writes to the `withdrawn` variable.
    }
}
```

Figure 11.1: Example contract vulnerable to the issue.

The `fixed_layout_size` for the above contract is `2048` bytes. Bob, the developer, mistakenly specifies `2000` bytes in the `space` annotation. The compiler compiles the code without any errors. Bob deploys the contract and calls the constructor. The constructor creates the data account with the size of `2000` bytes and initializes the account.

The deposit operations and other operations succeed without any errors. After some time, the contract accumulates assets. Bob tries to withdraw the assets using the `withdraw` function. The `withdraw` function writes to the `withdrawn` variable. The `withdrawn` variable is stored after the offset `2000` in the data account. The operation fails with out of bounds error. The funds are stuck in the contract.

Recommendations

Short term, update the `solana_deploy` function to check the `space` value during compilation if it is static and to add runtime checks to the constructor code if the space value is a runtime constant.

Long term, Implement the compiler to be strict and perform as many checks as possible. Develop the compiler with the assumption that the developer will make mistakes and write incorrect code.

12. The bump is not guaranteed to be at the end of seeds array

Severity: **Low**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-SOLCG-12
Target: `codegen/solana_deploy.rs`	

Description

The compiler, while constructing the seeds array using the constructor annotations, does not ensure that the bump value is placed at the end of the array. As a result, the computed account might not be a valid PDA and the contract initialization might fail.

The developer can specify the seeds and bump value for a PDA using the constructor annotations. The compiler uses the seeds in the specified order for signing the PDA account. It considers the bump value as just another seed value and includes it in the specified position.

```
for note in &func.annotations {
    match note {
        ConstructorAnnotation::Seed(seed) => {
            seeds.push(expression(seed, cfg, contract_no, None, ns, vartab, opt));
        }
        ConstructorAnnotation::Bump(bump) => {
            let expr = ast::Expression::Cast {
                loc: Loc::Codegen,
                to: Type::Slice(Type::Bytes(1).into()),
                expr: ast::Expression::BytesCast {
                    loc: Loc::Codegen,
                    to: Type::DynamicBytes,
                    from: Type::Bytes(1),
                    expr: bump.clone().into(),
                }
                .into(),
            };

            seeds.push(expression(&expr, cfg, contract_no, None, ns, vartab, opt));
        }
        _ => (),
    }
}
```

Figure 12.1: `codegen/solana_deploy.rs#L463-L484`

However, the bump value is expected to be the last seed and should be placed at the end of the array. The developer might work with the assumption that the compiler will place the bump value at the end irrespective of the position of its annotation. If the developer places a seed annotation after the bump annotation, the order of seeds used by the compiler will be different from the order expected by the developer.

Because the order of the seeds decides the derived PDA account, the derived address will be different than the expected and it might not be a valid account. The derived PDA account is needed for contract initialization. As a result, the contract might need to be redeployed after updating the position of bump annotation.

Exploit Scenario

