DD2525
A dive into WebAssembly Exploitation and Mitigation

Lucas Kristiansson
Marcus Samuelsson

June 24, 2025

Live Demo

A live demonstration of our project is available at:

https://vidarrio.github.io/DD2525/

Contents

1 Introduction

WebAssembly (Wasm) is a binary instruction format designed for secure, high-performance execution in the browser. It provides a universal compilation target for programming languages, including but not limited to Go, C++, and Rust. Wasm promises improved security through sandboxing, but concerns exist about potential vulnerabilities, like memory corruption or side-channel attacks. We plan to investigate the security landscape of Wasm, both in theory and practicality.

The security model of WebAssembly's has two specified goals:

1. Protect users from buggy or malicious modules.
2. Provide developers with useful primitives and mitigations for developing safe applications, within the constraints of (1).

1.1 Project Scope and Limitations

Our project aims to:

1. Analyse WebAssembly's security model and inherent risks.
2. Demonstrate practical exploitation.
3. Evaluate existing mitigation mechanisms.
4. Showcase a few real-world examples of Wasm exploitation.

2 Background

2.1 WebAssembly Concepts

WebAssembly is a low-level assembly-like language with a compact binary format. It has a few key concepts:

- **Values**: The basic data types in WebAssembly include integers (32-bit and 64-bit), floats (32-bit and 64-bit), a 128-bit wide vector type for packed data (of integers or floats), and opaque references that represent pointers to various entities. Unlike the other types, the sizes and layouts of the reference types are not observable.
- **Instructions**: Wasm is based on a stack machine architecture. Code consists of sequences of instructions that are executed in order. Instructions manipulate values on an implicit operand stack and are categorized into two main groups: simple instructions that perform basic operations on data, and control instructions that manage the flow of execution (loops, blocks, etc.).
- **Traps**: A trap is a mechanism for handling errors or exceptional conditions during execution. When a trap occurs, the WebAssembly runtime

stops execution and returns an error code. Traps can be triggered by various conditions, such as division by zero, stack overflow, or invalid memory access.

- **Functions**: Functions are the primary building blocks of WebAssembly modules. Each function takes a set of input parameters, performs computations, and returns a result. Functions can call each other, including recursive calls, and this results in an implicit call stack that cannot be accessed directly.
- **Tables**: A table is an array of opaque values of a particular element type. It allows programs to select values indirectly through a dynamic index operand, enabling emulation of function pointers. Currently, the only supported element type is a function reference or a reference to an external host value.
- **Linear Memory**: Linear memory is a contiguous, mutable array of bytes. It's created with an initial size and can be resized dynamically. A WebAssembly module can access linear memory using load and store instructions, and can read both aligned and unaligned data. A trap occurs if an access is out of bounds of the memory.
- **Modules**: A WebAssembly binary takes the form of a module, which is a collection of functions, tables, and linear memories, as well as global variables. A module can also import and export functions, allowing it to interact with other modules or the host environment. It can also define initialisation data for its linear memory and tables.
- **Embedder**: The embedder is the host environment that executes the WebAssembly module. We say that the embedder embeds the module, and it is responsible for providing the module with the necessary resources and context to execute. The embedder can be a web browser, a server-side runtime, or any other environment that supports WebAssembly. It is responsible for providing imports and handling exports.

2.2 WebAssembly Security Model

The WebAssembly security model is designed to provide a safe execution environment for untrusted code. It achieves this through a combination of techniques:

- **Memory Safety**: Linear memory is isolated from the host environment, preventing direct access to the host's memory. It has a fixed size and can only be accessed through load and store instructions. These instructions perform bounds checking to ensure that memory accesses are within the allocated range. This prevents accessing the host's memory or other modules' memory, but does not prevent accessing other buffers within the same module, so buffer overflows are still possible. The memory is non-executable, meaning that code cannot be executed from memory regions.

- **Control Flow Integrity (CFI)**: WebAssembly uses a control flow graph to ensure that the execution flow of a module follows a valid path. Generally, there are three types of external control flow transitions that need to be protected: direct function calls, indirect function calls, and returns. Direct and indirect calls are referred to as "forward edges" in the control-flow graph, while returns are referred to as "backward edges". During compilation, the compiler generates a control flow graph that describes the valid paths of execution, along with the set of expected call targets for each call site. The runtime checks that the actual call targets match the expected targets, erroring if they do not. This prevents control flow hijacking attacks, such as return-oriented programming (ROP) and jump-oriented programming (JOP). Wasm guarantees the safety of direct calls through the use of explicit function section indexes, and returns through the use of a protected call stack. Indirect calls can only target functions that are in the module's function table. This still lets through code reuse attacks against indirect calls.
- **Sandboxing**: WebAssembly modules run in a sandboxed environment, isolated from the host system. This prevents direct access to system resources, such as files, network sockets, and other sensitive data. The embedder is responsible for providing controlled access to these resources through imports and exports.
- **Deterministic Execution**: WebAssembly is designed to behave the same way across different platforms and environments. This means that the same WebAssembly module will produce the same results regardless of where it is executed. This is achieved through a well-defined specification and a set of runtime requirements.
- **Validation and Compilation**: Before execution, WebAssembly modules are validated to ensure they conform to the specification. This includes checking for well-formedness, type correctness, and control flow integrity. Once validated, modules are compiled into machine code for execution. The compilation process can also include optimisations to improve performance.

2.3 Potential Exploits and Vulnerabilities

Despite WebAssembly's robust security model, several classes of vulnerabilities remain possible within its execution environment:

- **Time-of-Check-to-Time-of-Use (TOCTOU)**: These race condition vulnerabilities can occur when WebAssembly interacts with the host environment, particularly when validation checks and operations occur in separate steps, creating exploitable timing windows.
- **Code Reuse Attacks**: While direct calls and returns are protected, indirect calls remain vulnerable to code reuse techniques. Attackers can

potentially chain together existing functions through the module's function table to execute unintended operations.

- **Side-Channel Attacks**: WebAssembly's deterministic execution model does not protect against timing, cache, or other side-channel attacks that can leak sensitive information without directly accessing it.

Additionally, several implementation-specific security concerns exist:

- **Low-Level Exploitation Impact**: The low-level nature of WebAssembly means that successful Remote Code Execution (RCE) exploits potentially provide greater control and capability to attackers compared to higher-level languages.
- **Debugging Complexity**: The binary format and compilation process can obscure code execution paths, making vulnerability detection and debugging more challenging compared to JavaScript applications.
- **Cross-Language Vulnerabilities**: WebAssembly modules compiled from languages like C++ may inherit memory safety issues from their source language, even when the WebAssembly sandbox itself is functioning correctly.

