Abstract:

Web Assembly is a new technology that allows web developers to run native C/C++ code on a webpage with high performance. It is an open, industry-wide effort to bring a safe, efficient assembly language to the web. This document provides the best practices and security considerations for developers who wish to integrate WebAssembly into their product.

Some of the memory associated bugs and exploitation techniques like stack smashing, ROP etc are obviated in WebAssembly programs. Although, the attacker may not perform direct code injection attacks, it is possible to hijack the control flow of a module using code reuse attacks against indirect calls.

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).

Format String bugs:

Format string bugs are a class of bugs that is normally associated with the code written in C and are not typically found in web applications. By default, Emscripten’s printf print to the JavaScript console. 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.

Emscripten’s printf does support the “%n” format type, which allows attackers to write, instead of just reading. Nowadays, we get only information disclosure issue because many C compilers disallow the “%n” modifier because of its security implications. The leaked memory resides insides the memory of the wasm application instance. If attacker can leak the memory from the wasm instance, then the sensitive information like password may be revealed.

void EMSCRIPTEN\_KEEPALIVE format\_string\_bug(char \*str) {

 char secret\_password[] = "MyP@ssw0rd!!";

 printf("you entered = ");

 printf(str);

 printf("\n");

}

Let’s enter the string “%x%x%x%x %x%x%x %x%x%x%x%x%x%x%x%x%x%%x”. This causes the memory leakage. To avoid this use printf(“%s”, str) instead of printf(str).

Integer Overflows/underflows

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.However, a more likely scenario would be the case where an integer overflow is leveraged to exploit a buffer overflow.

Stack Based Buffer Overflows:

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.

EM\_JS(void,overflowAlert ,(),{

 alert("overflow");

 });

 int main() {

 char bof0[] = "abc";

 char bof1[] = "123";

 strcpy(bof1,"BBBBBBB");

 if(strcmp(bof0,"abc"))

 overflowAlert();

 return 0;

 }

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.

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 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 dlmalloc implementation that performs the basic UNLINK(), has entirely deterministic allocation locations, and does not contain any serious non-assert()-based validation mechanisms. The current, dlmalloc-based heap implementation should be replaced with a security-hardened one such as PartitionAlloc.

Indirect function calls:

Emscripten provides a C/C++ API that offers 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 E M\_ASM() for calling JavaScript from C or C++. Cross-site scripting obviates the security of any WebAssembly program operating within the affected JavaScript context. So we need to ensure that these macros and functions are not misused.

Let us first observe emscripten\_run\_script() . If the attacker can control the string passed to emscripten\_run\_script(), they can conduct a cross-site scripting attack. However, this is not only the way to exploit a function. Function pointers can be used to exploit. If the attacker can overwrite a function pointer which is used to access a function with matching signature of target javascript interoperability function and is able to control the parameter, then they can achieve XSS by invoking emscripten\_run\_script() instead of target function. This attack occurs when there is a buffer overflow. This is because emscripten\_run\_script() is not a safer function and attacker makes use of it .

There are several functions similar 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)

Emscripten provides several methods of calling javascript from C/C++. The recommended way to use “inline Javascript” with the EM\_ASM family of macros available in emscripten.h .

Consider the following:

1 . int main() {

emscripten\_run\_script("alert('Hello, world!');");

return 0;

}

Javascript file is:

function \_emscripten\_run\_script(ptr) {

eval(Pointer\_stringify(ptr));

}

1. int main() {

((void)emscripten\_asm\_const\_int("alert('Hello, world!');" ));

return 0;

}

Javascript file is:

var ASM\_CONSTS = [function() { alert('Hello, world!'); }];

function \_emscripten\_asm\_const\_i(code) {

return ASM\_CONSTS[code]();

}

emscripten\_asm\_const\_int belongs to the family of EM\_ASM and lives in the header em\_asm.h, a header which is included in emscripten.h. em\_asm contains extensive macro logic used to determine the required function signature . Emscripten creates functions in its output JavaScript file that include the inline code instead of using eval(). These functions use a naming scheme indicating the signature.

Using the family functions of inline em\_asm provides the safer construct than 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 system calls provided by the Emscripten toolchain are:

• \_\_syscall6: close

• \_\_syscall54: ioctl

• \_\_syscall140: llseek

• \_\_syscall146: writev

These system calls do not allow the direct execution of JavaScript through eval() or through editing DOM via methods like document.write() or calling element’s innerHTML() method. However, the system call implementation for writev() can invoke the function mapped to Emscripten’s Module['print'], which can be overridden in an unsafe manner. Misuse of syscall146 or the other system calls available by default can lead to security problems, but these functions aren’t easy access routes to arbitrary JavaScript execution.

