2.3.4.9 Error Taxonomy

Return to Data Taxonomy

Overview

Return to Top

There are all kinds of errors that can occur in Systems and Applications. Some errors are more or less static in nature and must be addressed before the software can be released. Others can occur during the running of the System or Application. Some of these errors are the result of aborent use of resources such as **Out of Memory** or **Dividion by Zero** while others are the result the code not doing what it is intended auch as **Invalid Data**.

- Syntax Errors
- Runtime Errors
- Logic Errors

Syntax Errors

Return to Top

A **Syntax Errors** are errors in program or script Source Code not meeting (following), or deviating from the standardized Syntax associated with the Programming Language.

Syntax Errors are generally small grammatical mistakes in the Source Code and can seem as trivial as a single character (i.e., a missing semicolon, or quotations, parenthesis, braces, or brackets not balanced). In the C++ code below, there is a Syntax Error on each line. The errors are indicated by comments on the line:

```
#include <iostream>
using namespace std;

void main()
{ int x = 10                    // semicolon missed
  int y = 15.4;                 // Asigning an int with a floating point
number
  char delimeter = ",";         // Using a quotation mark instead of an
apostrophy for a character
  string myText = 'hello World'; // Using apostrophe instead of a quotation
Mark
  cout << " "<< meText;         // referencing a variable that is not
```

```
declared
} }                                  // Too many closing braces
```

Some software Integrated Development Environments (IDEs) check Source Code for Syntax Errors in real-time, while others only generate syntax errors when a program is compiled. Even if a source code file contains one small syntax error, it will prevent an application from being successfully compiled and deployed. Similarly, if you run a script through an interpreter, any **Syntax Errors** will prevent the script from completing. In most cases, the compiler or interpreter provides the location (or line number) of the **Syntax Error**, making it easy for the programmer to find and fix the error.

As a general rule, Strongly Typed Languages create more **Syntax Errors** than Weakly Typed Languages with the idea that these **Syntax Errors** help prevent harder to diagnose **Runtime Errors** or **Logic Errors**.

DIDO Specifics

Return to Top

Solidity Syntax errors are detected within Syntax Highlighting the Ethereum: Remix Project or other Integrated Development Environments (IDEs) such as Microsoft: Visual Studio Code (VS Code) with the Solidity Plug In.

A very important aspect of **Syntax Errors** caught within the IDE is that these errors are NOT caught on the blockchain, but rather just inside the IDE. This means there is no expenditure of Gas to catch these errors.

Runtime Errors

Return to Top

Runtime Errors refer to a class of errors occurring during the execution of a program or an Application. Runtime errors imply Bugs in the running code developers expected but were unable to correct. For example, when the Application can not allocate more memory, an **Out of Memory** is triggered, terminating the Application and resulting in a Core Dump. The Core Dump and a Debugger are some of the most powerful and useful tools in tracking down **Runtime Errors**.

Table 1: A common list of errors adapted from https://www.techslang.com/definition/what-is-a-runtime-error/

Error Name	Description
Memory Leak	Memory leaks happen when a program drains your computer's Dynamic Random Access Memory (DRAM). It often arises from unpatched software, such as when you fail to update the Operating System (OS) to the newest release.

Error Name	Description
Out of Memory	**Out of Memory** is an error when there is no longer any spare memory to allocate to programs, or the memory requested exceeds the amount of memory available. An **Out of Memory** error causes programs to terminate or even the entire computer to crash triggering a Core Dump. The **Out of Memory** occurs when • A computer has a small amount of Random Access Memory (RAM) • Too many Applications or hardware pieces running at once • A request is made for a large amount of Random Access Memory (RAM) that exceeds amount available **Note:** Deleting items from the hard drive does not alleviate this problem
Heap Error	A Heap Error is when code inadvertently overwrites control information memory management functions use to control heap usage. The application that you are debugging must have been built with the heap check capability.
Out of Stack Space	The Stack Memory is the working area of memory that grows and shrinks dynamically with the demands of your executing program. Whenever an Application, program, function, procedure or a declarative block is invoked, or a local Variable is created, memory is allocated from the Stack. Some common reasons for this error are: • There are too many active procedure call being made (i.e., functions, procedures, methods, etc.) • There is a request to allocate local variables that exceed the available stack space • There are too many fixed-length strings, which in most compilers is allocated from the Stack • There is a Recursion issue
Division by Zero Error	**Division by Zero (DIV/0)** can occur in any application where the quotient is a Variable value substituted at Run Time. It is an error typically associated with spreadsheets. When formula inputs in the spreadsheet are left blank, the total might display a **`DIV/0`** error. The cell formulas need to be formatted in a precise manner to produce the correct output.
Encoding Error	**Encoding errors** happen when you're rendering a file, say a video file, to convert it into a usable or accessible file format. This is due to the resource-intensive nature of the encoding process. Error messages linked to this type of error include "encoding overloaded" or "encoding failed."
Input/Output Device Error	Input/Output (I/O) device errors occur when issues arise with the read/write function of a device. Common causes include device malfunction, outdated drivers, OS incompatibility, and faulty universal serial bus (USB) ports. As a result, users would get a prompt saying that the device wasn't accessible, making it impossible to transfer or encode files into it. Usually, the memory drive or the computer only needs to be restarted to get rid of the issue.
Overflow Error	Overflow Error occurs when an attempt is made to place a value into memory an integer (whole number) that is too large for the integer data type in a given system. For exaample, putting a 32-Bit value into a 16-Bit data field). Also see: Wrap Around