2.4 Related Work

First of all, a shout-out to the WebAssembly specification, which is a great resource for understanding the language and its security model. The specification is available at https://webassembly.github.io/spec/core/.

In Lehmann et al., 2020, the authors analyse the binary security of WebAssembly and find that many classic vulnerabilities which are no longer exploitable in native code due to common mitigations are completely exposed in WebAssembly. They identify a set of attack primitives that enable writing arbitrary memory, overwriting sensitive data, and triggering unexpected behaviour by diverting control flow or manipulating the host environment. Their empirical risk assessment on real-world binaries shows these attack primitives are likely feasible in practice, revealing a lack of binary security protections in WebAssembly. The authors discuss potential protection mechanisms to mitigate these risks.

Since the paper came out in 2020, the WebAssembly community has made significant progress in addressing these issues, but it was still a good starting point for our research. The authors also mention that the WebAssembly community is aware of these issues and is actively working on improving the security model.

A more modern paper (Perrone et al., 2025) provides a comprehensive review of the WebAssembly security landscape. The authors analyse 121 papers across seven security categories, including security analysis, attack scenarios, use cases in cybersecurity, vulnerability discovery, security enhancements, and attack detection approaches. They note that despite WebAssembly's isolation guaran-

tees, several studies confirm it remains vulnerable to side-channel attacks and memory-based exploits when compiling unsafe code. Their review shows that crypto-mining detection and smart contract vulnerability discovery dominate the current research landscape, while significant gaps remain in analysing WebAssembly applications in other domains like web applications, data visualization, and gaming.

3 Experiments

In this section, we will present our experiments and findings regarding WebAssembly exploitation and mitigation. To conduct our experiments, we compiled unsafe and safe Rust code to WebAssembly and executed it in a browser environment. We used the following tools:

- **Rust**: We used the Rust programming language to write our code. Rust provides a strong type system and memory safety guarantees, and it is a popular choice for compiling to WebAssembly. Rust allows us to write both safe and unsafe code, which is useful for our experiments. We thought it highly relevant, since the course is about language based security, and Rust is a language that has been designed with security in mind, automatically mitigating many common vulnerabilities.
- **wasm-pack**: This is a tool for building and packaging Rust code for WebAssembly. It simplifies the process of compiling Rust to WebAssembly and generating the necessary JavaScript bindings.
- **React**: We used React to create a simple web application that loads and executes our WebAssembly modules. It is the most popular JavaScript library for building user interfaces, and so provides a fitting environment for our experiments.
- **TypeScript**: We used TypeScript to write our JavaScript code. TypeScript is a superset of JavaScript that adds static typing and other features, and is commonly used in modern web development. It allows us to write safer and more maintainable code.
- **Firefox**: We used the Firefox web browser to run our experiments. Firefox has good support for WebAssembly and provides developer tools for debugging and profiling WebAssembly modules. It is one of the most popular browsers, and therefore provides a good general environment for our experiments.

The experiments are categorised under a few different sections, each focusing on a specific aspect of WebAssembly exploitation and mitigation.

3.1 Memory Safety

This section focuses on the memory safety of WebAssembly and how it can be exploited. We will present a few different types of memory safety vulnerabilities, including buffer overflows, heap metadata corruption, and stack overflows. We will also discuss how these vulnerabilities can be exploited in WebAssembly.

3.1.1 Buffer Overflow

A buffer overflow occurs when a program writes more data to a buffer than it can hold, causing adjacent memory to be overwritten. In WebAssembly, while memory accesses are bounds-checked at the linear memory level, there is no protection between different buffers within the same memory region. This means that buffer overflows can still corrupt adjacent data structures, even though the module cannot access memory outside its own linear memory.

As shown in Listing 1, two adjacent buffers are allocated on the stack. The unsafe function allows writing user input starting at a specified position in the first buffer, without bounds checking. If the user input is too long, it will overwrite the contents of the second buffer, demonstrating a buffer overflow. In Rust's safe code, such overflows are prevented by bounds checks, but in unsafe code, these checks are bypassed.

```
#[wasm_bindgen]
pub fn unsafe_copy_user_data(input: &str, position: usize) ->
    String {
    let mut buffer1 = [b'A'; 16];
    let buffer2 = [b'B'; 16];
    unsafe {
        let buffer_ptr = buffer1.as_mut_ptr();
        for (i, byte) in input.bytes().enumerate() {
            *buffer_ptr.add(position + i) = byte;
        }
    }
    let result1 = String::from_utf8_lossy(&buffer1).to_string();
    let result2 = String::from_utf8_lossy(&buffer2).to_string();
    format!("\nBuffer 1: {}\nBuffer 2: {}", result1, result2)
}
```

Listing 1: Unsafe buffer overflow example in Rust

To mitigate this, Listing 2 shows a safe version of the function using Rust's bounds checking to prevent writing past the end of the buffer. This safe version will panic if an out-of-bounds write is attempted, preventing memory corruption. In our experiments, we observed that the unsafe version allows overwriting adjacent stack data, while the safe version reliably prevents such overflows.

```rust
#[wasm_bindgen]
pub fn safe_copy_user_data(input: &str, position: usize) ->
    String {
    let mut buffer1 = [b'A'; 16];
    let buffer2 = [b'B'; 16];
    for (i, byte) in input.bytes().enumerate() {
        buffer1[position + i] = byte;
    }
    let result1 = String::from_utf8_lossy(&buffer1).to_string();
    let result2 = String::from_utf8_lossy(&buffer2).to_string();
    format!("\nBuffer 1: {}\nBuffer 2: {}", result1, result2)
}
```

Listing 2: Safe buffer copy in Rust

3.1.2 Heap Metadata Corruption

Heap metadata corruption occurs when a buffer overflow on the heap overwrites data structures used by the memory allocator, or adjacent heap data. In native code, this can lead to powerful exploits. In Rust, safe code always checks bounds and will panic (trap) if you go out of bounds, preventing memory corruption. However, if you use `unsafe` Rust and bypass bounds checks, you can still demonstrate how writing past a heap buffer can corrupt adjacent data, just like in C/C++. WebAssembly itself does not add extra protection against heap overflows; it is Rust's safety that prevents them in safe code.

