Assignment of bachelor’s thesis

Title:	Security of the Lua Sandbox
Student:	Petr Adámek
Supervisor:	Ing. Josef Kokeš
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2022/2023

Instructions

1) Explain the concept of sandbox, use of sandbox in operating systems and programming languages.
2) Study the Lua language with a particular focus on its sandbox. Describe its features, properties and limitations.
3) Research the current state of vulnerabilities of Lua and the sandboxing techniques used to counter them.
4) Demonstrate one of the attacks described in the previous section.
5) Propose modifications to the language or its sandbox to improve their security.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 31 January 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Security of the Lua Sandbox

Petr Adámek

Department of Computer Systems
Supervisor: Ing. Josef Kokeš

May 10, 2022

Acknowledgements

I wish to thank my supervisor, Ing. Josef Kokeš, who helped me whenever I needed it. I could not have created this work without him.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 10, 2022 …………………

Czech Technical University in Prague
Faculty of Information Technology
© 2022 Petr Adámek. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Adámek, Petr. *Security of the Lua Sandbox.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstract

Lua is an easy to embed language which can be used to extend an application with a scripting environment. This thesis focuses on isolation of Lua scripts from sensitive parts of the application.

Sandboxing is commonly used for isolation of components in an application. This work covers some theory behind sandboxes and discusses how to properly implement a sandbox in Lua. A sandbox based on isolation of Lua functions and a sandbox based on isolation of the entire Lua interpreter are proposed as possible implementations.

An analysis of the Lua language, including the standard library, is performed, focusing on isolation and potential for escaping a sandbox. A simple tool for crawling and dumping the environment of a Lua function is created. The dumped values can be later analysed in an interactive environment. This allows for a comprehensive examination of values accessible from a sandbox.

Some freely available Lua sandbox implementations are analysed to show how the theory described here is used in practice. The theory is further demonstrated on a flaw found in the Lua sandbox of the OpenMW game engine.

Keywords Lua programming language, sandboxing, language-based security, language-based sandbox

Abstract

Lua je programovací jazyk, který lze použít pro rozšíření jiné aplikace o skriptovací prostředí. Tato práce se zabývá izolací Lua scriptů od citlivých částí aplikace.

Sandboxing se používá pro izolaci dílčích částí aplikace. Tato práce rozebírá teorii sandboxingu a diskutuje o tom, jak správně implementovat sandbox v programovacím jazyce Lua. Jako možná implementace je navržen sandbox založený na izolaci individuálních funkcí a sandbox založený na izolaci celého interpreteru.

Byla provedena analýza programovacího jazyka Lua a jeho standardních knihoven zaměřená na izolaci a potenciál na únik ze sandboxu. Následně byl vytvořen jednoduchý nástroj pro vypsání hodnot dostupných v prostředí funkce a jejich následnou analýzu v interaktivním prostředí. Toto umožňuje komplexní průzkum hodnot dostupných ze sandboxu.

Také byly provedeny analýzy několika volně dostupných implementací sandboxů pro ukázku, jak se navržené teoretické konstrukty v praxi používají. Nakonec je ukázána chyba v implementaci sandboxu v rámci herního enginu OpenMW.

Klíčová slova programovací jazyk Lua, sandboxing, language-based security, language-based sandbox

Contents

Introduction **1**

1 Sandboxes and Sandboxing Techniques **3**
1.1 Sandboxing in Operating Systems 4
1.2 Virtualization-based Isolation 5
1.3 Interpreters 5
1.4 User Space Sandboxing Mechanisms 5
1.5 Sandboxing in Specific Languages 9

2 Lua **11**
2.1 Language features 12
2.2 Sandboxing Mechanisms in Lua 16
2.3 Referencing Variables and Values 19
2.4 Read only values 20
2.5 Applying Resource Limits 21
2.6 C interface 21
2.7 Standard library safety 22
2.8 Summary 26

3 Attacking Lua **27**
3.1 Attack Surface 27
3.2 Sandboxing Methods 27
3.3 Bytecode Attack 28
3.4 Other Known Bugs 29
3.5 Notable Lua Interpreters 29
3.6 Sandbox Implementations 30

4 Automatic Environment Crawler **33**
4.1 Motivation 33

4.2 Dumper . . . 34
4.3 Analyser . . . 34
4.4 Limitations and Further Improvements . . . 35

5 OpenMW **37**
5.1 OpenMW Lua Sandbox Design . . . 37
5.2 Mistakes with makeReadOnly . . . 38
5.3 Accessible Insecure Garbage Collection Metamethod . . . 39

6 Suggested Improvements **41**
6.1 Sandboxes . . . 41
6.2 The Language . . . 42
6.3 Existing Solution . . . 42

7 Conclusion **45**

Bibliography **47**

A Acronyms **53**

B Contents of enclosed CD **55**

List of Figures

2.1 The state-based sandbox architecture 17
2.2 The function-based sandbox architecture 18

Introduction

'*We use the term **sandboxing** to describe the concept of confining a helper application to a restricted environment, within which it has free reign.*' [1]

Or as [2] define a sandbox, '*An encapsulation mechanism that is used to impose a security policy on software components.*'

Sandboxes are a common part of all computer systems where not all parts can be trusted. Whether they be virtual machines, an operating system or a web browser, all the mentioned tools use sandboxing in one way or another. [2]

Sandboxing enforces that all applications and users can do only what they are allowed to, restricting access to other applications and resources. In the context of an operating system that can mean, e.g., that a process is only allowed to open files which the user has rights for.

Sandboxing allows us much more freedom without the worry about safety we would have otherwise. Programs that use sandboxing allow for installing any plugins and modifications without worrying that they might be malicious, as the sandbox should prevent them from affecting and accessing anything important. But since we give up our security precautions in favour of using the sandbox it is vital for it to be correctly implemented.

This thesis will summarize some commonly used sandboxing techniques and describe how sandboxes are constructed in Lua and other programming languages.

Using this knowledge, different ways of constructing a simple sandbox using Lua will be described. The standard libraries provided by the language will be analysed for harmful functions and recommendations for restraining these functions will be given. Any other known vulnerabilities will also be mentioned.

Later, multiple sandbox implementations will be shown to serve for comparison with the implementation and restrictions proposed in this thesis. Ideally, an attack on one of the sandboxes will be shown to demonstrate mistakes to be wary of.

Chapter 1

Sandboxes and Sandboxing Techniques

Sandboxes are based on the concept of *protection domains*.

A protection domain is a collection of access rights to objects. An access right is the ability to perform a specific operation, e.g., reading a file. Objects are specific software (such as files or processes) and hardware (such as CPU instructions, drives and printers) resources. Code running within a protection domain may only operate on objects using the access rights of the domain. [3, p. 627–628]

Communication and switching between domains is restricted to ensure that the restrictions cannot be bypassed. What concept the domain encapsulates is not always the same. Users, individual processes, or even individual functions may be regarded as a domain depending on the system. [3, p. 628–629]

Each domain can be thought of as an individual sandbox. To ensure it works well, two problems must be handled — isolation and policy enforcement.

Isolation is necessary to ensure that all interactions the sandbox makes pass through a given interface which can then be subject to policy enforcement.

Policy enforcement is used to filter requests coming through the interface deciding what to do based on the policies enforced upon the domain. For example, a request to open a file is always done the same way but the access may be granted or denied.

Sandboxes employ a variety of techniques to enforce the isolation. These techniques vary based on what is being sandboxed and which hardware resources the sandbox has access to. Some languages may provide features that allow for employing different isolation or policy enforcement mechanisms.

Isolation between subjects within a single domain is not strictly necessary, as everything affected could be equally affected by all parties, while isolation between different domains is needed.

Having programs in separate domains with restricted rights means that a

bug in one of them does not necessarily compromise the rest of the system and all data in it. Separating programs like this reduces the impact any individual vulnerability has.

1.1 Sandboxing in Operating Systems

Modern operating systems (OS) are typically split into the privileged *kernel space* and the unprivileged, sandboxed *user space*. *Kernel* is the core of the OS responsible for, among other things, sandboxing running user programs.

Hardware-based mechanisms are used to ensure isolation. Paging is used to map virtual memory and thus restrict access to physical memory. [4, 5] Hardware ring levels forbid unsafe instructions from being executed. That makes using virtual memory mapping as a security measure possible as otherwise a program could rewrite its page table. [6]

1.1.1 Processes

Operating systems employ processes as their memory/privilege isolation primitive. Each process has its own virtual memory which is by default not shared with other processes. A lower-level execution primitive are threads. But while threads can be used for further parallelization of workload, they do not belong to a different protection domain — individual threads of a single process have no isolation between them. [4, 5]

When processes want to do more than just play around in their own address space, they must get access to resources from other processes or the kernel. But communication with other processes is done via mechanisms provided by the kernel. To summarize, all such access has to be done through the kernel. [7, 8]

1.1.2 Policy Enforcement

Even though each process is a protection domain by itself, it is by default not particularly restricted as it can affect the system as much as the user running it could.

All communication is restricted to the *system call* interface — requests that the kernel does something that a process cannot do by itself.

Just limiting what system calls are made using *ptrace* or a kernel space module and filtering their arguments is sufficient for creating a sandbox with fine-grained policy enforcement, though as Garfinkel [9] shows it comes with its share of difficulties.

There are other more sophisticated mechanisms in place allowing processes to opt-in to different limits on access to resources. These may include limiting process running time, restricting access to certain system calls altogether, and more. [10, 11, 12]

Restricting this communication and filtering what should and should not be allowed is what lets us create a process-based sandbox, such as the one Chromium uses. [13]

1.2 Virtualization-based Isolation

Virtualization means creating virtual versions of physical resources and thus hiding the physical versions behind a controlled layer. A similar term is *emulation* which is used when there is no physical version of the virtual resource and its functionality is simulated using software. [3, p. 40-41]

This allows for providing different instances of virtual resources to different components thus isolating them. An example would be the aforementioned virtual memory that processes are given access to instead of accessing physical memory directly.

1.2.1 Virtual Machines

As virtual machines are virtualized computers, they should be perfectly isolated as they cannot just access the computer that is running them. Such machines don't have to be running an operating system — virtualizing the processor, memory, and potentially other peripheries is what allows the isolation to work. [14]

It is to be noted that the sandboxed component still has to be granted an interface with which it can communicate with the outside world. If the interface is not sufficiently restricted, the component can do as much harm as any other process.

Sandboxing based on running a component in a minimalistic virtualized environment has been demonstrated. [15, 14, 16]

Some programming languages are designed to be compiled for a specific virtual machine such as the Java Virtual Machine. Such machines do not require a real-world equivalent and may be fully emulated. [3, p. 40-41]

1.3 Interpreters

An interpreter is a program that executes the code it is given without first transforming it into another language as a compiler would.

By definition, interpreters do not provide any security guarantees other than the ones the interpreted language gives.

1.4 User Space Sandboxing Mechanisms

If the OS does not provide sufficiently fine-grained control over permissions or the inter-process communication overheads are considered too large [17] other

isolation or policy enforcement mechanisms may be used. Ultimately, creating a sandbox running inside a process is necessary — escaping the sandbox would be equal to taking control of the process and still having to deal with the restrictions imposed upon it by the operating system.

A major obstacle in this path is the OS taking a process as a single protection domain even though internally different libraries and plugins, which should not have equal rights, are used.

