First let's open the html file, we can see it asks for a passcode.

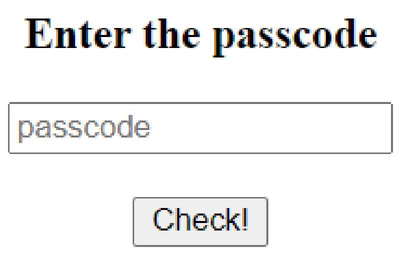
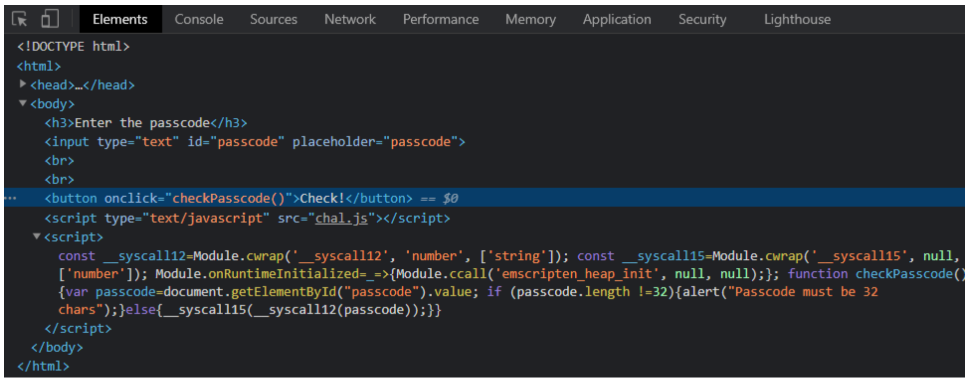


Fig1

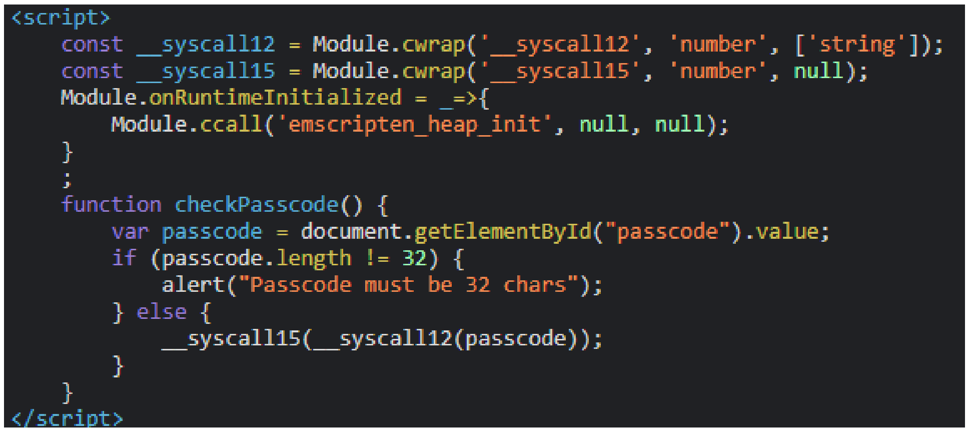
Let's check the DevTools to see what this button does.



```
<!DOCTYPE html>
<html>
▸<head>…</head>
▾<body>
    <h3>Enter the passcode</h3>
    <input type="text" id="passcode" placeholder="passcode">
    <br>
    <br>
    <button onclick="checkPasscode()">Check!</button> == $0
    <script type="text/javascript" src="chal.js"></script>
  ▾<script>
      const __syscall12=Module.cwrap('__syscall12', 'number', ['string']); const __syscall15=Module.cwrap('__syscall15', null,
      ['number']); Module.onRuntimeInitialized=_=>{Module.ccall('emscripten_heap_init', null, null);}; function checkPasscode()
      {var passcode=document.getElementById("passcode").value; if (passcode.length !=32){alert("Passcode must be 32
      chars");}else{__syscall15(__syscall12(passcode));}}
    </script>
  </body>
</html>
```

Fig2

It calls "checkPasscode" function. We can beautify the JS code using chrome's "PrettyPrint" to make it easier to read.