Listing 3 demonstrates a heap metadata corruption vulnerability. A single 32-byte heap buffer is allocated, with the first 16 bytes (region 1) filled with 'A' (`\x41`) and the next 16 bytes (region 2) filled with 'B' (`\x42`). These regions are adjacent in memory. If the input is too long or the position is too close to the end of region 1, the write will overflow into region 2, corrupting its contents. This simulates heap metadata or adjacent allocation corruption.

```rust
#[wasm_bindgen]
pub fn unsafe_heap_corruption(input: &str, position: usize) ->
    String {
    let mut buffer = vec![b'A'; 16];
    buffer.extend(vec![b'B'; 16]); // buffer[0..16] = 'A',
    buffer[16..32] = 'B'
    let (region1, region2) = buffer.split_at_mut(16);
    unsafe {
        let ptr = region1.as_mut_ptr();
        for (i, byte) in input.bytes().enumerate() {
            *ptr.add(position + i) = byte;
        }
```

```
    }
    let result1 = String::from_utf8_lossy(region1);
    let result2 = String::from_utf8_lossy(region2);
    format!("\nRegion 1: {}\nRegion 2: {}", result1, result2)
}
```

Listing 3: Unsafe heap metadata corruption in Rust

A safe version of the function, shown in Listing 4, uses Rust's bounds checking to prevent writing past the end of the first region. In our experiments, the unsafe version allowed us to overwrite the contents of region 2, while the safe version reliably prevented any out-of-bounds writes. This demonstrates that, although WebAssembly provides isolation at the module level, memory safety within a module still depends on the safety guarantees of the source language and its usage.

```
#[wasm_bindgen]
pub fn safe_heap_corruption(input: &str, position: usize) ->
    String {
    let mut buffer = vec![b'A'; 16];
    buffer.extend(vec![b'B'; 16]);
    let (region1, region2) = buffer.split_at_mut(16);
    for (i, byte) in input.bytes().enumerate() {
        // This will panic if position + i >= 16
        region1[position + i] = byte;
    }
    let result1 = String::from_utf8_lossy(region1);
    let result2 = String::from_utf8_lossy(region2);
    format!("\nRegion 1: {}\nRegion 2: {}", result1, result2)
}
```

Listing 4: Safe heap corruption prevention in Rust

3.1.3 Stack Overflow

A stack overflow occurs when a program's call stack exceeds its maximum size, typically due to uncontrolled recursion or excessive stack allocation. In WebAssembly, stack overflows are detected by the runtime, which triggers a trap and halts execution. This prevents classic stack smashing attacks seen in native code, but denial-of-service is still possible.

Listing 5 shows a minimal example of a stack overflow in Rust compiled to WebAssembly. The function recursively calls itself without a base case, causing the stack to grow until the WebAssembly runtime detects the overflow and traps:

```
#[wasm_bindgen]
```

```rust
#[allow(unconditional_recursion)]
pub fn stack_overflow() {
    // This will always overflow the stack and trap in WASM
    stack_overflow();
}
```

Listing 5: Stack overflow via infinite recursion in Rust

When this function is invoked, the WebAssembly runtime immediately halts execution upon detecting the stack overflow, raising a trap. Unlike buffer or heap overflows, this does not allow memory corruption or code execution, but it does terminate the module's execution. This demonstrates that, while WebAssembly's runtime enforces stack safety, stack overflows can still be used to cause denial-of-service by crashing the module. Since Wasm and the embedder shares the same stack space, this can also lead to a crash of the embedder, potentially crashing the browser tab. We have intentionally ignored Rusts unconditional recursion warning in this example, which would notify the developer of the potential for infinite recursion.

3.2 Side-channel Attacks

This section will focus on different types of side-channel attacks and how they can be used to gain access to potentially sensitive information. Side-channel attacks differ from other attacks that may attack the program directly; instead, they gather information about the program's execution or leverage overly explicit information provided during interaction.

3.2.1 Storage Attacks

Storage side-channel attacks are fairly easy to both execute and mitigate. These attacks work by analyzing the information provided by the application that may reveal useful data. This could include a message showing that a user already exists during registration, entering a license plate to park and getting the message that it is already parked in that area, etc. Any kind of information that may give the malicious user specific insights can be exploited.

In Listing 6, we examine a login page which either displays "Username does not exist" or "Incorrect password" depending on whether the entered username exists. From this information, an attacker can systematically test common usernames to determine which accounts exist in the system.

```rust
#[wasm_bindgen]
pub fn unsafe_check_credentials_storage(username: &str, password
    : &str) -> String {
    let stored_users = vec!["admin", "user", "guest"];

```

```
    // Key vulnerability: Different error messages leak information
    if !stored_users.contains(&username) {
        return "Error: Username does not exist".to_string();
    }

    // Simplified password logic
    let correct_password = match username {
        "admin" => "admin123",
        // Other cases omitted for brevity
        _ => "",
    };

    if password != correct_password {
        return "Error: Incorrect password".to_string();
    }

    "Login successful".to_string()
}
```

Listing 6: Unsafe storage side-channel: revealing username existence

Mitigating this issue is, as mentioned earlier, fairly simple. Instead of giving specific information when a log-in fails, something more generic can be used instead. For example, instead of having different error messages for incorrect usernames and passwords, it could simply say "Invalid username or password" for both cases. This is the implementation in Listing 7.

```
#[wasm_bindgen]
pub fn safe_check_credentials_storage(username: &str, password: &str) -> String {
    // Same code as unsafe version except:

    // Key security fix: Generic error message doesn't leak information
    if !stored_users.contains(&username) || password != correct_password {
        return "Error: Invalid username or password".to_string();
    }

    "Login successful".to_string()
}
```

Listing 7: Safe storage side-channel: using generic error messages

This issue is not something that WebAssembly could easily mitigate as a general solution, as it depends on the information that developers choose to display to their users. Developers must be vigilant to ensure these vulnerabilities are addressed in their applications.

3.2.2 Timing Attacks

Timing side-channel attacks measure the time for a computer to perform certain operations to gain information about the data that the system is processing. These attacks are often used to extract information about encryption keys, cryptographic algorithms, passwords, or API keys. This process can be challenging in real-world scenarios where numerous factors may influence operation timings. However, a dedicated attacker could use precise measuring tools and techniques to overcome these challenges. The most concerning aspect of timing attacks is their stealth, they often don't directly interact with the application in any suspicious or easily trackable way.