That means user space sandboxing must be implemented without using the hardware mechanisms that the OS uses and all the restrictions a process can opt-in to will apply to the entire process. The OS-based restrictions should not be overlooked. If the sandbox were somehow bypassed, those would still be effective in restricting what damage can be done.

All the isolation mechanisms attempt to isolate parts of a program from each other — essentially creating individual protection domains. Though if provided an insecure interface, they may still escape the sandbox and be used for malicious purposes.

1.4.1 Software Fault Isolation

As Gang [18] very well summarizes:

Software Fault Isolation is a technique based on running modified code. Either by dynamically rewriting the binary code which is being executed or by statically in-lining checks into the entire program, an input program is taken and transformed into a safer variant. [18]

This technique can be used to enforce both control flow and data flow integrity. For example, making sure only code inside the sandbox can be executed and only memory it owns can be read and written. [18]

It may be implemented as a compiler that adds the necessary checks into the resulting code. This method requires access to the source code. [18]

If compiled code is to be run directly, some form of a verifier is needed to ensure all checks have been inserted correctly. Assuming the compiler is correctly implemented the issue can be sidestepped by only allowing running code which was compiled by the compiler beforehand. [18]

Such guarantees come with some runtime speed overhead and with upkeep overhead as the compiler/rewriter/verifier need to be kept up to date. [18]

One prominent example of this approach is Native Client [19] which had been a part of the Chrome browser until 2019 when it got deprecated in favour of WebAssembly [20].

1.4.2 Language-based Sandboxing

Language-based sandboxing is based on some features or properties of a language. Much like how sandboxes restrict what resources can be accessed,

languages may restrict by simply not having expressions to describe certain unsafe operations.

It may also allow adding policies to using internal functions such as the ones from a language's standard library rather than only the system calls. For example, in Java the Security Manager [21] (which is now deprecated) could be used to check which exact function calls lead to this restricted call and decide if all of the callers had sufficient rights. Lua allows for wrapping of functions and restricting which parameters are OK and which are not or outright replacing the original functionality.

Two features that must be implemented correctly are *type safety* and *memory safety*. Otherwise, all restrictions a language imposes could be worked around.

1.4.2.1 Memory Safety

Memory safety is a difficult term to define — commonly defined using the errors it prevents rather than the properties a memory-safe language should have. [22]

The paper [23] and article [22] attempt to formalise this, but listing the errors it prevents gives us a more intuitive understanding of what it means.

To give a simplified explanation, it should not be possible to access undefined memory. Some common violations of this rule are *use after free*, *illegal free*, *using uninitialized memory*, *buffer overflow*, *null pointer dereference*. [22]

1.4.2.2 Type Safety

Type safety ensures that a programmer can trust that variable of a type always contains a valid value for that type.

Moreover, type safety ensures an operation with invalid types — such as calling a function with incorrect parameters — simply cannot be executed. These guarantees may be enforced both statically at compile time and dynamically at run time.[24]

A weak type system may be used to subvert memory safety as casting an integer to a pointer would allow for arbitrary memory access. Without memory safety a type system cannot be fully trusted as the underlying value of a type could be easily overwritten to be invalid.

Type confusion is a bug where a program expects a value of a different type than it receives and continues operating on it with the assumption that it is correctly typed.

1.4.2.3 Achieving Isolation

When memory safety and type safety get combined, a program cannot retrieve data from arbitrary memory locations. And if that is the case, all values and

functions a program interacts with must come from some sort of interface described by the language.

This guarantee is sufficient for achieving isolation from the rest of the system, but the interface may introduce bugs that break the isolation as the functions which are being interfaced may be written in an unsafe language and contain bugs or be inherently unsafe due to their functionality.

A component, in this case, may be an entire program that is compiled directly into machine code (one such case is described in section 1.5.1). In the case of a language running in an interpreter or a virtual machine, the entire interpreter may act as a component with restrictions imposed upon it rather than the language. This is further discussed in section 2.2.1.

While memory and type safety are still required, some languages may offer sandboxing or isolation as an integral part of the language. When only using memory and type safety, the entire runtime has to be constrained. The languages allow for segregating protection domains based on lower-level primitives.

Java, for example, uses individual classes or more specifically protection domains as their primitive. Their protection domains are sets of classes with the same privileges. [25, p. 2]

In Lua, each function is ultimately a protection domain. This will be discussed later in section 2.1.1.1.

Some languages may make isolation particularly hard by leaking potentially sensitive data from unexpected sources. A good example is Javascript in web browsers where — unless strict mode is enabled — the *window* object is leaked through many different calls. [26]

1.4.2.4 Policy Enforcement

Policy enforcement in language-based sandboxes differs from process-based sandboxes in a couple of ways. While the concepts are the same, details may differ.

First and foremost, the controlled interface is often different. Languages ship with their standard library which may be widely different from the system call interface. As such, the restrictions must be tailored to this interface rather than the resources or system calls.

Each value or function accessible with the language can be considered from a privilege standpoint — deciding whether each domain should be able to use those resources. The restrictions can of course be more fine-grained inside the functions should they be accessible at all.

Some languages may allow for hiding or replacing functions, effectively removing them from the interface entirely.

Policy enforcement may be a part of the language or its standard library. One such example is shown in section 1.5.2.

Other mitigations come from language features themselves. For example, Lua strings are immutable. Once a string value is retrieved it cannot be changed, only manipulated to create a new string. When checking the contents of a string in C, its value may be changed from another thread after it has been validated — a Time-of-check Time-of-use (TOCTOU) Race Condition [27].

1.5 Sandboxing in Specific Languages

This section covers some programming languages and sandboxing mechanics used in them. Many other languages also allow for sandboxing in some way.

Javascript is an example of a language where sandboxing is very useful for advertisements and other arbitrary third-party code. The language is however not designed to make sandboxing easy. It can be encapsulated in an *iframe* — thus sidestepping the need for a language-based sandbox — though that restricts it to a specific section of the page which may not always be wanted. Otherwise sandboxing javascript is very complex as there are many subtle ways in which bypassing a sandbox may be possible. [28, 26]

1.5.1 WebAssembly

An interesting example of isolation implementation is WebAssembly (Wasm). It is a 'binary instruction format for a stack-based virtual machine' [29], meaning it is something akin to the machine instructions a processor uses.

Wasm can be used as a compile target — multiple languages can be compiled into Wasm. [30].

Wasm isolation is based on the concept of 'Linear memory'. All memory accesses are bound to a restricted region indexed by a 32-bit integer. As has been shown, a bug can result in the region's data being corrupted making it untrustworthy — but only that region. Barring a bug in the virtual machine or the provided API, no resources outside its own memory space can be affected. A bug can result in malicious data being sent via the API as internally the memory safety is not guaranteed at all. [31]

Wasm does not provide any instructions for modifying the executable code. That means all functions have to be provided from outside of the sandbox.

An example of how Wasm can be used for sandboxing is the Firefox browser which uses it via the RLBox library to sandbox some helper libraries. It compiles to Wasm and then to machine code while retaining the security guarantees Wasm provides. [32]

1.5.2 Java and C#

While both Java and C# provide type safety [24], sandboxing of partially trusted code has been discontinued [33, 21] as it is very hard to get right. [25, 34]

Both languages employed a rather complicated security policy manager based on checking the stack for which classes have been used in the calling chain and determining if all of the classes have sufficient rights for the privileged operation. [25, 33]

Instead, using the system-provided access control and isolation mechanics is now being recommended. [33]

Java provides a module mechanism through which some sort of sandboxing may be possible [35], but this method is not straightforward nor simple.

Chapter 2

Lua

This chapter covers basic information about the language, its features, and concepts useful for creating a sandbox. Lua versions 5.1–5.4 have been checked and relevant differences between the versions will be mentioned.

Lua is designed to be an embedded language, meaning it is to be used to extend an application rather than to be used to make a program using only it. The language is heavily reliant on the API provided by the application, as even the standard library functions must be explicitly loaded by it. [36]

The program in which Lua is embedded is referred to as the host program.

There are two main APIs. The language API, which is provided to the program running in Lua, and the C API, which is used to interact with the interpreter from the host program.

The language can be embedded via either static linking or linking against a shared library. Both options provide the same functionality.

The PUC-Rio[1] implementation of Lua will be referred to as the official implementation. Other interpreters generally attempt to be compatible with PUC-Rio's C API and Lua libraries, not only the language syntax and semantics, while often providing their extensions.

All the interpreters mentioned in this work are composed of a language compiler and a virtual machine. Lua code is compiled into bytecode which is later executed by the virtual machine. The interpreters allow for loading of pre-compiled bytecode but provide no guarantees when executing intentionally maliciously crafted bytecode. [37, 38]

It has been shown that executing malicious bytecode is a serious security vulnerability that may lead to arbitrary code execution as mentioned in section 3.3. It is either assumed or outright claimed [37] that the bytecode created by the provided compiler will always be safe.

This is an application of the principles of Software Fault Isolation. The assumption here is that the compiler will inject appropriate restrictions into

[1] https://www.lua.org/

the bytecode upon compilation. It also implies that arbitrary changes to the values the virtual machine operates with during the execution could lead to further exploits, if the checks are not set up to handle this.

The Lua manual does not explicitly warn about the dangers of untrusted bytecode: the 5.1 manual [39] does not mention it at all, 5.2 [40] mentions that all verification of it has been removed and is insecure, and both the 5.3 [41] and the 5.4 [38] manuals merely claim it may crash the interpreter.

2.1 Language features

Lua is a dynamically typed language with no compile-time type checking. Each function has to check the validity of its arguments manually. Even the argument count is not checked.

After embedding into a program, the language runs in an environment called a *State*. This is an object representing the interpreter in which the code is then loaded and executed. Many APIs are provided, allowing for nigh-arbitrary access to all data and internal values of the interpreter.

A State is designed to only work with a single thread at a time. If multi-threading is desired, such programs have to be split into multiple States akin to creating additional processes on an OS. The problem is, transferring values between States is not trivial. To put it simply, values cannot be passed into the new environment by reference and instead must be copied with all the limitations and benefits it brings.

Not all language features are covered in this section, merely a subset of them considered relevant to this work. For example, Lua 5.4 introduces constant local variables which are not considered relevant for sandboxing as they are only local.

2.1.1 Values and their types

'*All values in Lua are **first class values**. This means that all values can be stored in variables, passed as arguments to other functions, and returned as results.*' [39, ch. 2.2]

Some values are very simple. The default value of all variables is *nil*. This is usually considered an absence of a value. As far as trivial data types are concerned, there is also *number*, *string*, and *boolean*. Lua strings are immutable and are not terminated by zero — they can contain any values. They are closer to constant byte arrays than to strings in this manner. [39, ch. 2.2]

Other values have more complex semantics.

Lua does not have C-like arrays, *tables* are used instead. Tables are associative arrays in which keys and values can be of any type except nil. [39, ch. 2.2]

2.1.1.1 Functions

Functions, whether provided by the host program or Lua are treated the same way in the language. The C API considers them to be distinct types. All functions can take an arbitrary number of arguments and may return as many values as they wish. [39, ch. 2.5.8]

All Lua functions may have local variables bound to the current scope and *upvalues* which are references to variables in the enclosing lexical scope. Variables not bound to any local or upvalue are stored in the function environment, which may be shared between functions. [39, ch. 2.6]

All executable code is loaded as a function. When loading a file, it is compiled as a vararg function which may then be invoked. In the manual, this executable block of code is called a 'chunk'.

