This has to be used only as a reference for your research. This work is under review and it should not be shared/published elsewhere.

MOHAMMAD GHAFARI

ACM Reference Format:
Mohammad Ghafari. 2022. This has to be used only as a reference for your research. This work is under review and it should not be shared/published elsewhere. . 1, 1 (May 2022), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

EXPLANATION OF SYMPTOMS

Environment: Access to Environment Variables. The following excerpt accesses the `PATH` environment variable.

```
printf("%s\n", getenv("PATH"));
```

In native code, this accesses a variable that is bound on Unix-like systems. However, in WebAssembly, the execution environment is cleared upon execution of a program. With the platform we are using to execute the test cases, it is required to explicitly bind these environment variables before executing this program. In case this is not performed, the output of this excerpt differs between native code and WebAssembly (where the empty string is printed).

Heuristic
Structural. Check for calls to `getenv`.

CWE 404: Improper Resource Shutdown. The following example illustrates CWE 404. A file is opened with the `open` system call. This returns a file descriptor as an `int`. The `fclose` function is used to close the file. However, a file descriptor must instead be closed with the `close` system call, while `fclose` expects a pointer to a `FILE` data structure. In WebAssembly, this code executes succesfully and the program runs to completion, while it crashes in its native version.

```
int data = open("file.txt", O_RDWR|O_CREAT, S_IREAD|S_IWRITE);
fclose((FILE *)data);
```

Heuristic
Structural. Check for calls to `open` followed by calls to `fclose` on the same value, after a cast to `FILE*`.

CWE390: Error Without Action. The following example illustrates CWE 390. A file is opened with the `fopen` function. The file is then closed, without checking that opening the file actually succeeded. This results in an error when executed in WebAssembly (`RuntimeError: uninitialized element`), but succeeds in its native version. The WebAssembly crash is unexpected, as the file should be created successfully after calling `fopen`. A closer inspection reveals that `fopen` indeed fails and sets `errno` to `ENOTUNIQ` (*Name not unique on network*), which is an unconventional error for `fopen`

Author's address: Mohammad Ghafari.

```
FILE *fileDesc = fopen("file.txt", "w+");
fclose(fileDesc);
```

Heuristic

Dataflow analysis required. Check for calls to `FILE*` that have been opened, and that are closed in a branch where they could be `NULL`. Could happen inter-procedurally too.

CWE 253: Incorrect Check of Function Return Value, with `fputs`. The following example illustrates one difference that arises due to the use of `musl` as the standard C library when compiling to WebAssembly. Function `fputs` is called, and its result is checked against the constant 0 to see if printing the string has failed. However, the specification of `fputs` states that it returns `EOF` (-1) upon error, and a non-negative number upon success. The `musl` library returns 0 as number upon success, while `glibc` returns the number of bytes printed. As a result, the WebAssembly version enters the branch and prints `fputs failed!`.

Heuristic

Structural/lightweight dataflow. Check for return value of `fputs` flowing to a comparison against 0.

CWE 415: Double Free. The following excerpt illustrates CWE 415. It allocates a memory region but frees it twice. In its native version, this program results in a double free error (`free(): double free detected in tcache 2`), while it runs successfully in WebAssembly.

```
char *data = (char *)malloc(100*sizeof(char));
free(data);
free(data);
```

Heuristic

Dataflow. Requires tracking allocated/freed buffers. Already implemented in Clang: https://clang.llvm.org/docs/analyzer/checkers.html#unix-malloc

CWE 675: Double `fclose`. The following excerpt illustrates CWE 675. A file is opened, and thereafter closed twice. In its native version, this program succeeds. In its WebAssembly version, it fails: the file is not opened successfully, resulting in the same error as we encountered in CWE 390 (`ENOTUNIQ`: *Name not unique on network*).

```
FILE * data = freopen("BadSource_freopen.txt","w+",stdin);
fclose(data);
fclose(data);
```

Heuristic

Same as for CWE 390.

CWE 688: Incorrect String Argument. The following program calls `printf` with `"%s"` as a format string, but incorrectly passes an `int` as argument. The native program fails with a segmentation fault, as it tries to read a string at an invalid

memory location. However, the WebAssembly binary succeeds, reading an empty string from the (invalid) memory location 5. This is because the int argument is treated as a string pointer of which the first element is likely '0' due to WebAssembly initializing its linear memory with zeroes. As a result, this is interpreted as passing the empty string to `printf`.

```
printf("%s", 5);
```

Heuristic

Type analysis: look for invalid types being passed as argument to format string functions. Already implemented in clang with `-Wformat`.

