Sneaking around with Web Assembly

dtm.uk/wasm

DTM August 10, 2024

DTM

Aug 10, 2024 • 17 min read

Photo by Pratik Patil / Unsplash

Introduction

WebAssembly (often abbreviated as WASM) is a binary instruction format designed as a portable compilation target for high-level programming languages like C, C++, and Rust, enabling deployment on the web for client and server applications. Introduced by the World Wide Web Consortium (W3C) in March 2017, WebAssembly aims to offer near-native performance. Alongside its binary format, WebAssembly features a human-readable text format known as WebAssembly Text Format (WAT), which facilitates debugging and learning. Today, WebAssembly enjoys widespread support across all major web browsers, including Chrome, Firefox, Safari, and Edge, making it a robust and versatile choice for developers looking to build high-performance client-side web applications.

Concepts

There are two key concepts to really understand when it comes to Web Assembly:

- **Memory**: Memory can be shared between WebAssembly (WASM) and JavaScript (JS) by creating a `WebAssembly.Memory` object. This object can be accessed directly in JS, allowing for efficient data exchange and manipulation.
- **Functions**: There are bi-directional function call capabilities between WASM and JavaScript. We can export functions from WASM and make them callable from JavaScript. Similarly, we can import JavaScript functions to be called and interact with the web browser from within WASM.

Digging into the specific available WASM instruction set (see Appendix of this post for a table) gives you some insight into the capabilities.

WASM Inline Loader

With this in mind, let's create a basic example by making a WebAssembly (WASM) module with one function. We will expose `console.log()` from JavaScript into WASM and then call it with a string from within the WASM module.

The `WebAssembly.instantiate` method accepts WASM bytes directly from JavaScript, allowing us to embed WebAssembly directly in our JavaScript code. To demonstrate this, we'll create a minimal WASM module. We'll start with `hello.wat` to create a function in WASM that logs "Hello World" to our browser console:

```
(module
  (import "env" "log" (func $log (param i32 i32)))
  (memory (export "memory") 1)
  (data (i32.const 16) "Hello, world!")
  (func (export "hello")
    i32.const 16
    i32.const 13
    call $log
  )
)
```

This WebAssembly Text (WAT) code defines a module that imports a JavaScript `log` function and sets up memory to store the string "Hello, world!". The `import` statement brings in the `log` function from the `env` environment, expecting two 32-bit integer parameters. The module declares and exports a memory segment of one page (64 KB) and initializes it with the string "Hello, world!" starting at offset 16. The `hello` function, which is exported, pushes the memory offset (16) and the string length (13) onto the stack and calls the imported `log` function, effectively logging "Hello, world!" to the console.

Next we need to convert WAT to WASM using the Web Assembly Binary Toolkit. On Mac you can install this using:

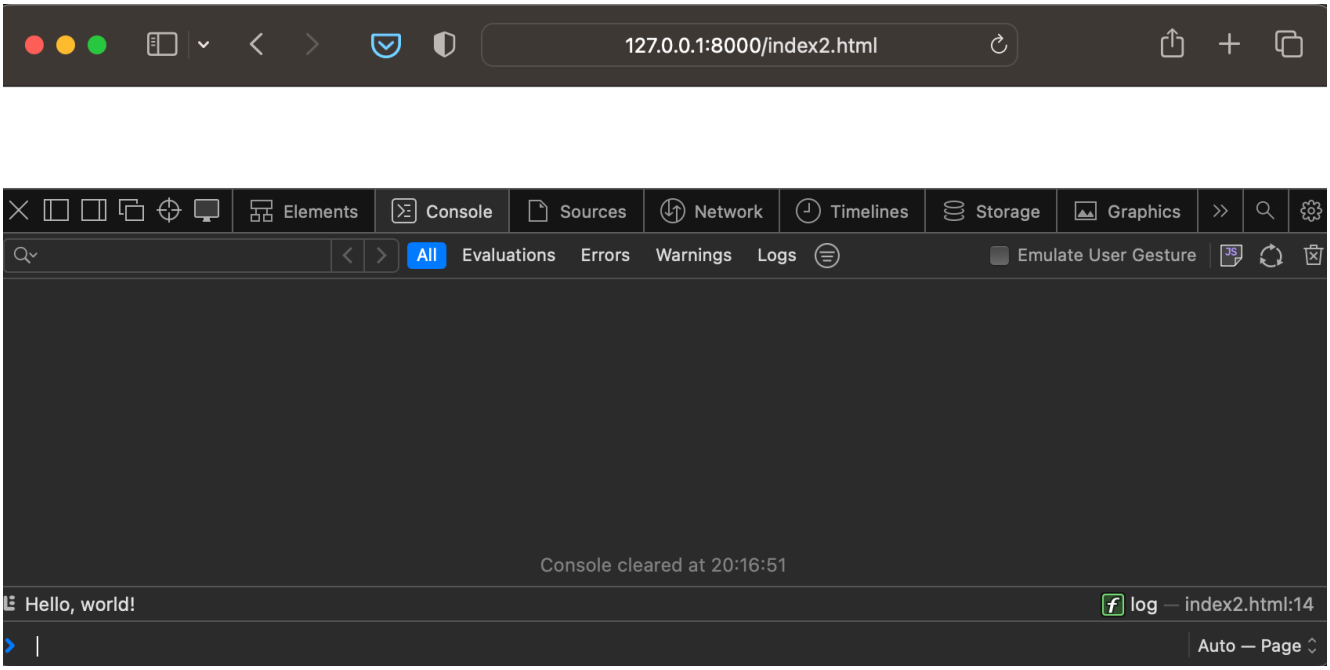
```
brew install wabt
```

We compile it and base64 the output, then send this to clip board so we can copy into our HTML template:

```
wat2wasm hello.wat
base64 -i hello.wasm | pbcopy
```