A function's environment is a table that can be manipulated in the same way any other table can. The exact implementation changed between Lua versions 5.1 and 5.2. It used to be a magical value associated with all functions. In version 5.2 the concept got simplified. Instead, the environment of a function is always bound as its first upvalue under the name _ENV. [39, ch. 2.3][40, ch. 2.2]

The global environment is a table which the State keeps a track of and many functions by default operate on it. Alternatively, in version 5.1, this was a table which was bound to the current thread rather than being universally global. Unless explicitly changed, the global environment is still stored in the _G variable and may be equal to the _ENV value. [39, ch. 2.3][40, ch. 2.2]

It is assumed that both local variables and upvalues in a function cannot be retrieved or affected by outside sources.

2.1.1.2 Threads

Another value type is coroutines referred to as *threads*. These may wrap a Lua function [40, ch. 6.2] or (since version 5.3 [41, ch. 6.2]) any function.

As has been mentioned before, this does not allow for executing code in a single State from multiple system threads at once.

2.1.1.3 Userdata

Finally, Lua has *userdata* values. These represent custom values, and their functionality depends purely on what the host program defines them as. There are two subtypes, *light userdata* and *full userdata*. Assumptions are made that userdata values cannot be created or modified directly in Lua, to ensure the integrity of data in the host. [40, ch. 2.1]

Light userdata is just a pointer with no additional associated values. Full userdata is a block of memory allocated by Lua which will be garbage collected by it eventually. Lua provides no way to modify or create these values — this can only be done by the C API. Full userdata may also have Lua values

associated with them. These had been called environment in Lua version 5.1 but will be referred to as *user values* as has been the case since. [40, ch. 2.1]

When referring to userdata in this thesis, full userdata are meant as these values carry very important properties, unlike light userdata.

2.1.1.4 Value References

'*Tables, functions, threads, and (full) userdata values are **objects**: variables do not actually **contain** these values, only **references** to them. Assignment, parameter passing, and function returns always manipulate references to such values; these operations do not imply any kind of copy.*' [38, ch. 2.1]

If any of the above-mentioned *objects* get passed into two separate sandboxes, the objects may be used for communication between them. This may be both desirable and undesirable as will be later discussed. The exact extent to which values may be modified and to what effect is covered in section 2.3.

2.1.2 Errors

Lua allows for throwing of errors. These errors work similarly to exceptions in other programming languages. In Lua, no dedicated object that must be thrown exists — any value may be thrown. The official interpreter and all its libraries only ever throw strings, but this is only by convention. [40, ch. 2.3]

Instead of a try-catch block, Lua utilises special calls in *protected mode*. These are interacted with as regular Lua function calls. They return a boolean, indicating whether an error occurred, and either the error value or all other function return values. From Lua, protected calls are made using *pcall*. [40, ch. 2.3]

Explicit throwing is done using the *error* function. [40, ch. 2.3]

Errors may be used for enforcing that a Lua function is forcibly terminated while leaving the interpreter in a consistent state. They may be also be used maliciously to subvert control flow at key moments or to attack error handlers. When implementing a handler, it is important not to assume the error will always be a string.

2.1.3 Metatables

All values in Lua can have an associated *metatable* but only tables and full userdata have individual metatables. All other values share a metatable with all other values of the same type. [38, ch. 2.4]

Values in the metatable are referred to as *metavalues* or *metamethods* if they are functions. A metatable is an ordinary table that is queried during some specific operations, most commonly if the operations were not defined on the value it is associated with, but exceptions exist.

Each operation is associated with a unique string key, for which the metatable will be queried using *raw access* — access which ignores metatables associated with the metatable. The standard keys are always preceded by two underscores. [38, ch. 2.4]

The use of metatables is not restricted to the core interpreter. Some functions may optionally respect custom metavalues. This can be done freely and carries no inherent impact unless a collision in the keys occurs.

One such case is the *getmetatable* method which becomes restricted if the *__metatable* metavalue is defined for the object operated upon. [39, 38]

2.1.4 Notable Metavalues

In the context of sandboxing, most of the outlined metavalues are not particularly interesting. They allow for leaking very little data. The *__add* operator is only invoked if the + operator would otherwise fail, as either of the arguments is not a number [38, ch. 2.4]. This is meaningless if argument checking is done based on types.

Potentially, a function which does not check the types of its arguments and instead use errors to ensure correct execution may be affected.

Most other metavalues work the same way.

2.1.4.1 Garbage Collection

If a userdata or table object is assigned a metatable with the *__gc* metavalue set, the object is marked for finalization. Just before the object is garbage-collected, the metatable of the object is queried for this metavalue. If this metavalue is a function, it is called with the to-be-collected object as the only argument. [41, ch. 2.5.1]

In version 5.1 this operation is only supported for userdata and the metatable does not have to contain the metavalue when being set, it may be defined later. [39, ch. 2.10.1][40, ch. 8.3]

As this metamethod operates with to-be-deleted objects, it is assumed to be able to deallocate objects or otherwise affect references. Care needs to be taken that this method never leaks to any sandbox or is resilient to being called with arbitrary values and the values are then kept in a safe state. Otherwise, errors such as use after free, double free, or more could arise.

Similar to this are to-be-closed local variables defined in Lua 5.4 with the *__close* metamethod. [38, ch. 3.3.8]

2.1.4.2 Hiding Real Values

The *__newindex* is used whenever key-value pair assignment is attempted to a non-existent key of a table or assignment to a non-table value. The *__index* is used whenever key indexing is attempted on a table with the key not present or on any non-table value. [38, ch. 2.4]

These two metamethods allow for restricting all writes and reads to a table. They can be bypassed by using raw access, however. They see large usage in the implantation of read-only objects.

Lua also allows for setting these metavalues to tables instead of functions. In that case, rather than calling a method, the corresponding operation is done on the stored table.

2.1.5 Dangers of Shared Metatables

If write access to a metatable is acquired, all metavalues may be changed. This in turn indirectly affects all values which have this metatable set. If an operator were to be replaced, the results may be spoofed, and the other argument retrieved and used arbitrarily.

Changing the __*gc* value will lead to all the objects being eventually leaked upon garbage collection.

Even only being able to read a metatable directly carries security implications as the metamethods may otherwise carry some assumptions about the objects they are called with.

2.2 Sandboxing Mechanisms in Lua

The Lua sandbox can be interpreted as the State which contains and executes all code in which Lua is contained. An alternative approach can be taken using purely language features. The former will be referred to as a state-based sandbox and the latter as a function-based sandbox.

Both approaches are not mutually exclusive and have their benefits and downsides. Neither is trivial to implement and keep isolated if a person does not know what they are doing and wishes to expose complex APIs to the sandbox.

A trivial sandbox with no provided APIs can be constructed very easily. Interacting with such a sandbox would be tedious and even basic, safe functionality of Lua would be restricted.

2.2.1 State-based Sandbox

The state-based approach isolates all individual components into separate States. This approach makes interaction between components more difficult, as Lua has no mechanisms for sharing data between States. All values which are to be shared have to either be copied or some other complex mechanism has to be made up. This approach allows for optimizations based on multi-threading.

The isolation is only dependent on the language not being able to access values not provided to the State. As such, the entire Lua State is treated

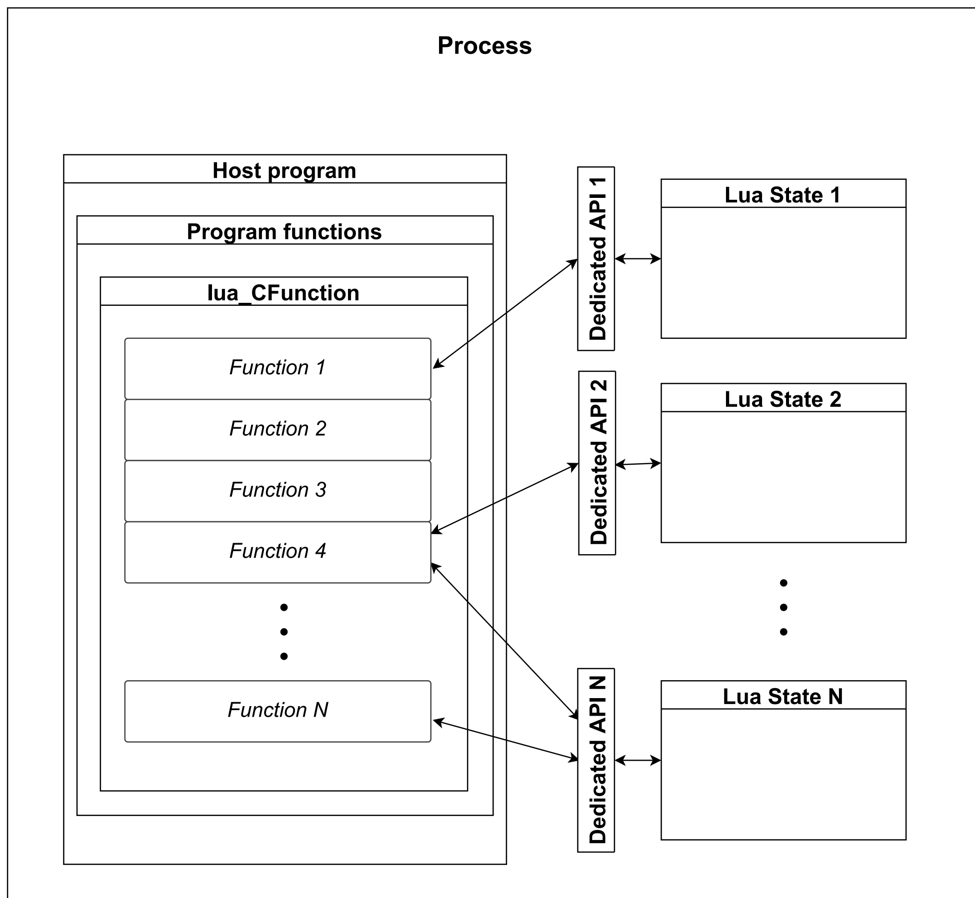

Figure 2.1: *A diagram showing the architecture of a program using a state-based sandbox. Each State has its own dedicated API. All communication is done through the API.*

as an unsafe protection domain, which should have no unrestricted privileged functions accessible.

Policy control with this sandbox is done exclusively in the host application, potentially replacing some of the standard library functions or removing them outright.

2.2.2 Function-based Sandbox

The function-based approach treats individual functions as protection domains.

A function cannot affect anything outside of its environment, upvalues, arguments provided to it and values returned from other functions. Functions created in this environment cannot access anything more than these values.