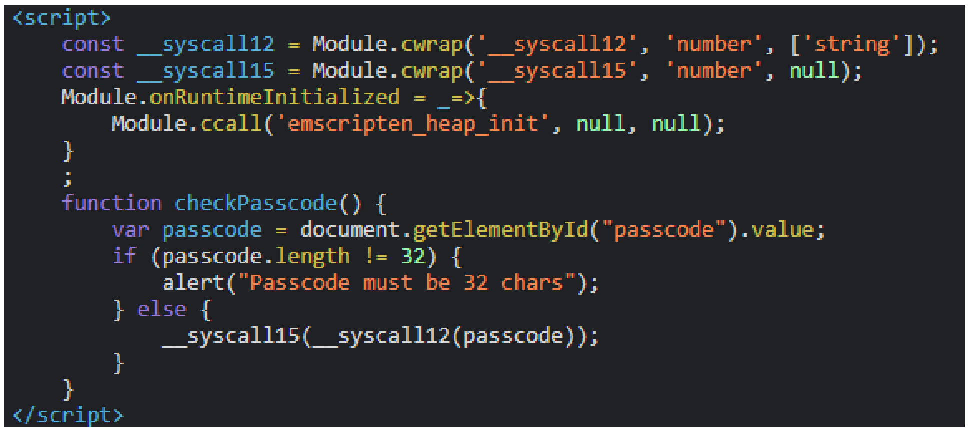
```
<script>
    const __syscall12 = Module.cwrap('__syscall12', 'number', ['string']);
    const __syscall15 = Module.cwrap('__syscall15', 'number', null);
    Module.onRuntimeInitialized = _=>{
        Module.ccall('emscripten_heap_init', null, null);
    }
    ;
    function checkPasscode() {
        var passcode = document.getElementById("passcode").value;
        if (passcode.length != 32) {
            alert("Passcode must be 32 chars");
        } else {
            __syscall15(__syscall12(passcode));
        }
    }
</script>
```

Fig3

So the passcode length must be 32 chars. If that's the case it will call "__syscall12" with passcode as its argument, and then pass the result to "__syscall15".

If we take a look at the page sources we can see 3 files, one of them is "chal.wasm" which is a web assembly file.

The way web assembly works is by exporting functions to be imported and called in javascript.

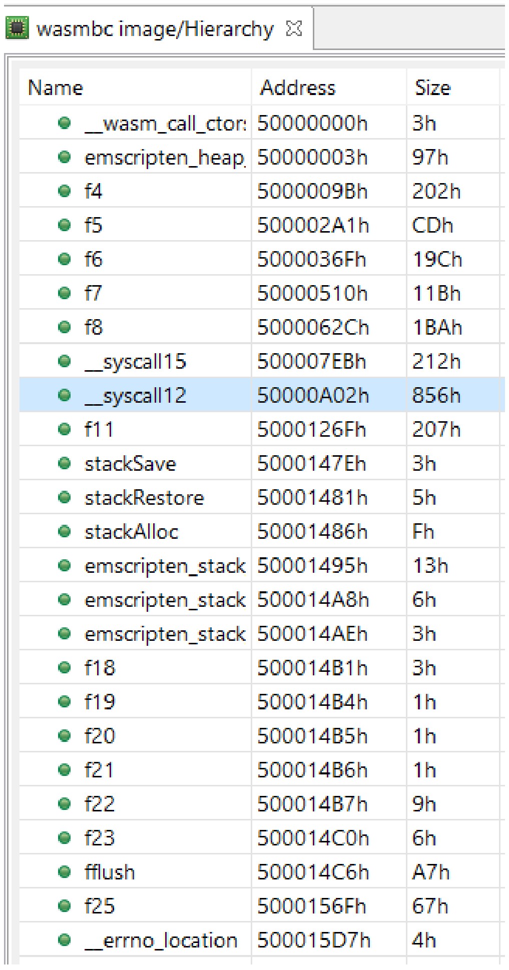


```
<script>
    const __syscall12 = Module.cwrap('__syscall12', 'number', ['string']);
    const __syscall15 = Module.cwrap('__syscall15', 'number', null);
    Module.onRuntimeInitialized = _=>{
        Module.ccall('emscripten_heap_init', null, null);
    }
    ;
    function checkPasscode() {
        var passcode = document.getElementById("passcode").value;
        if (passcode.length != 32) {
            alert("Passcode must be 32 chars");
        } else {
            __syscall15(__syscall12(passcode));
        }
    }
</script>
```