The final HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Minimal WASM Example</title>
</head>
<body>
    <script>
        const b64 =
'AGFzbQEAAAABCQJgAn9/AGAAAAILAQNlbnYDbG9nAAADAgEBBQMBAAEHEgIGbWVtb3J5AgAFaGVsbG8AAQoKAQgAQRB
BDRAACwsTAQBBEAsNSGVsbG8sIHdvcmxkIQAUBG5hbWUBBgEAA2xvZwIFAgAAAQA='; // Insert the base64
string from hello_world.wasm.b64 here
        const bytes = Uint8Array.from(atob(b64), c => c.charCodeAt(0));
        const imports = {
            env: {
                log: (ptr, len) => console.log(new TextDecoder('utf-8').decode(new
Uint8Array(wasm.memory.buffer, ptr, len)))
            }
        };
        let wasm;
        WebAssembly.instantiate(bytes.buffer, imports).then(result => {
            wasm = result.instance.exports;
            wasm.hello();
        }).catch(console.error);
    </script>
</body>
</html>
```



WAT Smuggling

I started playing around with Web Assembly by compiling some Rust code to WASM and looking at how I can call JavaScript from WASM and vice versa. Naturally I'm drawn to new any new offensive TTPs that could utilise web assembly. I did some searching around for anything already around and came across some research from some friends at delivr.to (Waves James and Alfie!) https://blog.delivr.to/webassembly-smuggling-it-wasmt-me-648a62547ff4. This blog takes a path of utilising Rust code to smuggle content back into HTML.

This is when I went down two rabbit holes, firstly how to make efficiencies for generated WASM size after compiling Rust and secondly getting my head around what are the available instructions in human readable WAT and therefore its associated WASM (See WAT Syntax).

Now I went on my journey to generate some WAT code that could be used to simply store binary content. The following Python script is what I ended up with:

```
import argparse
import math

def generate_wat(binary_data,chunk_size=1024):
    # Convert binary data to hexadecimal values suitable for WAT and split into chunks
    chunks = [binary_data[i:i +chunk_size] for i in range(0, len(binary_data),chunk_size)]
    hex_chunks = [''.join(f'\\{byte:02x}' for byte in chunk) for chunk in chunks]
    binary_length = len(binary_data)

    # Calculate the number of pages needed (1 page = 65536 bytes)
    num_pages = math.ceil(binary_length / 65536)

    wat_template = f"""
    (module
      (memory $mem {num_pages})
      (export "memory" (memory $mem))
    """

    for i, hex_chunk in enumerate(hex_chunks):
        wat_template += f"""
      (data (i32.const {i *chunk_size}) "{hex_chunk}")
    """

    wat_template += f"""
      (func $get_binary (result i32)
        (i32.const 0)
      )
      (export "get_binary" (func $get_binary))
      (func $get_binary_length (result i32)
        (i32.const {binary_length})
      )
      (export "get_binary_length" (func $get_binary_length))
    )
    """

    return wat_template

def main():
    parser = argparse.ArgumentParser(description='Embed a binary file into a WASM module.')
    parser.add_argument('binary_file', type=str, help='The binary file to embed.')
    parser.add_argument('output_wat', type=str, help='The output WAT file.')
    parser.add_argument('--chunk-size', type=int, default=1024,
                        help='The size of chunks to split the binary data into.')

    args = parser.parse_args()

    with open(args.binary_file, 'rb') as bin_file:
        binary_data = bin_file.read()

    wat_code = generate_wat(binary_data, args.chunk_size)

    with open(args.output_wat, 'w') as wat_file:
        wat_file.write(wat_code)

    print(f"WAT code has been written to {args.output_wat}")
```

```
if __name__ == "__main__":
    main()
```

Running it against a binary, in our case a GIF file, should generate some valid WAT code with the binary chunked up and two exported methods, get_binary and get_binary_length.

```
python3 watsmuggle.py giphy.gif output.wat
WAT code has been written to output.wat
Run the following command to generate the WASM module:
wat2wasm output.wat -o output.wasm
```

Now we can do the conversion:

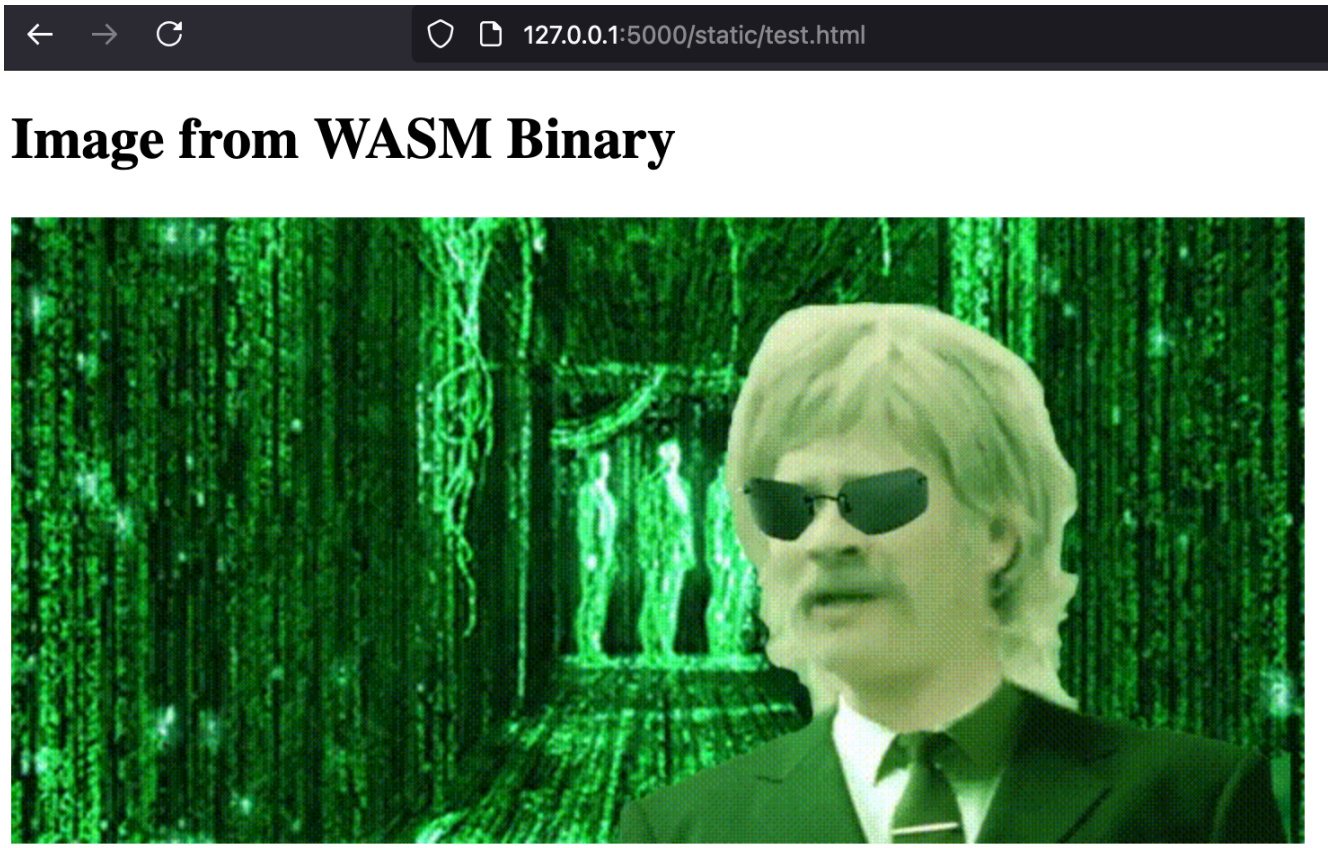
```
wat2wasm output.wat
```

Sweet! Now we have output.wasm and all we need now is HTML / JavaScript to load our Web Assembly and retrieve our image and render it:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Display Image from WASM</title>
</head>
<body>
  <h1>Image from WASM Binary</h1>
  <img id="wasm-image" alt="Image loaded from WASM binary">

  <script>
    fetch('output.wasm')
      .then(response=>response.arrayBuffer())
      .then(bytes=> WebAssembly.instantiate(bytes, {}))
      .then(results=> {
        const instance =results.instance;
        const memory = new Uint8Array(instance.exports.memory.buffer);
        const offset = instance.exports.get_binary();
        const imageLength = instance.exports.get_binary_length(); // Retrieve the image
length

        console.log('Memory buffer length:', memory.length);
        console.log('Offset:', offset);
        console.log('Image length:', imageLength);

        // Ensure the image length is within bounds
        if (offset + imageLength > memory.length) {
          throw new Error('Image length exceeds memory buffer size');
        }

        const binaryData = memory.slice(offset, offset + imageLength);

        // Create a Blob from the binary data and generate a URL
        const blob = new Blob([binaryData], { type: 'image/gif' }); // Change the type as
per your image format
        const url = URL.createObjectURL(blob);

        // Set the image src to the generated URL
        document.getElementById('wasm-image').src = url;
      })
      .catch(err=> console.error('Error loading WASM module:',err));
  </script>
</body>
</html>
```