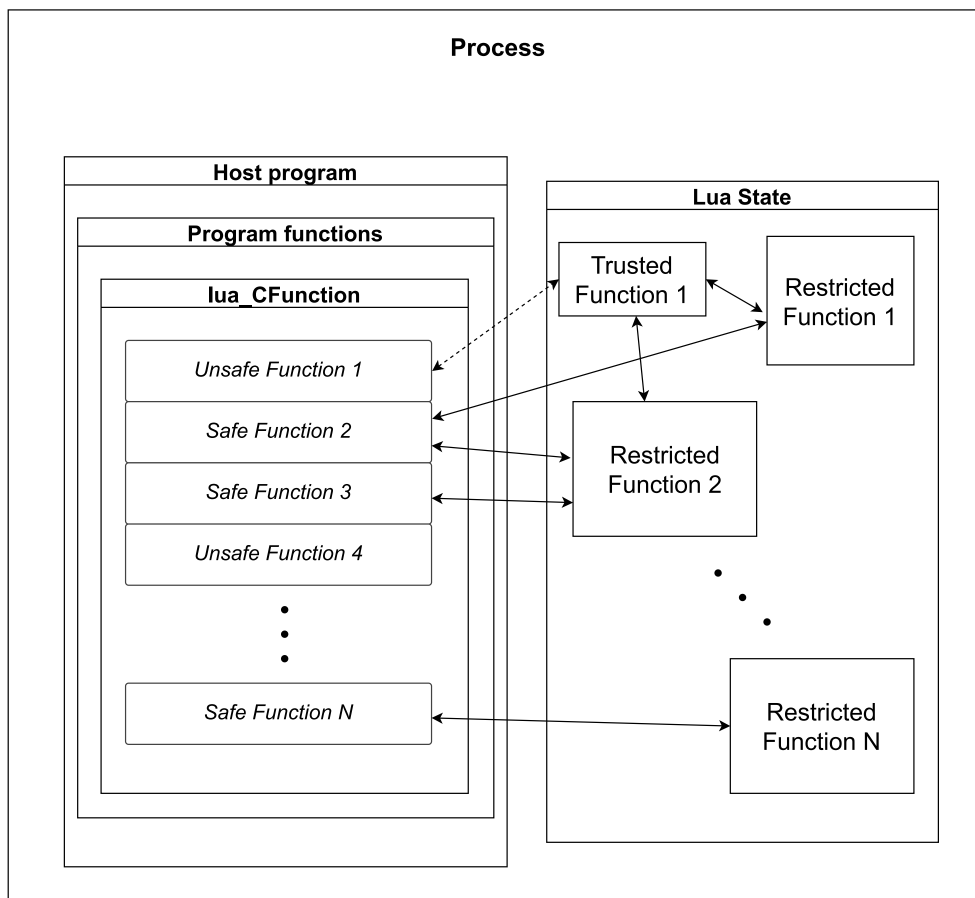

Figure 2.2: *A diagram showing the architecture of a program using a function-based sandbox. A single State contains both trusted and untrusted code. The untrusted, restricted code cannot access unsafe functions directly. Restrictions may be implemented both in the host and in Lua.*

This restriction means the functions returned or otherwise created from a domain are also automatically confined to the same domain.

This has large implications on how much harm functions injected into other protection domains can realistically do, as the function only grants the protection domain access to parameters passed into it. The replaced function may also return arbitrary values, but the damage which can be done is severely limited.

Policy enforcement has to be done by the functions provided to the individual protection domains. It can be done in Lua using techniques from section 3.2 or in the host program.

This means that a single State may contain privileged protection domains with access to dangerous APIs alongside untrusted user code. If incorrectly

isolated, the dangerous functions may be leaked or called with incorrect arguments.

Sharing values is very simple with this sandbox, as values can be passed as a function argument or saved in a shared table. This, however, allows for accidental sharing of values. Since shared objects can be modified, this may result in unwanted communication or even attacks between domains.

This entire sandbox trivially breaks down by making the 'debug' library accessible. The library contains functions that allow for accessing and changing all values accessible in a given Lua State.

2.3 Referencing Variables and Values

This section talks about how values may be shared between different functions and in turn sandboxes. It forms a basis of what needs to be isolated and what may be shared with what restrictions. It may also shine some light on issues with copying values between sandboxes.

2.3.1 Variables

A variable refers to an identifier under which a value is stored. The language does not provide a way to create arbitrary references to variables.

Local variables may be passed by reference only as upvalues which need to be bound on function creation. Upvalue variables a function contains may be referenced in the same manner.

In version 5.1 no built-in method for changing what variable upvalues are bound to exists. From version 5.2 functions for binding together multiple upvalues exist in the form of the *upvaluejoin* API. No API exists for creating a new upvalue from a local variable in a thread.

In all versions, the debug library and the C API allow for accessing and changing the values of upvalues and locals.

If these API are not readily accessible, some security guarantees can be inferred. Because the code of a function cannot be changed after creation, all local variables and upvalues can be assumed to be isolated from any outside tampering.

Global variables cannot be passed by reference individually. The environment of a function has to be shared. This works as if sharing just another table.

2.3.2 Values

Values other than objects are not considered at all in this section, since they do not get passed as reference and are implicitly copied.

Sharing tables implies all keys and values inside it can be retrieved and changed. Techniques restricting reads and writes to a table exist. Extra care

has to be taken when either the keys or values are objects as changing them changes the state of the table, even though it is not directly modified.

Dangers of functions passed as a reference depend on the restrictions placed upon a sandbox. The bytecode may not be changed at all, though it may be retrieved. The environment and upvalues may be retrieved and changed depending on provided functions.

Threads, coroutines, including the current running thread, represent the entire running stack. It is possible to retrieve and manipulate all functions on its stack. In this context, the function locals may also be retrieved and modified.

Full userdata have no inherent properties. Their use is defined by their metatable or specific program-defined functions which take userdata. The user values bound to these objects have to be shared explicitly via the C API. The values which are actually stored in the memory block may only be changed by the C API.

The individual metatables of tables and userdata also get shared by the virtue of the objects being shared.

2.4 Read only values

This section is only relevant for a function-based sandbox, as a state-based sandbox has no need for additional restrictions on shared values — all shared values are copied.

In the context of a function-based sandbox sharing values also has to be done. Sharing an object, which is passed by reference, without it being changed maliciously is important. One option is to copy over all values. This operation is expensive and not trivial. This leaves the option of somehow ensuring that these shared values are immutable.

This is very straightforward in the case of threads, functions and userdata. Functions that may manipulate these objects need to be restricted or removed outright.

Tables are a more complicated example. Steps have to be taken to ensure the metatable of a table does not get changed. Then the *__newindex* is defined to ensure no new key/value pairs can be assigned. This will not stop already existing keys from getting assigned a new value. Proxy objects are used for this.

By creating an empty table, all assignments are routed to the *__newindex* metavalue. To ensure values can still be retrieved, the original table is set as the *__index* metavalue. The danger of this approach comes in the form of raw access. Raw access methods can and will change the values of the proxy table. If these methods are accessible, a proxy table has to be created for each sandbox individually.

Alternatively, a method can be created in the host program which will provide a proxy userdata object. This object is guaranteed not to be mutable and may be shared freely. It is vital that the function creating the userdata cannot be used to change the metatables or other properties of unrelated userdata. Otherwise it could be used to break the assumption about mutability of userdata from Lua.

Userdata objects are by definition immutable. Their metatables may still be retrieved and potentially changed. This has to be restricted. Proxy tables will not work in this case as metavalues are queried for using raw access. So, for sensitive metatables the access has to be restricted outright.

2.5 Applying Resource Limits

For many applications, restricting how long a particular program may run is a valuable option. This prevents needless wasting of system resources for malfunctioning or actively malicious components.

2.5.1 Restricting operation time

A naive implementation might think this can be trivially done by setting a hook. Lua allows for setting a hook that will fire an event on some actions such as calling a function or after a set number of lines. While this can effectively prevent the code from running for too long it has its flaws. If a C function which takes a very long time is found, the time per line count will be much larger.

Even if that were solved, the issue of how to recover from the error comes. If the sandbox has access to pcall a single error will not be sufficient in returning to normal context as the error will simply be caught by the sandbox. Repeated throwing of errors or restricting of protected mode calls is required.

Restricting protected calls may be hard, as unrelated functions inside the sandbox may internally use protected calls.

2.5.2 Restricting memory allocations

Memory allocation restrictions can only be done per State in Lua. This is done using a custom allocator which counts the total memory used and refuses to allocate above the limit.

Lua simplifies the counting by using a *realloc*-like API that additionally passes the original block size as an argument.

2.6 C interface

The C API communicates with Lua using a virtual stack. When a function is invoked from Lua, it has a virtual stack assigned which is used for passing

variables to Lua. When a new State is created, values are added to it using the stack. The programmer has to take care that the virtual stack does not overflow by expanding its capacity as appropriate.

When checking if a function's Lua stack overflows, the stack space internal Lua functions use does not need to be considered. Lua ensures the stack may grow an extra *EXTRA_STACK* spaces for internal use. The extra space ensures that, unless a programmer has made a mistake beforehand, the virtual machine will always have enough space on the stack for internal calls, such as calling metamethods.

The API is partially error-prone as the stack prevents the C type-based static analysis from preventing mistakes. All standard Lua values are provided with type-checking retrieval functions, meaning they cannot be confused for data of another type. This does not apply to userdata. Lua only has one userdata type while the host program may use userdata for storing multiple different types of objects.

The C API also exposes a Registry, which is a special table associated with a Lua State that may contain confidential values. This table can be potentially passed into Lua as it is just a table.

Lua allows for some type-safety with userdata, if some guidelines are followed. The function *luaL_newmetatable* cretes a named metatable. The name and this metatable are stored in the Registry. When creating a userdata object it is assigned this metatable to make it associated with this type. Later *luaL_checkudata* can be used to check whether the metatable associated with a given userdata object is the same as is expected.

If the Registry is made accessible to Lua code, or it becomes possible to set a metatable to a userdata object, this type-checking will no longer work.

Neither the lifetime nor the location of Lua objects between calls is guaranteed. If an object needs to be persistent, it must be stored in Lua. The Registry is also used for this purpose. If this convention is not adhered to, this can lead to use-after-free bugs.

While high-level frameworks for Lua exist for creating bindings conveniently, it does not mean that these are safer. As is shown in section 5.3, it can be quite the opposite.

2.7 Standard library safety

This section covers all of the standard library functions and ways in which they may be abused in the context of both of the mentioned sandboxing approaches.

The functions are grouped by the library which includes them, rather than the name of the table they are located in, as some of the functions are placed in the global environment under no other table. Deprecated functions are not included. Only the latest releases of official Lua versions have been checked.

Lua implements all of these functions in C so they could be abused if incorrectly implemented as they are guaranteed to run in an unrestricted context. No major mistakes in the function implementation were found, so this section will mostly refer to the intended function semantic being dangerous. Coercing C functions to run for a very long time is relevant as mechanisms for restricting script runtime are restricted to Lua code itself.

The functions very commonly handle having more arguments than needed by simply ignoring the excess arguments.

2.7.1 Base

Allowing the *collectgarbage* function in and of itself does not sound like a big deal, but it greatly increases the attack surface on the garbage collection. Numerous bugs concerning garbage collection have been reported in the past [42] with one allowing for escaping the Lua State in version 5.4 [43]. Unless a good reason for keeping this function in the sandbox exists, removing it is recommended. Alternatively, restricting it to only accept the 'count' argument is safe.

Functions *dofile*, *loadfile* allow for loading of arbitrary files. These files may also be interpreted as Lua code and Lua bytecode. Even if the functions were restricted to only load uncompiled code, unrestricted filesystem access is still an issue. The *load* and *loadstring* functions allow for loading functions from strings. The strings may also be interpreted as bytecode, which needs to be restricted. All of these functions also, by default, load the function into the global environment. That needs to be prevented to retain the expectation that all functions created in a protection domain remain in the domain.

