MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Securing the mod system of BeamNG.drive

Master’s Thesis

BC. ADAM IVORA

Brno, Fall 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Securing the mod system of BeamNG.drive

Master's Thesis

BC. ADAM IVORA

Advisor: doc. RNDr. Petr Švenda, Ph.D.

Department of Computer Systems and Communications

Brno, Fall 2023

FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Adam Ivora

Advisor: doc. RNDr. Petr Švenda, Ph.D.

Acknowledgements

I'm in. (Hacker – MC Ride)

My most immense appreciation goes to doc. RNDr. Petr Švenda, Ph.D. who brought many fruitful remarks and ideas about the thesis to the table. Due to his support and encouragement, I managed to write a big chunk of the thesis more than a few days before the deadline, which is impressive. *Thank you so much, Petr*.

I could not have gotten this far without my family, and I am grateful to them. Appreciations to my mother, sister, brother-in-law, and my little niece, who support all my endeavours. They are the still point of the ever-turning world for me. *I will love you always*.

Big thanks to everyone who had enough courage to befriend me, and I am sorry I have been unavailable lately; will make it up to you real soon. I will not even try to list you this time for data protection and personal laziness reasons. *Special credit belongs to the group chat, my primary source of serotonin*.

Last but not least, thanks to all the people representing the Faculty of Informatics and to the institution as a whole for the impact it had on my life during the past six years and the opportunities it provided. With how formative these years have been for me, I should call it the Faculty of Formatics instead 👉🐸👉.

Don't cry because it happened. Smile because it's over.
(On Writing Final Theses – Author unknown)

Abstract

We delve into how videogames are modified by the community, a process known as “modding”. Then, we scrutinise the security problem of safeguarding the modding system of a videogame, which is akin to protecting against the effects of executing untrusted code.

Our focus is the vehicle simulator BeamNG.drive, which includes modding support in the Lua language. Thus, we analyse the Lua ecosystem from the perspective of both attackers and defenders, exploring the possibilities of running malicious code using Lua and the software mitigations that guard against these exploits.

The state of videogame security is analysed by creating the Game Malware Survey – a survey of 51 popular videogames with modding support and the history of relevant related malware incidents.

Furthermore, we probe into various approaches to mitigate the Lua exploits, concerned about the performance and efficacy of the mitigations. For that purpose, we developed the EMO (Exploit-Mitigation-Overhead) Test Bench application, enabling automated analysis of mitigation effectiveness on Lua implementations and performance overhead on BeamNG.drive.

We propose a short-term and a long-term strategy for securing the mod system based on our findings. We also released the Lua Exploit-Mitigation part of the EMO Test Bench to the open-source community.

Keywords

sandboxing, sandbox, Lua, LuaJIT, videogames, malware, process isolation, modding

Contents

1 Introduction **1**

2 Glossary **3**

3 Game Malware Survey **5**
3.1 Attacker Motivation 5
3.2 Case Studies 6
3.2.1 BeamNG.drive 6
3.2.2 Roblox 8
3.2.3 Minetest 9
3.2.4 Factorio 9
3.2.5 Other Lua Games 10
3.2.6 Minecraft 10
3.2.7 Cities: Skylines 11
3.3 Summary Table 11
3.3.1 Methodology 12
3.4 Statistics and Takeaways 16

4 Security in the Lua Ecosystem **17**
4.1 Lua 17
4.2 LuaJIT 18
4.3 Luau 18
4.4 Exploit Methods 18
4.5 Using Bytecode 19
4.6 Using FFI 20
4.7 Using the Standard Library 20
4.8 Exploits: Summary 21

5 Prevention and Mitigations **22**
5.1 Lua-specific Solutions 22
5.1.1 Disabling Loading Bytecode 23
5.1.2 Disabling FFI 23
5.1.3 Language Level Sandboxes 26
5.1.4 Source Code Level Sandboxes 26
5.2 Sandboxing Software 27
5.2.1 Windows Sandbox 27

5.2.2 Sandboxie . . . 28
5.3 Enforcing Limits Using the Operating System . . . 28
5.3.1 AppContainer Isolation . . . 28
5.3.2 Process Mitigation Policies . . . 29
5.4 Summary . . . 30

6 The EMO Test Bench 31
6.1 Requirements . . . 31
6.2 Lua Implementations . . . 32
6.3 Exploits . . . 33
6.4 Mitigations . . . 36
6.5 Exploit-Mitigation Benchmark . . . 37
6.6 Overhead . . . 38
6.7 Mitigation-Overhead Benchmark . . . 40
6.7.1 AppContainer Obstacles . . . 42

7 Benchmark Evaluation 43
7.1 Exploit-Mitigation: Mitigation Efficacy . . . 43
7.2 Mitigation-Overhead: Performance under Mitigations . 44
7.3 Mitigation Assessment . . . 47
7.4 Proposed Solution . . . 48

8 Conclusion 50

A Attachments 52
A.1 The EMO Test Bench . . . 52
A.2 Additional Data . . . 52
A.3 Video Showcase . . . 53

B Benchmark Parameters 54

Bibliography 56

List of Tables

4.1 Comparison of the different exploit methods. 21

5.1 Comparison of the different sandboxing technologies in the scope of sandboxing mods of BeamNG.drive. 30

7.1 The list of implementations/mitigations and whether they succeeded to mitigate the exploits. 45

7.2 The mean framerates of the median runs, their corresponding standard deviations, and overhead compared to no mitigation. 46

1 Introduction

Security vulnerabilities in videogames are not an uncommon thing: for example, remote code execution in Dark Souls [1] and also a recent example of a partial remote code execution bug being actively exploited in the computer game Grand Theft Auto V [2, 3]. Videogames often include an official way of extending the game's functionality, such as adding new maps, vehicles, or additional custom behaviour. These extensions are called "mods", meaning modifications, and the persons who make them are called modders. Even though some games do not provide official support for modding, the modders often devise a way of producing mods, as in Minecraft [4]. The work of modders, in general, adds content of high value that could not be obtained without modding communities [5].

BeamNG.drive is a game with official modding support. It is a realistic driving simulator with a real-time soft-body physics engine. It has been in development since 2012. The user base is extensive, with around 16 thousand daily players on the Steam platform [6]. Also, the modding community is quite alive, with the most popular mods being downloaded more than two million times [7]. Therefore, a popular mod including malware could seriously impact the user base and the company.

The work aims to protect BeamNG.drive players from malicious code that can be included in the mod files. The official mod repository is curated and checked for malicious code. However, detecting all possible malicious code is a non-trivial task. Also, unofficial websites with mod files exist, and the developers cannot curate content on these platforms.

The structure of the thesis is the following: Chapter 2 describes a few key terms used throughout the work in a high-level sense. Readers knowledgeable with the meaning of terms like *modding*, *Lua*, *BeamNG.drive*, or *sandboxing*, feel free to skip this chapter. In Chapter 3, we survey popular games with modding support, focusing on past malware incidents. We also thoroughly analyse selected games as case studies, discussing the "state of the art" of videogame security related to modding. Chapter 4 reviews the security of the Lua ecosystem, focusing on different Lua interpreters. Concrete examples of ways to

achieve malicious behaviour of Lua code are provided. We use the language's features and exploits, which get around the limitations of a secured Lua environment (sandbox escapes). Chapter 5 provides a list of the mitigations we can implement to protect us against these codes, including Lua-specific methods and protection on the process and operating-system level. The Exploit-Mitigation-Overhead (EMO) Test Bench, implemented in Python as part of the thesis, is described in Chapter 6, including its design needs, choices, architecture and setup. Chapter 7 is concerned with interpreting the EMO Test Bench run results and suggests a set of mitigations that fit BeamNG.drive requirements. Chapter 8 summarises our findings from the Game Malware Survey and the EMO Test Bench.

2 Glossary

The chapter is a short introduction to the key terms used throughout the work intended for those without a videogame or cybersecurity background.

Modding

Modding is a commonly used videogame-slang word, a shorthand for "modifying". Modding, in the more general sense, is the modification of hardware or software to extend its behaviour for the needs of the modder or other individuals. In the context of the thesis, we always use the word modding in the meaning of "computer game modding". Computer game modding is changing a game using programming and other software tools to add features not included in the game or to fix issues with the game. Modding is not done by the developers but by the players with the incentive and needed technical skills [8].

Some of the videogame mods change the experience of the game immensely, and there are examples of mods which became more popular than the original game (Counter-Strike as a mod of Half-Life, DoTA as a mod of Warcraft 3).

Lua

Lua is a self-proclaimed [9], powerful, efficient, lightweight, embeddable scripting language. It was implemented and is maintained by a team from the Pontifical Catholic University of Rio de Janeiro in Brazil.

Lua is a dynamically typed language. The original implementation (called Lua, or also PUC Lua) is the most popular, although there are a lot of reimplementations and languages based on Lua [10]. We discuss the differences between PUC Lua, LuaJIT and Luau from the security point of view in Chapter 4.

BeamNG.drive

BeamNG.drive is a dynamic soft-body physics vehicle simulator available on Steam[1] since 2015. The game includes a high level of customisation and modding capabilities [11]. BeamNG.drive uses Lua for most of the game's logic, including the modding system. The Lua implementation used is a customised version of LuaJIT [12].

BeamNG.drive is a primarily Windows application[2]. Therefore, we limit our analysis to the Windows operating system. Analysis of sandboxing methods for Linux is left as a potential future work.

Sandbox

In the context of this work, sandboxing is a security mechanism for separating running programs and limiting their behaviour.

The term "sandboxing" was introduced in 1993 as a technique to provide isolation against faults in extension code [13]. For our means, we can follow the definition of sandboxing by Goldberg et al.: "the concept of confining a helper application to a restricted environment, within which it has free reign" [14].

All modern web browsers use a form of sandboxing, such as the sandbox in Chromium [15]. The Android platform also uses a kernel-level Application Sandbox enforcing security between apps and the system [16]. Apple has a similar way of sandboxing applications for macOS and iOS [17]. Software solutions which provide a sandbox environment for arbitrary applications also exist (more in Section 5.2).

1. A videogame digital distribution service.
2. Linux support is in an experimental phase.

3 Game Malware Survey

The chapter analyses the state of the art of some of the most popular games in the modding community, focusing on games using Lua as the language of mods, as BeamNG is using Lua for the mod system. The chapter also includes a table comparing 51 popular games, their malicious mod mitigation techniques and security incidents in the past. We analyse the state of selected case studies from the ecosystem regarding malware incidents and protection in Section 3.2. We start by discussing the motivations of malware attacks using the modding systems of various computer games.

3.1 Attacker Motivation

Kaspersky, a cybersecurity company, published multiple reports about gaming-related cyber threats [18, 19, 20]. By reading the reports, we can extract the motivation of the attackers. Kaspersky report from 2022 [19] contains the top ten threats distributed worldwide under the guise of popular games. The most prevalent malware families are downloaders, adware and trojans. The leading malware family in the same report is `Trojan-PSW.MSIL.Reline/RedLine`, a credential and cryptocurrency-stealing software. Be aware that the report does not describe malware found in mods but all malware "branded" as belonging to one of the popular videogames, including phishing sites and fake cracks[1]. Therefore, not surprisingly, the primary motivation of the attackers is monetary. The attackers gain money either by stealing the cryptocurrency of the infected players or by mining it on their machines. Also, extorting the players with ransomware is a common technique for making money off malware.

However, the gaming industry also attracts another type of attacker who is not writing malware for financial gain. There have been cases where a reputed creator of mods for Cities: Skylines, a city-building videogame started packing malicious code with his mods, breaking mods by other creators on purpose, as discussed in Section 3.2.7. Also,

1. Illegally modified executables bypassing the anti-copying protection.

some exploits only try to be as loud as possible to let the developers know an exploit is available and urge them to fix it (Section 3.2.5).

The execution of untrusted code in computer games that allows modding is a wide attack vector. Some of these modding systems include auto-updating, which would allow an easy spread of a malicious update to many players in case a famous modder's account got compromised. Gaming computer hardware is better suited for intensive CPU/GPU tasks, and therefore, gaming computers are also a beneficial addition to botnet mining cryptocurrencies.

The popular malware payloads used are cryptocurrency miners, keyloggers, remote access trojans and ransomware. The definitions and explanations of the different types of malware are not in the scope of this thesis; more information is available in the Malwarebytes glossary [21] or other sources.

3.2 Case Studies

We describe the different situations of securing player-supplied code for selected games, focusing primarily on Lua mods and multiplayer code (Roblox, Factorio, Starbound, Garry's Mod). We also describe other popular games with mods, such as Minecraft and Cities: Skylines.

3.2.1 BeamNG.drive

BeamNG.drive is a vehicle simulation game; for a brief introduction, see Chapter 2. It uses a custom fork of LuaJIT, which is tweaked to sandbox the following features:

- file access is through a virtual filesystem, which only allows access to the game and user folder,
- loading external DLLs[2] through `package.loadlib` and similar functions is disabled,
- dangerous functions such as `os.execute` are replaced with empty implementations inside the source code,

2. Dynamic-link libraries.

- other parts of LuaJIT are sandboxed.

The in-game mod repository of BeamNG.drive is shown in Figure 3.1. Installing mods is done by simply clicking the "➕" button; no knowledge of Lua or security is needed to install the mod. Therefore, establishing a properly secure environment where the mods can run is essential.