Demo:

https://lab.k7.uk/wasm/gif_wasm.html



But we could also look to obfuscate/encrypt the binary in the Web Assembly too? Let's try with a basic XOR implementation in WAT/WASM:

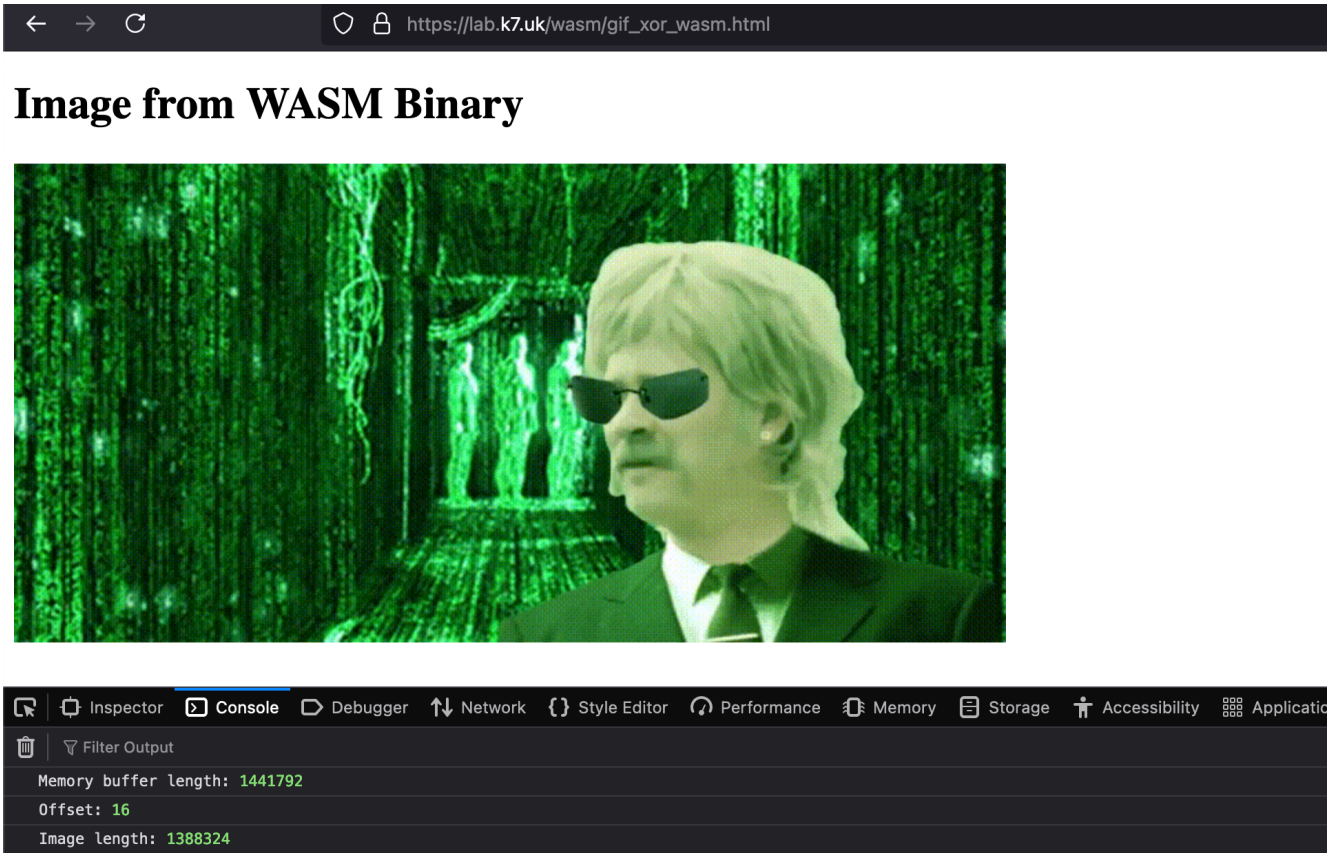
```
import argparse
import math
import os

def xor_encrypt_decrypt(data, key):
    key_len = len(key)
    return bytes([data[i] ^ key[i % key_len] for i in range(len(data))])

def generate_wat(encrypted_data, key, chunk_size=1024):
    # Convert encrypted data to hexadecimal values suitable for WAT and split into chunks
    chunks = [encrypted_data[i:i + chunk_size] for i in range(0, len(encrypted_data),
chunk_size)]
    hex_chunks = [''.join(f'\\{byte:02x}' for byte in chunk) for chunk in chunks]
    data_length = len(encrypted_data)

    # Calculate the number of pages needed (1 page = 65536 bytes)
    num_pages = math.ceil(data_length / 65536)

    key_hex = ''.join(f'\\{byte:02x}' for byte in key)

    wat_template = f"""
    (module
      (memory $mem {num_pages})
      (export "memory" (memory $mem))
      (data (i32.const 0) "{key_hex}")
    """

    for i, hex_chunk in enumerate(hex_chunks):
        wat_template += f"""
      (data (i32.const {i * chunk_size + len(key)}) "{hex_chunk}")
    """

    wat_template += f"""
      (func $get_binary (result i32)
        (call $decrypt (i32.const {len(key)}) (i32.const {len(key)}) (i32.const
{data_length}))
        (i32.const {len(key)})
      )
      (export "get_binary" (func $get_binary))
      (func $get_binary_length (result i32)
        (i32.const {data_length})
      )
      (export "get_binary_length" (func $get_binary_length))

      (func $decrypt (param $dst i32) (param $src i32) (param $len i32)
        (local $key_offset i32)
        (local $i i32)
        (local $key_len i32)
        (local $data_byte i32)
        (local $key_byte i32)

        (local.set $key_offset (i32.const 0))
        (local.set $key_len (i32.const {len(key)}))
        (local.set $i (i32.const 0))

        (block $outer
```

```
      (loop $inner
        (br_if $outer (i32.ge_u (local.get $i) (local.get $len)))

        (local.set $data_byte
          (i32.load8_u
            (i32.add (local.get $src) (local.get $i))
          )
        )

        (local.set $key_byte
          (i32.load8_u
            (i32.add (local.get $key_offset)
                     (i32.rem_u (local.get $i) (local.get $key_len)))
          )
        )

        (i32.store8
          (i32.add (local.get $dst) (local.get $i))
          (i32.xor (local.get $data_byte) (local.get $key_byte))
        )

        (local.set $i
          (i32.add (local.get $i) (i32.const 1))
        )

        (br $inner)
      )
    )
  )
  (export "decrypt" (func $decrypt))
)
"""

    return wat_template

def main():
    parser = argparse.ArgumentParser(description='Embed an encrypted binary file into a WASM
module.')
    parser.add_argument('binary_file', type=str, help='The binary file to embed.')
    parser.add_argument('output_wat', type=str, help='The output WAT file.')
    parser.add_argument('--chunk-size', type=int, default=1024,
                        help='The size of chunks to split the binary data into.')

    args = parser.parse_args()

    with open(args.binary_file, 'rb') as bin_file:
        binary_data = bin_file.read()

    key = os.urandom(16)
    encrypted_data = xor_encrypt_decrypt(binary_data, key)
    wat_code = generate_wat(encrypted_data, key, args.chunk_size)

    with open(args.output_wat, 'w') as wat_file:
        wat_file.write(wat_code)

    print(f"WAT code has been written to {args.output_wat}")
```

```
if __name__ == "__main__":
    main()
```