Listing 8 shows an implementation of a login system where each character of the password is checked individually, terminating the comparison as soon as a mismatch is found. This implementation is vulnerable to timing attacks, as an attacker can determine the password character by character by observing that the operation takes longer for each correct character in sequence. The example includes artificially amplified timing differences for demonstration purposes in practice, real timing differences might be in the nanosecond range and harder to measure.

```
#[wasm_bindgen]
pub fn unsafe_check_credentials_timing(username: &str, password: &str) -> TimingResult {
    // Shortened authentication check
    let correct_password = "admin123";

    // Key vulnerability: Early termination creates timing variability
    let mut simulated_time: f64 = 0.85; // Base processing time (milliseconds)
    for i in 0..std::cmp::min(password.len(), correct_password.len()) {
        if password.chars().nth(i) != correct_password.chars().nth(i) {
            break;  // Early termination
        }
        simulated_time += 0.03;
    }

    let is_correct = matching_chars == correct_password.len() &&
                     password.len() == correct_password.len();
```

```
17
18      // Simplified simulation code
19      TimingResult {
20          message: if is_correct { "Login successful" } else { "
    Invalid credentials" },
21          simulated_time_ms: simulated_time,
22      }
23  }
```

Listing 8: Unsafe timing side-channel: early termination in password comparison

Mitigating this vulnerability involves implementing constant-time operations, usually by ensuring that the code path executes the same operations regardless of input validity. In our mitigation shown in Listing 9, we ensure the function always processes all characters, hiding the point where a wrong letter might be entered and normalizing the operation time.

```
1   #[wasm_bindgen]
2   pub fn safe_check_credentials_timing(username: &str, password: &
    str) -> TimingResult {
3       let correct_password = "admin123";
4
5       // Key security fix: Constant-time comparison
6       let mut result = if password.len() == correct_password.len()
     { 1 } else { 0 };
7
8       // Always process all characters - no early termination
9       for i in 0..std::cmp::max(password.len(), correct_password.
    len()) {
10          let pass_char = if i < password.len() { password.chars()
    .nth(i).unwrap() as u32 } else { 0 };
11          let correct_char = if i < correct_password.len() {
    correct_password.chars().nth(i).unwrap() as u32 } else { 0
    };
12          result &= if pass_char == correct_char { 1 } else { 0 };
13      }
14
15      // Simplified simulation code
16      TimingResult {
17          message: if result == 1 { "Login successful" } else { "
    Invalid credentials" },
18          simulated_time_ms: 1.05,  // Always constant time
19      }
20  }
```

Listing 9: Safe timing side-channel: constant-time password comparison

Similar to storage attacks, this is not something that WebAssembly itself could counteract as it would compromise performance and would be impractical to implement as a general-purpose solution. Instead, developers must implement appropriate defences in their own code, such as the constant-time comparison technique shown above.

3.2.3 Cache Timing Attacks

Cache timing attacks work similarly to regular timing attacks (see Section 3.2.2), using time measurements to uncover confidential information. This attack specifically measures the time it takes to access data from memory, distinguishing between fast cache hits and slower cache misses. By analysing these timing differences, attackers can infer which memory locations are accessed frequently, potentially revealing the location of sensitive information like encryption keys.

Our example implementation in Listing 10 simulates different access times for different array locations, mimicking the behaviour of a system where frequently used data (such as a cryptographic key) might be found in the cache.

```
#[wasm_bindgen]
pub fn unsafe_cache_timing(probe_index: u32) ->
    CacheTimingResult {
    let buffer_size = 16;
    let mut lookup_table: Vec<u8> = vec![0; buffer_size];

    // Shortened table initialisation
    lookup_table[12] = 0xE4; // The "secret" key byte at index
    12

    let index = (probe_index as usize) % buffer_size;
    let accessed_value = lookup_table[index];

    // Key vulnerability: Timing differs based on accessed index
    const SECRET_INDEX: usize = 12;
    let simulated_time = if index == SECRET_INDEX {
        0.02  // 20 nanoseconds for cache hit (shown as ms)
    } else {
        0.12 + (index as f64 * 0.005)  // ~120-200 nanoseconds
    for cache miss (shown as ms)
    };

    // Simplified result code
```

```
    CacheTimingResult {
        message: format!("Accessed key byte: 0x{:02X}",
    accessed_value),
        simulated_time_ms: simulated_time,
        access_pattern: vec![0; buffer_size].tap_mut(|v| v[index
    ] = 1),
        memory_value: accessed_value,
    }
}
```

Listing 10: Unsafe cache timing side-channel: variable access times

The mitigation for cache timing attacks is similar to that for general timing attacks. Instead of accessing only the requested memory location, we access all possible locations to ensure consistent timing regardless of which value is being requested. Listing 11 demonstrates this approach.

```
pub fn safe_cache_timing(probe_index: u32) -> CacheTimingResult
    {
    let buffer_size = 16;
    let mut lookup_table: Vec<u8> = vec![0; buffer_size];

    // Shortened table initialisation
    lookup_table[12] = 0xE4; // The "secret" key byte at index
    12

    let index = (probe_index as usize) % buffer_size;

    // Key security fix: Access all elements regardless of
    target
    let mut accessed_value = 0;
    let mut dummy_accumulator = 0u8;

    for i in 0..buffer_size {
        // Access every element to prevent optimization
        let value = lookup_table[i];
        if i == index {
            accessed_value = value;
        } else {
            dummy_accumulator = dummy_accumulator.wrapping_add(
    value);
        }
    }

    // Use the dummy_accumulator in a way that doesn't affect
    the result
```

```
    // but prevents the compiler from optimizing away the accesses
    if dummy_accumulator == 255 {
        accessed_value = accessed_value.wrapping_add(0);
    }

    // Simplified result code
    CacheTimingResult {
        message: format!("Accessed key byte: 0x{:02X}", accessed_value),
        simulated_time_ms: 0.25,  // Always constant time
        access_pattern: vec![1; buffer_size],  // Access pattern shows all accessed
        memory_value: accessed_value,
    }
}
```

Listing 11: Safe cache timing side-channel: constant-time memory access

Implementing specific mitigations in WebAssembly for cache timing attacks would also compromise performance, which is why this responsibility typically falls to developers. They must consider the security-performance trade-off and implement appropriate defences based on their specific application's security requirements.

3.3 Code Injection

Code injection is a critical security vulnerability where attackers insert malicious code fragments through user-controlled inputs such as form fields, URL parameters, or API payloads. In web applications, these attacks primarily target the user's browser, executing malicious code within the victim's browsing context with the privileges of the legitimate website.

3.3.1 WASM-to-DOM Injection

WebAssembly modules cannot directly manipulate the Document Object Model (DOM), which serves as an important security boundary in web applications. This architectural restriction helps prevent unauthorized DOM manipulation and maintains separation between WASM computation and web page rendering.