Error Name	Description
Range Error	A **Range Error** occurs when a value is not in the set or range of allowed values. For example, a percent might only allow numbers between 0 and 100, and an attempt is made to store a -1 or 101 into the field.
Segmentation Fault	Segmentation Fault (SEGFAULT) is an error returned by Hardware (H/W) with memory protection that tells the Operating System (OS) that a memory access violation has occurred.
Undefined Object Error	An undefined object error happens when a program attempts to call a function for a PHP or JavaScript object (or a C++ variable) that isn't defined or assigned a value. The error also occurs for deeply nested objects. In simpler terms, the code "cannot read" or find where a property is because it does not exist or is buried several levels deep within the code.
Underflow	An **Underflow Error** occurs when a mathematical operation results in a number which is smaller than what the device is capable of storing. For example, an negative number being put into an unsigned number space. Also see: Wrap Around.
User Defined Exception	User Defined Exception occurs is part of the architecture and design of an Application, routine, methd or object.

There are always costs associated with **Runtime Errors** and many of these errors can be traced to poor specification of reqiurements, misuderstanding of what the requirements are, or ignorance of the requirements.

> *Many studies have shown that requirements errors are very costly. By one estimate (in an article by Donald Firesmith for the Software Engineering Institute), requirements errors cost US businesses more than $30 billion per year and often result in failed or abandoned projects and damaged careers. The common wisdom is to find and fix requirements errors early in the lifecycle of a project, but that is easier said than done. Furthermore, the actual cost of a requirement error has been hard to quantify in the past. This resulted in a "business-as-usual" approach rather than proactively creating programs to find these errors early[1)].*

There are some important tools, processes and resources available to help prevent some Runtime Errors before the code is even executed.

- Static Code Analysis using Code Review Tools
- Have a Requirement Traceability plan
- Following the guidance provided in the Non-Functional Requirements section on Modifiability
- Understanding and using the Common Weakness Enumeration (CWE)
- Conducting Peer Reviews
- Establish a Case Management process with Automated Bug and Issue Tracking tools such as Jira (Bug tracking system) or Bugzilla
- Make sure there are clean compiles with no errors, warnings or informational messages
- Conducting in-depth, detailed testing covering requirement conformity checks outlined in 4.3.3.5 Testability, including:
    1. Unit Testing
    2. Integration Testing
    3. End-to-End Testing (E2E Testing)

4. Smoke Testing
5. Sanity Testing
6. Regression Testing
7. Acceptance Testing
8. White Box Testing
9. Black Box Testing
10. Interface Testing
11. Interoperability Testing

DIDO Specifics

Return to Top

Runtime errors occur only after the Smart Contract Code has been compiled to a byte code, deployed to the blockchain and executed on the EVM. These runtime errors occur when Ethereum “thinks” that there is something wrong with the Smart Contract. If the Runtime error occurs during a Transaction, any State Changes made during the Transaction are canceled and the Transaction is reverted. Depending on the kind of error, all or some of the gas in the Transaction is consumed, or a portion is returned. Runtime errors are harder to troubleshoot than Syntax errors because the Solidity Compiler can not help with the diagnostics.

Runtime Errors are more serious than Syntax Errors since the contract has already been deployed to the blockchain, and it is not possible to just update the code by redeploying it.