Demo:

https://lab.k7.uk/wasm/gif_xor_wasm.html



JavaScript Obfuscation using Web Assembly

We've already shown how trivially Web Assembly can be used to embed content this makes it harder to statically analyse due to another layer of abstraction in a lesser known place.

Hang on though, what if we expose `eval()` from JavaScript to WASM - this is one trivial way we could directly execute arbitrary JavaScript from WASM.

I ended up with putting together this PoC to generate the HTML either from either JavaScript code passed on the command line or in a single JS source file:

```
import argparse
import os
import subprocess
import base64

def generate_wat(js_code):
    js_code_length = len(js_code)
    js_code_escaped =js_code.replace('\\', '\\\\').replace('\n', '\\n').replace('"', '\\"')

    wat_code = f"""
                (module
                  (import "env" "eval_js" (func $eval_js (param i32 i32)))
                  (memory $0 1)
                  (export "memory" (memory $0))
                  (export "hello" (func $hello))
                  (data (i32.const 16) "{js_code_escaped}")

                  (func $hello
                    ;; String pointer and length
                    (i32.store (i32.const 0) (i32.const 16)) ;; Store the string pointer at
memory offset 0
                    (i32.store (i32.const 4) (i32.const {js_code_length})) ;; Store the
string length at memory offset 4 ({js_code_length} characters)

                    ;; Call eval_js with the pointer and length
                    (call $eval_js
                      (i32.load (i32.const 0)) ;; Load the string pointer
                      (i32.load (i32.const 4)) ;; Load the string length
                    )
                  )
                )
                """
    return wat_code

def wat_to_wasm(wat_file,wasm_file):
    subprocess.run(['wat2wasm',wat_file, '-o',wasm_file], check=True)

def wasm_to_base64(wasm_file):
    with open(wasm_file, 'rb') as f:
        wasm_binary = f.read()
    return base64.b64encode(wasm_binary).decode('utf-8')

def generate_html(base64_wasm,output_html):
    html_template = f"""<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Eval</title>
</head>
<body>
    <script>
        const b64 = '{base64_wasm}';
        const bytes = Uint8Array.from(atob(b64), c => c.charCodeAt(0));
        const imports = {{
            env: {{
```

```
                    eval_js: (ptr, len) => {{
                    const jsCode = new TextDecoder('utf-8').decode(new
Uint8Array(wasm.memory.buffer, ptr, len));
                    eval(jsCode);
                }}
            }}
        }};
        let wasm;
        WebAssembly.instantiate(bytes.buffer, imports).then(result => {{
            wasm = result.instance.exports;
            wasm.hello();
        }}).catch(console.error);
    </script>
</body>
</html>
"""
    with open(output_html, 'w') as f:
        f.write(html_template)

def main():
    parser = argparse.ArgumentParser(description="Generate WAT, convert to WASM, Base64
encode, and embed in HTML")
    parser.add_argument('-c', '--code', type=str, help="JavaScript code as a command line
argument")
    parser.add_argument('-f', '--file', type=str, help="Path to a file containing JavaScript
code")
    parser.add_argument('-o', '--output', type=str, default="output.html", help="Output HTML
file name")

    args = parser.parse_args()

    if args.code:
        js_code = args.code
    elif args.file:
        if os.path.exists(args.file):
            with open(args.file, 'r') as file:
                js_code = file.read()
        else:
            print(f"Error: File '{args.file}' not found.")
            return
    else:
        print("Error: You must provide either JavaScript code or a source file.")
        return

    wat_code = generate_wat(js_code)
    wat_file = 'output.wat'
    wasm_file = 'output.wasm'

    with open(wat_file, 'w') as f:
        f.write(wat_code)

    wat_to_wasm(wat_file, wasm_file)
    base64_wasm = wasm_to_base64(wasm_file)
    generate_html(base64_wasm, args.output)

    print(f"HTML file generated and saved as '{args.output}'")
```