However, this security boundary can be circumvented when WebAssembly modules generate HTML content that is subsequently rendered by the host JavaScript environment. The vulnerability emerges at the WASM-JavaScript interface when:

- A WASM module processes user input and generates HTML strings containing unsanitised data
- The host application renders this HTML using DOM manipulation methods such as `innerHTML`, React's `dangerouslySetInnerHTML`, or similar APIs
- The rendered content executes malicious scripts or manipulates the DOM in unintended ways

This vulnerability is particularly concerning in applications that leverage WASM for performance-intensive content processing while delegating DOM rendering to JavaScript. Listings 12 and 13 demonstrate how this vulnerability can manifest in practice.

```rust
#[wasm_bindgen]
pub fn vulnerable_render_user_content(user_input: &str) ->
    String {
    // VULNERABILITY: Direct HTML generation with user input
    let html = format!(
        r#"<div class="user-content">
            <h3>User Content:</h3>
            <p>{}</p>
            <div class="metadata">Submitted by user</div>
        </div>"#,
        user_input  // No sanitization - dangerous!
    );

    format!("Generated HTML: {}", html)
}
```

Listing 12: Vulnerable WASM function generating unsanitized HTML content

```javascript
import * as wasm from "wasm-security-test";

const runVulnerableVersion = () => {
   const userInput = '<script>alert("XSS!")</script>';
   const htmlContent = wasm.vulnerable_render_user_content(userInput);

   // DANGER: Direct DOM insertion of WASM-generated HTML
   document.getElementById('content').innerHTML = htmlContent;
   // This executes the XSS payload in the user's browser!
};
```

Listing 13: JavaScript code executing WASM-generated HTML with DOM injection vulnerability

While allowing WASM modules to indirectly influence DOM content through HTML generation introduces security risks, this pattern can be made safe through proper input validation and sanitization. Defence strategies include

implementing Content Security Policy (CSP) headers to restrict script execution, or more fundamentally, sanitizing user input before HTML generation. Listing 14 demonstrates the latter approach with input sanitization performed within the WASM module.

```rust
#[wasm_bindgen]
pub fn secure_render_user_content(user_input: &str) -> String {
    // Sanitize the user input before using it
    let sanitized_input = sanitize_input(user_input);

    // Generate safe HTML with sanitized content
    let html = format!(
        r#"<div class="user-content-safe">
            <h3>Secure User Content:</h3>
            <p>{}</p>
            <div class="metadata">Safely rendered content</div>
        </div>"#,
        sanitized_input  // Properly sanitized!
    );

    format!("Generated safe HTML: {}", html)
}

// Helper function to sanitize user input
fn sanitize_input(input: &str) -> String {
    input
        .replace("&", "&amp;")    // Must be first to avoid double-encoding
        .replace("<", "&lt;")     // Prevent HTML tag injection
        .replace(">", "&gt;")     // Prevent HTML tag injection
        .replace("\"", "&quot;")  // Prevent attribute injection
        .replace("'", "&#x27;")   // Prevent attribute injection
        .replace("/", "&#x2F;")   // Prevent closing tag injection
        .chars()
        .filter(|c| c.is_ascii_graphic() || c.is_ascii_whitespace())
        .collect()
}
```

Listing 14: Secure WASM function with proper input sanitization

This vulnerability pattern illustrates that while WASM itself provides memory safety guarantees, security vulnerabilities can still emerge at the interface between WASM modules and the broader web application environment. Developers must remain vigilant about data flow across these boundaries, particularly

when user-controlled data influences content generation that will be rendered in security-sensitive contexts.

3.3.2 JavaScript Code Generation from WASM

A potentially more dangerous code injection vector emerges when WebAssembly modules dynamically generate JavaScript code that is subsequently executed by the host environment. This pattern occurs in applications where WASM handles computational logic while JavaScript manages DOM interactions, event handling, and API communications that cannot be performed directly from WebAssembly.

The vulnerability manifests when WASM modules accept user input and incorporate it into generated JavaScript functions without proper validation. Unlike the previous WASM-to-DOM injection which targets HTML rendering, this attack vector directly generates executable JavaScript code, potentially granting attackers complete control over the execution environment.

The security risk is amplified because the generated JavaScript executes with the full privileges of the web application's origin, including access to cookies, local storage, and the ability to make authenticated API requests. Listings 15 and 16 demonstrate how this vulnerability can be exploited.

```rust
#[wasm_bindgen]
pub fn vulnerable_generate_javascript(func_name: &str, user_code: &str) -> String {
    // VULNERABILITY: No validation or sanitization of user input
    let js_function = format!(
        "function {}() {{ {} }}",
        func_name, user_code  // Direct injection - extremely dangerous!
    );

    format!("Generated JS: {}", js_function)
}
```

Listing 15: Vulnerable WASM function generating unvalidated JavaScript code

```javascript
import * as wasm from "wasm-security-test";

const executeUserCode = () => {
    const userInput = 'alert("XSS Attack!"); document.location="https://attacker-site.test"';
    const jsCode = wasm.vulnerable_generate_javascript("userFunc", userInput);

    // DANGER: Execute WASM-generated JavaScript code
    const func = new Function(jsCode);
    func(); // This executes the malicious code with full privileges!
```

```
10 };
```

Listing 16: JavaScript code executing WASM-generated functions with code injection vulnerability

The mitigation for this attack is basically the same as for the WASM-to-DOM Injection(see Section 3.3.1), where it could be blocked using a properly setup Content Security Policy (CSP). But, similarly if the functionality is wanted, then the solution would be to use input validation of the code before allowing anything. This is the solution implemented in Listing 17 below.

```
1  #[wasm_bindgen]
2  pub fn secure_generate_javascript(user_function_name: &str,
       user_code: &str) -> String {
3      // Validate function name (alphanumeric only)
4      if !user_function_name.chars().all(|c| c.is_alphanumeric())
       {
5          return "Error: Function name must be alphanumeric only".
       to_string();
6      }
7
8      // Whitelist allowed operations for the user code
9      let allowed_operations = [
10         "return ", "console.log(", "Math.", "parseInt(", "
       parseFloat(",
11         "typeof ", "length", "+", "-", "*", "/", "(", ")", " ",
       ";"
12     ];
13
14     // Check if user code contains only allowed operations
15     let safe_code = if allowed_operations.iter().any(|&op|
       user_code.contains(op))
16         && !user_code.contains("alert")
17         && !user_code.contains("document")
18         && !user_code.contains("window")
19         && !user_code.contains("eval") {
20         user_code.to_string()
21     } else {
22         "console.log('Code blocked for security')".to_string()
23     };
24
25     let javascript = format!(
26         r#"function {}() {{
27             console.log("Safe user function executing...");
28             {};
29             return "Function executed safely";
30         }}"#,
```

```
        user_function_name, safe_code
    );

    format!("Generated safe JavaScript: {}", javascript)
}
```

Listing 17: Secure WASM function with comprehensive input validation for JavaScript generation

This vulnerability demonstrates another security challenge that WebAssembly cannot inherently mitigate. WASM modules cannot directly execute JavaScript, they can only generate strings that the host application chooses to execute. Therefore, preventing code injection through JavaScript generation remains the developer's responsibility when implementing such functionality. This vulnerability typically emerges from either deliberate design choices requiring dynamic code execution or from developers inadvertently creating unsafe interfaces without understanding the security implications.