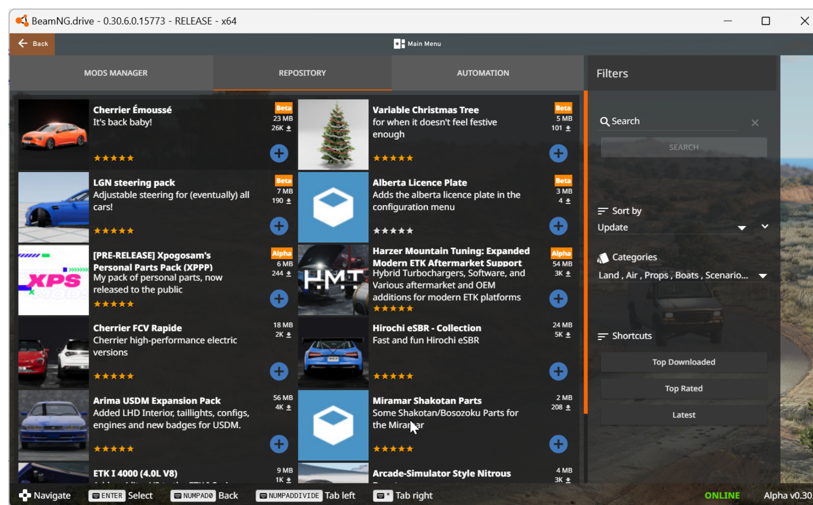

Figure 3.1: The BeamNG.drive mod repository.

BeamNG has a mod system that allows it to enhance the gaming experience with new vehicles, maps, or other features. The mods can be downloaded in-game and from the official mod repository website [7].

The mods are written in Lua, a dynamic programming language. For executing Lua code, BeamNG.drive uses LuaJIT [12], a just-in-time (JIT) compiler. Lua provides features to sandbox environments [22], but it is tricky to do it properly. Moreover, LuaJIT includes the FFI library, which allows calling external C functions and using C data structures from pure Lua code and is labelled as inherently unsafe by the developer of LuaJIT [23]. FFI is used heavily in BeamNG.drive.

3.2.2 Roblox

Roblox is an online game and game creation platform with over 66 million daily active users [24]. The players participate in "experiences" created by the community. Experience in Roblox is a collection of maps players can play through, including custom objects and scripting logic [25]. The concept of experiences is the same as that of mods, so experiences suffer from the same security problems.

The experiences are scripted using Luau [26], a gradually typed embeddable scripting language derived from Lua 5.1. The experience code runs on every user's computer, so proper sandboxing is crucial.

During the early Roblox days, Luau (Section 4.3) had not been created yet, and the language used for the experiences was Lua 5.1. The 2012 blog post "Bye Bye Bytecode" [27] is an example of a Lua 5.1 vulnerability affecting the game, as Roblox developers had to turn off the loading of bytecode as it led to arbitrary code execution.

Luau tries to solve the Lua 5.1 sandboxing problems by [28]:

1. removing parts of the standard Lua library,
    - `io` library: file manipulation,
    - `package` library: facilities for loading and building modules,
    - file and environment access functions from the `os` library,
    - most of the `debug` library, which contains functions for debugging or profiling,
    - the `dofile` and `loadfile` functions, which parse Lua code;
2. removing access to function bytecode,
3. reducing the functionality of the garbage collector API.

The developers also focus on high performance and implement various optimisations to the Lua interpreter and compiler [29]; they do not use JIT compilation in contrast with LuaJIT. The main reason for no JIT is that there is an extra risk involved in JITs, which is supported by some existing attacks [30].

3.2.3 Minetest

Minetest is an open-source voxel game engine and also a game with the same name [31]. Minetest has a modding API in Lua (uses either PUC-Lua 5.1 or LuaJIT, depending on the compilation configuration), an official modding repository called ContentDB [32] and supports multiplayer, where the mods of the server are being downloaded to the clients. Minetest tries to provide a sandboxed environment for the mods.

The mitigations that were already implemented are [33]:

- blocking direct access to the filesystem, only letting Lua access a VFS[3],
- not loading the `io` library,
- not loading the `package` library,
- not allowing access to the `debug` library,
- only allowing access to safe functions in the `os` library.

The mitigations are still a work in progress. The to-do list of the sandbox features to implement, as of November 2023, includes:

- limiting how much data can be stored in mod storage,
- disabling LuaJIT and JIT compilation,
- a permission system for mods.

Although there is effort put into sandboxing the environment, multiple vulnerabilities were found in Minetest in the past, including a remote code execution issue [34].

3.2.4 Factorio

Factorio uses Lua 5.2.1 to implement its mod system. To allow determinism and improve security, it removes the `loadfile` and `dofile` functions. Factorio also removes the `coroutine.`, `io.` and `os.` libraries

3. Virtual file system.

and provides its version of the `package.` library [35]. A prior remote code execution in Factorio was caused by the `package.cpath` table being exposed [36] and writable. The table exposure allowed the attackers to load DLLs from arbitrary locations. Factorio allows mods to write files from Lua to a mod-specific folder. By changing the `package.cpath` to point at that mod-specific folder, the attacker (in this case, the mod creator) could run arbitrary code.

3.2.5 Other Lua Games

Examples of other games supporting modding in Lua are Starbound and Garry's Mod. We did not find any traces of historical Starbound exploits, but Garry's Mod has been a target of remote code execution attacks before [37, 38]. Although the exploit was not part of a mod per se (it was malicious Lua code sent by clients to a server), the underlying principles and mitigations are the same. The incident is intriguing as the malicious code only spammed the players with the message "fix it vinh", trying to draw attention to the remote code execution issue [38].

A similar non-Lua case is that of Dark Souls 3 [1], where the malware creator interrupted a famous player's stream to crash his game. According to the article, the malware creator tried to contact the developer about the issue, but he was ignored, so he hacked streamers to draw attention to the problem.

We continue with the analysis of two other popular games that do not use Lua as their modding language.

3.2.6 Minecraft

Minecraft is a sandbox[4] infinite-generated world exploration game. Minecraft support for modding is not official; players wanting to download and use mods have to download Minecraft modding API called MinecraftForge [39].

Although it is unofficial, the modding community of Minecraft is one of the largest ones, with over 100,000 mods available on the mod repository CurseForge [40]. The mods are written in Java, and

4. Sandbox game is a game without any predetermined goal. Sandbox games are not related to sandboxing as defined in Chapter 2.

any sandboxing whatsoever is missing, ironically, despite the sandbox genre of the game. That makes including malware code in Minecraft trivial; the distribution of such malicious mod becomes the greatest obstacle for the attackers.

3.2.7 Cities: Skylines

Cities: Skylines is a single-player, open-ended city-building simulation published by Paradox Interactive. Distribution of the mods is done through the Steam Workshop, a standard method to distribute mods of games supporting the Steam platform. The mods are written in C# and use the Unity framework APIs. They are not sandboxed in any way, and there have already been cases of malicious mods being distributed on the platform [41, 42]. The game has an official modding API. However, the official wiki page contains a disclaimer that the mods are not executed in a sandbox and that caution is recommended when running mods [43].

Steam Workshop is considered to be one of the safe places to download mods. As we see with Cities: Skylines and also other games (for example, Dota 2 [44]), there have been cases in the past where Steam Workshop mods included remote code execution malware, which was distributed to the players.

3.3 Summary Table

A table of 51 games and their cybersecurity-related incident follows. We focus on games that include some support for modding, but we do not limit ourselves to security incidents related to malicious mods. Some of these allowed remote code execution using the multiplayer feature, an even more severe attack vector, as no user interaction is usually needed other than playing the game.

It serves as an overview of how others in the industry “do it”. The table does not try to be exhaustive and include all the existing mod repositories or issues. Still, it should provide a good overview of what security problems arise in computer games supporting mods.

3.3.1 Methodology

We combined several popular services distributing videogame mods: Steam Workshop, NexusMods, and CurseForge.

For the Steam Workshop games, we took the top 20 most-played Steam Workshop games from the SteamDB website [45]. Note that not all games that have support for Steam Workshop support a fully-fledged modding experience – for example, Dota 2 has support only for new 3D models of existing content and no support for scripts changing the game's behaviour. For NexusMods, we took the top 20 games sorted by the number of downloads [46]. For CurseForge, we took the top ten games sorted by the number of different mods, as the information about downloads is unavailable [40]. We removed duplicates and the game WildStar, which was shut down in 2018.

We added four other games with their custom self-hosted mod repository: BeamNG.drive (subject of the work), Factorio, Roblox, and Minetest. This process results in a table of 51 games.

For all the games in the list, we searched the Internet for phrases such as "<GAME> exploit", "<GAME> malware", and similar terms. These results were then manually verified to search for exploits related either to the mod system or multiplayer. Links to relevant articles and sources are provided in the table. The table is also included as an attachment of the thesis in the CSV format (Appendix A.2).

Columns Description

- **Game:** Name of the game. If the name is **bold**, the game uses Lua for modding support.
- **Mods:** The number of mods found during the data-gathering phase. This number is a lower bound for the available mods. With games in multiple studied mod repositories, we use the repository with the highest number of mods.
- **Supp:** ✓ if the game officially supports the modding and the modding APIs are exposed, or official modding tools are available, ✓ [5] with a footnote if official modding tools are not pro-

5. Like this.

vided, but modding is endorsed by the game creators, ✗ if modding is purely unofficial.

- **MP:** Whether the game has online multiplayer capability. Multiplayer can be used as the means of delivering malware, sometimes with the use of mod synchronisation among players.

- **Mitigations / Tools / Notes:** The mitigations that the game uses for protecting against malware or other notes about the game's security. If empty, it does not mean that mitigations are not in place but that we found no information about them. It also mentions official tools for modding for some of the games.

- **Exploits:** Links to the game's publicly disclosed vulnerabilities/-exploits related to mods or multiplayer code. ✗ means that no reports of vulnerabilities/exploits were found, which does not mean these vulnerabilities/exploits do not exist. ✓ means that although we found no reports of malware in these games, we expect that writing exploits for the game is possible, as we suspect no sandboxing features are implemented.

	Game	Mods	Supp	MP	Mitigations / Tools / Notes	Exploits
1	**Roblox**	>40,000,000	✓	✓	Luau (Section 4.3)	[47][6]
2	**Garry's Mod**	1,722,352	✓	✓		[38]
3	Counter-Strike: Global Offensive	384,454	✗	✓		[48]
4	Cities: Skylines	318,248	✓	✗	Mods run arbitrary C# code.	[42]
5	Minecraft	127,127	✗	✓	Mods run arbitrary Java code.	[49]
6	Left 4 Dead 2	120,631	✗	✓		[50]
7	Rust	103,440	✓[7]	✓		✗
8	Unturned	93,887	✓	✓		✗
9	Skyrim	69,500	✓	✗		✗
10	Skyrim Special Edition	60,000	✓	✗		✗
11	Fallout 4	48,100	✓	✗		✗
12	Hearts of Iron IV	42,176	✓	✗		[52]
13	Dota 2	41,357	✓	✓		[44]
14	DayZ	38,074	✓	✓		✗
15	Team Fortress 2	34,590	✗	✓	Cosmetic item mods only.	[50]
16	Oblivion	31,000	✓	✗		✗
17	RimWorld	30,039	✓	✗	Mods run arbitrary C# code.	✓
18	Fallout New Vegas	29,000	✓	✗	Garden of Eden Creation Kit tool.	✗
19	Stellaris	26,400	✓	✓		✗
20	**BeamNG.drive**	21,495	✓	✗	Virtual filesystem, Lua patches.	✗
21	Euro Truck Simulator 2	18,017	✓	✓		✗
22	**Project Zomboid**	16,722	✓	✓	No mentions of sandboxing found.	✗
23	Fallout 3	16,000	✓	✗		✗
24	ARK: Survival Evolved	15,842	✓	✗		✗
25	Stardew Valley	11,300	✗	✓	Mods run arbitrary C# code.	✓
26	**World of Warcraft**	10,750	✓	✓		✗

6. Vulnerability was not inside the mod system, but inside the creators' toolbox.
7. Modding guidelines are published [51].

27	Morrowind	10,200	✓	✗		✗
28	The Sims 4	9,225	✓[8]	✗		✗
29	Sid Meier's Civilization VI	7,408	✓	✓		✗
30	Blade & Sorcery	5,900	✓	✓	BasSDK, mods run custom C# code.	✓
31	The Witcher 3	5,900	✓	✗	The Modkit tool.	✗
32	Cyberpunk 2077	5,800	✓	✗	The REDmod tool.	[54]
33	**Factorio**	5,705	✓	✓	Bytecode execution patches [55].	[36]
34	Monster Hunter: World	5,100	✗	✗		✗
35	Terraria	5,033	✓	✓	The tModLoader tool, custom C# code.	[56]
36	Mount & Blade II	4,280	✓	✓	Modding Kit tool, arbitrary C# code.	✓
37	Dragon Age: Origins	3,400	✓	✗	Dragon Age Toolset.	✗
38	Dragon Age: Inquisition	2,900	✗	✓		✗
39	StarCraft II	2,288	✓	✓	StarCraft II Editor – only a map editor.	✗
40	Blade & Sorcery: Nomad	2,100	✓	✗	BasSDK, mods run custom C# code.	✓
41	Kerbal Space Program	2,075	✓	✗[9]	Mods run arbitrary C# code.	[57]
42	**Minetest**	1,929	✓	✓	Virtual filesystem, limits libraries [33].	[34]
43	Monster Hunter Rise	1,900	✗	✓		✗
44	Valheim	1,800	✗	✓	Mod support at "own risk" [58].	[59]
45	Dragon Age 2	1,400	✗	✗		✗
46	American Truck Simulator	1,249	✓	✓		✗
47	Warframe	798	✗	✓	Cosmetic item mods only.	✗
48	Subnautica	666	✗	✗	Mods run arbitrary C# code.	✓
49	Rocket League	524	✗	✓	Mods not supported, but allowed [60].	✗
50	World of Tanks	431	✓	✓		[61]
51	Minecraft Bedrock	319	✓	✓	"Add-ons" with limited privileges.	✗

8. Supports a framework that makes installing mods easier [53].
9. An unofficial multiplayer mod exists.

3.4 Statistics and Takeaways

Seven of the 51 (13.7%) games included in the survey use Lua for modding support. Four out of the seven mentioned have had security issues in the past. If we count Steam Workshop and games with their mod repositories, we have 24 games using "official modding repositories". Nevertheless, 38 games (74.5%) have official modding support or endorsement from the developers, which means that third-party mod repositories such as NexusMods or CurseForge are also often accepted as a go-to mod distribution platform by developers.