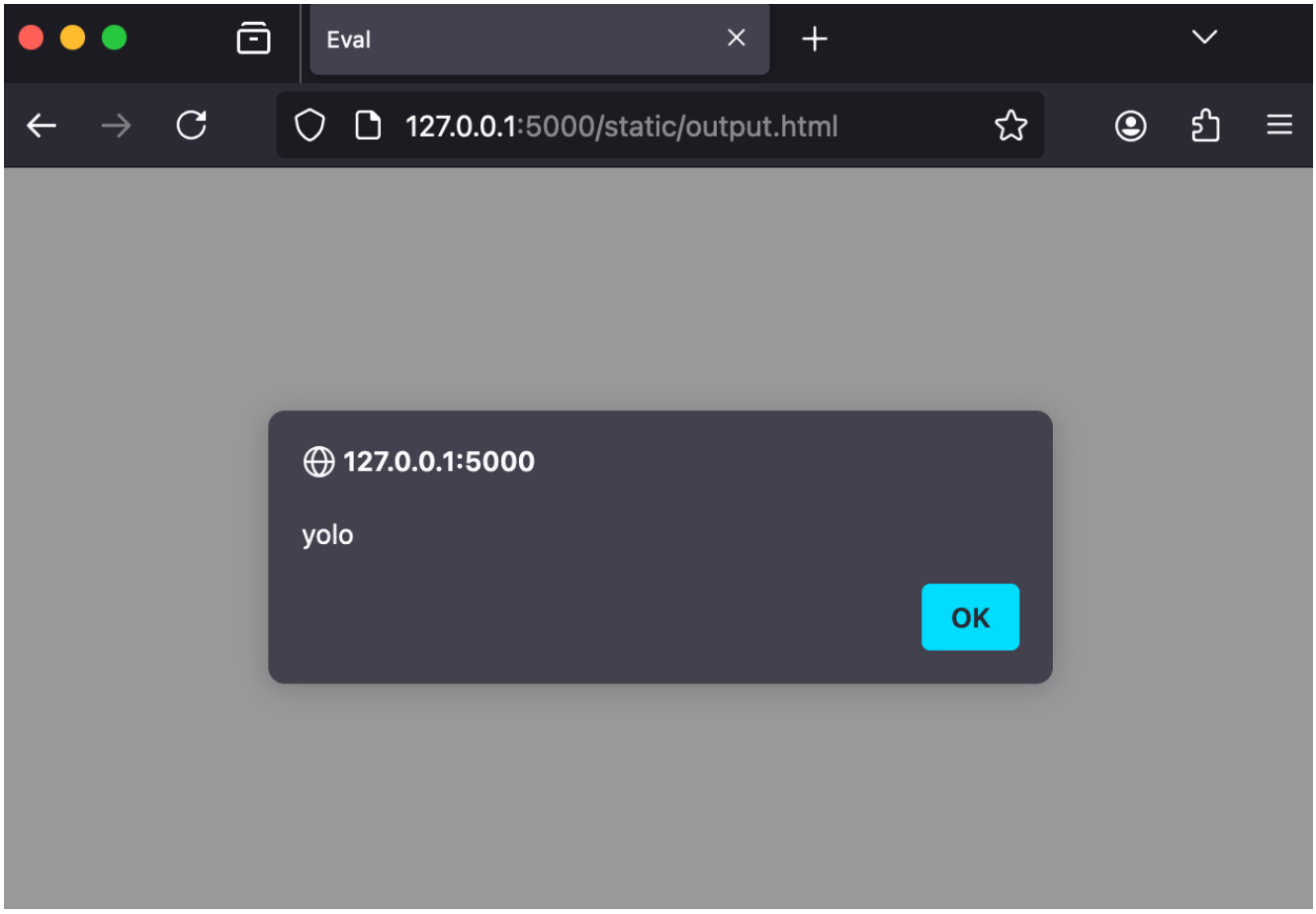
```
if __name__ == "__main__":
    main()
```

Let’s give this a try:

```
% python3 eval.py -h
usage: eval.py [-h] [-c CODE] [-f FILE] [-o OUTPUT]

Generate WAT, convert to WASM, Base64 encode, and embed in HTML

options:
  -h, --help            show this help message and exit
  -c CODE, --code CODE  JavaScript code as a command line argument
  -f FILE, --file FILE  Path to a file containing JavaScript code
  -o OUTPUT, --output OUTPUT
                        Output HTML file name

% python3 eval.py -c "alert('yolo')"
HTML file generated and saved as 'output.html'
```

Here’s the resulting HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Eval</title>
</head>
<body>
    <script>
        const b64 =
'AGFzbQEAAAABCQJgAn9/AGAAAAIPAQNlbnYHZXZhbF9qcwAAAwIBAQUDAQABBxICBm1lbW9yeQIABWhlbGxvAAEKHgE
cAEEAQRA2AgBBBEENNgIAQQAoAgBBBCgCABAACwsTAQBBEAsNYWxlcnQoJ3lvbG8nKQ==';
        const bytes = Uint8Array.from(atob(b64), c => c.charCodeAt(0));
        const imports = {
            env: {
                    eval_js: (ptr, len) => {
                    const jsCode = new TextDecoder('utf-8').decode(new
Uint8Array(wasm.memory.buffer, ptr, len));
                    eval(jsCode);
                }
            }
        };
        let wasm;
        WebAssembly.instantiate(bytes.buffer, imports).then(result => {
            wasm = result.instance.exports;
            wasm.hello();
        }).catch(console.error);
    </script>
</body>
</html>
```



Demo:

https://lab.k7.uk/wasm/eval.html

But is it really obfuscation if we can run strings on the WASM and get:

```
% strings output.wasm
eval_js
memory
hello
alert('yolo')
```

It's time to roll our previous XOR example in. Here's an updated PoC which XORs the JavaScript:

```
import argparse
import os
import subprocess
import base64

def xor_encrypt_decrypt(data, key):
    key_len = len(key)
    return bytes([data[i] ^ key[i % key_len] for i in range(len(data))])

def generate_wat(encrypted_data, key, chunk_size=1024):
    # Convert encrypted data to hexadecimal values suitable for WAT and split into chunks
    chunks = [encrypted_data[i:i + chunk_size] for i in range(0, len(encrypted_data),
chunk_size)]
    hex_chunks = [''.join(f'\\{byte:02x}' for byte in chunk) for chunk in chunks]
    data_length = len(encrypted_data)