```
contract Test {

    @payer(...)
    @space(64)
    @seed("A")
    @bump("x")
    @seed("B")
    constructor() { [...] }
    [...]
}
```

Figure 12.2: Example contract vulnerable to the issue.

Bob, the developer, expects the seeds array for the PDA to be `[“A”, “B”, “x”]`. The seeds used by the compiler for the PDA will be `[“A”, “x”, “B”]`. Bob provides the account derived from his seeds. The compiler tries to sign the instruction with the computed seeds resulting in a different PDA. The create account instruction is not signed by the account and the instruction fails.

The PDA derived using the compiler's order of the seed might not be valid PDA and the data account cannot be created using it. The contract needs to be redeployed with corrected annotations.

Recommendations

Short term, raise an error if the developer places the bump annotation before a seed annotation. Otherwise, consider placing the bump value at the end irrespective of the annotation's position.

Long term, Implement the compiler considering the expectations of the developer. Document the instances where the compiler diverges from these expectations.

13. Appending state variables to Solang contracts affects their storage layout

Severity: **High**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-SOLCG-13
Target: `https://solang.readthedocs.io`	

Description

Adding new state variables to the Solang contract will change the storage layout. This is different from Ethereum Solidity contracts. Developers not aware of the difference might brick their contract by updating it with a contract containing additional state variables.

The Solang contract uses Solana account data, a linear bytearray, for storage. The bytearray is divided into two sections using the offset `contract.fixed_layout_size`. The space from offset `0` to `fixed_layout_size` is used for storing fixed size storage variables. The space from the `fixed_layout_size` index is considered to be a heap and is used for storing dynamic size variables.

The `fixed_layout_size` depends on the contract's fixed size state variables. The state variables are stored in the defined order. Appending new fixed size variables would increase the `fixed_layout_size`. The new variables will be stored from the old `fixed_layout_size` offset.

As a result, if the contract is updated with a contract containing new fixed size state variables, the new variables will be stored in the heap space of the old contract. This corrupts the heap and results in a invalid state for the contract.

Exploit Scenario

Bob, the developer of a contract, adds new fixed size state variables to the contract and updates the old contract using the new contract. Bob executes a function which writes to the first variable of the new state variables. The first variable is stored at the start of the heap of the old contract. The function overwrites the heap and corrupts the contract's state. The contract becomes unusable.

Recommendations

Short term, add developer documentation to inform the issues with updating to a contract with new state variables.

Long term, list the differences between Ethereum Solidity contracts and Solang contracts. Review the effects of these differences and document the issues stemming from the differences.

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.

- **Reorganize the repository so that the root manifest is virtual, i.e., a workspace only.** Currently, the root manifest describes both a package and a workspace (figure B.1). The current organization complicates commands such as `cargo test`, as `--workspace` must be passed for the command to apply to the whole workspace, and not just the root package.

```
[package]
name = "solang"
...
[workspace]
members = ["solang-parser", "tests/wasm_host_attr"]
```

Figure B.1: `Cargo.toml#L1–L103`

- **Have the build script check that the correct version of `llvm-config` is referred to by PATH.** The expected version of `llvm-config` has the SBF target. The build script could run `llvm-config --targets-built` and verify that SBF appears in the output (see figure B.2). Currently, if the wrong `llvm-config` is referred to by PATH, the build script will complete without error.

```
$ llvm-config --targets-built
AArch64 AMDGPU ARM AVR BPF Hexagon Lanai Mips MSP430 NVPTX PowerPC RISCV SBF
Sparc SystemZ VE WebAssembly X86 XCore
```

Figure B.2: Output produced by the expected (patched) version of `llvm-config`

- **Adopt a consistent import format.** (See figure B.3.) Doing so will make it easier to determine what symbols are imported and from where. Rustfmt's (unstable) `imports_granularity` and group_imports configurations could help with this.

```
use self::{
    cfg::{optimize_and_check_cfg, ControlFlowGraph, Instr},
    dispatch::function_dispatch,
    expression::expression,
    solana_accounts::account_collection::collect_accounts_from_contract,
    vartable::Vartable,
};
...
use crate::codegen::cfg::ASTFunction;
```

Figure B.3: `codegen/mod.rs#L27–L43`

- **Run Clippy's pedantic lints in CI.** As previously reported (TOB-SOLANG-3 in the "Solang Parser and Semantic Analysis" report), Clippy's pedantic lints produce many warnings when applied to the codebase. Addressing them would improve the quality of the code. Example warnings appear in figures B.4 through B.7.

```
warning: redundant closure
    --> src/codegen/cfg.rs:1802:14
     |
1802 |         .map(|stmt| stmt.reachable())
     |              ^^^^^^^^^^^^^^^^^^^^^^^ help: replace the closure with the
method itself: `sema::ast::Statement::reachable`
     |
     = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#redundant_closure_fo
r_method_calls
```

Figure B.4: Warning produced by `redundant_closure_for_method_calls`

```
warning: implicitly cloning a `Vec` by calling `to_vec` on its dereferenced
type
    --> src/codegen/expression.rs:2519:16
     |
2519 |         value: id.to_vec(),
     |                ^^^^^^^^^^^ help: consider using: `id.clone()`
     |
     = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#implicit_clone
```

Figure B.5: Warning produced by `implicit_clone`

```
warning: used `cloned` where `copied` could be used instead
  --> src/codegen/solana_accounts/account_management.rs:21:14
   |
21 |             .cloned()
   |              ^^^^^^ help: try: `copied`
   |
   = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#cloned_instead_of_co
pied
```

Figure B.6: Warning produced by `cloned_instead_of_copied`