We found historical malware incidents in 16 out of the 51 games (31.4%) in the Game Malware Survey. These incidents are related either to the mod system or to the multiplayer. Both of these cases are interesting for our study, as the prevention against them is similar. In addition to these 16 games, there are at least six more (11.8%) games with "suspected exploitability" – that is, no information about sandboxing is provided, and these games usually allow running of arbitrary C# code. Out of the $16 + 6 = 22$ games, 15 have official modding support/endorsement; thus, the incidents are not limited only to unofficial modding.

We can summarise our findings as follows. There are popular games with mod support that are not concerned about mod security. On the other hand, there are also a lot of security-positive examples where the developers are trying to sandbox the system somehow. However, proper sandboxing is not trivial, and incidents or sandbox escapes still happen in games that are trying to create a secure modding environment (Minetest and Garry's Mod are two examples of that phenomenon). Roblox took a different path by creating a new custom Lua interpreter called Luau focused on security which proved meaningful for the use case of Roblox. However, the performance difference between Luau and LuaJIT can be an issue if maximal performance is sought.

4 Security in the Lua Ecosystem

We discuss the security protections against running untrusted code of standard Lua implementations used in the videogame industry: the pristine implementation (Lua, or PUC-Lua), LuaJIT, and Luau.

Petr Adámek's bachelor thesis [62] overviews different ways to construct a Lua sandbox. Paul Florence and Lucien Menassol explore (in French) the attack surface of the modding APIs in videogames, with a heavy focus on Lua in their study [63]. Scientific literature about the chapter's topic is scarce, as is the case in general with cybersecurity in the gaming industry. Continuing with the chapter, we discuss the concrete methods of exploiting Lua in Section 4.4: using bytecode, the FFI library, and the Lua standard library.

Lua is often used for modding computer games because it is embeddable but still performant, small, and portable. The simplicity and low overhead of Lua is a factor in this; it is also the reason why many different Lua implementations exist [10].

4.1 Lua

Lua, or PUC-Lua, is the original and the most popular implementation of Lua, created in 1993 and maintained by a team at the PUC-Rio University (more info in Chapter 2).

The lua-users wiki, a website created by the community of the Lua programming language, contains a section about sandboxing [22]. Still, it is outdated (the guide does not work for Lua 5.2 and above) and incomplete. The current version is Lua 5.4.

PUC-Lua is not prone to exploits leveraging the FFI library (Section 4.6) because the library is only available for LuaJIT. However, multiple cases exist of escaping sandboxes using crafted bytecode; see Section 4.5. Without proper sandboxing, functions from the standard library can be used to execute malicious behaviour (Section 4.7).

4.2 LuaJIT

LuaJIT is a Just-In-Time Compiler for Lua and comes with a high-performance interpreter. It has been in development since 2005, primarily by Mike Pall and is used as a scripting middleware in games and other performance-critical applications.

LuaJIT is fully compatible with Lua 5.1, but it also contains selected features from Lua 5.2, Lua 5.3 and several extension modules, such as support for bitwise operations or the FFI[1] library, which allows calling external C functions from Lua code [64, 65]. New development seemed halted in the past [66], but as of November 2023, LuaJIT is actively developed and maintained with a rolling release system [67]. Development on LuaJIT is also done by OpenResty, a widespread nginx distribution, which has forked the LuaJIT repository and is providing extra API functions for the language [68].

4.3 Luau

Luau is an implementation based on Lua 5.1 that provides performance and security [26]. As mentioned in Section 3.2.2, Luau was created by the developers of the Roblox game. However, Luau is also being used in other videogames, Alan Wake 2 being a notable and recent example [69].

External Luau scripts are run by millions of players daily in Roblox. Thus, Luau got field-tested, and there do not seem to be signs of exploitable security-related bugs as of November 2023.

4.4 Exploit Methods

We call an exploit some chunk of code that executes unexpected, often malicious behaviour. If the developers decide to allow modders to execute arbitrary code, then they also have to prevent these exploits. That can be done by limiting access to unsecure modules/functions and also by sandboxing the Lua implementation. If the Lua implementation is not sandboxed, the attacker could call operating-system commands

1. Foreign Function Interface.

(`os.execute`) or manipulate the filesystem using the Lua-included functions.

4.5 Using Bytecode

Lua bytecode is the intermediate language to which the Lua implementations compile the Lua code. Bytecode is a list of internal instructions, the Lua virtual machine processes these instructions. Every Lua implementation uses a different bytecode format, and the formats are not compatible with each other. Pristine Lua used to include a bytecode verifier, which was supposed to protect against bytecode exploitation. It was removed in Lua 5.2 because the team could not deliver a verifier that would be effective enough [70].

Remarkably, loading bytecode is allowed by default, and to deactivate it, the developers have to use the `loadfile` function for loading Lua with an extra parameter. Moreover, the `dofile` function, also used for executing Lua chunks, does not support the option of disabling bytecode [71]. The FAQ section of the LuaJIT website states that loading untrusted bytecode is unsafe and that the recommended way to sandbox Lua is to do it on the process level [23].

Peter Cawley published bytecode exploits for LuaJIT [30], Lua 5.1 [72], and also Lua 5.2 [73, 74]. Another exploit for Lua 5.1 has been published by a different author [75]. There also exists a sandbox escape using bytecode for more recent Lua 5.4 [76], which suggests that mitigation of external bytecode is not a priority for the developer team and that loading unverified bytecode should not be allowed.

Another LuaJIT sandbox escape using bytecode was published in late 2022 [77], so the attack vector is valid even for recent versions of LuaJIT. The code for these exploits is too long to be included as a part of the thesis; consult the references of this section for the proofs-of-concepts.

The difficulty of using this exploit method is quite high, as the payloads are version and operating-system-specific, and it is necessary to find an existing bug in the bytecode interpreter to exploit it.

4.6 Using FFI

The Foreign Function Interface library allows calling external C functions from Lua code. It is integrated into LuaJIT and unavailable for other Lua implementations [65]. The FFI library is said to be inherently unsafe by the LuaJIT author [23].

Peter Cawley discusses using FFI to escape Lua-level sandboxes in his article about malicious LuaJIT bytecode. According to him, if the sandbox exposes the FFI library, then arbitrary code execution is trivial [30]. We can see such an example of a script that executes arbitrary Windows shellcode[2] in Figure 6.3. Due to the nature of the FFI library, this is only one of the different ways to run arbitrary code. We can use other functions from the Win32 API, such as `LoadLibraryA`, to execute arbitrary code from the `DllMain` method or `ShellExecuteA` to execute an arbitrary file.

The difficulty of making LuaJIT exploits using FFI is lower than that of using bytecode, as the exploits are using the standard functions of the FFI library. Still, for a successful exploit, the attackers need to know how to operate the operating-system-specific APIs to execute malicious code.

4.7 Using the Standard Library

There are many possibilities with the standard library of Lua/LuaJIT to run arbitrary malicious code, as is the case with most modern programming languages. In Figure 4.1, we can see three such ways:

1. executing an operating-system-specific shell which runs a command,
2. loading a dynamic library that runs its `DllMain` method on load,
3. writing an executable file to a location that will autorun the executable on the next start.

Using the standard library of Lua is the most accessible type of exploit for an attacker, but it is also the method that is being protected

2. A piece of machine code used to spawn a (malicious) process.

```lua
-- execute a system command
os.execute('start cmd /k call "C:/Malware/malware.exe"')

-- load an arbitrary DLL (DllMain is executed)
package.loadlib('C:/Malware/malware.dll', '*')

-- copy a file to a location that will be run on startup
file = io.open('C:/Malware/malware.exe', 'r')
content = file:read('*a')
file = io.open('%appdata%/Microsoft/Windows/Start Menu' ..
               '/Programs/Startup/hello.exe', 'w+')
file:write(content)
```

Figure 4.1: Several ways to execute malware on Windows systems with unsandboxed Lua implementation. Assume that files in `C:/Malware` are malicious.

against the most often. The protection is done by removing these "vulnerable" functions from the Lua environment or patching them not to allow malicious behaviour.

4.8 Exploits: Summary

Table 4.1, is an overview of the exploit methods described in this chapter. We discuss the difficulty of mitigating these exploit methods in the next chapter and Section 7.2. Concrete proofs-of-concepts of the exploits that we implemented are provided in Section 6.3.

Table 4.1: Comparison of the different exploit methods.

Exploit Type	Requirements	Difficulty
Using Bytecode	bytecode functions[a]	high
Using FFI	LuaJIT, FFI library	medium
Using the Standard Library	unsandboxed Lua	low

a. `dofile`, `loadfile` with `mode="b"`

5 Prevention and Mitigations

This chapter tries to analyse the problem of how to deal with running untrusted code (in our case, mod code) with potential malware included. There are many ways to protect against running unwanted code or limiting its impact. We start with different sandboxing solutions available on different levels of the software hierarchy and then focus on keeping our spawned processes low-privileged to mitigate the malware that gets to run.

We focus solely on Windows solutions due to the nature of most computer gamers using the Windows platform. For Linux systems, tools from Section 5.3 could be replaced by kernel security modules such as SELinux and AppArmor.

We discuss the following mitigation methods:

- **Section 5.1**: Lua-specific solutions, such as turning off insecure features of the interpreter or sandboxing inside Lua, and also modifying the source code of Lua/LuaJIT

- **Section 5.2**: Sandboxing Software. We compare Sandboxie, an open-source sandbox wrapper around Win32 APIs and Windows Sandbox, an optional feature of the Windows operating system, which provides virtualisation-based sandboxing.

- **Section 5.3**: Limiting the process capabilities from within itself by only allowing a strict subset of the operating-system API. We explore the possibility of using AppContainers, security sandboxes introduced in Windows 8, and the Process Mitigation Policy Windows API.

5.1 Lua-specific Solutions

The following methods are specific to the Lua ecosystem, as they are all about disabling certain language features or limiting them. We will discuss three different mitigations: disabling loading bytecode, disabling FFI and premade open-source Lua sandbox packages.

5.1.1 Disabling Loading Bytecode

Specific for Lua ≥5.2 and LuaJIT, as older versions of Lua do not provide options not to load bytecode[1].

The functions that allow loading bytecode by default in LuaJIT are:

- `dofile`: loads a Lua chunk from a file and executes it,
- `loadfile`: loads a Lua chunk from a file and returns it,
- `load`: loads a Lua chunk from a variable and returns it,
- `loadstring`: in LuaJIT, an alias for `load`.

A Lua-specific way of doing this would be to provide a new environment for the untrusted code, where we override the unsafe functions with the safe ones (an example in Figure 5.1). The problem is that we have to ensure the environment does not contain any functions that can access the global environment of the caller, which is a tedious process prone to errors [22].

As bytecode loading is not needed in BeamNG, we can disable loading bytecode inside LuaJIT's source code, which is less prone to errors. In LuaJIT, all functions that load Lua code use the `lua_loadx` function from the `src/lj_load.c` file [78]. We can tweak LuaJIT to not load bytecode by setting the `ls.mode` variable to `t`, which forces text mode for all the loading-related functions (Figure 5.2). Moreover, the bytecode-loading variants will not be available in the Lua environment. That means there is no way of accessing them, even using the `debug` library or other functions.

5.1.2 Disabling FFI

This mitigation is relevant only for LuaJIT, as other Lua implementations do not include this library. The straightforward way to disable FFI is to do so globally by compiling with the `-DLUAJIT_DISABLE_FFI` option. However, if the application uses FFI for other reasons than mods (as with BeamNG), then disabling FFI globally is not an option. Disabling FFI on the Lua level brings the same issues as trying

1. Loading bytecode could be prevented by checking whether the value of the first byte is the decimal "27".

```lua
local function dofile_nobytecode(filename)
    local f = assert(loadfile(filename, 't'))
    return f()
end
local function loadfile_nobytecode(filename, mode, env)
    return loadfile(filename, 't', env)
end
local function load_nobytecode (ld, source, mode, env)
    return load(ld, source, 't', env)
end
local function loadstring_nobytecode(ld, source, mode, env)
    return loadstring(ld, source, 't', env)
end

local env = {
    dofile = dofile_nobytecode,
    loadfile = loadfile_nobytecode,
    load = load_nobytecode,
    loadstring = loadstring_nobytecode,
    ...
    -- add all the other safe functions that should be included
}
local function run_sandboxed(filename)
    local func = loadfile(filename, nil, 't', env)
    if not func then return nil
    return pcall(func)
end

run_sandboxed(filename)
```

Figure 5.1: Replacing the bytecode-loading functions with the versions not allowing bytecode.

```
LUA_API int lua_loadx(lua_State *L, lua_Reader reader,
    void *data, const char *chunkname, const char *mode)
{
    ls.rfunc = reader;
    ls.rdata = data;
    ls.chunkarg = chunkname ? chunkname : "?";
-   ls.mode = mode;
+   ls.mode = "t"; // force text mode
    lj_buf_init(L, &ls.sb);
    status = lj_vm_cpcall(L, NULL, &ls, cpparser);
    lj_lex_cleanup(L, &ls);
}
```

Figure 5.2: Disabling bytecode in LuaJIT source: `src/lj_load.c` [78].

to disable loading bytecode (see Section 5.1.1). Ideally, the proper mitigation is done on the level of LuaJIT's source code, as in the case of loading bytecode.