Out of Gas	The **out of gas** error occurs when all the gas allotted for the transaction is consumed before the transaction could complete.
Revert	The **revert** error occurs when a transaction is terminated in the Smart Contract by a **Revert** statement. See: revert
Invalid OpCode	1. **`assert()`** uses the **`0xfe`** opcode to cause an error condition 2. **`require()`** uses the **`0xfd`** opcode to cause an error condition **Note:** neither **`0xfe`** nor the **`0xfd`** opcodes in the yellow paper, you won’t find them. This is why you see the **`invalid opcode`** error, because there’s no specification for how a client should handle them.
Invalid Jump	**`Invalid-Jump`** is an old error replaced by **`revert`**. However, it still occurs when older published contracts (Solidity versions prior to 0.4. 10) are used. The EVM calls a jump opcode with an invalid-jump destination. **Note:** This is caused by malformed inputs, usually during contract creation.
Stack Overflow / Underflow	EVM is a stack-based machine, and thus performs all computations in a data area called the stack. All in-memory values are also stored in the stack. It has a maximum depth of 1024 elements and supports the word size of 256 bits. https://www.datasciencecentral.com/the-ethereum-virtual-machine-evm/. An **Overflow** occurs when the stack exceeds the 1024 elements maximum size. An **Underflow** occurs when the stack is empty and a **pop** is called.

Logic Errors

Return to Top

Logic Errors occur when there is a fault in the logic expressed by the Software (SW) or when the (i.e., **''if''**-**''then''** statements) through the SW creates a situation which is wrong or incorrect. The chance of a **Logic Error**, increases with the number of Decision Points found with the SW code.

Logic Errors can originate anywhere in the System Lifecycle from its conception to its execution.

Sometimes **Logic Errors** are reported as a Bug, however, not all “Bugs” are discovered or reported. Logic errors resulting in a Crash are generally quite obvious and are generally reported. However, Bugs which do not result in a crash but can only be found with a detailed forensic analysis of the Datastores are some of the hardest to discover and even track down.

Sometimes a **Logic Error** is introduced when a developer misinterprets the requirements or inadvertently implements the boolean calculus used in the Control Flow operations. Some examples:

- Assuming that memory is **`nulled`** before it is used
- Incorrectly specifying the logical comparison operators (i.e., **<**, ⇐, **==**, **===**, **'!='**, **>**, or **>=**). For example, specifying a numerical check for a percent and specifying **`p >= 0 && p < 100`** instead of **`p >= 0 && p ⇐ 100`**
- Incorrectly using the equivalent operators (i.e., **==**, **===**). **==** converts the variable values to the same type before performing comparison. This is called **Type Coercion**. **===** does not do any type conversion (coercion) and returns true only if both values and types are identical for the two variables being compared.
- Incorrectly using the assignment operator, **=**, instead of the comparison operator, **==**
- Assuming that indexing arrays at one
- Incorrect incrementing a pointer instead of the value stored at the pointer
- Assuming a string is terminated by a **`null`**
- Comparing an empty string, **“”** to a **`null`** string

Sometimes, a **Logic Error** is a **Bug** because it introduces a Weakness making the System or Application Vulnerable for malicious Exploitation. These kinds of “Bugs” could exist for a long time (i.e., years) before they are actually exploited.

Software (SW), Firmware, and Hardware (H/W) weakness types that have security ramifications are also potential **Logic Errors**. These Weaknesses can be flaws, faults, Bugs, or other errors in Software, Firmware, or Hardware and could have been introduced during any phase of the System Lifecycle from conceptualization, requirements specifications, architecture, code, or implementation.

Currently, there are 900+ Weaknesses defined by the Common Weakness Enumeration (CWE) most of which can be found through Static Code Analysis and Testing

DIDO Specifics

Return to Top

Logic Errors occur once the Smart Contract is deployed and it occurs because there is a problem with the logic of the Smart Contract. An important difference between a Solidity **Runtime Error** and a **Logic Error** is that the Ethereum Virtual Machine (EVM) does not consider it to be an error and the EVM assumes the Smart Contract is executing as expected. **Logic Errors** are either dangerous or produce false or eroneous results. For example, The Reentrancy Attack of a Decentralized Autonomous Organization (DAO) called The DAO Project had a Smart Contract attack 2016. The attack was a **Logic Error** and not a **Runtime Error**, a **Syntax Error** or a problem with the EVM.

Logic Errors are the hardest to fix becuase there are no tools that can examine a Smart Contract and find the Logic Errors. There are efforts underway at Ethereum called the Ethereum Formal Verification.