    # Calculate the number of pages needed (1 page = 65536 bytes)
    num_pages = (data_length + 65535) // 65536

    key_hex = ''.join(f'\\{byte:02x}' for byte in key)

    wat_template = f"""
    (module
      (import "env" "eval_js" (func $eval_js (param i32 i32)))
      (memory $0 {num_pages})
      (export "memory" (memory $0))
      (data (i32.const 0) "{key_hex}")
    """

    for i, hex_chunk in enumerate(hex_chunks):
        wat_template += f"""
      (data (i32.const {i * chunk_size + len(key)}) "{hex_chunk}")
    """

    wat_template += f"""
      (func $hello
        (local $key_offset i32)
        (local $i i32)
        (local $key_len i32)
        (local $data_byte i32)
        (local $key_byte i32)
        (local $len i32)
        (local $ptr i32)

        ;; Initialize variables
        (local.set $key_offset (i32.const 0))
        (local.set $key_len (i32.const {len(key)}))
        (local.set $len (i32.const {data_length}))
        (local.set $ptr (i32.const {len(key)}))

        (block $outer
          (loop $inner
            (br_if $outer (i32.ge_u (local.get $i) (local.get $len)))

            ;; Load the encrypted byte
            (local.set $data_byte
```

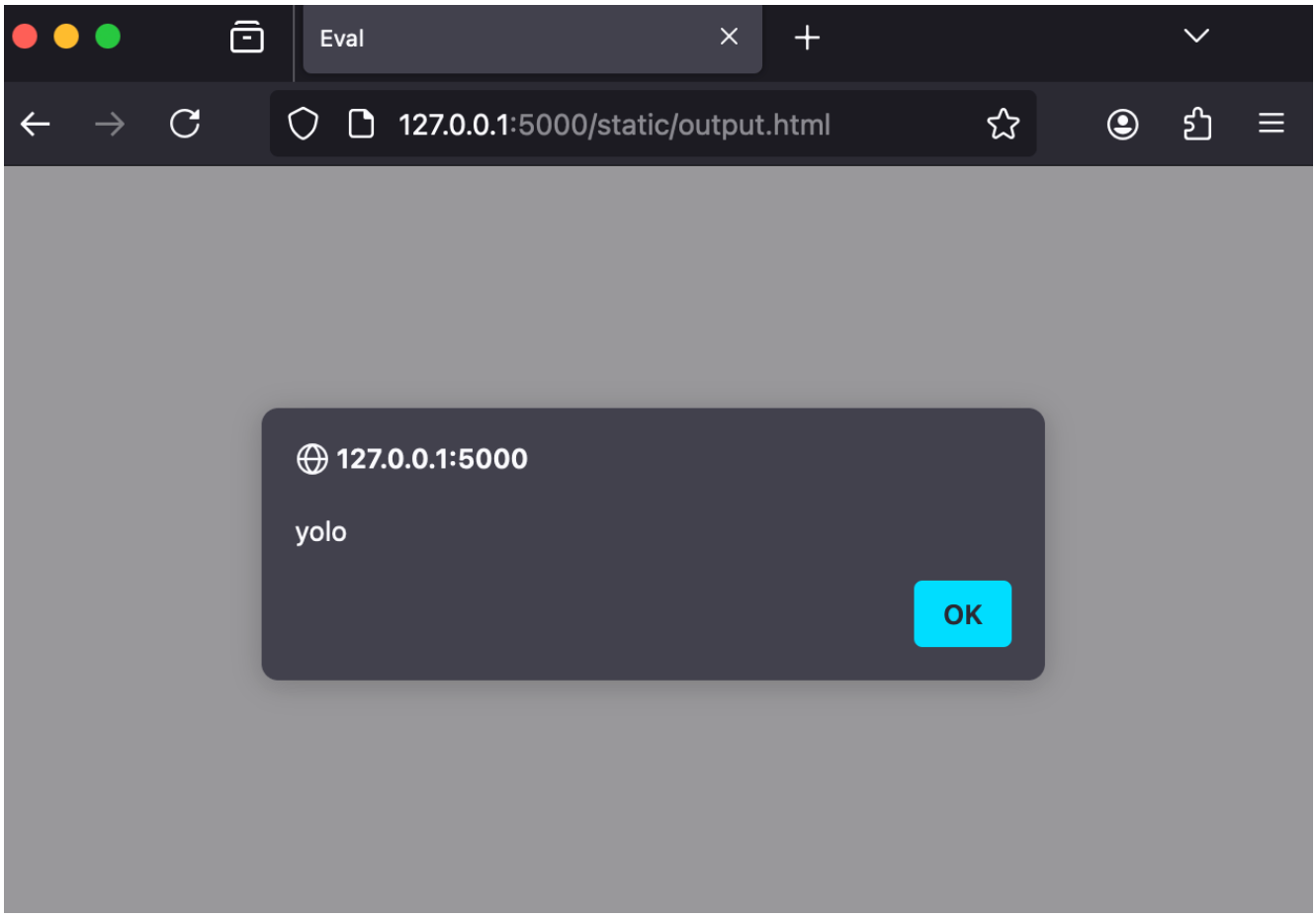
```
              (i32.load8_u
                (i32.add (local.get $ptr) (local.get $i))
              )
            )

            ;; Load the key byte
            (local.set $key_byte
              (i32.load8_u
                (i32.add (local.get $key_offset)
                         (i32.rem_u (local.get $i) (local.get $key_len)))
              )
            )

            ;; XOR decrypt the byte
            (i32.store8
              (i32.add (local.get $ptr) (local.get $i))
              (i32.xor (local.get $data_byte) (local.get $key_byte))
            )

            ;; Increment the index
            (local.set $i
              (i32.add (local.get $i) (i32.const 1))
            )

            (br $inner)
          )
        )

        ;; Call eval_js with the pointer and length
        (call $eval_js
          (i32.const {len(key)}) ;; Pointer to the decrypted data
          (i32.const {data_length}) ;; Length of the decrypted data
        )
      )
      (export "hello" (func $hello))
    )
    """

    return wat_template

def wat_to_wasm(wat_file, wasm_file):
    subprocess.run(['wat2wasm', wat_file, '-o', wasm_file], check=True)

def wasm_to_base64(wasm_file):
    with open(wasm_file, 'rb') as f:
        wasm_binary = f.read()
    return base64.b64encode(wasm_binary).decode('utf-8')

def generate_html(base64_wasm, output_html):
    html_template = f"""<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Eval</title>
</head>
```