Fig4

We can completely depend on chrome DevTool for the analysis, but here I'm gonna use [JEB Decompiler]() to statically analyze this wasm file.

NOTE: function parameters are reversed in JEB because of the wasm way to push parameters to the stack (first parameter is pushed first unlike x86 calling convention).

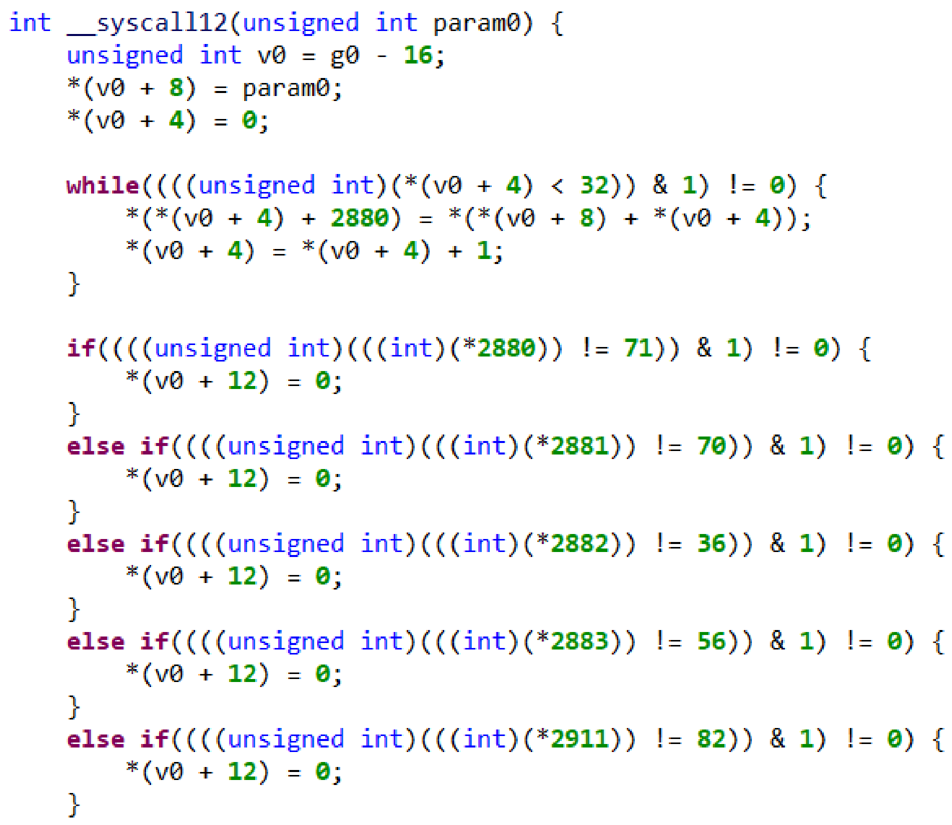


wasmbc image/Hierarchy

Name	Address	Size
__wasm_call_ctor:	50000000h	3h
emscripten_heap_	50000003h	97h
f4	5000009Bh	202h
f5	500002A1h	CDh
f6	5000036Fh	19Ch
f7	50000510h	11Bh
f8	5000062Ch	1BAh
__syscall15	500007EBh	212h
__syscall12	50000A02h	856h
f11	5000126Fh	207h
stackSave	5000147Eh	3h
stackRestore	50001481h	5h
stackAlloc	50001486h	Fh
emscripten_stack	50001495h	13h
emscripten_stack	500014A8h	6h
emscripten_stack	500014AEh	3h
f18	500014B1h	3h
f19	500014B4h	1h
f20	500014B5h	1h
f21	500014B6h	1h
f22	500014B7h	9h
f23	500014C0h	6h
fflush	500014C6h	A7h
f25	5000156Fh	67h
__errno_location	500015D7h	4h

Fig5

JEB did a great job detecting all the functions, so let's start by looking at "__syscall12".



```
int __syscall12(unsigned int param0) {
    unsigned int v0 = g0 - 16;
    *(v0 + 8) = param0;
    *(v0 + 4) = 0;

    while((((unsigned int)(*(v0 + 4) < 32)) & 1) != 0) {
