Starknet Ecosystem Smart Contract Security Vulnerabilities

Introduction

Cairo, the programming language powering StarkNet's Layer 2 scaling solution, has gained significant traction in the blockchain ecosystem. As with any emerging technology, understanding its potential vulnerabilities is crucial for developers and auditors working in the Web3 space. This article delves into some of the key vulnerabilities and quirks of the Cairo language that every smart contract developer and auditor should be aware of.

1. Integer Overflow and Underflow

Unlike Solidity, Cairo uses modular arithmetic with a prime field of size P = 2^251 + 17 * 2^192 + 1. This means that all arithmetic operations are performed modulo P. While this prevents traditional overflow and underflow, it introduces a new class of vulnerabilities.

Consider the following code:

```
JavaScript
func add_numbers(a: felt, b: felt) -> felt {
    return a + b;
}
```

If $a + b >= P$, the result will wrap around to a smaller number. This behavior can lead to unexpected results, especially when dealing with financial calculations or token transfers.

Mitigation:

- Always check that the result of arithmetic operations falls within expected ranges. Use assert statements to enforce bounds:

```
JavaScript
func safe_add(a: felt, b: felt) -> felt {
    let res = a + b;
    assert(res >= a);  // Check for wrap-around
    return res;
}
```

2. Re-entrancy

While Cairo doesn't have direct external calls like Solidity, re-entrancy can still occur through the use of system calls, particularly when interacting with other contracts.

Example of potential re-entrancy:

```
JavaScript
func withdraw(amount: felt) {
    let balance = get_balance();
    if (amount <= balance) {
        transfer_to_user(amount);
        set_balance(balance - amount);
    }
}
```

If *transfer_to_user* involves a system call that could invoke other contracts, it might lead to re-entrancy.

Mitigation:

- Implement checks-effects-interactions pattern and use reentrancy guards when necessary.

```
JavaScript
@l1_handler
func handle_deposit(from_address: felt, amount: felt) {
    // Effects
    let (balance) = balances.read(from_address);
    balances.write(from_address, balance + amount);

    // State update to prevent reentrancy
    let (processed) = deposit_processed.read(from_address);
    with_attr error_message("Deposit already processed") {
        assert processed = 0;
    }
    deposit_processed.write(from_address, 1);

    // Interactions
    process_deposit(from_address, amount);
}
```

3. Uninitialized Storage

Cairo's storage model differs significantly from Solidity's. Storage slots are not automatically initialized to zero. Reading from an uninitialized storage slot returns an arbitrary value, which can lead to unexpected behavior.

```
JavaScript
struct Balance {
    value: felt,
}

func get_balance() -> felt {
    let balance: Balance = Balance(0);
    balance.read();
    return balance.value;
}
```

If *balance* was never written to, *get_balance* might return an arbitrary non-zero value.

Mitigation:

- Always initialize storage variables before first use. Consider using a boolean flag to check if a storage variable has been initialized.

4. Lack of Decimal Support

Cairo operates on field elements (felts) and doesn't have native support for decimals or floating-point numbers. This can lead to precision loss and rounding errors, especially in financial applications.

```
JavaScript
func calculate_interest(principal: felt, rate: felt) -> felt {
    return (principal * rate) / 100;  // Potential loss of precision
}
```

Mitigation:

- Use fixed-point arithmetic and clearly document the decimal places being used. Consider using libraries that implement robust fixed-point math.

5. Unexpected Behavior with Negative Numbers

Cairo's felt type can represent both positive and negative numbers, but their behavior might not always be intuitive, especially when used in comparisons or as array indices.

```
JavaScript
func is_positive(x: felt) -> felt {
    if (x > 0) {
        return 1;
    } else {
        return 0;
    }
}
```

This function might return unexpected results for large positive numbers that wrap around to negative values in Cairo's field.

Mitigation:

- Be explicit about the range of values expected and use appropriate bounds checking.

6. Lack of Standard Libraries

Cairo, being a relatively new language, lacks the extensive standard libraries that developers might be accustomed to in other languages. This often leads to developers implementing common functionalities from scratch, increasing the risk of vulnerabilities.