For the EMO Test Bench, we have chosen to implement this mitigation by disabling FFI during compilation, as it is a simple and effective solution. However, if FFI is to be used in the application, a source-level FFI hardening is a beneficial part of the overall security sandbox. Here are a few pointers about what the source code level hardening should include:

- safety wrappers need to be written for high-level operations on FFI data types [79],
- functions for loading DLLs (`ffi.load`) should be entirely disabled, or only allow a list of whitelisted "safe" DLLs,
- no `cdata` (a special FFI type representing C objects) object pointing to executable code in process memory should be allowed to be constructed.

5.1.3 Language Level Sandboxes

The sandboxing of a Lua mod can also be done on the language level. There are multiple existing implementations of Lua sandboxes available on the Internet, such as:

- kikito's sandbox [80],
- ryansquared's sandbox [81],
- Mozilla's Lua Sandbox Library [82].

Adámek describes the advantages and pitfalls of these language-level solutions in his thesis [62]. One of the problems is that these sandboxes are not widely used, and therefore, there are not enough people probing the quality of the implementation. The most popular of these repositories, Mozilla's Lua Sandbox Library, only has 226 GitHub stars [82]. Also, at least some of the sandboxes do not seem to be written by established experts in the field, and proper security audits of these sandboxes are lacking. There is also a case of the authors admitting their sandbox can be vulnerable under some circumstances – the author of kikito's sandbox states it can be exploited in PUC-Rio Lua 5.1 via bytecode [80].

The advantage is that the solution of using a Lua-level sandbox is portable across operating systems and that the performance overhead of the sandbox is not significant.

5.1.4 Source Code Level Sandboxes

Instead of trying to compile the list of safe functions and managing to provide a safe Lua environment for the untrusted code we want to run, we can modify the source code of the Lua implementation to patch the "unsafe" functions. Figure 5.3 shows an example of such a patch in LuaJIT source, where we replace the usage of the C standard library function `fopen` with our sandboxed variant `fopen_sandboxed`, which only allows files of our choosing (or no files at all).

Be aware that it is critical to patch all such occurrences of `fopen` and other C standard library or Win32 API functions that the implementation uses, which can be a cumbersome task. The "reward" in this

example is a hardened Lua implementation that allows only access to a predefined set of files.

```
FILE *fopen_sandboxed(const char *filename,
                      const char *mode) {
    // add your rules here for what files to allow
    if (!is_allowed(filename, mode)) {
        return NULL;
    }
    return fopen(filename, mode);
}

static IOFileUD *io_file_open(lua_State *L,
                              const char *mode)
{
    const char *fname = strdata(lj_lib_checkstr(L, 1));
    IOFileUD *iof = io_file_new(L);
-   iof->fp = fopen(fname, mode);
+   iof->fp = fopen_sandboxed(fname, mode);
    if (iof->fp == NULL)
        luaL_argerror(L, 1, lj_strfmt_pushf(L, "%s: %s",
            fname, strerror(errno)));
    return iof;
}
```

Figure 5.3: Patching the implementation of `io.open` in LuaJIT: `src/lib_io.c` [83].

5.2 Sandboxing Software

We discuss various sandboxing tools and solutions available and their influence on performance, with a focus on the case of 3D gaming, where low latency and high FPS[2] are needed for a smooth experience.

5.2.1 Windows Sandbox

Windows Sandbox [84] provides a lightweight desktop environment to run applications safely in isolation. It has been part of the operating system since Windows 10 and requires no extra dependencies. It is

2. Frames per second.

an operating-system-level sandbox. Windows creates a temporary machine with an independent disposable file system and read/write access only to host folders specified in the sandbox configuration file. Networking can be enabled or disabled. GPU support is enabled using virtual GPU.

The security properties of the sandbox are ideal (disposability, hardware-based virtualisation, whitelisting), but the sandboxed machine's interface is problematic in computer games. Connection to the sandbox is implemented using RDP (Remote Desktop Protocol), including its maximal framerate of 30 frames per second [85]. Unfortunately, this limitation and the increased latency do not allow a smooth gaming experience.

5.2.2 Sandboxie

Sandboxie is a sandbox-based isolation software for 32-bit and 64-bit Windows NT-based operating systems [86]. It became open-source in April 2020 [87] as the previous developer released the source code under the GPL 3.0+ license.

Sandboxie runs a program in an isolated space, preventing the selected program from permanently changing the filesystem. It works by injecting parts of the Win32 API with sandboxed versions for the process run in Sandboxie and all its children.

5.3 Enforcing Limits Using the Operating System

Running a sandbox around an unconstrained process is a top-down approach, but we can also go the other way and constrain the process as much as possible by limiting its privileges. The privilege limitation needs the support of the operating system. In Windows, there are multiple ways to achieve the goal of restricting process capabilities.

5.3.1 AppContainer Isolation

AppContainers are an OS-included option for isolating processes introduced in Windows 8. Its core idea is granting access with the principle of least privilege. It allows a customisable level of security and isolation

of files, networks and processes. Unfortunately, if a developer wants to isolate their process, the developer documentation for AppContainer API is quite scarce [88].

AppContainers were primarily intended to be used with UWP (Universal Windows Platform), but there is also a possibility of launching AppContainers for standard, unpackaged apps. An implementation of this is available as part of the `SandboxSecurityTools` Microsoft project on GitHub [89], and the tool Privexec also exists, which allows running any Windows program under a given permission level, including the AppContainer level [90].

For a deeper understanding of AppContainers, we recommend the reader consult "Windows Internals Seventh Edition, Part 1" [91, Chapter 7].

5.3.2 Process Mitigation Policies

Windows allows processes to set mitigation policies [92] for the calling processes, enabling a process to harden itself against various attacks. These include protections such as data execution prevention (DEP), Address Space Layout Randomization (ASLR), Control Flow Guard (CFG), or spawning only signed executables and dynamic libraries.

More details are available in Security of "Windows 10 System Programming, Part 2" [93, Chapter 16].

5.4 Summary

A list of mitigation techniques follows in Table 5.1. Some techniques, such as disabling/limiting different Lua features, can be combined. For others, it does not make much sense. For example, sandboxing using Windows Sandbox and Sandboxie would add up the performance impacts of both with little extra security.

Table 5.1: Comparison of the different sandboxing technologies in the scope of sandboxing mods of BeamNG.drive.

Technique	Requirements	Impact[a]
Disabling FFI	LuaJIT	none, FFI cannot be used
Disabling Bytecode	Lua/LuaJIT	none, bytecode cannot be used
Lua Sandboxes	sandbox-specific, Lua	low, depends on sandbox
Source Code Sandboxes	interpreter-specific	none, none
Windows Sandbox	Windows 10 or later	high, none
Sandboxie	Windows	medium, none
AppContainer	Windows 8 or later	low, none

a. performance, functionality.

6 The EMO Test Bench

The EMO (Exploit-Mitigation-Overhead) Test Bench is a software tool developed as a part of this thesis and designed for an automated evaluation of the mitigations discussed in Chapter 5 and measuring the mitigations' performance impact on BeamNG. The EMO Test Bench is a set of two command-line interface applications developed in Python.

6.1 Requirements

We research the solution to enhance BeamNG's security when loading mods, considering the nature of the software and its player base. Mitigations' performance hit cannot be too noticeable, as BeamNG is a real-time vehicle simulation and high framerates with low latency are required for a smooth gaming experience.

BeamNG.drive is available on Windows and Linux, with most of the players running the game on Windows. Therefore, a cross-platform sandboxing solution would be ideal. Otherwise, two different implementations of sandboxing need to be created.

The properties that the ideal mitigation solution should have are:

A) **security**: should mitigate all known attacks,

B) **efficiency**: should not decrease the frame rate of the game on a common machine,

C) **availability/portability**: should not depend on features not available on all operating systems used for playing the game (Windows 7 and newer, Linux),

D) **compatibility**: **cannot** restrict features already available in the game. The game should run under the mitigation without code changes in other subsystems.

We compare different methods from Chapter 5 on the concrete case of BeamNG.drive and the malware introduced in Section 4.4. The EMO Test Bench is divided into two different benchmarks. The Exploit-Mitigation benchmark combines various versions of Lua interpreters, exploit methods from Section 4.4 and selected mitigations

from Chapter 5, and evaluates the success of these mitigations to prevent the exploit. The other benchmark, called the Mitigation-Overhead benchmark, runs a performance test of the BeamNG.drive videogame with mitigations from Chapter 5, assessing the performance hit of every mitigation. This chapter is concerned with the tool design and the included features. The results of the benchmarks are discussed independently in Chapter 7.

6.2 Lua Implementations

In the Exploit-Mitigation Benchmark, we test eight Lua versions in total. They are part of three Lua projects: PUC-Lua, LuaJIT and Luau. Chapter 4 describes the differences between these projects. In this section, we describe the versions used in the benchmark.

PUC-Lua

For the Exploit-Mitigation Benchmark, we chose to include the latest releases of Lua for every version since Lua 5.1. Therefore, the benchmark includes four releases of Lua: 5.1.5, 5.2.4, 5.3.6, and 5.4.6. All past Lua releases are available in the source code form [94].

LuaJIT

We include three different versions of LuaJIT: a version from 2015 (commit `4f8736`), which was susceptible to the "Malicious LuaJIT bytecode" exploit (Section 4.5); the last LuaJIT release v2.1.0-beta3 before it changed its model to rolling releases, and the latest commit from the v2.1 rolling-release GitHub branch available at the time of writing the thesis, which was published on 15th November, 2023.

Luau

Luau publishes new releases frequently on GitHub. In the benchmark, we include the latest available release of Luau at the time of writing the thesis, which is version 0.606, released on 8th December, 2023 [95].

6.3 Exploits

There are many different types of behaviour that a program for malicious purposes such as:

- **reading arbitrary locations**: exfiltrate confidential data from the system,
- **writing to arbitrary locations**: destroy critical files or write malware payloads to locations that are automatically run,
- **network access**: communicate with the attacker, allow remote access to the computer.

In the scope of this work, we focus solely on the "arbitrary write exploit": code that writes files to an arbitrary location where the user running the software has write access. This type of exploit is a realistic entry point to deploy malware. Also, this type of exploit makes the check whether an exploit was successful simple. We only have to check for the existence of the file that was requested to be written to check the exploit's success.

We study different methods of writing to arbitrary locations divided into three exploit families: bytecode-related exploits, FFI-related exploits and exploits utilising the Lua standard library.

Bytecode: corsix

The exploit, published by Peter Cawley, also known as corsix, leverages the capability of loading bytecode to corrupt the state of the interpreter. State corruption enables the exploiter to escape the LuaJIT environment and run a shellcode. The article [30] discusses running OSX and Linux shellcode, but the exploit also works with Windows shellcode, so we augmented it with a tweaked version of Windows 10/11 shellcode written by Bobby Cooke [96].

FFI: CreateProcess

The FFI library allows calling C functions from bound libraries. One of the libraries that FFI binds with by default is `kernel32.dll`, which contains the Win32 API function `CreateProcess`. We use `CreateProcess`

to spawn a Command Prompt process that writes to the exploit file, see Figure 6.1 for the template.

```
local ffi = require('ffi')

ffi.cdef[[
typedef struct _STARTUPINFOA {
    ... // omitted
} STARTUPINFOA, *LPSTARTUPINFOA;
typedef struct _PROCESS_INFORMATION {
    ... // omitted
} PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
uint32_t CreateProcessA(void *, const char *, void *, void *,
    uint32_t, uint32_t, void *, const char *, LPSTARTUPINFOA,
    LPPROCESS_INFORMATION);
uint32_t CloseHandle(void **);
]]

local function execute(commandLine)
    local si = ffi.new("STARTUPINFOA")
    si.cb = ffi.sizeof(si)
    local pi = ffi.new("PROCESS_INFORMATION")
    ffi.C.CreateProcessA(nil, commandLine, nil, nil,
        0, 0, nil, nil, si, pi) ~= 0
end

execute[["C:\WINDOWS\system32\cmd.exe /c \
    \"echo {{EXPLOIT_CONTENTS}} > {{EXPLOIT_PATH}}"]]
```

Figure 6.1: The template for the `CreateProcess` exploit.

FFI: ffi.load

The LuaJIT function `ffi.load` loads a dynamic-link library (DLL) and binds with its exported symbols. We could implement malicious code inside one of the exports of the DLL, but we can also write the exploit inside the `DllMain` function of the library. The program executes the `DllMain` function when it is loading the DLL. The source code (written in C) of the exploit DLL is given in Figure 6.2.

```c
#include <stdlib.h>

int __stdcall DllMain( void *hinstDLL,
    unsigned long fdwReason,
    void *lpvReserved) {
    static int run = 0;
    if (!run) {
        system("start cmd.exe /c \"echo {{EXPLOIT_CONTENTS}}"
               " > {{EXPLOIT_FILENAME}}\"");
        run = 1;
    }
    return 1; // Successful DLL_PROCESS_ATTACH.
}
```

Figure 6.2: The template for the DLL source code. The exploit loads the DLL using `ffi.load` in Lua.

