WebAssembly Security

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 attacks

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.

If attacker can leak the memory from the wasm instance, then the sensitive information like password may be revealed.

If we use “%n”, then error occurs (*uncaught exception: Runtime error: The application has corrupted its heap memory area (address zero)!)*

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).

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.

Checkings need to be implemented if we are uaing strcpy .

Buffer overread via Integer overflow

Integer overflows can exist in both higher-level languages such Java and in lower-level languages such as C. Due to bounds-checking verifications built-in to higher-level languages, the existence of an integer overflow issue will typically not give an attacker the opportunity to read data that he should not have access to. However, with Wasm the situation is different. The implications of an integer overflow in a Wasm application written in C may be greater than if the same program would have been written in a higherlevel language. Depending on how the program is implemented, an integer overflow in a Wasm application may give the opportunity to read data that is adjacent in memory [4], that the user should not have access to. The following Wasm program that has an integer overflow vulnerability that allows a malicious user to leak memory from adjacent variables .

void EMSCRIPTEN\_KEEPALIVE buffer\_overread(int start\_pos, int end\_pos) {

char buf[200];

char secret\_password[256] = "Mypass@word";

char msg[256] = "Hi, hello everyone";

unsigned char e = end\_pos;

if (e > strlen(msg)) {

printf("Do not try to read past the end.\n"); }

else { snprintf(buf,(end\_pos - start\_pos) + 1,"%s",&msg[start\_pos]);

printf("Contents: %s\n",buf); }

}

The string in variable ‘msg’ has a length of 32 bytes, and a check is applied to ensure that the end position does not go past that. However, due to an integer overflow when casting datatypes, entering 256 will flow over to zero. This allows us to read the value of the variable ‘secret\_password’ in an adjacent variable, by asking the program to print out the characters between positions 256 and 270.

Use After Free(Similar to C/C++. )

UAF vulnerabilities can be used to transfer execution to the beginning of another function in Wasm, but not to arbitrary locations in the code.

Race Conditions

Arise due to SharedArrayBuffer. Can be mitigated by using atomics and mutex locks. Occurs when we are using pthreads where multiple threads were accessing and modifying a global variable.

Hijacking the control flow

The attacker tricks the program to call a method on instance of wrong type. If the program reads an instance from malicious data source, it does not check whether the instance is appropriate or not and calls some object method on it. By taking this as an advantage, the attacker controls the program.

Avoid emscripten\_run\_script:

WebAssembly modules can interact with their host environment in various ways to cause externally visible effects. One such way is to invoke the notorious eval function of a JavaScript host environment, which interprets a given string as code. To access eval, WebAssembly modules compiled via Emscripten can use, e.g., emscripten\_run\_script, which executes JavaScript code in the host environment, both in browsers. In browsers, any function that allows to add code to the document (e.g., document.write) can serve as an eval-equivalent for constructing exploits. An attacker may inject malicious code by overwriting the argument passed to an eval-like function. For example, suppose a WebAssembly usually invokes eval with a “constant” string of code stored in linear memory, then an attacker could overwrite that constant with malicious code. This may lead to cross site scripting(XSS).

So, to avoid this use emscripten\_asm family functions. The construction comparision of both is given below .

emscripten\_asm family functions have safe construction than emscripten\_run\_script family functions.

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]();

}

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.