The Solidity Remix Project Integrated Development Environment (IDE) has a Plug In for Static Code Analysis called the **Remix-analyzer**.

remix-analyzer is the library which works underneath of **Remix-IDE Solidity Static Analysis** plugin.

remix-analyzer is an NPM package. It can be used as a library in a solution supporting Node.js. Find more information about this type of usage in the remix-analyzer repository

Table 2: Solidity Static Analysis[2)]

Category	Name of Weakness	Description
Security	Transaction origin: **`tx.origin`** is used	**`tx.origin`** is useful only in very exceptional cases. If you use it for authentication, you usually want to replace it by “msg.sender”, because otherwise any contract you call can act on your behalf. Example: `require ( tx.origin == owner );`

	Check effects: Potential reentrancy bugs	Potential Violation of Checks-Effects-Interaction pattern can lead to Reentrancy Attack vulnerability.
		Example:
		`// sending ether first`
		`msg.sender.transfer ( amount );`
		`// updating state afterwards`
		`balances [ msg.sender ] -= amount;`

Category	Name of Weakness	Description
	Inline assembly: Inline assembly used	Use of inline assembly is advised only in rare cases. Example: `assembly { // retrieve the size of the code, this needs assembly let size := extcodesize(_addr) } // End assembly`
	Block timestamp: Semantics maybe unclear	**now** does not mean current time. **now** is an alias for **block.timestamp**. **block.timestamp** can be influenced by miners to a certain degree, be careful. Example: `// using now for date comparison if ( startDate > now ) { isStarted = true; } // End if // using block.timestamp uint c = block.timestamp;`
	Low level calls: Semantics maybe unclear	Use of low level **call**, **callcode** or **delegatecall** should be avoided whenever possible. **send** does not throw an exception when not successful, make sure you deal with the failure case accordingly. Use **transfer** whenever failure of the ether transfer should rollback the whole transaction. Example: `x.call ( 'something' ); x.send ( 1 wei );`
	Blockhash usage: Semantics maybe unclear	**blockhash** is used to access the last 256 block hashes. A miner computes the block hash by “summing up” the information in the current block mined. By summing up the information in a clever way a miner can try to influence the outcome of a transaction in the current block. Example: `bytes32 b = blockhash(100);`
	Selfdestruct: Beware of caller contracts	selfdestruct can block calling contracts unexpectedly. Be especially careful if this contract is planned to be used by other contracts (i.e. library contracts, interactions). Selfdestruction of the callee contract can leave callers in an inoperable state. Example: `selfdestruct(address(0x123abc..));`

Category	Name of Weakness	Description

Gas & Economy	Gas costs: Too high gas requirement of functions	Never use **this** to call functions in the same contract, it only consumes more gas than normal local calls.
		Example:
		
		`contract test`
		`{ function callb() public`
		`  { address x;`
		`  this.b(x);`
		`  } // End function callb`
		`  function b (address a )`
		`  public`
		`  returns ( bool )`
		`  {} // End function b`
		`}`

	Delete on dynamic Array: Use require/assert appropriately	The **delete** operation when applied to a dynamically sized array in Solidity generates code to delete each of the elements contained. If the array is large, this operation can surpass the block gas limit and raise an OOG exception. Also nested dynamically sized objects can produce the same results.
		Example:
		
		`contract arr`
		`{ uint[] users;`
		`  function resetState()`
		`  public`
		`  { delete users;`
		`  } // End function resetState`
		`} // End contract arr`

Category	Name of Weakness	Description

	For loop over dynamic array: Iterations depend on dynamic array's size	Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully: Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit, which can stall the complete contract at a certain point. Additionally, using unbounded loops can incur in a lot of avoidable gas costs. Carefully test how many items at maximum you can pass to such functions to make it successful.
		Example:
		
		`contract forLoopArr`
		`{ uint[] array;`
		
		`  function shiftArrItem`
		`    ( uint index )`
		`    public`
		`    returns ( uint[] memory )`
		`  { for ( uint i = index;`
		`                i < array.length;`
		`                i++`
		`        )`
		`     { array [ i ] = array [i + 1 ];`
		`     } // End for i`
		`     return array;`
		`  } // End function shiftArrItem`
		`} // End contract forLoopArr`

Category	Name of Weakness	Description

	Ether transfer in loop: Transferring Ether in a for/while/do-while loop	Ether payout should not be done in a loop. Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit, which can cause the complete contract to be stalled at a certain point. If required, make sure that the number of iterations are low, and you trust each address involved.
		Example:
		