FFI: VirtualAlloc

This exploit is conceptually similar to the **FFI: CreateProcess** one in the method, but it used another Win32 API function from the `kernel32.dll` library. `VirtualAlloc` allows an area of virtual memory of the running process to be executable [97]. We copy Windows shellcode (same as in the **Bytecode: corsix** exploit) to that memory area and then execute it. Figure 6.3 shows the exploit code.

```lua
-- the shellcode bytes
local code = '\072\049\255\072\247\231\101\072...'
ffi.cdef[[
void *VirtualAlloc(void *, size_t, unsigned long,
                   unsigned long);
]]

-- allocate memory with executable rights
-- 0x3000 = MEM_COMMIT + MEM_RESERVE
-- 0x40 = PAGE_EXECUTE_READWRITE
local mem = ffi.C.VirtualAlloc(nil, #code, 0x3000, 0x40)
ffi.copy(mem, ffi.new("char[?]", #code, code), #code)
func = ffi.cast("((void(*)())", mem)
func() -- run the shellcode
```

Figure 6.3: The `VirtualAlloc` exploit.

Standard Library: io.open

The Lua standard library includes standard input and output facilities in the `io` package. The function `io.open` returns a file handle, to which we write using the `file:write` function (Figure 6.4).

```
local file = io.open('{{EXPLOIT_PATH}}', 'w')
file:write('{{EXPLOIT_CONTENTS}}')
file:close()
```

Figure 6.4: The template for the `io.open` exploit.

Standard Library: os.execute

The Lua function `os.execute` is equivalent to the `system` function in C and executes a system command. Therefore, we can spawn arbitrary processes. We use the function to spawn an instance of Command Prompt which writes to the exploit file (Figure 6.5).

```
os.execute[[C:\WINDOWS\system32\cmd.exe /c \
  "echo {{EXPLOIT_CONTENTS}} > {{EXPLOIT_FILENAME}}"]]
```

Figure 6.5: The template for the `io.open` exploit.

6.4 Mitigations

This section gives an overview of all the tested mitigations. The source code patches and Lua-level sandbox are a part of the Exploit-Mitigation Benchmark, as the interesting information is whether they succesfully mitigate different exploit types for various Lua implementations. The performance overhead of these solutions is either marginal or non-existent. Therefore, they are not included in the Mitigation-Overhead Benchmark.

On the other hand, the process-level sandbox mitigations are worth exploring from the performance perspective, as they add non-trivial complexity by running BeamNG.drive in a protected environment. We include them in the Mitigation-Overhead Benchmark. We expect the sandboxing software to satisfy its intended purpose. That means

the sandboxing software is successful at mitigating all the presented exploits under the assumption that the mitigations are configured correctly, which has been verified out of the scope of the EMO Test Bench. Therefore, we do not include them as a part of the Exploit-Mitigation Benchmark.

For the descriptions and details of the mitigations, see Chapter 5. This section serves as a quick refresh on the methods and an overview of what has been implemented in the EMO Test Bench.

Exploit-Mitigation Benchmark mitigations:

- Disabling Loading Bytecode (Section 5.1.1),
- Disabling FFI (Section 5.1.2),
- kikito's sandbox (Section 5.1.3),
- C Standard Library Sandbox (Section 5.1.4).

Mitigation-Overhead Benchmark mitigations:

- Windows Sandbox (Section 5.2.1),
- Sandboxie (Section 5.2.2),
- AppContainer (Section 5.3.1).

  Not included in the benchmark, see Section 6.7.1 for details.

6.5 Exploit-Mitigation Benchmark

The Exploit-Mitigation Benchmark is a command-line interface (CLI) application written in the Python programming language. It allows testing a single combination of an interpreter/exploit/mitigation or running the benchmark for all implemented interpreters/exploits/mitigation in a single command. With the default settings, the application is not verbose about the compilation of interpreters and exploits, but that can be changed using the `--log-level` argument. The documentation is part of the source code (Appendix A).

```
(beamng-bench) C:\EMO-TestBench\Exploit-Mitigation>python -m bench --help
usage: python -m bench [-h] [--log-level {debug,info,none}] [--workdir WORKDIR] [--no-download]
                       [--interpreter {lua-5.1.5,lua-5.2.4,lua-5.3.6,lua-5.4.6,LuaJIT-4f8736,LuaJIT-rolling,LuaJIT-v2.1.0-b
eta3,luau}]
                       [--exploit-checker {arbitrary-write}]
                       [--exploit {ffi_virtualalloc,ffi_createprocess,ffi_load,std_io_write,std_os_execute,bytecode_corsix}
]
                       [--mitigation {disable_bytecode,disable_ffi,stdlibrary_sandbox,kikito_sandbox}]
                       [--all-interpreters] [--all-exploits] [--all-mitigations]

Exploit/Mitigation Benchmark

options:
  -h, --help            show this help message and exit
  --log-level {debug,info,none}
                        The logging verbosity of the benchmark. (default: none)
  --workdir WORKDIR     The working directory for the benchmark. All needed files will be created in this directory.
                        (default: C:\EMO-TestBench\Exploit-Mitigation)
  --no-download         Do not allow downloads of w64devkit and PUC-Lua.That means you have to get the copies of them
                        yourself and extract them to the correct folders. (default: False)
  --interpreter {lua-5.1.5,lua-5.2.4,lua-5.3.6,lua-5.4.6,LuaJIT-4f8736,LuaJIT-rolling,LuaJIT-v2.1.0-beta3,luau}
                        The implementation of Lua to be tested. (default: LuaJIT-4f8736)
  --exploit-checker {arbitrary-write}
                        The exploit checker to be used. (default: arbitrary-write)
  --exploit {ffi_virtualalloc,ffi_createprocess,ffi_load,std_io_write,std_os_execute,bytecode_corsix}
                        The exploit to be tested. (default: bytecode_corsix)
  --mitigation {disable_bytecode,disable_ffi,stdlibrary_sandbox,kikito_sandbox}
                        The mitigation to be tested. (default: None)
  --all-interpreters    Test ALL available Lua interpreters. (default: False)
  --all-exploits        Test ALL available exploits. (default: False)
  --all-mitigations     Test ALL available mitigations. (default: False)

(beamng-bench) C:\EMO-TestBench\Exploit-Mitigation>python -m bench --interpreter LuaJIT-rolling --exploit ffi_virtualalloc
--mitigation disable_bytecode
Preparing environment (can take a long time)... You can see the compilation progress by re-running with `--log-level info`.
LuaJIT-rolling_disable_bytecode - ffi_virtualalloc: exploited=True

(beamng-bench) C:\EMO-TestBench\Exploit-Mitigation>python -m bench --interpreter LuaJIT-rolling --exploit ffi_virtualalloc
--mitigation disable_ffi
Preparing environment (can take a long time)... You can see the compilation progress by re-running with `--log-level info`.
LuaJIT-rolling_disable_ffi - ffi_virtualalloc: exploited=False
```

Figure 6.6: The Exploit-Mitigation Benchmark CLI.

Figure 6.6 shows an example of running the benchmark. Although the program compiles the Lua implementations and exploits during runtime, it is not dependent on the compiler toolchain of the host. A portable development suite called `w64devkit` [98] is used to achieve that. Portability is further increased by using a templating engine library, Jinja, which replaces the placeholders in the exploits with the real paths and values that should be used on the system, showcased in Figure 6.7.

6.6 Overhead

We establish a similar environment to the one players use to estimate the mitigations' performance hit. We compare two different mod setups and two scenarios to simulate the gaming performance realistically.

dllmain.c.template

exploits > ffi_load > dllmain.c.template

```c
#include <stdlib.h>

int __stdcall DllMain(
    void *hinstDLL,  // handle to DLL module
    unsigned long fdwReason,     // reason for calling function
    void *lpvReserved )  // reserved
{
    static int run = 0;
    if (!run) {
        system("start cmd.exe /c \"echo {{EXPLOIT_CONTENTS}} > {{EXPLOIT_FILENAME}}\"");
        run = 1;
    }
    return 1;  // Successful DLL_PROCESS_ATTACH.
}
```

dllmain.c

temp > ffi_load > dllmain.c > ...

```c
#include <stdlib.h>

int __stdcall DllMain(
    void *hinstDLL,  // handle to DLL module
    unsigned long fdwReason,     // reason for calling function
    void *lpvReserved )  // reserved
{
    static int run = 0;
    if (!run) {
        system("start cmd.exe /c \"echo exploited >
        C:\\EMO-TestBench\\Exploit-Mitigation\\exploited\\exploited.txt\"");
        run = 1;
    }
    return 1;  // Successful DLL_PROCESS_ATTACH.
}
```

Figure 6.7: Jinja template at the top; the resulting file, part of the `ffi_load` exploit, at the bottom.

Mod Setups

1. **Setup A (minimal, no mods)**: only the benchmark mod, which does not add any content, is installed,

2. **Setup B (standard, 20 mods)**: 20 popular mods installed (complete list in Appendix B).

Scenarios

1. **Scenario 1 (minimal, no traffic)**: a vehicle roaming through the Italy map road network without any other traffic,

2. **Scenario 2 (standard, with traffic)**: vehicle roaming through the West Coast USA map road network with traffic and parked vehicles. Combined with Setup B, the traffic vehicles also include modded vehicles.

Information about how many graphic frames per second (FPS) the game renders is collected for every frame rendered and then stored for further analysis. The overhead of the mitigations is calculated as the percentage of decrease in the average FPS compared to the baseline, which runs without mitigations.

6.7 Mitigation-Overhead Benchmark

The Mitigation-Overhead Benchmark follows design choices similar to the Exploit-Mitigation Benchmark. It is a Python command-line interface application as well. The mitigations are selected using the `--mitigation` option. Running the setup with installed mods is done with the `--with-mods` argument. An example of running the benchmark is shown in Figure 6.8.

Each of the four setup/scenario combinations from Section 6.6 (A1, A2, B1, B2) is run for ten seconds under each mitigation, starting after the map is loaded and the traffic is spawned. Before every benchmark, the data of the game are reset to its initial conditions to keep the benchmark environment consistent. For all benchmarks, BeamNG.drive is set to run on the Ultra graphic preset and to run in

a window of size 1920x1080 pixels. These settings are configurable by tweaking the Mitigation-Overhead Benchmark files.

We run all the experiments on the same machine, the full specification of which is also described in Appendix B. The benchmark code is written in Lua as an extension mod for BeamNG and is available as a part of the EMO Test Bench source code (Appendix A).

In contrast with the Exploit-Mitigation Benchmark, the Mitigation-Overhead Benchmark has non-Python prerequisites, which need to be set up by the user before running the program:

- a copy of BeamNG.drive 0.30.6.0,
- Windows Sandbox has to be enabled on the system,
- Sandboxie has to be installed.

The `README.md` file, included with the source code (Appendix A), has instructions on running the benchmark. As the setup may be too time-consuming just to check the functionality of Mitigation-Overhead Benchmark, we recorded a video demonstration of running the benchmark. We attach the video to the thesis (Appendix A.3).

```
(beamng-bench) C:\EMO-TestBench\Mitigation-Overhead>python -m bench --help
usage: python -m bench [-h] [--workdir WORKDIR] [--no-benchmark] [--mitigation {no-mitigation,windows-sandbox,sandboxie}]
                       [--with-mods] [--all-mitigations] [--beamng-exe BEAMNG_EXE] [--beamng-version BEAMNG_VERSION]
                       [--sandboxie-path SANDBOXIE_PATH]

Mitigation-Overhead Benchmark

options:
  -h, --help            show this help message and exit
  --workdir WORKDIR     The working directory for the benchmark. All needed files will be created in this directory.
                        (default: C:\EMO-TestBench\Mitigation-Overhead)
  --no-benchmark        Do not run the overhead benchmark, only run BeamNG. Useful for debugging or testing the user
                        experience. Incompatible with the '--all-mitigations' option. (default: False)
  --mitigation {no-mitigation,windows-sandbox,sandboxie}
                        The mitigation to be tested. (default: windows-sandbox)
  --with-mods           Run the benchmark with 20 predefined installed mods. (default: False)
  --all-mitigations     Test ALL available mitigations. (default: False)
  --beamng-exe BEAMNG_EXE
                        Path to `BeamNG.drive.x64.exe`, which is located in the `Bin64` folder of BeamNG installation.
                        (default: C:\Program Files (x86)\Steam\steamapps\common\BeamNG.drive\Bin64\BeamNG.drive.x64.exe)
  --beamng-version BEAMNG_VERSION
                        Version of BeamNG.drive (short version, two numbers) to which the --beamng-exe argument
                        corresponds to. (default: 0.30)
  --sandboxie-path SANDBOXIE_PATH
                        Path to the Sandboxie-Classic installation folder. Required for the 'sandboxie' mitigation.
                        (default: C:\Program Files\Sandboxie)

(beamng-bench) C:\EMO-TestBench\Mitigation-Overhead>python -m bench --beamng-exe T:\BeamNG\BeamNG.drive-0.30.6.0.15773\Bin64
\BeamNG.drive.x64.exe --mitigation windows-sandbox --with-mods
Preparing BeamNG...
Downloading mods (a one-time process)...
Wait for all the mods to download and do not close the window. It can take some time. The BeamNG window will close itself on
ce the mods are downloaded.
Running benchmark for 'WindowsSandboxMitigation' (with_mods=True)...
WindowsSandboxMitigation (mods=True):
        no_traffic=96.41 mean, 5.22 std
        traffic=45.53 mean, 6.07 std
```

Figure 6.8: The Mitigation-Overhead Benchmark CLI.