```
warning: it is more concise to loop over containers instead of using explicit
iteration methods
  --> src/codegen/strength_reduce/mod.rs:91:29
   |
91 |     for (block_no, vars) in block_vars.into_iter() {
   |                             ^^^^^^^^^^^^^^^^^^^^^^ help: to write this
more concisely, try: `block_vars`
   |
   = help: for further information visit
```

```
https://rust-lang.github.io/rust-clippy/master/index.html#explicit_into_iter_l
oop
```

Figure B.7: Warning produced by `explicit_into_iter_loop`

- **Eliminate the unnecessary use of `mut` in figure B.8.**

```
warning: variable does not need to be mutable
   --> src/codegen/subexpression_elimination/mod.rs:165:13
    |
165 |         let mut cur_block = &mut cfg.blocks[*block_no];
    |             ----^^^^^^^^^
    |             |
    |             help: remove this `mut`
    |
    = note: `#[warn(unused_mut)]` on by default
```

Figure B.8: Warning produced by `unused_mut`

- **Change the use of `borrow_mut` to `borrow` in figure B.9.** Using `borrow_mut` unnecessarily could result in a panic. (Note: `unnecessary_borrow_mut` is a Dylint lint.)

```
warning: borrowed reference is used only immutably
   -->
src/codegen/subexpression_elimination/available_expression_set.rs:368:43
    |
368 |         for (child_id, node) in &var_node.borrow_mut().children {
    |                                           ^^^^^^^^^^^^ help: use:
`borrow()`
    |
    = note: `#[warn(unnecessary_borrow_mut)]` on by default
```

Figure B.9: Warning produced by `unnecessary_borrow_mut`

- **Eliminate the unnecessary call to `as_bytes` in figure B.10.** (Note: `unnecessary_conversion_for_trait` is a Dylint lint.)

```
warning: the receiver implements the required traits
  --> src/codegen/events/solana.rs:34:23
   |
34 |         hasher.update(discriminator_image.as_bytes());
   |                       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ help: use:
`&discriminator_image`
```

Figure B.10: Warning produced by `unnecessary_conversion_for_trait`

- **Eliminate the duplicate dependencies that appear in the root manifest.** Packages sha2 and `tempfile` appear as both regular and "dev" dependencies. It is sufficient that they appear as just regular dependencies.

```
[dependencies]
...
tempfile = "3.4"
...
sha2 = "0.10"
...
[dev-dependencies]
...
sha2 = "0.10"
...
tempfile = "3.3"
```

Figure B.11: `Cargo.toml#L18–L85`

- **Eliminate the corner case that can cause `test_mul_within_range_signed` to fail.** If `first_operand_rand` is `-2^(N-1)` and `second_op` is `-1`, the multiplication will overflow.

```
#[test]
fn test_mul_within_range_signed() {
    let mut rng = rand::thread_rng();
    for width in (8..=256).step_by(8) {
        ...
        // The range of values that can be held in signed N bits is [-2^(N-1),
2^(N-1)-1]. Here we generate a random number within this range and multiply it
by -1, 1 or 0.
        let first_operand_rand = rng.gen_bigint(width - 1).sub(1_u32);
        println!("First op : {first_operand_rand:?}");

        let side = vec![-1, 0, 1];
        // -1, 1 or 0
        let second_op = BigInt::from(*side.choose(&mut rng).unwrap());
        println!("second op : {second_op:?}");
```

Figure B.12: `tests/solana_tests/primitives.rs#L989–L1011`

- **Replace the call to `BigUint::pow` followed by `truncate_biguint` (figure B.13) with just one call to `BigUint::modpow` (figure B.14).** Doing so will make the uint test more efficient.

```
let mut res = a.clone().pow(n);
truncate_biguint(&mut res, width);
```

Figure B.13: `tests/solana_tests/primitives.rs#L543–L544`

```
let res = a
     .clone()
     .modpow(&BigUint::from(n), &BigUint::from(2u64).pow(width as u32));
// truncate_biguint(&mut res, width);
```

Figure B.14: Proposed change to the code in figure B.13

- **Check both sides of the boundary condition in the code in figure B.15, i.e., add code like in figure B.16 to `transfer_fails_not_enough`.** Doing so will help increase confidence in the `transfer_fails_not_enough` test.