        *(*(v0 + 4) + 2880) = *(*(v0 + 8) + *(v0 + 4));
        *(v0 + 4) = *(v0 + 4) + 1;
    }

    if((((unsigned int)(((int)(*2880)) != 71)) & 1) != 0) {
        *(v0 + 12) = 0;
    }
    else if((((unsigned int)(((int)(*2881)) != 70)) & 1) != 0) {
        *(v0 + 12) = 0;
    }
    else if((((unsigned int)(((int)(*2882)) != 36)) & 1) != 0) {
        *(v0 + 12) = 0;
    }
    else if((((unsigned int)(((int)(*2883)) != 56)) & 1) != 0) {
        *(v0 + 12) = 0;
    }
    else if((((unsigned int)(((int)(*2911)) != 82)) & 1) != 0) {
        *(v0 + 12) = 0;
    }
```

Fig6

This function takes one argument (our passcode) and assigns it to (v0+8), then it loops from 0 to 32 and assigns the stack values starting from address 2912 to that passcode.

From now on I will refer to stack values as "S+offset".

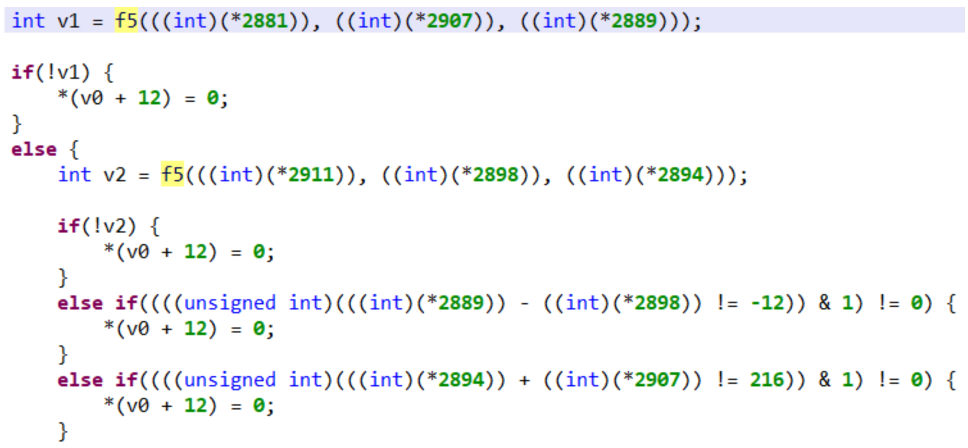
So the passcode starts from "S+2912" to "S+2943".

Next it checks the first 4 chars and last char for the flag format characters.

# Constraint1: S+2912 == 'G'
# Constraint2: S+2913 == 'F'
# Constraint3: S+2914 == '$'
# Constraint4: S+2915 == '8'
# Constraint5: S+2943 == 'R'

Note that (v0+12) is used here as the return value.