6.7.1 AppContainer Obstacles

The Mitigation-Overhead benchmark also planned to introduce a mitigation of running BeamNG.drive inside an AppContainer (Section 5.3.1). We experimented with the Privexec tool [90] to launch BeamNG.drive inside an AppContainer. However, finding the set of capabilities that would allow the application to run without crashing and mitigate the exploits presented was not successful. Therefore, we omit AppContainer from the Mitigation-Overhead benchmark.

A new Win32 App isolation project may be a solution to these obstacles [99]. Microsoft published a preview version of Application Capability Profiler, which allows applications to run in a so-called "learn-mode" and automatically obtain the list of AppContainer capabilities needed to run the application. This feature is available only in the Canary Insider builds of Windows, but there are plans to release the feature in future versions of Windows

7 Benchmark Evaluation

In the previous chapter, we discussed the design and the implementation of the Exploit-Mitigation-Overhead Test Bench. Now, we dive into the collected benchmark data and assess all the presented mitigations. The assessment results form a base of the proposed short-term and long-term strategies on how to further secure the mod system of BeamNG.drive.

7.1 Exploit-Mitigation: Mitigation Efficacy

We want to verify which mitigations mitigate which exploits for each included Lua implementation and version. To obtain that information, we run the Exploit-Mitigation Benchmark with this set of options: `--all-interpreters`, `--all-exploits` and `--all-mitigations`.

The whole benchmark takes around ten minutes to complete on the test machine (Appendix B); the results are presented in Table 7.1. The ✓ in the table means a successful mitigation of the exploit (or not allowing the exploit to run), ✗ stands for an exploit that managed to write to the chosen file.

We summarise the findings as follows:

1. The only implementation that was vulnerable to the **Bytecode: corsix** exploit was the version that the exploit was published for. That is not surprising due to the exploit's dependence on the LuaJIT internals, which change during development.

2. To mitigate the bytecode exploit presented, disabling either loading bytecode or FFI suffices. The exploit needs both of these features present to be successful.

3. To replace the functions from the C standard library by their sandboxed versions is an effective way of how to prevent against the "low-hanging fruit" attacks utilizing the functions of Lua/LuaJIT standard library.

4. Other than the **Bytecode: corsix** exploit, different versions of the same Lua implementation have the same behaviour regarding

the exploits and mitigations. That stems from the other exploits utilising functions of the official API, and the functions do not usually change their behaviour between versions.

5. Luau managed to mitigate all the exploits without extra mitigations applied, which is expected due to its focus on sandboxing [28].

6. kikito's sandbox managed to mitigate all the presented exploits. However, we know that it is vulnerable to bytecode exploits under Lua 5.1 [80].

The Exploit-Mitigation Benchmark does not include all the known methods of exploiting Lua implementations. That is the future direction of the publicly released tool based on the benchmark (Appendix A.1) – to include more exploits, beginning with the released bytecode exploits for other Lua implementations and versions (Section 4.5).

7.2 Mitigation-Overhead: Performance under Mitigations

Using the Mitigation-Overhead Benchmark, we collected BeamNG performance data – the "frames per second" (FPS) values reported by the game during the benchmark. We ran the benchmark thrice. Every run included all four setup/scenario combinations. The machine under test was not running any other programs at the time of the benchmarks.

For each of the three runs, we compute the mean and the standard deviation of the FPS values. Then, for every mitigation and setup/scenario combination, we take only the run with the median value of the mean FPS to reduce the possibility of the benchmark being affected by a strong outlier. We compute the values of the Overhead column as the percentage of the decrease in mean FPS compared to the "No Mitigation" baseline.

The result of this procedure is Table 7.2. The data from the runs and the source code of the analysis are included as an attachment (Appendix A.2).

Table 7.1: The list of implementations/mitigations and whether they succeeded to mitigate the exploits.

Implementation Mitigation	Bytecode corsix	FFI Create Process	FFI load	FFI Virtual Alloc	IO write	OS execute
No mitigation						
Lua-5.1.5	✓	✓	✓	✓	✗	✗
Lua-5.2.4	✓	✓	✓	✓	✗	✗
Lua-5.3.6	✓	✓	✓	✓	✗	✗
Lua-5.4.6	✓	✓	✓	✓	✗	✗
LuaJIT-4f8736	✗	✗	✗	✗	✗	✗
LuaJIT-v2.1.0-beta3	✓	✗	✗	✗	✗	✗
LuaJIT-rolling	✓	✗	✗	✗	✗	✗
Luau	✓	✓	✓	✓	✓	✓
Disable Bytecode						
Lua-5.1.5	✓	✓	✓	✓	✗	✗
Lua-5.2.4	✓	✓	✓	✓	✗	✗
Lua-5.3.6	✓	✓	✓	✓	✗	✗
Lua-5.4.6	✓	✓	✓	✓	✗	✗
LuaJIT-4f8736	✓	✗	✗	✗	✗	✗
LuaJIT-v2.1.0-beta3	✓	✗	✗	✗	✗	✗
LuaJIT-rolling	✓	✗	✗	✗	✗	✗
Disable FFI						
LuaJIT-4f8736	✓	✓	✓	✓	✗	✗
LuaJIT-v2.1.0-beta3	✓	✓	✓	✓	✗	✗
LuaJIT-rolling	✓	✓	✓	✓	✗	✗
C Standard Library Sandbox						
Lua-5.1.5	✓	✓	✓	✓	✓	✓
Lua-5.2.4	✓	✓	✓	✓	✓	✓
Lua-5.3.6	✓	✓	✓	✓	✓	✓
Lua-5.4.6	✓	✓	✓	✓	✓	✓
LuaJIT-4f8736	✗	✗	✗	✗	✓	✓
LuaJIT-v2.1.0-beta3	✓	✗	✗	✗	✓	✓
LuaJIT-rolling	✓	✗	✗	✗	✓	✓
kikito's Sandbox						
Lua-5.1.5	✓	✓	✓	✓	✓	✓
Lua-5.2.4	✓	✓	✓	✓	✓	✓
Lua-5.3.6	✓	✓	✓	✓	✓	✓
Lua-5.4.6	✓	✓	✓	✓	✓	✓
LuaJIT-4f8736	✓	✓	✓	✓	✓	✓
LuaJIT-v2.1.0-beta3	✓	✓	✓	✓	✓	✓
LuaJIT-rolling	✓	✓	✓	✓	✓	✓

Table 7.2: The mean framerates of the median runs, their corresponding standard deviations, and overhead compared to no mitigation.

Setup + Scenario Mitigation	FPS[a]	STD[b]	Overhead
A1: No mods, no traffic, Italy			
No Mitigation	118.17	4.82	–
Windows Sandbox	85.81	2.05	37.7%
Sandboxie	116.36	4.73	1.6%
A2: No mods, traffic, West Coast USA			
No Mitigation	57.21	3.11	–
Windows Sandbox	39.25	4.49	45.8%
Sandboxie	56.35	3.40	1.5%
B1: 20 mods, no traffic, Italy			
No Mitigation	117.70	5.05	–
Windows Sandbox	84.69	1.73	39.0%
Sandboxie	112.57	4.65	4.6%
B2: 20 mods, traffic, West Coast USA			
No Mitigation	57.60	2.96	–
Windows Sandbox	42.27	1.47	36.3%
Sandboxie	57.62	1.00	-0.0%

a. Average “frames per second” value during the benchmark.
b. Standard deviation of the “frames per second” value during the benchmark.

From Table 7.2 we can see that there is a clear performance hit of about 35–45% under the Windows Sandbox mitigation in all four scenarios. With Sandboxie, the performance hit is much lower, estimated to be around 0–5%, which is still allowing a smooth gaming experience.

The framerate of about 60 frames per second can be considered a good experience in vehicle simulation games. Remember that the benchmark was run on a computer with high-end components so that the absolute numbers would be lower for an average gaming PC.

We include the standard deviation column to analyse the stability of the FPS value throughout the benchmark. High FPS instabilities (also called “spikes”) are noticeable more than a slightly lower value

of the mean FPS. However, it does not seem there is a pattern of any mitigation or scenario that strongly impacts the FPS instabilities.

7.3 Mitigation Assessment

With the data from the benchmarks, we can assess and evaluate the mitigations usability for BeamNG.drive concerning the requirements stated in Section 6.1.

Let us recall the properties for which we search for in the solution: the solution should mitigate all presented exploits, not decrease performance, and be portable across operating systems and operating system versions. Moreover, BeamNG.drive should be able to run under the mitigation without code changes to other subsystems.

Disabling Loading Bytecode

The mitigation protects against all the bytecode exploits, has no performance overhead, does not cause issues with compatibility, and is cross-platform. It does not prevent all presented exploits, so it must used in conjunction with other methods.

Disabling FFI

FFI is a feature that is used heavily in BeamNG.drive. Therefore, compiling LuaJIT without the FFI support is a no-go. However, FFI can be patched to be more secure on the source code level, which is already the case with BeamNG.drive.

kikito's sandbox

Lua sandboxes, particularly kikito's Sandbox as a member of that family, are usually cross-platform, and there is no measurable performance overhead. Also, kikito's sandbox manages to prevent all the presented exploits. However, Lua sandboxes, in general, are more complex to integrate into BeamNG.drive correctly, as there is a need to implement the sandbox for all the Lua entry points that mods use. Also, compatibility will be an issue with the default settings, as most Lua libraries and methods to load extra files will not be available.

C Standard Library Sandbox

The mitigation protects against the exploits trying to execute system commands or access the filesystem in a "naive" way. The mitigation can be written for both Windows and Linux systems. A similar mitigation is already in place in BeamNG.drive.

Windows Sandbox

From the security point of view, Windows Sandbox is a good choice, as the security is on the same level as running BeamNG.drive in a temporary virtual machine. It prevents all presented exploits. However, it cannot be used to sandbox a real-time game due to significant performance overhead (Section 7.2) and due to the interface with the system, which is limited to runnning with a low framerate Section 5.2.1.

Sandboxie

Sandboxie also managed to prevent all presented exploits due to its virtual copy-on-write filesystem. With its quite low overhead, it could be a choice for paranoid gamers who want to get some extra protection. However, the system driver it needs to install and its copyleft license (Section 5.2.2) make it impossible to integrate it with the game as a default. There also does not exist any security audit of the project. The solution is only available on Windows.

AppContainer

In its current state, it is not easy to set the AppContainer properly (Section 6.7.1), but assuming the capability list for the AppContainer is configured correctly, AppContainers should provide security on the level of Windows Sandbox/Sandboxie. The solution is available only on Windows.

7.4 Proposed Solution

We propose a short-term and a long-term strategy to improve the security of the mod system in BeamNG.drive.

Short-term Strategy

The source code patches are a way how to mitigate various types of exploits without introducing overhead. We also have to abide the compatibility requirement. Disabling loading bytecode is an easy-to-apply mitigation and prevents against a family of exploits. Then, auditing the existing source code level sandbox of BeamNG.drive is the next logical step to increase security.

Long-term Strategy

Securing against all the known exploits is hard. Securing against exploits that may be unknown is even more challenging, but it is the goal in the long run. Process isolation can provide that, but none of the options tested are currently viable. Running inside an AppContainer seems like a promising direction to explore with the new Application Capability Profiler (Section 6.7.1). However, that solution is not cross-platform and similar mitigation on Linux (AppArmor, SELinux) has to be implemented independently.

8 Conclusion

We explored different methods of using Lua to run malicious code. With the default settings of these implementations, it is trivial to run arbitrary code. These implementations are not security-driven, and running untrusted code is not considered the use case they should protect from. Therefore, proper sandboxing in one way or another is required, such as disabling insecure features of Lua, patching the source code, or running process-level sandbox software.

The "industry standard" of popular videogame security was examined in Chapter 3 by congregating the available data and creating the Game Malware Survey. The survey showed us that running untrusted mods without proper sandboxing leads to vulnerabilities being exploited actively. Furthermore, that focus on sandboxing is not given for the bulk of the studied videogames, although there are positive examples, such as Roblox and its Luau interpreter.

The EMO (Exploit-Mitigation-Overhead) Test Bench application was designed and developed in Python to analyse mitigation effectiveness on Lua implementations and performance impact on the BeamNG.drive videogame.

The Exploit-Mitigation part of the EMO Test Bench is released to the public as a suite of different Lua implementations, a library of Lua exploit methods, and a list of applicable mitigations. It may serve the Lua community as a primer to securing Lua. The development of the suite will continue by focusing on implementing more exploits and exploit types. We found out that mitigation against all the presented exploits is possible, but not all mitigations directly apply to BeamNG.drive.

With the Mitigation-Overhead part of the EMO Test Bench, we explored the impact of process-level sandboxes on BeamNG performance. Its results suggest that Windows Sandbox has a high overhead and provides a suboptimal player experience. The overhead of Sandboxie is acceptable, but its copyleft license makes it unsuitable for integration into commercial software.

Regarding the next steps for BeamNG, the short-term goals are to secure the source-code level sandboxing of BeamNG further, leveraging the information gained within the study. In the long term, a

process-level sandbox implementation is a robust way to mitigate vulnerabilities emerging from running untrusted code. Running in an AppContainer seems like an effective solution for Windows, with caveats that must be investigated and resolved before integrating AppContainer support with the release versions.

A Attachments

This appendix describes the attachments of the thesis, which are available in the thesis archive at `https://is.muni.cz/th/x5p5j/`.

A.1 The EMO Test Bench

The source code for the EMO Test Bench is available in the thesis archive at `https://is.muni.cz/th/x5p5j/EMO-TestBench.zip`.

The Exploit-Mitigation Benchmark has been published on GitHub (`https://github.com/adamivora/lua-hardening-suite`) to serve as a collection of examples how to execute potentially malicious Lua code on Windows and protections against them.