The pair of *get/setmetatable* functions allow for accessing metatables unless *__metatable* is defined. This may have severe security implications for userdata objects and proxy objects.

The *print* function allows for arbitrary writes to stdout. This may be unwanted since it can facilitate communication with other programs.

All raw access methods *raw** will have implications for shared tables.

The *next* method is used to iterate a table using raw access in all versions. If trying to restrict the function, the first return value of the *pairs* function also has to be restricted as it is this function.

Functions *ipairs*, *pairs* allow for iterating over a table using raw access in version 5.1. In version 5.2 they still use raw access but may be overridden by the metamethods *__ipairs* and *__pairs*. Since version 5.3 the ipairs function respects metatables when querying and its metamethod is deprecated.

The function *tostring* takes a single argument and converts it to a string. For many types, it returns a value in the form 'type: address'. The exact object of which an address is collected varies but if given a good function, getting a pointer to where C functions are allocated in memory is nigh guaranteed. This pretty much breaks or at least reduces the security guarantees

which come from using Address space layout randomization. If the *__tostring* metamethod is defined for the argument, it is called instead.

The *_G* variable is potentially unsafe if not set to the same value as the function environment. It may also cause issues if the environment is otherwise not directly accessible — this could lead to the metatable of the environment table getting leaked.

The *get/setfenv* functions allow for retrieving and resetting the environments of functions. They are only present in version 5.1 but have severe implications if not restricted at all. They can provide access to the global environment and more.

The function *unpack* is using raw access but is only present in version 5.1. It was moved to the table library later.

The function *newproxy* is undocumented, only present in version 5.1, and breaks the assumption that userdata objects cannot be created from Lua. It does not, however, have other security implications.

2.7.2 Coroutine

No dangerous functions have been found in this library. Until version 5.4, when both C and Lua functions may be turned into coroutines, it can be used for detecting the type of a function.

2.7.3 Debug

As the manual clearly states, this library is not meant to be used in usual Lua code and is meant for debugging only.

If a pure state-based sandbox is used, it may be possible to include this library as well. Almost all functions in this library break some basic assumption about Lua code. Do not include this library in a function-based sandbox.

In a state-based sandbox, allowing access to this library may not lead to an immediate escape from the sandbox, but extra care has to be taken when interacting with Lua. Specifically, the Registry, which should be a safe place to store values, will be compromised. This has other implications, since the Registry also stores a copy of all loaded libraries, meaning that some forms of deletion will still result in data getting leaked. Moreover, no metatable will be secure.

If the debug library is allowed in full, especially when concerning manipulation of local values and upvalues, further research needs to be done to ensure this cannot be used to attack the virtual machine as will be hinted at in section 3.3.

The function *debug.traceback* may be safe if string-based knowledge of the call stack is not considered essential.

2.7.4 Package

This package is unsafe in its entirety. It allows for loading of both Lua files and native compiled libraries.

2.7.5 IO

This library is not to be included in any sandbox. It allows for accessing the filesystem and even launching other programs. Even leaking opened file userdata objects is unsafe as they contain callable functions for reading and writing to the opened file in the metatable.

2.7.6 OS

The os library is also unsafe except for a few functions. I will not be listing why each function is supposed to be restricted, as it is self-evident. I do not believe it is possible to reasonably restrict these functions aside from removing them from the environment outright.

Only the functions *os.clock*, *os.time*, *os.difftime* should be allowed.

Special care has to be taken with the function *os.date*. The standard library uses the C functions *gmtime* and *localtime* which may not be thread-safe.

2.7.7 Bit32

This library is endemic to the 5.2 version. It was later superseded by native binary operators. It contains no unsafe functions.

2.7.8 Utf8

Nothing of note was found in this library.

2.7.9 Table

The tables library is mostly safe. Up until version 5.3 it ignored metamethods of its arguments, meaning that all operations were done using raw access and it could only operate on tables. It only ever operates on numeric keys, but it could be an issue for a function-based sandbox that depends on rawset and rawget not being accessible for numeric keys. Rawset can be emulated with repeated calls to *table.insert* and rawget with calls to *table.remove*. The deprecated foreach functions also allow for looping over keys using raw access.

The 5.1 version of *table.insert* does not check if pos has a non-negative value. So the call ‘table.insert(, -2147483647, ”err”)’ will result in a long freeze of the application. Of the other mentioned interpreters, LuaJit suffers from the same issue even though the call finishes in a reasonable time.

2.7.10 Math

Not much is to be said about the math library. It is a straightforward wrapper for the C math library. Of note are the *random* and *randomseed* methods which could very well affect other sandboxes or the program itself. As such, they are potentially unsafe for use in both types of sandboxes. Though it is to be noted that the C random functions are not meant to be used for security purposes in any case.

2.7.11 String

Including this library automatically sets the metatable of all strings to one which has the strings library as the *__index* metamethod. This is significant for security purposes, as this creates another way in which the strings library may be referenced. Alternativelly the fields of the metatable could be modified leading to further issues.

Of interest is the *string.dump* function which dumps the bytecode of a Lua function. This bytecode can be analysed externally to understand the internals of a sandbox. Some information is also retrieved if an error occurs while dumping. In all versions, an error while dumping cannot happen unless the function is a C function, thus betraying that this is an interesting function to try to exploit.

2.8 Summary

Functions that invalidate basic assumptions about the code need to be carefully restricted. These basic assumptions are: the Registry is not accessible from Lua, metatables of userdata cannot be set from Lua and are made inaccessible if they contain unsafe values, user values associated with userdata are not accessible from Lua, and finally local values, upvalues and environments of functions cannot be changed and retrieved by other functions.

In a true state-based sandbox the assumption about functions is not strictly necessary as far as the language is concerned. The virtual machine may also require that some of these guarantees are not broken. If alternative approaches are taken to verify the integrity of userdata, restrictions on the Registry and userdata may also be lifted. Though care needs to be taken so not only the global environment but also the Registry does not contain any exploitable values.

A sandbox may add additional restrictions, especially for raw access methods, if they would violate the integrity of shared data.

Loading of bytecode needs to be forbidden unless the source is guaranteed to be safe. Functions with access to the filesystem and ones which allow for executing files should also be avoided unless properly restricted.

Chapter 3

Attacking Lua

In the previous section, the guarantees expected from Lua were mentioned, both for the expected integrity of values when interacting with Lua from C and for the assumptions that need to be fulfilled for securely restricting values using Lua.

This chapter covers some Lua sandboxes and methods which may be used to attack Lua or these sandboxes.

3.1 Attack Surface

When a sandbox of either type is constructed, all the functions Lua may access are the attack surface. As Lua provides no inherent type-checking, no attack vector can be trivially dismissed.

Since the interpreters are composed of many parts, the code in a sandbox will always have the option of trying to attack the implementation of the interpreter, rather than contending with the language-based sandbox.

The loading of malicious bytecode is one example of such an attack, it attacks the implementation of the virtual machine. In a much similar manner, the compiler and garbage collector could be attacked. No currently known, unfixed vulnerabilities of this type exist.

The bug reports of the interpreter of choice must be regularly checked. Save for checking the implementation of the interpreter, this is the only way attacks on the implementation can be secured against.

3.2 Sandboxing Methods

Whatever method is chosen for sandboxing, it is important to sandbox functions using a whitelist rather than a blacklist. By using a blacklist, some unsafe functions may inadvertently get included after an architectural change or an update.

One exploit of this type is quite recent. Due to a difference in a package on Debian, the unsafe Lua library package was loaded and not hidden. [44]

It is vital that all libraries and functions included after the fact are also secured since they may pose the same security vulnerabilities as the standard library may.

When a function-based sandbox is created, many dangerous functions can be wrapped by using the following pattern.

```lua
local _unsafeFunction = unsafeFunction;
function unsafeFunction(...)
  -- do argument sanitisation
  return _unsafeFunction(...)
end
```

This replaces the unsafeFunction in the current environment. The original unsafe function is stored only as an upvalue of this function and cannot be retrieved.

Simply erasing unsafe functions is also possible:

```lua
for key,value in pairs(os) do
  if(key ~= 'time') then
    os[key] = nil;
  end
end
```

This irreversibly removes all values but the one associated with key ‘time’ from the table *os* from the current environment.

It is to be noted that if the original functions were ever stored anywhere else, these protections may be bypassed by retrieving them from there.

Both of these methods may be utilized in a state-based sandbox if the relevant assumptions are not violated.

3.3 Bytecode Attack

Bytecode being insecure has been mentioned multiple times in this thesis. Exploiting of Lua bytecode has been shown possible for PUC-Rio Lua 5.1 [45, 46], Lua 5.2 [47] and LuaJIT [48].

Attacks on the official interpreter use multiple assumptions in the virtual machine. In both attacks the bytecode instructions concerning for loops make assumptions about the types of values when inside a for loop. The types are only checked when entering the loop. Bypassing the check or overwriting them inside the loop leads to addresses of Lua values leaking, which will be needed later. The attack on version 5.1 breaks an assumption in the bytecode for creation of functions, eventually rewriting a function pointer with a crafted string. The version 5.2 attack instead exploits the assignment operator, over-

writing the function that is supposed to be called when returning from a call, with a specially crafted string. [45, 47]

This shows a way in which the debug library may be used to attack an implementation. If the execution were paused at a precise point in a for loop, a value of a local value changed, and then execution were resumed, the equivalent of the first part of the attack would be reached. This could be done by using the hooks briefly mentioned in the section about limiting process time, as these get called just before an instruction gets executed.

No publicly available verifier of bytecode exists for any interpreter of any version at the time of writing this thesis.

3.4 Other Known Bugs

This section only contains known bugs for the PUC-Rio interpreter. It shows why using an outdated version of an interpreter can be extremely dangerous.

For this reason, using outdated versions of the PUC-Rio interpreter, and especially the 5.1 version, is not recommended. Outdated major versions are explicitly not updated even for security concerns. And updating the latest version is just as important as new exploits commonly appear. An example of this is an attack on the garbage collector in Lua 5.4.0-5.4.3 [43]. The latest bugs do not necessarily get a minor version increase when they are fixed — they just get fixed and listed in the bugs section [42].

Numerous bugs such as crashing [42, #5.2.0-5 , #5.3.4-2 , #5.3.4-3], memory hoarding [42, #5.2.0-1] and stack overflow [42, #5.2.2-1] exist for the latest Lua 5.1.

Lua 5.2 does not fare much better, returning incorrect values in some cases [42, #5.3.0-3], crashes [42, #5.3.0-3 , #5.3.3-1] and even has incorrect bytecode generation [42, #5.3.4-1].

Lua 5.3 is still not perfect with crashes [42, #5.4.0-9 , #5.4.0-11] being possible.

Note that the mentioned bugs are not an exhaustive list and contain only the most important issues. Most of these do not require any dangerous functions and are never going to be fixed in the corresponding versions by the authors.

3.5 Notable Lua Interpreters

Multiple different Lua interpreters exist aside from the official implementation. When deciding to implement a sandbox in Lua, the usage of a different interpreter may make it easier. Note that this list is far from exhaustive — the chosen interpreters were either commonly mentioned when doing research for this thesis or have interesting extensions.