3.4 Sandbox Escapes

WebAssembly's security model is fundamentally built on sandboxing, meaning modules are designed to be isolated from the host environment and can only interact with it through well-defined interfaces. However, this security boundary can be compromised when host applications inadvertently expose dangerous functionality to WebAssembly modules. These sandbox escape vulnerabilities represent some of the most critical security risks in WebAssembly deployments, as they allow modules to break out of their intended isolation and access sensitive host resources.

3.4.1 Dangerous Host Function Exposure

The most direct form of sandbox escape occurs when host applications expose dangerous native functions to WebAssembly modules through the import mechanism. This vulnerability arises from the fundamental design of WebAssembly's host interface: modules can only call functions that the host explicitly provides through imports, but if the host mistakenly exposes sensitive operations, malicious modules can exploit these interfaces to escape their sandbox.

Common dangerous functions that should never be exposed to untrusted WebAssembly modules include:

- `eval()` and similar code execution functions
- File system access functions (`read`, `write`, `delete`)
- Network access functions (`fetch`, `XMLHttpRequest`)
- Browser storage manipulation (`localStorage`, `sessionStorage`)

- Operating system command execution interfaces
- Memory management functions outside the WASM linear memory

Listing 18 demonstrates how a WebAssembly module can exploit dangerous host function exposure to execute arbitrary JavaScript code, make unauthorized network requests, and manipulate browser storage.

```rust
use wasm_bindgen::prelude::*;

// External JavaScript functions that we'll import (dangerous!)
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = window)]
    fn eval(code: &str) -> JsValue;

    #[wasm_bindgen(js_name = "dangerousFetch")]
    fn dangerous_fetch(url: &str) -> JsValue;

    #[wasm_bindgen(js_name = "dangerousStorageWrite")]
    fn dangerous_storage_write(key: &str, value: &str) ->
    JsValue;
}

// UNSAFE: WASM gets access to dangerous host functions
#[wasm_bindgen]
pub fn unsafe_eval_access(code: &str) -> String {
    // Execute arbitrary JavaScript code!
    match std::panic::catch_unwind(|| {
        let result = eval(code);
        format!("EXECUTED: JavaScript code '{}' returned: {:?}",
     code, result)
    }) {
        Ok(success) => success,
        Err(_) => format!("ERROR: JavaScript code '{}' caused an
     error", code)
    }
}

#[wasm_bindgen]
pub fn unsafe_fetch_access(url: &str) -> String {
    // Make unauthorized network requests!
    match std::panic::catch_unwind(|| {
        let result = dangerous_fetch(url);
        format!("EXECUTED: Network request to '{}' initiated:
    {:?}", url, result)
    }) {
```

```
        Ok(success) => success,
        Err(_) => format!("ERROR: Network request to '{}' failed", url)
    }
}

#[wasm_bindgen]
pub fn unsafe_storage_access(key: &str, value: &str) -> String {
    // Write to browser storage!
    match std::panic::catch_unwind(|| {
        let _write_result = dangerous_storage_write(key, value);
        format!("EXECUTED: Wrote '{}' = '{}' to storage", key, value)
    }) {
        Ok(success) => success,
        Err(_) => format!("ERROR: Storage access for key '{}' failed", key)
    }
}
```

Listing 18: Dangerous host function exposure allowing sandbox escape

The secure approach involves providing only safe, restricted host functions to WebAssembly modules. Instead of exposing raw system capabilities, host applications should implement carefully designed APIs that validate all inputs, enforce access controls, and limit the scope of operations. Preventing dangerous host function exposure requires careful architectural planning during the design phase. Host applications should follow the principle of least privilege, exposing only the minimum necessary functionality to WebAssembly modules. Additionally, all exposed functions should implement comprehensive input validation, rate limiting, and access controls to prevent abuse even if they are legitimately needed.

3.5 Control Flow Integrity Bypasses

Control Flow Integrity (CFI) is a critical security mechanism designed to prevent code-reuse attacks by ensuring that program execution follows legitimate control flow paths. WebAssembly implements several CFI mechanisms, including call and return address validation, type checking for indirect calls, and bounds checking for function tables. However, several attack vectors can potentially bypass these protections, particularly when WebAssembly modules interact with host environments or when module boundaries are not properly enforced.

3.5.1 Host Export Exposure

One of the most straightforward CFI bypass techniques involves exploiting improperly exposed host exports. When WebAssembly modules export functions intended for specific use cases, but the host environment makes these exports accessible in unintended contexts, attackers can call sensitive functions directly rather than through their intended control flow paths.

This vulnerability is particularly concerning because it circumvents the module's internal access controls and intended API design. Malicious code can invoke exported functions with arbitrary parameters, potentially triggering unintended behavior or accessing sensitive functionality without proper authorization checks.

To prevent host export exposure vulnerabilities, developers should carefully audit which functions are marked for export from WebAssembly modules. Internal helper functions, administrative operations, and any function that performs privileged operations should not be exported unless absolutely necessary. When functions must be exported, they should implement comprehensive input validation and authorization checks at the function entry point, treating all callers as potentially malicious. Additionally, modules should use a facade pattern where only a minimal set of carefully designed public functions are exported, while sensitive operations remain encapsulated within internal, non-exported functions that can only be accessed through controlled pathways.

3.5.2 Indirect Call Logic Flaws

WebAssembly's indirect call mechanism uses function tables to enable dynamic function dispatch. While this provides flexibility, it also creates opportunities for CFI bypasses when the logic controlling these indirect calls contains flaws. Attackers can potentially manipulate table indices or exploit validation logic to call unintended functions.

Listing 19 demonstrates a vulnerable implementation where insufficient bounds checking allows attackers to access functions beyond the intended scope.