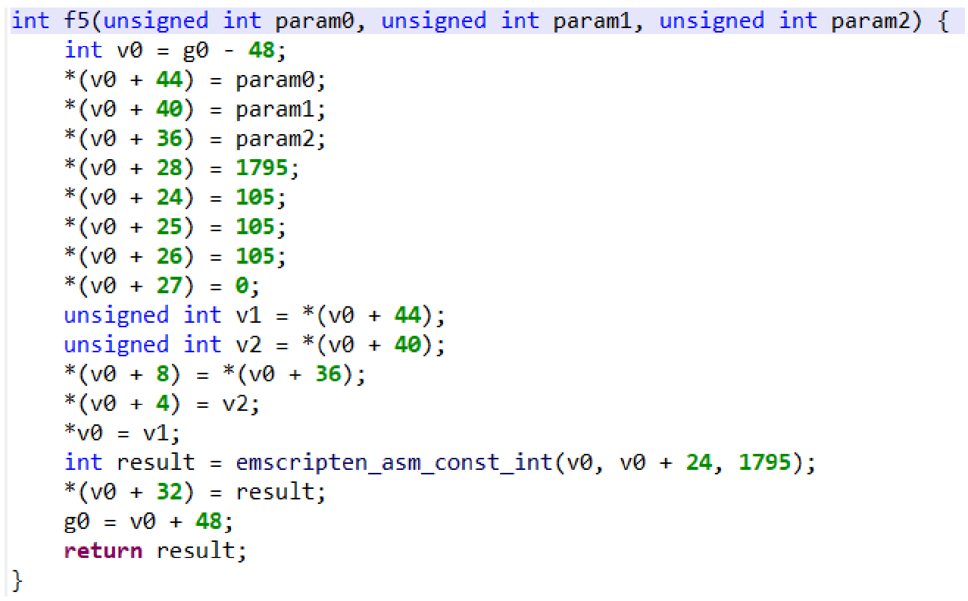
After these checks we see 2 function calls to "f5".



```
int v1 = f5(((int)(*2881)), ((int)(*2907)), ((int)(*2889)));

if(!v1) {
    *(v0 + 12) = 0;
}
else {
    int v2 = f5(((int)(*2911)), ((int)(*2898)), ((int)(*2894)));

    if(!v2) {
        *(v0 + 12) = 0;
    }
    else if((((unsigned int)(((int)(*2889)) - ((int)(*2898)) != -12)) & 1) != 0) {
        *(v0 + 12) = 0;
    }
    else if((((unsigned int)(((int)(*2894)) + ((int)(*2907)) != 216)) & 1) != 0) {
        *(v0 + 12) = 0;
    }
```

Fig7

This function takes 3 arguments and passes them to “emscripten_asm_const_int”.



```
int f5(unsigned int param0, unsigned int param1, unsigned int param2) {
    int v0 = g0 - 48;
    *(v0 + 44) = param0;
    *(v0 + 40) = param1;
    *(v0 + 36) = param2;
    *(v0 + 28) = 1795;
    *(v0 + 24) = 105;
    *(v0 + 25) = 105;
    *(v0 + 26) = 105;
    *(v0 + 27) = 0;
    unsigned int v1 = *(v0 + 44);
    unsigned int v2 = *(v0 + 40);
    *(v0 + 8) = *(v0 + 36);
    *(v0 + 4) = v2;
    *v0 = v1;
    int result = emscripten_asm_const_int(v0, v0 + 24, 1795);
    *(v0 + 32) = result;
    g0 = v0 + 48;
    return result;
}
```

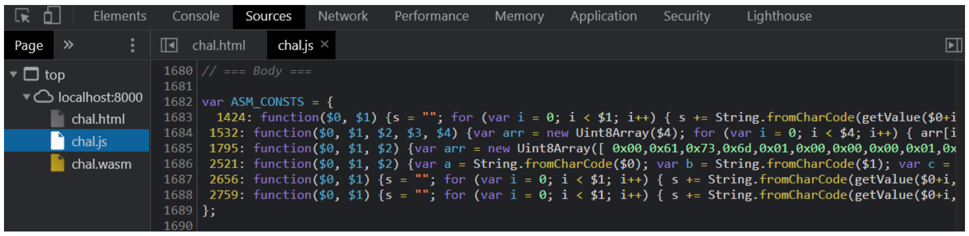
Fig8

Emscripten is a compiler that compiles C and C++ code to web assembly. It provides methods to connect and interact between JavaScript and compiled C or C++ code.