The official implementation does not hinder sandboxing in any way, but it is not made a priority, as can be seen by the number of dangerous functions in the standard library. It is, however, maintained and has a long history behind it.

LuaJIT[2] is an interpreter of Lua which in addition to compilation into bytecode then does just-in-time compilation into machine code. It is not designed for sandboxing. The interpreter is compatible with Lua version 5.1 with some extensions from later versions included.

The Luau interpreter[3] is used in the game Roblox where Lua is extensively used as a scripting language. It is an interpreter of Lua written in C++ which retains complete backward compatibility with Lua version 5.1. [49]

It ships with a sandboxed environment out of the box. None of the libraries considered dangerous for a state-based sandbox are included in the interpreter. They may be included by a host program, but an implementer has to explicitly create and register them. For its interactive interpreter, it ships a restricted implementation of *require* which only loads Lua files.

A much more interesting are some of the non-standard shipped sandboxing features. It replaces the unsafe garbage collection metamethods stored in the metatable of an object by using its own mechanic which is constrained to the C API [37]. It also makes read-only tables an integral part of the virtual machine, thus avoiding proxy objects.

It also modifies the language to optionally support type-checking at compile time.

The Luau interpreter itself is sufficiently constrained to be considered secure after including its standard libraries, unlike the official Lua interpreter.

3.6 Sandbox Implementations

Many different Lua sandbox implementations may be encountered based on the use case and strategy used. Some which were found interesting are listed here.

3.6.1 Kikito Lua-Sandbox

A sandbox by Kikito [4]. It explicitly warns that for Lua version 5.1, disabling bytecode is not possible. This is not completely true. It cannot be done portably, but if we were to restrict ourselves to the PUC-Rio 5.1 implementation, it is possible to restrict this behaviour.

Citing the Lua Bugs page '*To avoid running precompiled code from untrusted sources, raise an error if the first byte in the stream is the escape character (decimal 27).*' [42]

[2] https://luajit.org/
[3] https://luau-lang.org/
[4] https://github.com/kikito/lua-sandbox

Other interpreters running this version of Lua have generally extended the language to allow for restricting the loading of bytecode.

Code in this sandbox is correctly run in a restricted environment that contains no unsafe functions as outlined previously. By default, the code is loaded from a string, though the exact same principle can be applied when running code from a file.

This sandbox allows for restricting the runtime of a function using the quota mechanism. It only attempts to throw one error so it can be bypassed. A possible solution to this is outlined in a pull request[5] I have made for the repository. The attempted solution is based on repeated throwing of errors until the sandbox is returned from.

While it does state why some functions were restricted, it fails to warn that if *getmetatable* were to be passed into the sandbox, all other sandboxes could be affected as the library tables are shared via proxy tables. It only mentions it could affect global objects. This would not affect the global versions of the libraries.

3.6.2 RyanSquared Lua-Sandbox

This sandbox[6] ignores all Lua guarantees and instead proceeds to enforce limits using the operating system.

No research has been done into how resilient this sandbox is, as it is out of the scope of this thesis. It serves as a reminder that using the operating system for sandboxing may be just as or even more effective than using Lua.

3.6.3 Factorio

The game Factorio[7] utilizes Lua not only for mods but also for internal implementations.

It uses a very unrestrictive state-based sandbox. It loads different Lua files with different accessible values in different stages of the game initialisation [50]. But many of the potentially unsafe libraries, including the debug package, are included [51]. As I have previously mentioned, in a state-based sandbox this is not a critical mistake, but the implementation has to work around this.

The inclusion of bytecode loading is very surprising. Factorio either uses an interpreter with a built-in bytecode verification or has made a verifier of its own. A brief test of loading explicitly malicious bytecode was done using functions from [52], as [47] is not applicable due to its usage of coroutines. Malicious bytecode was prevented from loading, while known safe bytecode of a dumped function was loaded without issue. The unsafe bytecode was also tested in the PUC-Rio 5.2 interpreter and immediately resulted in a crash.

[5] https://github.com/kikito/lua-sandbox/pull/12

[6] https://github.com/RyanSquared/lua-sandbox

[7] https://factorio.com/

This may also fix the potential attack mentioned in section 3.3 but has not been tested.

Potential resource exhaustion exploits are not an issue in this game as it has no competitive multiplayer and they cannot be used for any gains.

In my scan of the environment, I have found no garbage collection metamethods and as a state-based sandbox is utilized, shared objects are a non-issue.

3.6.4 Libluabox

This sandbox[8] is meant to be a stripped-down version of official Lua 5.1. Much like the Luau interpreter, it does not contain unsafe functions. Unfortunately, while it seems to be correctly removing unsafe libraries, it fails to consider the bugs in the original interpreter. None of the bugs listed in [42] are fixed. It should be avoided.

[8] https://gitlab.com/numzero/libluabox

Chapter 4

Automatic Environment Crawler

A tool was created to analyse sandboxes made in Lua. It attempts to log every value which can be found in a function's environment. The logged values can then be more effectively inspected.

This tool is split into two parts. An environment dumper and a dump analyser. The rationale behind separating the two modules is that interaction with the sandbox may be severely restricted. This way, only retrieving a string in some way is necessary.

Examples of dumped environments can be found in the attached medium. Environments dumped from Factorio and from OpenMW, which is mentioned in the next chapter, are included.

4.1 Motivation

Analysing the environment of a function is something which will always be done when analysing a Lua sandbox. All Lua code is loaded as a function, after all.

No matter the type of sandbox used, the environment will contain any and all values that may be interacted with.

By getting a full enumeration of all the accessible values in an environment, we get a comprehensive list of all values exposed to the sandbox.

All of this could be done by hand by checking the source code of the host program and seeing which values are provided. Dumping every value programmatically instead can both speed up the research of a sandbox and may also show otherwise overlooked values.

4.2 Dumper

Using functions from the standard library, the tool attempts to crawl the entire environment of a function.

Then these crawled values are serialised into a string, which is later transferred into the analyser.

4.2.1 Crawling

All values are recursively explored, starting from a value which is passed into the dumper. This value is typically the environment of a function.

To ensure shared metatables are not missed, dummy instances of values of all types that may be created from Lua are also analysed.

All values are queried for their metatables using the *getmetatable* function.

An attempt is made to get the environment and upvalues of functions. The *getfenv* function and the debug library are used for this.

Getting all values accessible from tables and userdata is a difficult task. A part of the values may be gleaned from the metatable. Many others have to be explicitly queried for. Tables can be explicitly walked using raw access. This is done using the *next* function and emulating a for loop. Raw access is meaningless for userdata.

To make full use of the potentially overloaded *pairs* function, multiple things have to be done. Firstly, it has a well-defined interface. This allows for calling it and then checking if the return values leak anything new. Secondly, it is used in a for loop to retrieve keys and values of tables and userdata.

Tables and userdata are further explicitly tested for common metatable keys as metatables may easily contain sensitive functions.

4.2.2 Serialising

The only major difficulty with serialising is getting a unique identifier for all values.

The *tostring* method can be used if it is not restricted and still leaks pointers. Alternatively, a function which keeps track of all values may be used. This function will create some identifier and consistently return it for the same calling value.

4.3 Analyser

The analyser part of the tool is designed to run separate from the dumper. It loads one or two strings created by the dumper.

The loaded environments can be then scanned for shared values and more. A full enumeration of values of a given type can be listed. This can in used to find any differences between the environments.

The ways a value may be reached along with other information about how it can be used later are accessible in the analyser. For example, all the paths to the string *__gc* and all the values it points to in all tables may be listed.

4.4 Limitations and Further Improvements

As of now, this tool cannot effectively serve as the only point of analysis. While it can reliably analyse most types, functions stop it far too easily.

Even though values that functions return can be considered to be a part of the environment, there is no way to retrieve them automatically. This means that functions can be very effectively used to employ security through obscurity.

A prominent example of security through obscurity is proxy tables with no overload for the *pairs* function. The hidden tables cannot be walked, but if the key is known, a value is trivial to extract.

In Lua, not all cases of this are a bad security practice. If the required keys are objects, they have to be explicitly passed from a protection domain which contains them.

These function-based obfuscations could be worked around in a couple of ways.

Getting a database of common table keys and explicitly querying for them would cover a subset of possible values. Just the names used in the standard library would help to an extent.

A couple of methods that allow for getting the exact type of function have been mentioned. The dumper could be extended to make use of them. This would differentiate which functions implementations can likely be found as plain Lua code and which must be searched for in the host source code.

No fuzzing or any other testing is currently done for found functions. For example, using known error messages, the expected parameters of these functions could be retrieved.

Even though the scanning is not perfect, the tool was very useful in the analysis done in the next chapter.

Chapter 5

OpenMW

OpenMW[9] is an open-source game engine that is designed to be compatible with the engine of the game *The Elder Scrolls III: Morrowind*.

In the next version, a new Lua scripting interface will be added using LuaJIT and the C++ Sol2 framework[10].

An analysis was done to see if the standard library functions are restricted properly as outlined in this thesis. This includes making unsafe functions inaccessible and having proper isolation between sandboxes on the Lua side. An extensive analysis of the provided C++ API was not done except for the *require* function that was concluded to be safe.

5.1 OpenMW Lua Sandbox Design

OpenMW utilizes a function-based sandbox — all scripts are run in the same Lua State. While most of the dangerous libraries mentioned in section 2.7 are not present in the global environment, the few that are can create a lot of issues.

I have incorrectly claimed in an issue[11] made for one of the exploits that a malicious global function would not be able to do harm. This is not true. The debug library is included and functions for loading bytecode are not restricted.

Functions provided to the user sandboxes are a well-restricted subset of the unsafe functions.

Compared to official Lua 5.1, the *pairs* and *ipairs* functions have been replaced with alternatives implementations which may be overloaded by the *__pairs* and *__ipairs* metamethods respectively.

Individual Lua scripts should be isolated from directly affecting one another except for the explicit interface provided by the engine.

[9] https://openmw.org/en/

[10] https://github.com/ThePhD/sol2

[11] https://gitlab.com/OpenMW/openmw/-/issues/6694

Standard libraries which are to be shared are loaded into the global environment and shared based using proxy userdata objects to ensure non-mutability. The *require* function also returns shared tables protected in the same manner. The function responsible for establishing this restriction is called *makeReadOnly*.

5.2 Mistakes with makeReadOnly

The implementation of *makeReadOnly* was done incorrectly. The original table was getting leaked in multiple ways. This resulted in the global environment and environments of all other sandboxes getting compromised.

All foreign and internal OpenMW scripts loaded from the filesystem are sandboxed. Only two types of functions run in the global environment. Functions provided by the host program which contain no usage of these compromised libraries. Functions defined in the global scope that also do not interact with any of the compromised libraries. Moreover, these functions have most of their dependencies stored as upvalues.

The section 2.2.2 briefly mentions how much an injected function can realistically do. Since no function interacts with the compromised values, the vulnerability resulted in no further exploits.

A mod that asserts whether these vulnerabilities are still present can be found in the medium attached to this thesis.

5.2.1 Accessible Metatables

The first flaw of *makeReadOnly* were unprotected metatables. The *getmetatable* method was left unchanged, and no protection was offered in the form of a defined __*metatable* value in the metatables. Since the original table was stored as the __*index* field of the proxy userdata, accessing the original table was a trivial operation.

```
originalTable = getmetatable(wrappingUserdata).__index
```

The metatable of string values was also accessible. It had not been protected at all.