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn safe_function() -> i32 {
    42
}

#[wasm_bindgen]
pub fn secret_function() -> i32 {
    1337  // Sensitive value that should not be accessible
}

```

```rust
#[wasm_bindgen]
pub fn call_by_index(index: usize) -> i32 {
    // VULNERABILITY: Exposes all functions in table, including sensitive ones
    let table: [fn() -> i32; 2] = [safe_function, secret_function];
    if index < table.len() {
        table[index]()  // Attacker can call index 1 to access secret_function
    } else {
        -1 // Invalid index
    }
}
```

Listing 19: Vulnerable indirect call implementation allowing CFI bypass

The secure implementation explicitly controls which functions can be called through indirect mechanisms, implementing proper access controls and validation. Listing 20 shows this approach.

```rust
#[wasm_bindgen]
pub fn call_by_index_cfi(index: usize) -> i32 {
    // SECURE: Only expose approved functions through indirect calls
    match index {
        0 => safe_function(),
        // secret_function is NOT exposed through this interface
        _ => -1,  // Explicitly reject all other indices
    }
}
```

Listing 20: Secure indirect call implementation with proper CFI protection

3.5.3 Function Table Manipulation

WebAssembly's function tables can be exported and modified by host environments, creating another vector for CFI bypasses. When modules export their function tables, malicious host code can potentially modify these tables to redirect function calls to unintended targets. This attack vector is particularly dangerous because it can redirect control flow within the WebAssembly module itself, potentially bypassing internal security checks.

The key insight is that exported tables represent a shared resource between the module and host, and modifications to these tables can have far-reaching effects on the module's internal control flow. Attackers with access to exported tables can essentially perform a form of return-oriented programming (ROP) or jump-oriented programming (JOP) within the WebAssembly environment.

Defending against function table manipulation requires careful consideration of what should be exported. Modules should avoid exporting function tables unless absolutely necessary, and when they must be exported, the host environment should implement strict access controls and validation before allowing any modifications.

Additionally, modules can implement runtime integrity checks to detect unauthorized modifications to their function tables, though this adds performance overhead and complexity to the implementation.

3.6 Time-of-Check-to-Time-of-Use

Time-of-Check-to-Time-of-Use (TOCTOU) vulnerability is mentioned in the WebAssembly documentation as a potential security risk (webassembly.org, 2025). The TOCTOU vulnerability occurs when there is a timing gap between validating access permissions (time-of-check) and actually accessing the resource (time-of-use), creating a race condition window that attackers can exploit (Prakash, 2024).

The attack scenario involves an unauthorized user attempting to access a protected resource. Initially, the permission check correctly denies access, but during the delay between the permission validation and the actual resource access, an attacker can modify the permission cache to grant access. This race condition allows the attacker to bypass the original security decision. Listing 21 demonstrates this vulnerability, where a user attempts to access a file that should be outside their permissions.

In this demonstration, we artificially create a delay to make the race condition easier to exploit and understand. In real-world applications, these timing gaps occur naturally from normal operations such as network I/O, database queries, file system operations, or operating system thread scheduling. These natural delays are typically measured in microseconds or milliseconds, but are sufficient for sophisticated automated attacks that can attempt the race condition thousands of times per second.

```
#[wasm_bindgen]
pub fn vulnerable_file_access(filename: &str, user_id: &str) ->
    String {
    // TIME OF CHECK: Validate permission
    let key = format!("{}:{}", user_id, filename);
    let has_permission = permission_cache()
        .lock()
        .unwrap()
        .get(&key)
        .copied()
        .unwrap_or_else(|| has_default_permission(filename,
    user_id));

```

```
    if !has_permission {
        return format!("ACCESS DENIED: User {} cannot access {}", user_id, filename);
    }

    // VULNERABILITY: Delay creates TOCTOU window
    for i in 0..1000 { let _ = i; }

    // TIME OF USE: Access file (vulnerability window!)
    match file_system().lock().unwrap().get(filename) {
        Some(content) => format!("SUCCESS: Read file '{}': {}", filename, content),
        None => format!("ERROR: File '{}' not found", filename)
    }
}

// EXPLOIT: Modify permissions during TOCTOU window
#[wasm_bindgen]
pub fn modify_permission_during_delay(user_id: &str, filename: &str, grant_access: bool) -> String {
    let key = format!("{}:{}", user_id, filename);
    permission_cache().lock().unwrap().insert(key.clone(), grant_access);
    format!("RACE CONDITION: Modified permission for '{}' to {}", key, grant_access)
}
```

Listing 21: Vulnerable TOCTOU implementation with exploitable race condition window

The mitigation for TOCTOU vulnerabilities, shown in Listing 22, employs an atomic check-and-use operation. Instead of relying on a modifiable permission cache, the secure implementation uses immutable permission rules that cannot be altered at runtime. Crucially, the permission check and resource access occur within a single mutex lock scope, ensuring that no external code can modify the system state between the security decision and resource access. This atomic operation eliminates the race condition window entirely.

```
#[wasm_bindgen]
pub fn secure_file_access(filename: &str, user_id: &str) -> String {
    let files = file_system().lock().unwrap();

    // Check permission at the exact moment of use (atomic operation)
    if !has_default_permission(filename, user_id) {
```