CWE 590: Freeing Invalid Memory. The following program allocates memory on the stack using `alloca`, and tries to free it with `free`. However, `free` should be used to free heap-allocated memory. As a result, this program crashes in its native version. However, it runs to completion in WebAssembly. This could come from a different implementation of `free` in musl.

```
char * dataBuffer = (char *)alloca(100*sizeof(char));
free(dataBuffer);
```

Heuristic

Dataflow analysis required. Check for a buffer allocated with `alloca`, flowing into a call to `free`.

CWE 690: Dereferencing Null. Same as CWE 390 (uses fopen).

CWE 685: Incorrect Number of Arguments. The following program calls `printf` with too few arguments, while the provided format string expects two arguments. In WebAssembly, it works and treats the second string as null. In native, it crashes as it cannot provide a value for the second string.

```
printf("%s %s", "foo");
```

Heuristic

Already supported in clang with `-Wformat`.

CWE 761: Freeing Pointer Not At Start of Allocated Region. The following program allocates heap memory with `malloc`, and then moves the allocated pointer further in the allocated region. It then tries to free the memory by passing this incremented pointer as argument, which is invalid: `free` should be called on the pointer to the initial allocated region. This crashes in native, but works in WebAssembly, which could come from a different implementation of 'free' in musl.

```
char *data = (char *)malloc(100*sizeof(char));
data++;
free(data);
```

Heuristic

Requires a heavyweight analysis that tracks allocated regions.

CWE 469: Pointer Subtraction. This program performs invalid pointer manipulation by subtracting two different pointers: the `slash` variable points to the slash in the first string, but is mistakenly used to compute the index of the slash in `string2`. This prints a different value in WebAssembly and in its native version, most probably due to difference in memory layouts, i.e., `string1` and `string2` do not have the same offsets in both binaries.

```
char string1[] = "a/b";
char string2[] = "a/b";
char *slash = strchr(string1, '/');
printf("%d\n", slash - string2);
```

Heuristic

Probably requires a heavyweight pointer analysis?

CWE 121: Stack-Based Buffer Overflow. The following program contains a buffer overflow due to an incorrect allocation: `alloca` is used to allocate 10 bytes, while it should be used to allocate 10 integers (hence, `sizeof(int) * 10` bytes). When a buffer is copied into the badly allocated memory region, this overflows. In its native version, this program crashes as the stack is detected as being smashed. The WebAssembly version runs the program successfully.

```
int *data = alloca(10);
fir (int i = 0; i < 10; i++) {
  data[i] = 0;
}
```

Heuristic

Cannot be detected statically with decent precision (this is an entire research problem on its own).

CWE 126: Buffer Overread. The following program illustrates a buffer overread: the `data` string is filled with 150 times the 'A' character. 99 of them are copied into the `dest` buffer, but no extra null-terminating character is added. Hence, when printing the `dest` buffer, `printf` will continue printing the string until it encounters the byte 0. In WebAssembly, since the memory is initialized with 0s, the likeliness of having a byte 0 right after the string, and we encounter this in practice: only 99 As are printed. In its native version however, the string contains random garbage after the 99 first bytes, and `printf` prints much more characters.

```
char data[150] = "AAAAAAAA..."; // 150 As
char dest[100];
strncpy(dest, data, 99);
printf("%s\n", dest);
```

CWE 758: Undefined Behaviour. The following program has undefined behaviour according to the C standard. It allocates a pointer, but does not initialize it. When dereferencing this pointer to print a string, what will be printed is

therefore undefined. In native, the name of the program is being printed, while in WebAssembly, the empty string is printed.

```
char **pointer = alloca(sizeof(char *));
printf("%s\n", *pointer);
```

Heuristic

Requires some form of heavyweight pointer analysis?

CWE 457: Use of Uninitialized Variable. Similarly, the following program results in undefined behaviour because it reads data from an uninitialized behaviour. The observation is the same as previously: the WebAssembly output is the empty string, while the native output is the name of the program.

```
char *data;
printf("%s\n", data);
```

Heuristic

A simple dataflow analysis to detect uninitialized data should be enough.

Wide Characters. The following program simply writes a wide string to the console. While in its native version, the string is indeed written to the console, the WebAssembly version does not print anything. It actually requires calling `fwide` to tell the console that wide characters will be printed. This is likely due to a difference of libc.

```
wprintf(L"%ls\n", L"AAAA");
```

Heuristic

A simple grep for wprintf should give already some info. To more precisely detect this pattern, a dataflow analysis is required to detect calls to wprintf not preceded by a call to fwide.