Multiple solutions to this issue exist. The use of the __*metatable* field could be enforced to ensure protection. This would result in more upkeep down the line. As the program keeps sensitive functions in the metatables of other userdata objects, a blanket approach was deemed more appropriate.

Getmetatable was restricted to only work for tables and no other value.

5.2.2 Incorrect pairs overload

Both the *pairs* and *ipairs* method were affected but the latter has the same behaviour so only the former will be described.

When implementing *makeReadOnly*, retaining a consistent API by making pairs callable on the wrapping userdata is desirable. The sandbox did not take into account that the standard library *pairs* function returns the original table.

By writing out all the calls happening behind the scenes, we get three equal expressions.

```
pairs(wrappingUserdata)
pairs(getmetatable(wrappingUserdata).__index)
pairs(originalTable)
```

In a for loop this looks safe as the second returned value is not accessible, but it can be explicitly retrieved.

The solution that was agreed upon looks as follows:

```
function pairsForReadOnly(v)
  local next, t, key = pairs(getmetatable(v).__index)
  return function(_, k) return next(t, k) end, v, key
end
```

This retains the original properties of the pairs function while ensuring the original table does not get leaked.

5.3 Accessible Insecure Garbage Collection Metamethod

Since the raw access methods are accessible in the sandbox, any shared table is mutable even if protected via a metatable.

After all of the previously mentioned issues had been solved, a basic scan of the environment was done to see if this proves to be an issue. Using the previously mentioned tool, four shared tables were found. Two of those tables were metatables and two were tables contained in them. The cause was the Sol framework.

The Sol framework does not take into account that making metatables or even metamethods accessible to the code is dangerous. All Sol-mapped userdata objects in the project contain this behaviour. Mapping is created via the *sol::new_usertype* and *sol::make_object* functions. When these userdata objects are explicitly queried for metatable keys, the associated metavalues will be returned. The *__index* metavalue sometimes contains the metatable itself, resulting in the observed behaviour. Even objects which do not leak the entire metatable leak their metatable fields. Specific examples are given in the issue[12] description.

Among these fields is the *__gc* metamethod. The mapping of this method consistently points to the C++ template function:

[12] https://gitlab.com/OpenMW/openmw/-/issues/6698

```
template <typename T>
int usertype_alloc_destruct(lua_State* L) {
  void* memory = lua_touserdata(L, 1);
  memory = align_usertype_pointer(memory);
  T** pdata = static_cast<T**>(memory);
  T* data = *pdata;
  std::allocator<T> alloc {};
  std::allocator_traits<std::allocator<T>>
                                    ::destroy(alloc, data);
        return 0;
}
```

The first value from the Lua virtual stack is to be interpreted as a userdata object. If it is not userdata, *NULL* is returned. That results in immediate program termination due to a page fault after null pointer dereference.

In case it is a userdata object, Sol internally uses another layer of indirection and gets the actual location of the object in memory. Then it calls the destructor of the object in that memory location in a roundabout C++ way. Unlike how Lua does its mappings, there are no attempts to ensure the userdata was of the correct type in the first place.

This results in type confusion bugs. Calling another object's destructor on an object is possible. Calling a destructor on an object repeatedly is also possible. The function has no safeguards against this at all. While most objects in the API contain trivial destructors, making this harmless, this is not always the case.

Namely *LuaUtil::LuaStorage::SectionMutableView* that contains an interesting value *std::shared_ptr<Section>* as its first member variable can be used. Repeatedly invoking its destructor results in the Section object being freed and destroyed while still being used by other parts of the program.

Some of the other structs can be used to create arbitrary in place of the shared pointer. For example, the *LuaUtil::Vec4* object allows for setting arbitrary values to the 4 internal floats. In theory, an attacker could forge the 16 bytes in the vector in such a way that they contain meaningful values (OpenMW makes no attempts to mitigate issues mentioned in 2.7.1) which then get the *Section* destructor called upon them and the memory gets freed.

In practice, this is more difficult as the *std::shared ptr* contains another layer of indirection and the destructor of *Section* is not trivial.

As of now, making the game crash at will using this method is possible. Freeing data structures and then continuing using them, bringing about use-after-free vulnerabilities, is also realistic. No method of exploiting this in a meaningful way purely from Lua was found.

At the time of writing this thesis, the issue has not yet been resolved.

Chapter 6

Suggested Improvements

Possible improvements of sandboxing in Lua can be generally split into two types. One option is improvements done to the language itself on an interpreter level, the other is improvements made by sandbox implementers.

As of now, the language can be used for sandboxing, but the potential ways a sandbox may be implemented are not documented. Pitfalls which have to be avoided when designing a sandbox also are not listed anywhere and each sandbox implementation has to pave its own way.

6.1 Sandboxes

Sandbox implementations tend to not give exact specifications of why and how the sandbox works. When the user does not know why a restriction was placed onto a function or why a function was removed, the same behaviour may be brought back by another function. A sandbox which cannot be modified, while useful for basic isolation, is not flexible enough for many purposes.

Mentioning that using a state-based sandbox is a possibility would also be very useful. For example, the only semi-official page concerning sandboxes[13] makes no not of such an approach.

The page gives some good recommendations for creating a function-based sandbox but fails to show the dangers of functions reliably. For example, the *rawget* function is marked unsafe while the function *next* is regarded as fine even though it can be used to achieve the same functionality.

No such semi-official page exists for Lua versions later than 5.1.

The getmetatable function is also often regarded as universally unsafe, while that is not necessarily true for a state-based sandbox. By allowing its usage at least for tables, many commonly used features of the language become possible to use.

[13] https://lua-users.org/wiki/SandBoxes

A list of potentially unsafe functions on its own is meaningless without an understanding of which guarantees the sandbox requires. Instead, such a list has to be specified for each sandbox implementation.

A sandbox could also be described very precisely by an interpreter itself. If sandboxign were a priority, the interpreter could keep a subset of the standard library that is guaranteed not to directly interact with the OS and retains guarantees which the virtual machine requires.

6.2 The Language

Modifying the language so sandboxing becomes a bigger priority is possible, some parts of the design are inherently unsafe.

The most notable of these is the need for garbage collection metamethods being accessible from Lua. As of now, the language provides no way to guarantee that would ensure a value, which is loaded in a State, cannot be reached from any code in the State. As a result, all metatables have to be carefully secured to ensure no vulnerable methods get leaked. The garbage collection methods are an especially bad case of this, as they may often leave the destroyed objects in an unusable state.

By adding a way to keep these functions only for the C API, this issue could be prevented entirely.

An issue for all function-based sandboxes are shared tables. Currently, all sandboxes have to implement their own methods for making them read-only. This can be an error-prone operation as the issues in OpenMW show. By adding read-only as a part of the interpreter, much of this complexity could be avoided.

Lua version 5.1 does not have any way in which functions can be safely loaded from a string while being confined to the same function environment. This has been resolved in later versions but remains an issue for interpreters using this version.

6.3 Existing Solution

The Luau interpreter solves many of the mentioned issues.

The changes Luau did solve the most obvious issues a sandbox in the official Lua interpreter has to contend with.

The option to make tables read-only not only by restricting each function which may be using raw access and having to use proxy tables, but instead being done on an interpreter level is extremely helpful. It takes all the complexity of ensuring the metatable nor any properties of a table can be changed and transparently locks the table down without having to resort to complex interactions which may be broken by accidentally exposing a new API. This

essentially takes essentially multiple different API and combines them to only have to care for one function which may break everything.

Luau also removes the dangerous __gc metamethod from the API in all forms. While it prevents the metamethod from being callable for tables, that is not an issue for Luau, as the only version it is officially fully compatible with is 5.1 [53].

The interpreter does not solve the issue in a more generic way. Other metamethods may still be leaked into the sandbox.

Chapter 7

Conclusion

This thesis explores common sandboxing techniques and how they may apply to a sandbox in Lua. Based on this, different ways of constructing a sandbox are described.

Lua lets us create a language-based sandbox using functions and overwriting of variables. It allows us fine-grained control of functions and resources accessible to the user code, which is then run in isolation. That allows us to control which variables are accessible to different functions. We can thus segregate the untrusted user-provided code from our own trusted code while controlling which functions and resources the user can access.

Alternatively, a Lua State-based sandbox may be implemented. This approach requires preventing dangerous functions from getting loaded into the Lua State in the first place. The entire Lua State is assumed to contain malicious code.

Lua does not, however, contain mechanisms for access control on a more fine-grained level — such as which files should be accessible. That has to be implemented on a separate layer by individual programs embedding it.

The Lua standard libraries were analysed for potentially dangerous functions which may leak data or otherwise compromise a sandbox. Based on this knowledge, a tool that automatically crawls all values a Lua function may access was created. It was successfully used to show vulnerabilities in the implementation of a Lua sandbox of the OpenMW game engine.

Bibliography

1. GOLDBERG, Ian; WAGNER, David; THOMAS, Randi; BREWER, Eric. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In: *Proceedings of the Sixth USENIX UNIX Security Symposium*. 1996. Available also from: `https://www.usenix.org/legacy/publications/library/proceedings/sec96/full_papers/goldberg/goldberg.pdf`.
2. MAASS, Michael; SALES, Adam; CHUNG, Benjamin; SUNSHINE, Joshua. A systematic analysis of the science of sandboxing. *PeerJ Computer Science*. 2016, vol. 2, e43. Available from DOI: `10.7717/peerj-cs.43`.
3. SILBERSCHATZ, Abraham; GALVIN, Peter; GAGNE, Greg. Chapter 14 Protection [online]. In: *Operating System Concepts, 9th edition*. John Wiley & Sons, 2013. ISBN 978-1-118-06333-0. Available also from: `https://archive.org/details/operating-system-concepts-9th-edition/`.
4. KARL-BRIDGE-MICROSOFT; V-KENTS; MSATRANJR. *About Processes and Threads [online]* [online]. 2021-07 [visited on 2022-02-27]. Available from: `https://docs.microsoft.com/cs-cz/windows/win32/procthread/about-processes-and-threads`.
5. THE KERNEL DEVELOPMENT COMMUNITY. *Memory Management Concepts overview* [online]. [N.d.] [visited on 2022-02-27]. Available from: `https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html`.
6. INTEL. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1* [online]. 2016 [visited on 2022-02-26]. Available from: `https://www.intel.com/design/processor/manuals/253668.pdf`.