```
7           return format!("ACCESS DENIED: User {} cannot access {}"
        , user_id, filename);
8       }
9
10      // Immediately use the resource while holding the same lock
11      match files.get(filename) {
12          Some(content) => format!("SECURE SUCCESS: Read file
        '{}': {}", filename, content),
13          None => format!("ERROR: File '{}' not found", filename)
14      }
15  }
16
17  // Immutable permission rules (cannot be modified at runtime)
18  fn has_default_permission(filename: &str, user_id: &str) -> bool
         {
19      match filename {
20          "public.txt" => true,
21          "config.txt" | "user_data.txt" => user_id == "admin",
22          _ => false
23      }
24  }
```

Listing 22: Secure TOCTOU mitigation using atomic check-and-use operation

In the context of WebAssembly applications, TOCTOU vulnerabilities are particularly relevant because WASM modules often interact with host environments through foreign function interfaces, creating natural timing gaps. When WASM code calls JavaScript APIs, accesses browser storage, or communicates with servers, these operations introduce delays that can be exploited. Additionally, WASM's sandboxed execution model means that security decisions often involve cross-boundary validation, where the timing between permission checks in the host environment and resource access in the WASM module can create race condition opportunities. Proper mitigation requires careful design of the WASM-host interface to ensure atomic security operations.

4 Real world examples

In the following section, we point to a couple real-life examples of Wasm vulnerabilities connected to concepts mentioned earlier.

4.1 Code Injection via Export Names in Node.js (CVE-2023-39333)

In 2023, a critical code injection vulnerability was discovered in Node.js's support for experimental WebAssembly modules. When Node.js is launched with

the `-experimental-wasm-modules` flag, it allows importing WebAssembly modules as ECMAScript modules. The vulnerability arises from insufficient sanitisation of export names: specially crafted names in a malicious WebAssembly module could be interpreted as valid JavaScript identifiers and injected directly into the runtime namespace (Node.js Security Team, 2023).

This leads to a violation of the Wasm security model, where sandboxed modules are assumed to have limited interaction with host functionality. In practice, a malicious module could inject arbitrary JavaScript code and potentially access resources outside its intended boundary. This case demonstrates how WebAssembly's safety guarantees can be undermined by flawed integrations in host environments.

4.2 TOCTOU in Wasmtime (CVE-2024-47813)

In 2024, researchers identified a critical race condition in the `wasmtime` WebAssembly runtime. The vulnerability was rooted in the engine's internal type registry. Specifically, when multiple threads concurrently created and dropped type instances (e.g., `FuncType` or `ArrayType`) on the same `wasmtime::Engine` object, a double unregistration of types could occur (Snyk Security Team, 2024). This situation exemplifies a Time-of-Check to Time-of-Use (TOCTOU) issue, where the state verification becomes stale due to concurrent modifications, leading to corruption of internal registry structures. Although the guest WebAssembly code remains sandboxed, the flaw underscores the challenges of ensuring thread safety in complex host environments.

5 Conclusion

Our investigation into WebAssembly's security landscape reveals a complex balance of architectural security guarantees alongside implementation-dependent vulnerabilities. WebAssembly employs techniques such as sandboxing, memory isolation, and control flow integrity to combat many types of security concerns, but our demonstrations reveal that significant challenges remain across multiple attack vectors.

WebAssembly employs module-level memory isolation, but vulnerabilities within modules remain possible, especially when unsafe code practices are employed. This is especially common in memory-unsafe languages such as C/C++, or like we used, unsafe Rust. This shows that while WebAssembly's security boundaries prevent cross-module exploits, we still must employ good coding practices within each module. Using a memory safe language like Rust (non-unsafe Rust that is) can be a great way to inherently prevent these vulnerabilities.

Another security angle is that of side-channel attacks. WebAssembly doesn't do anything to inherently protect against these types of attacks. We demonstrate storage, timing, and cache timing attacks in our demos, illustrating how

sensitive information can be extracted through analysis of execution and response patterns. For the cache timing attacks especially, one needs to be aware of compiler optimizations to avoid revealing information even when trying to implement non-information revealing code.

Perhaps the most critical security risks we identified are sandbox escape vulnerabilities, which directly undermine WebAssembly's fundamental security model. Our demonstrations show how dangerous host function exposure can allow malicious modules to break out of their intended isolation and access sensitive host resources like eval(), fetch(), and storage APIs. These vulnerabilities highlight the critical importance of carefully designing the host-WASM interface and following the principle of least privilege when exposing functionality to WebAssembly modules.

Control flow integrity bypasses represent another significant threat vector that can circumvent WebAssembly's built-in CFI protections. We documented three main attack patterns: host export exposure, indirect call logic flaws, and function table manipulation. These attacks demonstrate that even with WebAssembly's architectural CFI mechanisms, developers must implement additional access controls and validation to prevent unauthorized function calls and control flow redirection.

Our code injection attacks demonstrate how WASM-generated content can be unsafely executed in the host environment if care is not taken to validate and sanitize user inputs. This turns what is supposed to be a safe sandbox environment into a launching point for attacks against the host environment. Whether through DOM manipulation or dynamic JavaScript generation, these vulnerabilities emphasize the need for comprehensive input validation and Content Security Policy implementations.

In addition to this, we also investigate how concurrency vulnerabilities can manifest in the form of TOCTOU attacks, supported by the real-world example CVE-2024-47813. Race conditions must be thought about even in the WASM sandbox, and considering your host environment is essential for this. Using concurrency mechanisms such as locks, mutexes, and semaphores are some ways to handle these types of vulnerabilities.

WebAssembly is a great way to get performant, cross-platform modules into many types of environments. For the web, it provides a great sandbox architecture which prevents a multitude of problematic weaknesses inherent to browsers. However, for successful use, developers must take responsibility for careful implementation of both WASM and embedder side code, actively thinking about the different types of exploits they might want to prevent. Our live demonstration at https://vidarrio.github.io/DD2525/ provides hands-on examples for some of these vulnerabilities and their mitigations, hopefully making some developers more aware of how they manifest and how they themselves can think about mitigating weaknesses in their future work.

6 Contribution

We worked together on finding different vulnerabilities while communicating over discord. The report was then split up such that each person wrote about their respective exploits, while the general parts where written together. The coding was done together on the common parts of the website, and individually for the exploits that each person found. Both of us made sure to understand all the vulnerabilities.

6.1 Disclaimer

A Large Language Model was employed for help with website styling. All security implementations and explanatory texts were written by us.

References

Lehmann, Daniel, Johannes Kinder, and Michael Pradel (Aug. 2020). "Everything Old is New Again: Binary Security of WebAssembly". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, pp. 217–234. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

Node.js Security Team (2023). *CVE-2023-39333: Code Injection via WebAssembly Export Names in Node.js*. https://nvd.nist.gov/vuln/detail/CVE-2023-39333. Accessed: 2025-05-28. URL: https://nvd.nist.gov/vuln/detail/CVE-2023-39333.

Perrone, Gaetano and Simon Pietro Romano (2025). "WebAssembly and security: A review". In: *Computer Science Review* 56, p. 100728. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2025.100728. URL: https://www.sciencedirect.com/science/article/pii/S157401372500005X.

Prakash, Arun (2024). *Understanding TOCTOU Vulnerability: A Timeless Security Risk*. https://arunprakashpj.medium.com/understanding-toctou-vulnerability-a-timeless-security-risk-c2be4fb10b3b. Accessed: June 24, 2025.

Snyk Security Team (2024). *CVE-2024-47813: Time-of-check to time-of-use (TOCTOU) in wasmtime*. https://security.snyk.io/vuln/SNYK-DEBIANUNSTABLE-RUSTWASMTIME-10004904. Accessed: 2025-05-28. URL: https://security.snyk.io/vuln/SNYK-DEBIANUNSTABLE-RUSTWASMTIME-10004904.

webassembly.org (2025). *Security*. https://webassembly.org/docs/security/. [Online; accessed 17-05-25].

A Source Code Repository

The complete source code for this project, including all WebAssembly security implementations and the interactive website, is available under the project directory at:

https://github.com/vidarrio/DD2525