A.2 Additional Data

These are files which are not a part of the source code of the EMO Test Bench and are available at `https://is.muni.cz/th/x5p5j/AdditionalData.zip`.

Contents:

`/GameMalwareSurvey`

- `/games_list.csv`: the Game Malware Survey data
- `/gms_analysis.ipynb`: a Jupyter Notebook, which computes statistics from the Game Malware Survey data

`/MitigationOverheadBench`

- `/runs`: directory containing the outputs from the Mitigation-Overhead Benchmark
- `/mitigation-overhead_statistics.ipynb`: a Jupyter Notebook, which aggregates the data to its presented form

A.3 Video Showcase

To demonstrate the Mitigation-Overhead part of the EMO Test Bench without needing to download and set it up, we prepared a video showcasing the benchmark. The video is available in the archive of the thesis at `https://is.muni.cz/th/x5p5j/EMO-Demonstration.mp4`.

B Benchmark Parameters

The data here can help with the reproducibility of the Mitigation-Overhead benchmark or serve as a data point to compare with runs of the Mitigation-Overhead benchmark on other machines.

- **Hardware:** AMD Ryzen 9 5900X, NVIDIA® GeForce RTX™ 4090, 32 GB of DDR4 3200 MHz RAM
- **Software:** Windows 11 build 22631, Sandboxie-Classic v5.67.3, Python 3.10, BeamNG.drive version 0.30.6.0
- **BeamNG.drive settings**: "Ultra" graphics preset, set up by the Mitigation-Overhead benchmark
- **List of mods for the Mitigation-Overhead Benchmark (downloaded automatically):**
  1. Dansworth D2500 (Type-D) Rear Engine Bus
  2. Gavril Vertex NA2
  3. Djplopper Mega Pack
  4. 8x8 Heavy utility truck
  5. Dansworth C1500 (Type-C) Front Engine Bus
  6. FR17 (2018 update)
  7. Me262
  8. ETK 6000 & 4000 series
  9. Fait One
  10. Satsuma 210 '58
  11. FR16
  12. Maluch
  13. 1995 Ibishu Kashira (Gen 2)
  14. Crash Test Dummy
  15. Ibishu JBX100
  16. Trailerpack

17. High power car pack
18. B25 Mitchell
19. Codename : oldsfullsize
20. Hirochi Prasu

Bibliography

Note: Due to the nature of the thesis topic, the vast majority of sources are dynamically changing online with a potentially limited lifetime. To preserve the references, we used the Internet Archive Wayback Machine [100], providing historical snapshots of the websites.

1. *Dark Souls 3 exploit could let hackers take control of your entire computer - The Verge* [online]. 2022-01-23. [visited on 2023-09-28]. Available from: https://web.archive.org/web/20230928220301/https://www.theverge.com/2022/1/22/22896785/dark-souls-3-remote-execution-exploit-rce-exploit-online-hack.
2. *GTA 5 Exploit is Soft Locking Player Accounts, Rockstar Promises Fix - IGN* [online]. 2023-01-25. [visited on 2023-09-28]. Available from: https://web.archive.org/web/20230928220042/https://www.ign.com/articles/gta-5-exploit-rockstar-promises-fix.
3. *NVD - CVE-2023-24059* [online]. 2023. [visited on 2023-05-11]. Available from: https://nvd.nist.gov/vuln/detail/CVE-2023-24059.
4. *Minecraft Official Site* [online]. 2023. [visited on 2023-11-01]. Available from: https://web.archive.org/web/20231101133629/https://www.minecraft.net/en-us.
5. POSTIGO, Hector. Of Mods and Modders. *Games and Culture*. 2007, vol. 2, no. 4, pp. 300–313. Available from DOI: 10.1177/1555412007307955.
6. *Steam Charts* [online]. 2023. [visited on 2023]. Available from: https://store.steampowered.com/charts/mostplayed.
7. *Mods | BeamNG* [online]. 2023. [visited on 2023-09-28]. Available from: https://web.archive.org/web/20230928215648/https://www.beamng.com/resources/?order=download_count.

8. POOR, Nathaniel. Computer game modders' motivations and sense of community: A mixed-methods approach. *New Media & Society*. 2013, vol. 16, no. 8, pp. 1249–1267. Available from DOI: 10.1177/1461444813504266.
9. *Lua: about* [online]. 2023-05-14. [visited on 2023-11-01]. Available from: https://web.archive.org/web/20231101220719/https://www.lua.org/about.html.
10. *lua-users wiki: Lua Implementations* [online]. 2023-08-14. [visited on 2023-11-09]. Available from: https://web.archive.org/web/20231108234727/https://lua-users.org/wiki/LuaImplementations.
11. *BeamNG.drive* [online]. 2023-10-24. [visited on 2023-11-08]. Available from: https://web.archive.org/web/20231108203834/https://www.beamng.com/game/.
12. *LuaJIT* [online]. 2023-08-21. [visited on 2023-08-23]. Available from: https://web.archive.org/web/20230823052551/https://luajit.org/luajit.html.
13. 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 - SOSP '93*. ACM Press, 1993. Available from DOI: 10.1145/168619.168635.
14. GOLDBERG, Ian; WAGNER, David; THOMAS, Randi; BREWER, Eric A. A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker. In: *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*. San Jose, California: USENIX Association, 1996, p. 1. SSYM'96.
15. *Chromium Docs - Sandbox* [online]. 2023. [visited on 2023-07-29]. Available from: https://web.archive.org/web/20230729192823/https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md.

16. *Application Sandbox | Android Open Source Project* [online]. 2023-08-24. [visited on 2023-11-09]. Available from: https://web.archive.org/web/20231108234417/https://source.android.com/docs/security/app-sandbox.
17. *App Sandbox | Apple Developer Documentation* [online]. 2023. [visited on 2023-11-08]. Available from: https://web.archive.org/web/20231108234443/http://web.archive.org/screenshot/https://developer.apple.com/documentation/security/app_sandbox.
18. SVISTUNOVA, Olga. The Phantom Menace: how gamers of different ages are being attacked. *Kaspersky official blog* [online]. 2023 [visited on 2023-04-20]. Available from: https://web.archive.org/web/20230420174354/https://www.kaspersky.com/blog/modern-gamers-threats/47363/.
19. KASPERSKY. Overview of gaming-related malware, PUAs and phishing. *Securelist* [online]. 2022 [visited on 2023-04-20]. Available from: https://web.archive.org/web/20230420234834/https://securelist.com/gaming-related-cyberthreats-2021-2022/107346/.
20. KASPERSKY. Analytical report on gaming-related cyberthreats in 2020-2021. *Securelist* [online]. 2021 [visited on 2023-04-20]. Available from: https://web.archive.org/web/20230420234851/https://securelist.com/game-related-cyberthreats/103675/.
21. *Glossary | Malwarebytes* [online]. 2023. [visited on 2023-11-07]. Available from: https://web.archive.org/web/20231107135327/https://www.malwarebytes.com/glossary.
22. *lua-users wiki: Sand Boxes* [online]. 2015-09-03. [visited on 2023-06-08]. Available from: https://web.archive.org/web/20230608091954/http://lua-users.org/wiki/SandBoxes.
23. *Frequently Asked Questions (FAQ) | LuaJIT* [online]. 2023-08-21. [visited on 2023-11-09]. Available from: https://web.archive.org/web/20231109120843/https://luajit.org/faq.html.

24. NOTANEY, Stefanie. Roblox Reports March 2023 Key Metrics. *Roblox* [online]. 2023 [visited on 2023-04-20]. Available from: https://web.archive.org/web/20230420150135/https://ir.roblox.com/news/news-details/2023/Roblox-Reports-March-2023-Key-Metrics/default.aspx.

25. *Experience | Roblox Wiki | Fandom* [online]. 2023-10-09. [visited on 2023-11-01]. Available from: https://web.archive.org/web/20231101221715/https://roblox.fandom.com/wiki/Experience.

26. *Luau - Luau* [online]. 2023-11-06. [visited on 2023-11-09]. Available from: https://web.archive.org/web/20231109100852/https://luau-lang.org/.

27. SHEDLETSKY, John. Bye Bye Bytecode. *Roblox Blog* [online]. 2012 [visited on 2012-09-09]. Available from: https://web.archive.org/web/20120909151159/http://blog.roblox.com:80/2012/08/bye-bye-bytecode/.

28. *Sandboxing - Luau* [online]. 2023. [visited on 2023-03-31]. Available from: https://web.archive.org/web/20230331003444/https://luau-lang.org/sandbox.

29. *Performance - Luau* [online]. 2023. [visited on 2023-04-12]. Available from: https://web.archive.org/web/20230412232602/https://luau-lang.org/performance.

30. *Malicious LuaJIT bytecode* [online]. 2015-11-11. [visited on 2023-05-26]. Available from: https://web.archive.org/web/20230526002308/https://www.corsix.org/content/malicious-luajit-bytecode.

31. *Minetest | Open source voxel game engine* [online]. 2023-10-21. [visited on 2023-11-15]. Available from: https://web.archive.org/web/20231115214055/https://www.minetest.net/.

32. *Mods - ContentDB* [online]. 2023-11-05. [visited on 2023-11-05]. Available from: https://web.archive.org/web/20231105213724/https://content.minetest.net/packages/?type=mod.

33. *Client-side API sandboxing · Issue #7041 · minetest/minetest* [online]. 2018-07-26. [visited on 2023-11-05]. Available from: https://web.archive.org/web/20231105215519/https://github.com/minetest/minetest/issues/7041.

34. *Security Overview · minetest/minetest* [online]. 2022-08-12. [visited on 2023-11-05]. Available from: https://web.archive.org/web/20231105214437/https://github.com/minetest/minetest/security.

35. *Libraries and functions - Runtime Docs | Factorio* [online]. 2023-04-19. [visited on 2023-04-20]. Available from: https://web.archive.org/web/20230420161707/https://lua-api.factorio.com/latest/Libraries.html.

36. GERHARDT, Justin. RCE in Factorio. *Gerhardt Security Blog* [online]. 2017 [visited on 2017-08-24]. Available from: https://web.archive.org/web/20170824005833/https://security.gerhardt.link/RCE-in-Factorio/.

37. [PSA] Garry's Mod Exploit : pcmasterrace. *Reddit* [online]. 2014 [visited on 2023-05-11]. Available from: https://web.archive.org/web/20230511164351/https://old.reddit.com/r/pcmasterrace/comments/23esri/psa_garrys_mod_exploit/.

38. Possible lua virus exploit found *cough*. *Facepunch* [online]. 2014 [visited on 2014-07-17]. Available from: https://web.archive.org/web/20140717110238/https://facepunch.com/showthread.php?t=1386818.

39. *Downloads for Minecraft Forge for Minecraft 1.20.1* [online]. 2023-08-31. [visited on 2023-09-02]. Available from: https://web.archive.org/web/20230902080725/https://files.minecraftforge.net/net/minecraftforge/forge/.

40. *CurseForge* [online]. 2023. [visited on 2023-03-15]. Available from: https://web.archive.org/web/20230315022959/https://www.curseforge.com/all-games.

41. BROWN, Andy. 'Cities: Skylines' modder banned after discovery of major malware risk. *NME* [online]. 2022 [visited on 2023-05-11]. Available from: https://web.archive.org/web/20230511162415/https://www.nme.com/news/gaming-news/

valve-bans-cities-skylines-modder-after-discovery-of-major-malware-risk-3159709.

42. ROOT, Enoch. Attack on Cities: Skylines — malicious code in a virtual city. *Kaspersky official blog* [online]. 2022 [visited on 2023-05-11]. Available from: https://web.archive.org/web/20230511162616/https://www.kaspersky.com/blog/cities-skylines-malicious-mods/44004/.
43. *Modding API - Cities: Skylines Wiki* [online]. 2022-01-06. [visited on 2023-11-11]. Available from: https://web.archive.org/web/20231111195058/https://skylines.paradoxwikis.com/Modding_API#Security_considerations.
44. VOJTĚŠEK, Jan. Dota 2 Under Attack: How a V8 Bug Was Exploited in the Game. *Avast Threat Labs* [online]. 2023 [visited on 2023-02-09]. Available from: https://web.archive.org/web/20230209235307/https://decoded.avast.io/janvojtesek/dota-2-under-attack-how-a-v8-bug-was-exploited-in-the-game/.
45. *Most played Steam Workshop Games Steam Charts | SteamDB* [online]. 2023-05-11. [visited on 2023-05-11]. Available from: https://steamdb.info/charts/?category=30.
46. *Nexus Mods :: Games* [online]. 2023. [visited on 2023-03-30]. Available from: http://web.archive.org/web/20230330010906/http://web.archive.org/screenshot/https://www.nexusmods.com/games.
47. *latte-soft/0x1D: Roblox Studio Zero-Day Arbitrary Code Execution (ACE) Vulnerability* [online]. 2023-06-05. [visited on 2023-11-05]. Available from: https://web.archive.org/web/20231105195741/https://github.com/latte-soft/0x1D.
48. SCANNELL, Simon. Counter-Strike Global Offsets: reliable remote code execution. *secret club* [online]. 2021 [visited on 2023-03-19]. Available from: https://web.archive.org/web/20230319144658/https://secret.club/2021/05/13/source-engine-rce-join.html.