One of these methods is using “EM_ASM” which allows us to call javascript code inside C code.

The function “emscripten_asm_const_int” takes a pointer to arguments as the first parameter, and an index to the javascript function as the third parameter.

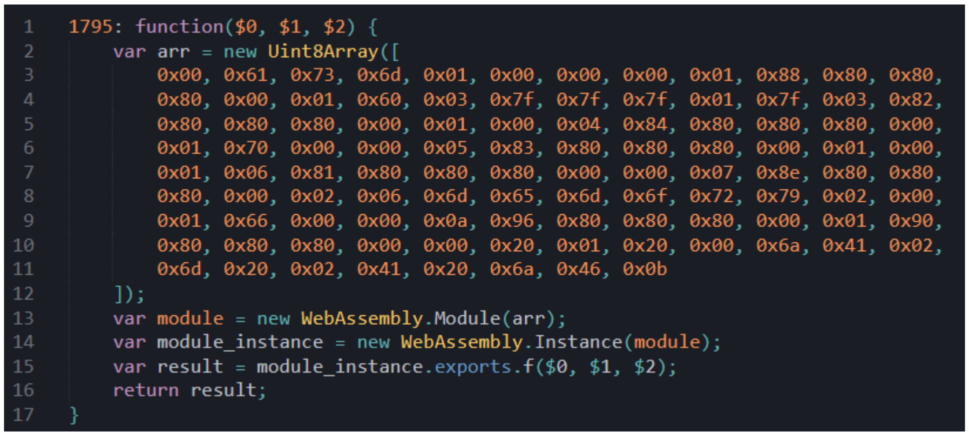
These indexes are mapped to javascript functions in a variable called “ASM_CONSTS” inside “chal.js” file.



```
// === Body ===

var ASM_CONSTS = {
  1424: function($0, $1) {s = ""; for (var i = 0; i < $1; i++) { s += String.fromCharCode(getValue($0+i
 1532: function($0, $1, $2, $3, $4) {var arr = new Uint8Array($4); for (var i = 0; i < $4; i++) { arr[i
 1795: function($0, $1, $2) {var arr = new Uint8Array([ 0x00,0x61,0x73,0x6d,0x01,0x00,0x00,0x00,0x01,0x
 2521: function($0, $1, $2) {var a = String.fromCharCode($0); var b = String.fromCharCode($1); var c =
 2656: function($0, $1) {s = ""; for (var i = 0; i < $1; i++) { s += String.fromCharCode(getValue($0+i,
 2759: function($0, $1) {s = ""; for (var i = 0; i < $1; i++) { s += String.fromCharCode(getValue($0+i,
};
```

Fig9

Now let’s take a closer look at our desired function at index 1795.