7. ORACLE. *POSIX Interprocess Communication* [online]. Copyright © 2001, 2012, Oracle and/or its affiliates. [Visited on 2022-02-27]. Available from: `https://docs.oracle.com/cd/E26502_01/html/E35299/svipc-posixipc.html`.
8. STEVEWHIMS; V-KENTS; DCTHEGEEK; MIJACOBS; MSATRANJR. *Interprocess Communications* [online]. 01/07/2021 [visited on 2022-02-27]. Available from: `https://docs.microsoft.com/en-us/windows/win32/ipc/interprocess-communications`.
9. GARFINKEL, Tal. Traps and pitfalls: Practical problems in system call interposition based security tools. In: *In Proc. Network and Distributed Systems Security Symposium*. 2003. Available also from: `https://cs155.stanford.edu/papers/traps.pdf`.
10. KARL-BRIDGE-MICROSOFT; HENKE37; S1CKB0Y1337; DREWBATGIT; V-KENTS; DCTHEGEEK; TURINGCOMPL33T; MSATRANJR. *Process Security and Access Rights* [online]. 01/07/2022 [visited on 2022-02-27]. Available from: `https://docs.microsoft.com/en-us/windows/win32/procthread/process-security-and-access-rights`.
11. MICROSOFT. *SetProcessMitigationPolicy function* [online]. 10/13/2021 [visited on 2022-02-27]. Available from: `https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-setprocessmitigationpolicy`.
12. THE KERNEL DEVELOPMENT COMMUNITY. *Seccomp BPF* [online] [visited on 2022-02-27]. Available from: `https://docs.kernel.org/userspace-api/seccomp_filter.html`.
13. GOOGLE. *Sandbox* [online] [visited on 2022-02-27]. Available from: `https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/design/sandbox.md`.
14. GOONASEKERA, Nuwan; CAELLI, William; FIDGE, Colin. LibVM: An architecture for shared library sandboxing. *Software: Practice and Experience*. 2014, vol. 45. Available from DOI: `10.1002/spe.2294`.
15. GOONASEKERA, Nuwan; CAELLI, William; FIDGE, Colin. A Hardware Virtualization Based Component Sandboxing Architecture. *Journal of Software*. 2012, vol. 7. Available from DOI: `10.4304/jsw.7.9.2107-2118`.
16. QIANG, Weizhong; CAO, Yong; DAI, Weiqi; ZOU, Deqing; JIN, Hai; LIU, Benxi. Libsec: A hardware virtualization-based isolation for shared library. In: *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2017, pp. 34–41.

17. WAHBE, Robert; LUCCO, Steven; ANDERSON, Thomas E.; GRAHAM, Susan L. Efficient Software-Based Fault Isolation. In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. Asheville, North Carolina, USA: Association for Computing Machinery, 1993, pp. 203–216. SOSP '93. ISBN 0897916328. Available from DOI: 10.1145/168619.168635.

18. TAN, Gang. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends® in Privacy and Security*. 2017, vol. 1, no. 3, pp. 137–198. Available from DOI: 10.1561/3300000013.

19. YEE, Bennet; SEHR, David; DARDYK, Gregory; CHEN, J. Bradley; MUTH, Robert; ORMANDY, Tavis; OKASAKA, Shiki; NARULA, Neha; FULLAGAR, Nicholas. Native Client: A Sandbox for Portable, Untrusted X86 Native Code. *Commun. ACM*. 2010, vol. 53, no. 1, pp. 91–99. ISSN 0001-0782. Available from DOI: 10.1145/1629175.1629203.

20. GOOGLE. *WebAssembly Migration Guide* [online] [visited on 2022-04-17]. Available from: https://developer.chrome.com/docs/native-client/migration/.

21. MULLAN, Sean. *JEP 411: Deprecate the Security Manager for Removal* [online]. Updated 2022/04/25 13:08 [visited on 2022-04-27]. Available from: https://openjdk.java.net/jeps/411.

22. HICKS, Michael. *What is memory safety?* [Online]. July 21, 2014 · 7:09 am [visited on 2022-03-10]. Available from: http://www.pl-enthusiast.net/2014/07/21/memory-safety/.

23. AMORIM, Arthur Azevedo de; HRITCU, Catalin; PIERCE, Benjamin C. The meaning of memory safety. 2017. Available from arXiv: 1705.07354 [cs.PL].

24. SALKA, C. Programming languages and systems security. *IEEE Security Privacy*. 2005, vol. 3, no. 3, pp. 80–83. Available from DOI: 10.1109/MSP.2005.77.

25. COKER, Zack; MAASS, Michael; DING, Tianyuan; LE GOUES, Claire; SUNSHINE, Joshua. Evaluating the Flexibility of the Java Sandbox. In: *Proceedings of the 31st Annual Computer Security Applications Conference*. Los Angeles, CA, USA: Association for Computing Machinery, 2015, pp. 1–10. ACSAC 2015. ISBN 9781450336826. Available from DOI: 10.1145/2818000.2818003.

26. TALY, Ankur; ERLINGSSON, Úlfar; MITCHELL, John C.; MILLER, Mark S.; NAGRA, Jasvir. Automated Analysis of Security-Critical JavaScript APIs. In: *2011 IEEE Symposium on Security and Privacy*. 2011, pp. 363–378. Available from DOI: 10.1109/SP.2011.39.

27. MITRE. *CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition* [online]. Page Last Updated: July 20, 2021 [visited on 2022-03-23]. Available from: https://cwe.mitre.org/data/definitions/367.html.
28. MAFFEIS, Sergio; TALY, Ankur. Language-based isolation of untrusted JavaScript. In: *2009 22nd IEEE Computer Security Foundations Symposium*. Port Jefferson, NY, USA: IEEE, 2009. Available from DOI: 10.1109/CSF.2009.11.
29. *WebAssembly* [online] [visited on 2022-03-14]. Available from: https://webassembly.org/.
30. *WebAssembly* [online]. Last modified: Apr 16, 2022 [visited on 2022-04-27]. Available from: https://developer.mozilla.org/en-US/docs/WebAssembly.
31. LEHMANN, Daniel; KINDER, Johannes; PRADEL, Michael. Everything Old is New Again: Binary Security of WebAssembly. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 217–234. ISBN 978-1-939133-17-5. Available also from: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.
32. NARAYAN, Shravan; DISSELKOEN, Craig; GARFINKEL, Tal; FROYD, Nathan; RAHM, Eric; LERNER, Sorin; SHACHAM, Hovav; STEFAN, Deian. Retrofitting fine grain isolation in the Firefox renderer (extended version). 2020. Available from arXiv: 2003.00572 [cs.CR].
33. GEWARREN; DOTNET-BOT. *Code Access Security* [online]. 03/30/2017 [visited on 2022-03-20]. Available from: https://docs.microsoft.com/en-us/previous-versions/dotnet/framework/code-access-security/code-access-security.
34. IMMO. *Porting to .NET Core* [online]. February 10th, 2016 [visited on 2022-03-20]. Available from: https://devblogs.microsoft.com/dotnet/porting-to-net-core/.
35. PRESSLER, Ron. *Security and Sandboxing Post SecurityManager* [online]. on April 23, 2021 [visited on 2022-03-20]. Available from: https://inside.java/2021/04/23/security-and-sandboxing-post-securitymanager/.
36. FIGUEIREDO, Luiz Henrique de; IERUSALIMSCHY, Roberto; FILHO, Waldemar Celes. The design and implementation of a language for extending applications. 2015. Available from https://www.lua.org/semish94.html (reprint from Proceedings of XXI Brazilian Seminar on Software and Hardware (1994) 273–283.)
37. ROBLOX. *Sandboxing* [online]. © 2022 Roblox [visited on 2022-04-24]. Available from: https://luau-lang.org/sandbox.

38. IERUSALIMSCHY, Roberto; FIGUEIREDO, Luiz Henrique de; CELES, Waldemar. *Lua 5.4 Reference Manual* [online]. 2022 [visited on 2022-04-19]. Available in html format along with the source code from `https://www.lua.org/ftp/lua-5.4.4.tar.gz` in the location /doc/manual.html.
39. IERUSALIMSCHY, Roberto; FIGUEIREDO, Luiz Henrique de; CELES, Waldemar. *Lua 5.1 Reference Manual* [online]. 2012 [visited on 2022-04-19]. Available in html format along with the source code from `https://www.lua.org/ftp/lua-5.1.5.tar.gz` in the location /doc/manual.html.
40. IERUSALIMSCHY, Roberto; FIGUEIREDO, Luiz Henrique de; CELES, Waldemar. *Lua 5.2 Reference Manual* [online]. 2015 [visited on 2022-04-19]. Available in html format along with the source code from `https://www.lua.org/ftp/lua-5.2.4.tar.gz` in the location /doc/manual.html.
41. IERUSALIMSCHY, Roberto; FIGUEIREDO, Luiz Henrique de; CELES, Waldemar. *Lua 5.3 Reference Manual* [online]. 2020 [visited on 2022-04-19]. Available in html format along with the source code from `https://www.lua.org/ftp/lua-5.3.6.tar.gz` in the location /doc/manual.html.
42. PUC-RIO. *Bugs* [online]. Last update: Tue Apr 26 09:54:42 UTC 2022 [visited on 2022-04-27]. Available from: `https://www.lua.org/bugs.html`.
43. MITRE. *CVE-2021-44964.* [Available from MITRE, CVE-ID CVE-2021-44964.]. Copyright © 1999–2022 [visited on 2022-04-24]. Available from: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44964`.
44. MITRE. *CVE-2022-0543.* [Available from MITRE, CVE-ID CVE-2022-0543.]. Copyright © 1999–2022 [visited on 2022-04-25]. Available from: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0543`.
45. CAWLEY, Peter (corsix). *Exploiting Lua 5.1 on 32-bit Windows* [online]. © 2022 GitHub, Inc. [Visited on 2022-04-27]. Available from: `https://gist.github.com/corsix/6575486`.
46. MITRE. *CVE-2015-4335.* [Available from MITRE, CVE-ID CVE-2022-0543.]. Copyright © 1999–2022 [visited on 2022-04-27]. Available from: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0543`.
47. CAWLEY, Peter (corsix). *Exploiting Lua 5.2 on x64* [online]. © 2022 GitHub, Inc. [Visited on 2022-04-27]. Available from: `https://gist.github.com/corsix/49d770c7085e4b75f32939c6c076aad6`.

48. CAWLEY, Peter (corsix). *Malicious LuaJIT bytecode* [online]. on November 11, 2015 [visited on 2022-04-27]. Available from: `https://www.corsix.org/content/malicious-luajit-bytecode`.
49. ROBLOX. *Why Luau?* [Online]. © 2022 Roblox [visited on 2022-04-25]. Available from: `https://luau-lang.org/why`.
50. WUBE SOFTWARE. *Data lifecycle* [online]. Copyright © Wube Software [visited on 2022-04-27]. Available from: `https://lua-api.factorio.com/latest/Data-Lifecycle.html`.
51. WUBE SOFTWARE. *Libraries and functions* [online]. Copyright © Wube Software [visited on 2022-04-27]. Available from: `https://lua-api.factorio.com/latest/Libraries.html`.
52. GBIP. *GitHub*. Research Project : video games modding vulnerabilities [online]. de4a551 on 20 Apr 2020 [visited on 2022-04-27]. Available from: `https://github.com/gbip/lua_attack`.
53. ROBLOX. *Compatibility* [online]. © 2022 Roblox [visited on 2022-04-25]. Available from: `https://luau-lang.org/compatibility`.

Appendix A

Acronyms

API Application programming interface

OS Operating system

Appendix B

Contents of enclosed CD

- `readme.md` .... the file with CD contents description
- `Implementation` ...... the directory with all practical parts of this thesis
  - `OpenMWEscape` the directory with a proof of concept mod for OpenMW
  - `Analyser` .... the directory containing the Lua source code of the environment analyser
- `Text` .... the thesis text and LaTeX source codes directory
  - `BP_Adamek_Petr_2022.pdf` .... the thesis text in PDF format