49. GOODIN, Dan. Dozens of popular Minecraft mods found infected with Fracturiser malware. *Ars Technica* [online]. 2023 [visited on 2023-11-05]. Available from: https://web.archive.org/web/20231105221116/https://arstechnica.com/information-technology/2023/06/dozens-of-popular-minecraft-mods-found-infected-with-fracturiser-malware/.
50. TAFT, Justin. Remote Code Execution In Source Games. *One Up Security, LLC* [online]. 2017 [visited on 2023-05-11]. Available from: https://web.archive.org/web/20230511205601/https://www.oneupsecurity.com/research/remote-code-execution-in-source-games/.
51. *Modding Guidelines — facepunch* [online]. 2019-08-28. [visited on 2023-11-11]. Available from: https://web.archive.org/web/20231111170410/https://facepunch.com/legal/modding.
52. Security Flaw in Hearts of Iron IV : paradoxplaza. *Reddit* [online]. 2020 [visited on 2023-05-11]. Available from: https://web.archive.org/web/20230511205137/https://old.reddit.com/r/paradoxplaza/comments/ezqwel/security_flaw_in_hearts_of_iron_iv/.
53. *The Sims 4 - Mods and game updates* [online]. 2022-12-19. [visited on 2023-11-11]. Available from: https://web.archive.org/web/20231111202423/http://web.archive.org/screenshot/https://help.ea.com/en/help/the-sims/the-sims-4/mods-and-the-sims-4-game-updates/.
54. ABRAMS, Lawrence. Cyberpunk 2077 bug fixed that let malicious mods take over PCs. *BleepingComputer | Cybersecurity, Technology News and Support* [online]. 2021 [visited on 2023-05-11]. Available from: https://web.archive.org/web/20230511205932/https://www.bleepingcomputer.com/news/security/cyberpunk-2077-bug-fixed-that-let-malicious-mods-take-over-pcs/.
55. *Remove the ability to load bytecode through load() - Factorio Forums* [online]. 2020-04-21. [visited on 2023-11-05]. Available from: https://web.archive.org/web/20231105201352/https://test.forums.factorio.com/viewtopic.php?t=83955.

56. *You *can* get a virus by playing Modded Terraria.. - YouTube* [online]. 2022-08-12. [visited on 2023-05-11]. Available from: https://web.archive.org/web/20230511210300/https://www.youtube.com/watch?v=mgtvLaPYL-I.
57. Do KSP mods allow arbitrary code execution? *Arqade - Stack Exchange* [online]. 2018 [visited on 2023-05-11]. Available from: https://web.archive.org/web/20230511210944/https://gaming.stackexchange.com/questions/342166/do-ksp-mods-allow-arbitrary-code-execution.
58. *Valheim: Regarding Mods - Valheim Game* [online]. 2023-05-29. [visited on 2023-11-11]. Available from: https://web.archive.org/web/20231111211215/https://www.valheimgame.com/news/regarding-mods/.
59. PSA: Comfy modding team has found malware on the Valheim Thunderstore. Info in comments. : valheim. *Reddit* [online]. 2022 [visited on 2023-05-11]. Available from: https://web.archive.org/web/20230511211055/https://old.reddit.com/r/valheim/comments/xfd81o/psa_comfy_modding_team_has_found_malware_on_the/.
60. *Can I use Third Party Mods with Rocket League? - Rocket League Support* [online]. 2023-11. [visited on 2023-11-11]. Available from: https://web.archive.org/web/20231111215231/https://www.epicgames.com/help/en-US/rocket-league-c5719357623323/technical-support-c7261971242139/can-i-use-third-party-mods-with-rocket-league-a5720159878043.
61. SANTERI. Using mods in World of Tanks is a security risk. *Web & SEO Designers Forum* [online]. 2019 [visited on 2023-05-11]. Available from: https://web.archive.org/web/20230511211617/https://forum.webseodesigners.com/games-f20/using-mods-in-world-of-tanks-is-a-security-risk-t1604.html.
62. ADÁMEK, Petr. *Security of the Lua Sandbox*. 2022. Available also from: https://web.archive.org/web/20230928214742/https://dspace.cvut.cz/handle/10467/101898. Bache-

lor's Thesis. Czech Technical University in Prague, Faculty of Information Technology.

63. FLORENCE, Paul; MENASSOL, Lucien. *Étude de la surface d'attaque des APIs de modding dans les jeux vidéo* [online]. 2020. [visited on 2023-09-28]. Institut National des Sciences Appliquées de Toulouse. Available from: `https://web.archive.org/web/20230928215401/https://raw.githubusercontent.com/gbip/lua_attack/master/report/report.pdf`.
64. *Frequently Asked Questions (FAQ) | LuaJIT* [online]. 2023-09-15. [visited on 2023-11-23]. Available from: `https://web.archive.org/web/20231122225906/https://luajit.org/extensions.html`.
65. *FFI Library | LuaJIT* [online]. 2023-08-21. [visited on 2023-09-02]. Available from: `https://web.archive.org/web/20230902075825/https://luajit.org/ext_ffi.html`.
66. WALTERBELL. My love-hate relationship with LuaJIT (2015) | Hacker News. *Hacker News* [online]. 2016 [visited on 2023-04-20]. Available from: `https://web.archive.org/web/20230420164301/https://news.ycombinator.com/item?id=12573981`.
67. *Status | LuaJIT* [online]. 2023-08-21. [visited on 2023-11-07]. Available from: `https://web.archive.org/web/20231107191251/https://luajit.org/status.html`.
68. *openresty/luajit2: OpenResty's Branch of LuaJIT 2* [online]. 2023. [visited on 2023-09-28]. Available from: `https://web.archive.org/web/20230928215127/https://github.com/openresty/luajit2`.
69. NORTHLIGHT. How Northlight makes Alan Wake 2 shine [online]. 2023 [visited on 2023-11-09]. Available from: `https://web.archive.org/web/20231109095712/https://www.remedygames.com/article/how-northlight-makes-alan-wake-2-shine`.

70. *future of bytecode verifier | lua-l archive* [online]. 2009-03-04. [visited on 2023-11-09]. Available from: https://web.archive.org/web/20231109122410/http://lua-users.org/lists/lua-l/2009-03/msg00039.html.

71. *dofile | Lua 5.4 Reference Manual* [online]. 2023-05-02. [visited on 2023-11-09]. Available from: https://web.archive.org/web/20231109135431/https://www.lua.org/manual/5.4/manual.html#pdf-dofile.

72. *Exploiting Lua 5.1 on 32-bit Windows.md · GitHub* [online]. 2013-09-26. [visited on 2023-11-09]. Available from: https://web.archive.org/web/20231109124059/https://gist.github.com/corsix/6575486.

73. *Exploiting Lua 5.2 on x64 · GitHub* [online]. 2016-09. [visited on 2023-11-09]. Available from: https://web.archive.org/web/20231109124118/https://gist.github.com/corsix/49d770c7085e4b75f32939c6c076aad6.

74. *Bytecode abuse in Lua 5.2 (-work4) - lua-l archive* [online]. 2010-08-21. [visited on 2023-11-09]. Available from: https://web.archive.org/web/20231109132835/http://lua-users.org/lists/lua-l/2010-08/msg00487.html.

75. *Pwning Lua through 'load'* [online]. 2017-01-01. [visited on 2023-04-09]. Available from: https://web.archive.org/web/20230409072741/https://saelo.github.io/posts/pwning-lua-through-load.html.

76. *Lua 5.4.4 Sandbox Escaping & Type confusion caused by the absence of type check | lua-l archive* [online]. 2021-10-25. [visited on 2023-11-09]. Available from: https://web.archive.org/web/20231109133052/http://lua-users.org/lists/lua-l/2021-10/msg00104.html.

77. *LuaJIT Sandbox Escape: The Saga Ends* [online]. 2022-12-30. [visited on 2023-11-06]. Available from: https://web.archive.org/web/20231106193716/https://0xbigshaq.github.io/2022/12/30/luajit-sandbox-escape/.

78. *LuaJIT/src/lj_load.c at 43d0a19158ceabaa51b0462c1ebc97612b420a2e · LuaJIT/LuaJIT · GitHub* [online]. 2023-08-20. [visited on 2023-11-26]. Available from: https://github.com/LuaJIT/LuaJIT/blob/43d0a19158ceabaa51b0462c1ebc97612b420a2e/src/lj_load.c#L56.

79. *FFI Semantics | LuaJIT* [online]. 2023-08-21. [visited on 2023-11-30]. Available from: https://web.archive.org/web/20231130191255/https://luajit.org/ext_ffi_semantics.html.

80. *kikito/lua-sandbox: A lua sandbox for executing non-trusted code* [online]. 2021-11-04. [visited on 2023-09-28]. Available from: https://web.archive.org/web/20230928211839/https://github.com/kikito/lua-sandbox.

81. *RyanSquared/lua-sandbox: A very quick sandbox module written for Lua 5.3, Linux compatible* [online]. 2017-02-05. [visited on 2020-12-02]. Available from: https://web.archive.org/web/20201202182355/https://github.com/RyanSquared/lua-sandbox.

82. *mozilla-services/lua_sandbox: Generic Lua sandbox for dynamic data analysis* [online]. 2021-04-06. [visited on 2023-11-06]. Available from: https://web.archive.org/web/20231106210800/https://github.com/mozilla-services/lua_sandbox.

83. *LuaJIT/src/lib_io.c at 43d0a19158ceabaa51b0462c1ebc97612b420a2e · LuaJIT/LuaJIT · GitHub* [online]. 2023-08-20. [visited on 2023-11-26]. Available from: https://github.com/LuaJIT/LuaJIT/blob/43d0a19158ceabaa51b0462c1ebc97612b420a2e/src/lib_io.c#L87.

84. *Windows Sandbox | Microsoft Learn* [online]. 2023-05-25. [visited on 2023-09-02]. Available from: https://web.archive.org/web/20230902081621/https://learn.microsoft.com/en-us/windows/security/application-security/application-isolation/windows-sandbox/windows-sandbox-overview.

85. *Frame rate is limited to 30 FPS in remote sessions - Windows Server | Microsoft Learn* [online]. 2023-02-23. [visited on 2023-05-11]. Available from: https://web.archive.org/

web/20230511184613/https://learn.microsoft.com/en-us/troubleshoot/windows-server/remote/frame-rate-limited-to-30-fps.

86. *Sandboxie-Plus | Open Source sandbox-based isolation software* [online]. 2023. [visited on 2023-08-31]. Available from: https://web.archive.org/web/20230831222005/https://sandboxie-plus.com/.

87. [*IMPORTANT*] *Sandboxie Open Source Code is available for download - Sandboxie Forum* (*Read Only*) *- Sandboxie* (*Read Only*) *- Sophos Community* [online]. 2020-04-08. [visited on 2023-06-17]. Available from: https://web.archive.org/web/20230617112707/https://community.sophos.com/sandboxie/f/forum/119641/important-sandboxie-open-source-code-is-available-for-download.

88. ASHCRAFT, Alvin. AppContainer Isolation - Win32 apps. *Microsoft Learn* [online]. 2021 [visited on 2023-05-11]. Available from: https://web.archive.org/web/20230511215044/https://learn.microsoft.com/en-us/windows/win32/secauthz/appcontainer-isolation.

89. *GitHub - microsoft/SandboxSecurityTools: Security testing tools for Windows sandboxing technologies* [online]. 2023-09-08. [visited on 2023-11-06]. Available from: https://web.archive.org/web/20231106220046/https://github.com/microsoft/SandboxSecurityTools.

90. *GitHub - M2Team/Privexec: Run the program with the specified permission level* [online]. 2023-11-29. [visited on 2023-12-12]. Available from: https://web.archive.org/web/20231212171534/https://github.com/M2Team/Privexec.

91. YOSIFOVICH, Pavel. *Windows Internals Seventh Edition, Part 1*. Microsoft Press, 2017.

92. *GetProcessMitigationPolicy function* (*processthreadsapi.h*) *- Win32 apps | Microsoft Learn* [online]. 2022-11-01. [visited on 2023-06-23]. Available from: https://web.archive.org/web/20230623134030/https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getprocessmitigationpolicy.

93. YOSIFOVICH, Pavel. *Windows 10 System Programming, Part 2*. Independently Published, 2021.
94. *Lua: download area* [online]. 2023-05-18. [visited on 2023-12-11]. Available from: https://web.archive.org/web/20231211202421/https://www.lua.org/ftp/.
95. *Releases · luau-lang/luau* [online]. 2023-12-08. [visited on 2023-12-11]. Available from: https://web.archive.org/web/20231211201920/https://github.com/luau-lang/luau/releases.
96. *GitHub - boku7/x64win-DynamicNoNull-WinExec-PopCalc-Shellcode: 64bit WIndows 10 shellcode dat pops dat calc - Dynamic & Null Free* [online]. 2023-03-08. [visited on 2023-12-11]. Available from: https://web.archive.org/web/20231211203325/https://github.com/boku7/x64win-DynamicNoNull-WinExec-PopCalc-Shellcode.
97. *VirtualAlloc function (memoryapi.h) - Win32 apps | Microsoft Learn* [online]. 2022-07-27. [visited on 2023-12-11]. Available from: https://web.archive.org/web/20231211212729/https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc.
98. *GitHub - skeeto/w64devkit: Portable C and C++ Development Kit for x64 (and x86) Windows* [online]. 2023-12-08. [visited on 2023-12-12]. Available from: https://web.archive.org/web/20231212001253/https://github.com/skeeto/w64devkit/tree/master.
99. WESTON, David. Public Preview : Improve Win32 app security via app isolation. *Windows Developer Blog* [online]. 2023 [visited on 2023-12-11]. Available from: https://web.archive.org/web/20231211185700/https://blogs.windows.com/windowsdeveloper/2023/06/14/public-preview-improve-win32-app-security-via-app-isolation/.
100. *Internet Archive: Wayback Machine* [online]. 2023-11-08. [visited on 2023-11-08]. Available from: https://web.archive.org/web/20231108130620/https://archive.org/web/.