Use Clang’s CFI

When compiling, using Clang’s Control Integrity flag ( -fsanitize=cfi) can prevent some of the function pointer manipulation issues.

Time-of-check-to-time of-use attacks(TOCTTOU):

Time-of-check-to-time-of-use is a file based “race condition” that occurs when a resource is checked for a particular value, such as whether a file exists or not, and that value then changes before the resource is used, invalidating the results of the check.

Errors can occur when the status changes unexpectedly, either maliciously or unintentionally, between a check and a subsequent operation. TOCTTOU attacks can lead to unauthorized access to resources, such as read and write. This sort of attack is difficult to detect. TOCTTOU race conditions are most common in UNIX file systems.

The main cause of these vulnerabilities is the lack of control in an operating system’s file-system API ad so it’s not that much easy to resolve. Most OSes change the order that instructions and processes are actually executed to improve efficiency. A programmer has to achieve atomicity of two operations using an API that isn’t designed for such process. The challenge is therefore ensuring the file system state, can not change between two system calls.

As a preventive step, most UNIX systems have adopted variants of common file system calls that operate on file handles rather than file names. These calls end in “at” like openat and statat because file handles are a private mapping to a file, they can not be changed by another program and so are not subject to race conditions with other applications. Microsoft added transaction support (TxF) to their TFS file system. Within a transaction, all updates are kept isolated until committed, when they are atomically published to the rest of the system.

Attacker may inject malicious code like:

|  |  |
| --- | --- |
| Victim | Attacker |
| **if** (access("file", W\_OK) != 0) {  exit(1);  }  fd = open("file", O\_WRONLY);  *// Actually writing over /etc/passwd*  write(fd, buffer, **sizeof**(buffer)); | *//*  *//*  *// After the access check*  symlink("/etc/passwd", "file");  *// Before the open, "file" points to the password database*  *//*  *//* |

In this example, an attacker can exploit the race condition between the access and open to trick the victim into overwriting an entry in the system password database. Although this sequence of events requires precise timing, it is possible for an attacker to arrange such conditions without too much difficulty.

Preventive steps:

 The general technique is to use exception handling instead of checking In this case there is no check, and failure of assumptions to hold are detected at use time, by an exception.

File locking is a common technique for preventing race conditions for a single file, but it does not extend to the file system namespace and other metadata, nor does locking work well with networked filesystems, and cannot prevent TOCTOU race conditions. File locking can’t automatically roll back a failed operation, which requires transaction support by the OS.

Side Channel Attacks (SCAs):

In this scenario, the attacker attacks based on the information gained from the implementation of computer system rather than the software bugs. They work by monitoring the emissions produced by electric circuits when the victim’s computer is being used. SCAs aim at extracting secrets from a chip or system through measurement and analysis of physical parameters like supply current, execution time. Side channel attacks include spying on the power consumption of an electronic device to steal encryption key or acoustic attacks that record the sound of a user’s key strokes to steal their passwords.

“Meltdown and Spectre CPU vulnerabilities ” are the timing attacks which are possible in Wasm. They rely on the attacker’s ability to measure precise time intervals. The attacker utilizes the internal native functions for measuring the time intervals such as “SharedArrayBuffer” and performance.now().

The mitigations to prevent the vulnerabilities from being exploited via maliciously-crafted web pages, such as JavaScript-based proof-of-concept was developed by researchers that could read the memory of the host browser process. One of the mitigations added was disabling or reducing the precision of time counters in browsers. For example, Mozilla reduced the precision nof performance.now() to 20 microseconds as of Firefox 57 and further to the default of 2 milliseconds in Firefox 59. The new Web Assembly standard poses a threat to these mitigations because Web Assembly has a support for threads and shared memory, very accurate timers can be created. Chrome disabled SharedArrayBuffer in Chrome 63 but now re-enabled in Chrome versions where Site Isolation is on by default.

The impact of successfully exploiting Spectre can be greatly reduced by preventing sensitive data from ever sharing a process with attacker-controlled code. The Chrome team has been working on a feature to achieve this called “Site Isolation”. Site Isolation ensures that pages from different websites are always put into different processes, each running in a sandbox limiting what the process is allowed to do. Where possible, prevent cookies from entering the renderer process' memory by using the SameSite and HTTPonly cookie attributes, and by avoiding reading from document.cookie.

Optimization

Enabling the optimizer can remove some of Emscripten’s built-in functions that can be used for exploits involving function pointer manipulation.