Mitigation:

- Use OpenZeppelin boilerplate contracts written in Cairo for the Starknet ecosystem and always thoroughly audit any external code before integration.

7. Pedersen Hash Collisions

Cairo uses the Pedersen hash extensively, which, while efficient for zk-proofs, is not cryptographically secure against collision resistance in the classical sense.

```
JavaScript
func store_data(key: felt, value: felt) {
    let hash = pedersen(key);
    storage.write(hash, value);
}
```

In theory, an attacker could find collisions in the Pedersen hash, potentially leading to storage conflicts.

Mitigation:

- For critical applications, consider using additional hash functions or implementing multi-layer hashing schemes.

8. Implicit Conversions and Type Safety

Cairo's type system, while strong in many aspects, allows for some implicit conversions that might lead to unexpected behavior.

```
JavaScript
func transfer(amount: felt) {
    let balance: u256 = get_balance();
    if (amount <= balance) {
        // Potential issues if amount > 2^256 - 1
```

```
        set_balance(balance - amount);
    }
}
```

Here, *amount* (a felt) is implicitly converted to a *u256*, which might lead to unexpected results for large values of *amount*.

Mitigation:

- Be explicit about type conversions and use appropriate bounds checking.

9. External Contract Calls

Cairo's approach to external calls requires careful handling:

```JavaScript
// Vulnerable implementation
@external
func execute_call(contract_address: felt, selector: felt, calldata_len: felt,
calldata: felt*) {

    // No validation of return data
    let response = call_contract(
        contract_address=contract_address,
        function_selector=selector,
        calldata_size=calldata_len,
        calldata=calldata
    );
}
```

- Implement proper validation:

```JavaScript
@external
func execute_call(contract_address: felt, selector: felt, calldata_len: felt,
calldata: felt*) -> (success: felt) {
    let response = call_contract(
        contract_address=contract_address,
        function_selector=selector,
        calldata_size=calldata_len,
        calldata=calldata
  );
    match response {
        Success(data) => {
            validate_response(data);
            return (1,);
        },
        Failure(error) => {
            return (0,);
        }
    }
}
```

10. Arithmetic Overflow Handling

Cairo handles arithmetic differently from Solidity. While Solidity has clear overflow behavior (depending on the compiler version), Cairo's behavior with felt type can be counterintuitive:

```JavaScript
// This might not behave as expected
func increment_counter() {
    let (current) = counter.read();
    // Vulnerable: No check for felt maximum value
    counter.write(current + 1);
}
```

- To properly handle arithmetic, implement explicit bounds checking:

```javascript
const BOUND = 2 ** 251 - 1;
func safe_increment{syscall_ptr: felt*, range_check_ptr}() {
    let (current) = counter.read();
    with_attr error_message("Counter overflow") {
        assert_le(current + 1, BOUND);
    }
    counter.write(current + 1);
}
```

Testing Considerations

When auditing Cairo contracts, consider these testing approaches:

1. Fuzzing Storage States: Test storage patterns with None values and edge cases.

2. Arithmetic Boundary Testing: Check behavior near felt bounds.

3. L1 Handler Testing: Simulate complex L1-L2 interaction scenarios.

4. Memory Allocation Analysis: Monitor memory usage in recursive functions.

Conclusion

Cairo, as a language designed for zk-STARKs and efficient Layer 2 scaling, introduces novel concepts and potential vulnerabilities that differ from traditional smart contract languages like Solidity. Developers and auditors must be acutely aware of these unique characteristics to build secure applications on StarkNet.

As the Cairo ecosystem evolves, new vulnerabilities and best practices will undoubtedly emerge. Staying updated with the latest developments, participating in the community, and rigorously testing smart contracts are crucial steps in ensuring the security of Cairo-based applications.

Remember, this article covers only a subset of potential vulnerabilities. Always conduct thorough audits and consider the specific context of your application when assessing security risks.

Tags:

- #CairoLang
- #SmartContractSecurity
- #StarkNet
- #CairoSecurity
- #StarknetSecurity
- #CairoLangAudit
- #Web3Security
- #BlockchainDevelopment