```
let res = vm.function_must_fail(
    "transfer",
    &[
        BorshToken::FixedBytes(new.to_vec()),
        BorshToken::Uint {
            width: 64,
            value: BigInt::from(104u8),
        },
    ],
);
assert!(res.is_err());
```

Figure B.15: `tests/solana_tests/balance.rs#L256-L266`

```
    let res = vm.function_must_fail(
        "transfer",
        &[
            BorshToken::FixedBytes(new.to_vec()),
            BorshToken::Uint {
                width: 64,
                value: BigInt::from(103u8),
            },
        ],
    );
    assert!(res.is_ok());
```

Figure B.16: Proposed change to the code in figure B.13

A similar recommendation applies to the `transfer_fails_overflow` test (see figure B.17).

```
let res = vm.function_must_fail(
    "transfer",
    &[
        BorshToken::FixedBytes(new.to_vec()),
        BorshToken::Uint {
            width: 64,
            value: BigInt::from(104u8),
        },
    ],
);
assert!(res.is_err());
```

Figure B.17: `tests/solana_tests/balance.rs#L297-L307`

- **Rename the following methods to better communicate what they do:**

    - `function_must_fail` → function_may_fail
    - `edges` → successors
    - `clone_for_parent_block` → deep_clone

- **Correct the grammar in the comments in figures B.18 and B.19.**

```
/// When a reaching definition change, we remove the variable node and all its
descendants from
/// the graph
```

Figure B.18: *`codegen/subexpression_elimination/available_expression_set.rs#L358-L359`*
("change" should likely be "changes")

```
/// Regenerate instructions after that we exchanged common subexpressions for
temporaries
```

Figure B.19: *`codegen/subexpression_elimination/instruction.rs#L203`*
("exchanged" should likely be "exchange")

- **Swap the comments in figure B.20, which appear to be associated with the wrong functions.**

```
/// Get the maximum unsigned value in a set
pub(super) fn set_max_signed(set: &HashSet<Value>) -> Option<BigInt> {
...
/// Get the maximum signed value in a set
pub(super) fn set_max_unsigned(set: &HashSet<Value>) -> BigInt {
```

Figure B.20: *`codegen/strength_reduce/value.rs#L69-L95`*

- **Correct the typo in `expresson_known_bits` (figure B.21).**

```
fn expresson_known_bits() {
```

Figure B.21: *`codegen/strength_reduce/tests.rs#L30`*
(`expresson` should be `expression`)

- **Rewrite the code in figure B.22 to use `unwrap` or `expect`.** Doing so will make the code more clear.

```
if let Some(block_vars) = block_vars.get_mut(&edge) {
    ...
} else {
    unreachable!();
}
```

Figure B.22: `codegen/dead_storage.rs#L149–L171`

- **Use named constants in place of magic numbers throughout the code.** Doing so will make the code more clear. Examples where magic numbers are used appear in figures B.23 though B.25.

```
let lamports_runtime_constant = (128 + space_runtime_constant) * 3480 * 2;
```

Figure B.23: `codegen/solana_deploy.rs#L342`

```
flow[block_1] = BigRational::from_integer(1000.into());
```

Figure B.24: `codegen/subexpression_elimination/anticipated_expressions.rs#L118`

```
&& BigRational::from_integer(2000.into()) == *flow_magnitude
```

Figure B.25: `codegen/subexpression_elimination/anticipated_expressions.rs#L161`

In some cases, even replacing `0` with a named constant would make the code more clear. For example, in figure B.26, `0` might be replaced with `ENTRY_BLOCK`.

```
vars[0].clone()
```

Figure B.26: `codegen/dead_storage.rs#L114`

- **Add a comment explaining why it is acceptable that the `highest_set_bit` function (figure B.27) returns 0 for both 0 and 1.** While this behavior doesn't appear to cause a problem now, the function could easily be misused in future code.

```
fn highest_set_bit(bs: &[u8]) -> usize {
    for (i, b) in bs.iter().enumerate().rev() {
        if *b != 0 {
            return (i + 1) * 8 - bs[i].leading_zeros() as usize - 1;
        }
    }

    0
}
```

Figure B.27: `codegen/strength_reduce/mod.rs#L569–L577`