		`contract etherTransferInLoop`
		`{ address payable owner;`
		
		`  function transferInForLoop`
		`    ( uint index )`
		`    public`
		`  { for ( uint i = index;`
		`                i < 100;`
		`                i++`
		`          )`
		`    { owner.transfer ( i );`
		`    } // End for i`
		`  } // End function transferInForLoop`
		
		`  function transferInWhileLoop`
		`    ( uint index )`
		`    public`
		`  { uint i = index;`
		`    while ( i < 100 )`
		`    { owner.transfer(i);`
		`      i++;`
		`    } // End while loop`
		`  } // End function transferInWhileLoop`
		`} // End contract etherTransferInLoop`

ERC	ERC20: ‘decimals’ should be ‘uint8’	ERC20 Contracts **`decimals`** function should have **`uint8`** as return type.
		Example:
		
		`contract EIP20`
		`{ uint public decimals = 12;`
		`} // End contract EIP20`

Miscellaneous	Constant/View/Pure functions: Potentially constant/view/pure functions	It warns for the methods which potentially should be constant/view/pure but are not.
		Example:
		
		`function b`
		`  ( address a )`
		`  public`
		`  returns ( bool )`
		`  { return true;`
		`  } // End function b`

Category	Name of Weakness	Description

	Similar variable names: Variable names are too similar	It warns on the usage of similar variable names.
		Example:
		
		`// Variables have very similar names voter and voters.`
		`function giveRightToVote`
		`  ( address voter )`
		`  public`
		`{ require ( voters [ voter ].weight == 0 );`
		`  voters [ voter ].weight = 1;`
		`} // End function giveRightToVote`

	No return: Function with 'returns' not returning	It warns for the methods which define a return type but never explicitly return a value.
		Example:
		
		`function noreturn`
		`  ( string memory _dna )`
		`  public`
		`  returns (bool)`
		`{ dna = _dna;`
		`} // End function noreturn`

	Guard conditions: Use 'require' and 'assert' appropriately	Use **`assert(x)`** if you never ever want **`x`** to be **`false`**, not in any circumstance (apart from a bug in your code). Use **`require(x)`** if **`x`** can be **`false`**, due to e.g. invalid input or a failing external component. Example: `assert(a.balance == 0);`

	Result not used: The result of an operation not used	A binary operation yields a value that is not used in the following code. This is often caused by confusing assignment (**=**) and comparison (**==**).
		Example:
		
		`c == 5;`
		` // or`
		`a + b;`

Category	Name of Weakness	Description

	String Length: Bytes length != String length	Bytes and string length are not the same since strings are assumed to be UTF-8 encoded (according to the ABI definition) therefore one character is not necessarily encoded in one byte of data.
		Example:
		
		`function length`
		`  ( string memory a )`
		`  public`
		`  pure`
		`  returns ( uint )`
		`{ bytes memory x = bytes ( a );`
		`  return x.length;`
		`} // End function length`

	Delete from dynamic array: 'delete' on an array leaves a gap	Using **`delete`** on an array leaves a gap. The length of the array remains the same. If you want to remove the empty position you need to shift items manually and update the length property.
		Example:
		
		`contract arr`
		`{ uint[] array = [ 1, 2, 3 ];`
		`  function removeAtIndex()`
		`  public`
		`  returns ( uint[] memory )`
		`  { delete array[1];`
		`  return array;`
		`  } // End function removeAtIndex`
		`} // End contract arr`

	Data Truncated: Division on int/uint values truncates the result	Division of integer values yields an integer value again. That means e.g. `10 / 100 = 0` instead of `0.1` since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.
		Example:
		
		`function contribute()`
		`  payable`
		`  public`
		`{ uint fee = msg.value * uint256 (`
		`feePercentage / 100 );`
		`  fee = msg.value * ( p2 / 100 );`
		`} // End function contribute`

DIDO Specifics

Return to Top

This section is an overview of how to find and fix bugs in general.

	Solidity Compiler	Solidity Debugger	Solidity Linter	Tests
Syntax Errors	**X**		**X**	
Runtime Errors		**X**		**X**
Logic Errors		**X**	**X**	**X**

☐ [char]Review

1)

Joe Marasco, Stickyminds, What Is the Cost of a Requirement Error?, 26 June 2007, Accessed 27 January 2022, https://www.stickyminds.com/article/what-cost-requirement-error

2)

Solidity Static Analysis, Remix-IDE, Accessed: 4 February 2022