```
<body>
    <script>
        const b64 = '{base64_wasm}';
        const bytes = Uint8Array.from(atob(b64), c => c.charCodeAt(0));
        const imports = {{
            env: {{
                eval_js: (ptr, len) => {{
                    const jsCode = new TextDecoder('utf-8').decode(new
Uint8Array(wasm.memory.buffer, ptr, len));
                    eval(jsCode);
                }}
            }}
        }};
        let wasm;
        WebAssembly.instantiate(bytes.buffer, imports).then(result => {{
            wasm = result.instance.exports;
            wasm.hello();
        }}).catch(console.error);
    </script>
</body>
</html>
"""
    with open(output_html, 'w') as f:
        f.write(html_template)

def main():
    parser = argparse.ArgumentParser(description="Generate WAT, convert to WASM, Base64
encode, and embed in HTML")
    parser.add_argument('-c', '--code', type=str, help="JavaScript code as a command line
argument")
    parser.add_argument('-f', '--file', type=str, help="Path to a file containing JavaScript
code")
    parser.add_argument('-o', '--output', type=str, default="output.html", help="Output HTML
file name")

    args = parser.parse_args()

    if args.code:
        js_code = args.code
    elif args.file:
        if os.path.exists(args.file):
            with open(args.file, 'r') as file:
                js_code = file.read()
        else:
            print(f"Error: File '{args.file}' not found.")
            return
    else:
        print("Error: You must provide either JavaScript code or a source file.")
        return

    key = os.urandom(16)
    encrypted_data = xor_encrypt_decrypt(js_code.encode('utf-8'), key)
    wat_code = generate_wat(encrypted_data, key)
    wat_file = 'output.wat'
    wasm_file = 'output.wasm'
```

```
    with open(wat_file, 'w') as f:
        f.write(wat_code)

    wat_to_wasm(wat_file, wasm_file)
    base64_wasm = wasm_to_base64(wasm_file)
    generate_html(base64_wasm, args.output)

    print(f"HTML file generated and saved as '{args.output}'")

if __name__ == "__main__":
    main()
```

Let's generate an XORed version:

```
% python3 xor_eval.py -c "alert('yolo')"
HTML file generated and saved as 'output.html'
% strings output.wasm
eval_js
memory
hello
```



Demo:

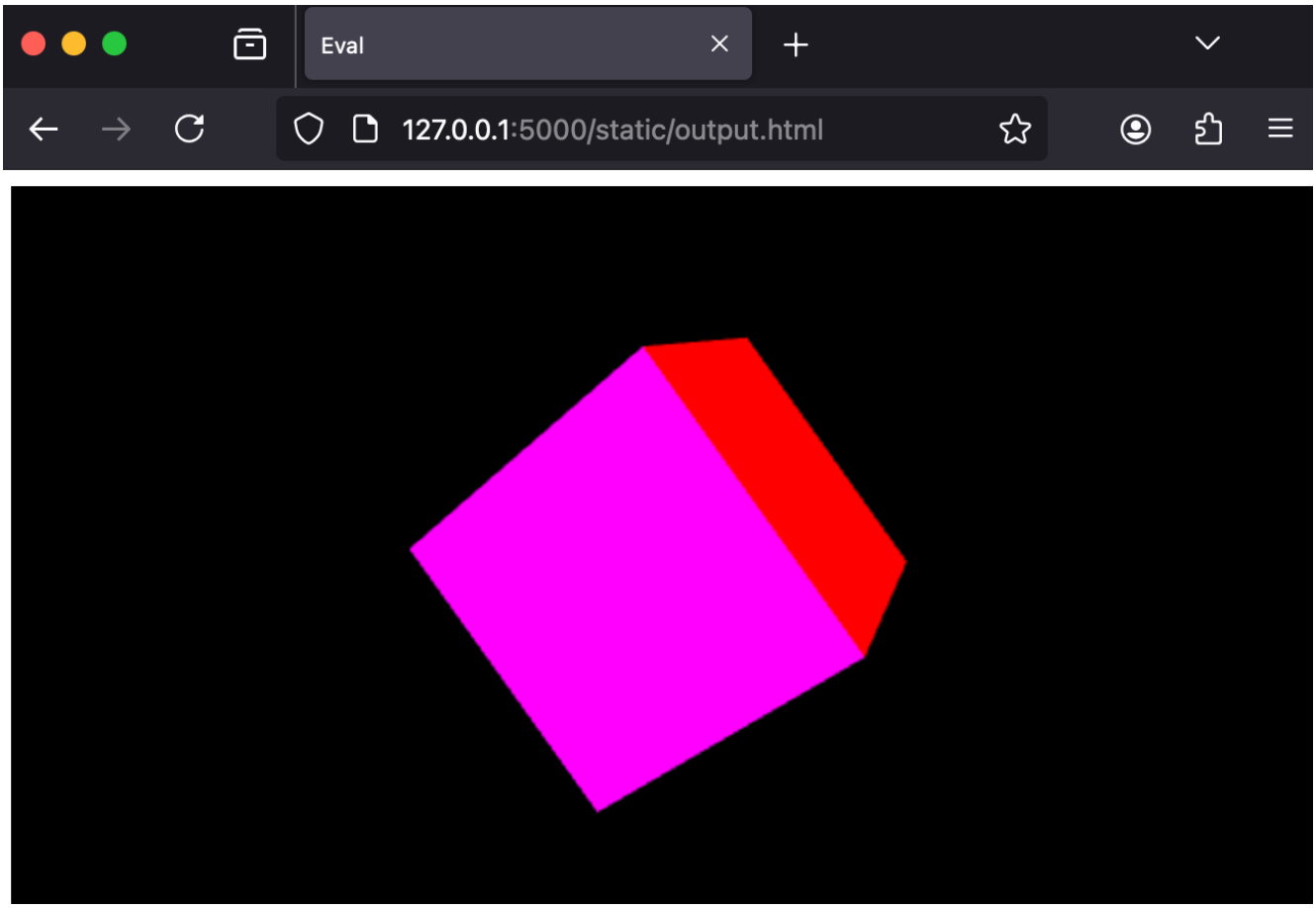
https://lab.k7.uk/wasm/xor_eval.html

Here's the HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Eval</title>
</head>
<body>
    <script>
        const b64 =
'AGFzbQEAAAABCQJgAn9/AGAAAAIPAQNlbnYHZXZhbF9qcwAAAwIBAQUDAQABBxICBm1lbW9yeQIABWhlbGxvAAEKVgF
UAQd/QQAhAEEQIQJBDSEFQRAhBgJAA0AgASAFTw0BIAYgAWotAAAhAyAAIAEgAnBqLQAAIQQgBiABaiADIARzOgAAIAF
BAWohAQwACwtBEEENEAALCygCAEEACxBHawnuTv6N06Lc6sB0boXwAEEQCw0mB2ycOtaqqs2whedd';
        const bytes = Uint8Array.from(atob(b64), c => c.charCodeAt(0));
        const imports = {
            env: {
                eval_js: (ptr, len) => {
                    const jsCode = new TextDecoder('utf-8').decode(new
Uint8Array(wasm.memory.buffer, ptr, len));
                    eval(jsCode);
                }
            }
        };
        let wasm;
        WebAssembly.instantiate(bytes.buffer, imports).then(result => {
            wasm = result.instance.exports;
            wasm.hello();
        }).catch(console.error);
    </script>
</body>
</html>
```

Let’s test a more substantial JavaScript file:

```
python3 xor_eval.py -f test.js
HTML file generated and saved as 'output.html'
```



Demo:

https://lab.k7.uk/wasm/cube.html

Appendix

References

WAT Syntax

Category	Syntax	Description
Module Definition	(module ...)	Defines a WebAssembly module.
Import Function	(import "env" "func" (func $func_name (param i32) (result i32)))	Imports a function named func from env namespace, with parameters and return type.
Export Function	(export "func_name" (func $func_name))	Exports a function with the name func_name.
Memory	(memory $mem_name 1)	Defines a memory block of 1 page (64KiB).

Category	Syntax	Description
Export Memory	(export "memory" (memory $mem_name))	Exports the memory block.
Data Segment	(data (i32.const 16) "Hello, world!")	Initializes memory at offset 16 with the string "Hello, world!".
Function Definition	(func $func_name (param $x i32) (result i32) ...)	Defines a function with parameters and return type.
Local Variables	(local $var_name i32)	Declares a local variable of type i32.
Call Function	call $func_name	Calls a function named $func_name.
Get Local	local.get $var_name	Pushes the value of a local variable onto the stack.
Set Local	local.set $var_name	Pops the top value off the stack and stores it in a local variable.
Const Value	i32.const 10	Pushes a constant value (10) onto the stack.
Arithmetic Ops	i32.add, i32.sub, i32.mul, i32.div_s	Performs arithmetic operations on the top values of the stack.
Comparison Ops	i32.eq, i32.ne, i32.lt_s, i32.gt_s, i32.le_s, i32.ge_s	Compares the top values of the stack and pushes the result.
Control Structures	if (result i32) ... else ... end	Conditional execution.
	loop $label ... br_if $label ... end	Loop with a conditional break.
Load from Memory	i32.load (i32.const 0)	Loads an i32 value from memory at offset 0.
Store to Memory	i32.store (i32.const 0) (i32.const 42)	Stores an i32 value (42) at memory offset 0.
Memory Size	(memory.size)	Returns the current size of memory.
Memory Grow	(memory.grow (i32.const 1))	Grows the memory by the specified number of pages.
Loop	(loop $label ... br $label ... end)	Defines a loop with a label.
Block	(block $label ... end)	Defines a block with a label.
Branch	br $label	Unconditionally branches to a label.

Category	Syntax	Description
Branch if	br_if $label	Conditionally branches to a label if the top of the stack is non-zero.
Return	return	Returns from the current function.
Drop	drop	Pops and discards the top value of the stack.
Select	select	Pops the top three values from the stack and pushes either the second or third value based on the first value.
Start	(start $func_name)	Specifies a function to be called automatically when the module is instantiated.

Rust Example

One final example demonstrates how to build a small WASM file from Rust code.

Cargo.toml

```
[package]
name = "hello_wasm"
version = "0.1.0"
edition = "2018"

[lib]
crate-type = ["cdylib"]

[profile.release]
lto = true
opt-level = "z"
codegen-units = 1

[dependencies]
wasm-bindgen = "0.2"
wee_alloc = "0.4.5"
```

hello.rs

```
extern crate wasm_bindgen;

use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn hello_world() {
    log("Hello, world!");
}

#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_name = log, js_namespace = console)]
    fn log(s: &str);
}
```

Compilation using wasm-pack:

```
wasm-pack build --target web
cd ./pkg/
base64 -i hello_wasm_bg.wasm | pbcopy
```

Calling the generated WASM in the same way, although you will have to align the bindings to what is expected for the imported function(s):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Rust WASM Example</title>
</head>
<body>
    <script>
        const b64 =
'AGFzbQEAAAABCQJgAn9/AGAAAAIiAQN3YmcaX193YmdfbG9nX2QwZDU1ZTA5OGVmYmFkMzUAAAMDAgABBQMBABEHGAI
GbWVtb3J5AgALaGVsbG9fd29ybGQAAgovAiEAIABC+oScg7LWqZRTNwMIIABC9Kmo3LKD6YKHfzcDAAsLAEGAgMAAQQ0
QAAsLowIBAEGAgMAAC5kCSGVsbG8sIHdvcmxkIWNhbGxlZCBgT3B0aW9uOjp1bndyYXAoKWAgb24gYSBgTm9uZWAgdmF
sdWUAAAAAAAAAAAAEAAAABAAAAL3J1c3QvZGVwcy9kbG1hbGxvYy0wLjIuNi9zcmMvZGxtYWxsb2MucnNhc3NlcnRpb24
gZmFpbGVkOiBwc2l6ZSA+PSBzaXplICsgbWluX292ZXJoZWFkAEgAEAApAAAAqAQAAAkAAABhc3NlcnRpb24gZmFpbGV
kOiBwc2l6ZSA8PSBzaXplICsgbWF4X292ZXJoZWFkAABIABAAKQAAAK4EAAANAAAAbGlicmFyeS9zdGQvc3JjL3Bhbml
ja2luZy5yc/AAEAAcAAAAiwIAAB4Abwlwcm9kdWNlcnMCCGxhbmd1YWdlAQRSdXN0AAxwcm9jZXNzZWQtYnkDBXJ1c3R
jHTEuODAuMCAoMDUxNDc4OTU3IDIwMjQtMDctMjEpBndhbHJ1cwYwLjIwLjMMd2FzbS1iaW5kZ2VuBjAuMi45MgAsD3R
hcmdldF9mZWF0dXJlcwIrD211dGFibGUtZ2xvYmFscysIc2lnbi1leHQ=';
        const bytes = Uint8Array.from(atob(b64), c => c.charCodeAt(0));
        const imports = {
            env: {},
            wbg: {
                __wbg_log_d0d55e098efbad35: (ptr, len) => console.log(new TextDecoder('utf-
8').decode(wasm.memory.buffer.slice(ptr, ptr + len))) // fix up the generated binding!
            }
        };
        let wasm;
        WebAssembly.instantiate(bytes.buffer, imports).then(result => {
            wasm = result.instance.exports;
            wasm.hello_world();
        }).catch(console.error);
    </script>
</body>
</html>
```