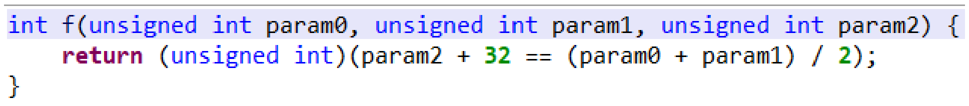


```
1795: function($0, $1, $2) {
    var arr = new Uint8Array([
        0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, 0x01, 0x88, 0x80, 0x80,
        0x80, 0x00, 0x01, 0x60, 0x03, 0x7f, 0x7f, 0x7f, 0x01, 0x7f, 0x03, 0x82,
        0x80, 0x80, 0x80, 0x00, 0x01, 0x00, 0x04, 0x84, 0x80, 0x80, 0x80, 0x00,
        0x01, 0x70, 0x00, 0x00, 0x05, 0x83, 0x80, 0x80, 0x80, 0x00, 0x01, 0x00,
        0x01, 0x06, 0x81, 0x80, 0x80, 0x80, 0x00, 0x00, 0x07, 0x8e, 0x80, 0x80,
        0x80, 0x00, 0x02, 0x06, 0x6d, 0x65, 0x6d, 0x6f, 0x72, 0x79, 0x02, 0x00,
        0x01, 0x66, 0x00, 0x00, 0x0a, 0x96, 0x80, 0x80, 0x80, 0x00, 0x01, 0x90,
        0x80, 0x80, 0x80, 0x00, 0x00, 0x20, 0x01, 0x20, 0x00, 0x6a, 0x41, 0x02,
        0x6d, 0x20, 0x02, 0x41, 0x20, 0x6a, 0x46, 0x0b
    ]);
    var module = new WebAssembly.Module(arr);
    var module_instance = new WebAssembly.Instance(module);
    var result = module_instance.exports.f($0, $1, $2);
    return result;
}
```

Fig10

It initializes a long byte array then creates a new instance of WebAssembly Module with this array as its argument. This is used to compile and execute byte arrays of WASM at runtime.

To analyze these raw bytes, we can save them as a wasm file and use JEB for the analysis.



```
int f(unsigned int param0, unsigned int param1, unsigned int param2) {
    return (unsigned int)(param2 + 32 == (param0 + param1) / 2);
}
```

Fig11

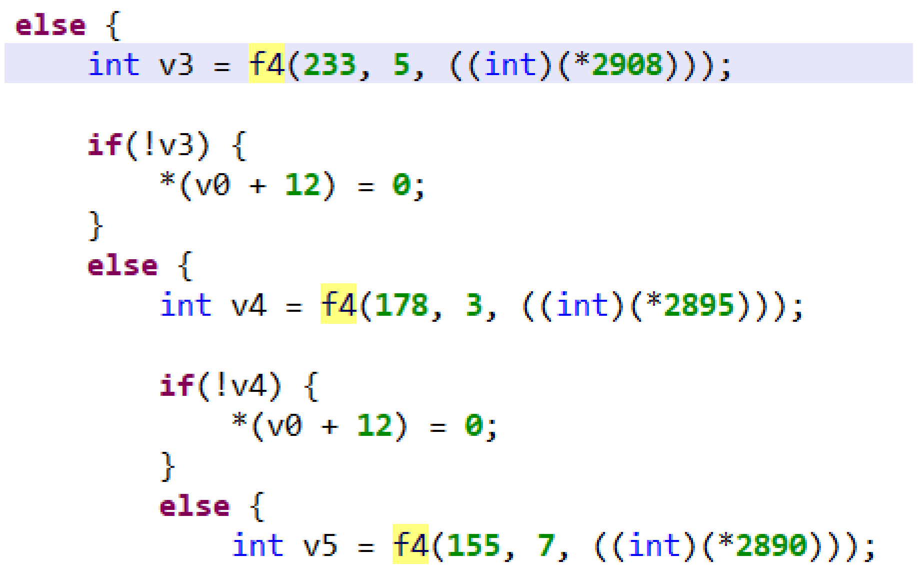
That’s much better, we just need to look back at the parameter of “f5” function to see what offsets of the passcode gets passed to the function above (see Fig7).

# Constraint6: ((S+2881) + 32) == (((S+2889) + (S+2907)) / 2)
# Constraint7: ((S+2911) + 32) == (((S+2894) + (S+2898)) / 2)

We have 2 more constraints at the end of Fig7.

# Constraint8: ((S+2889) - (S+2898)) == -12
# Constraint9: ((S+2894) + (S+2907)) == 216

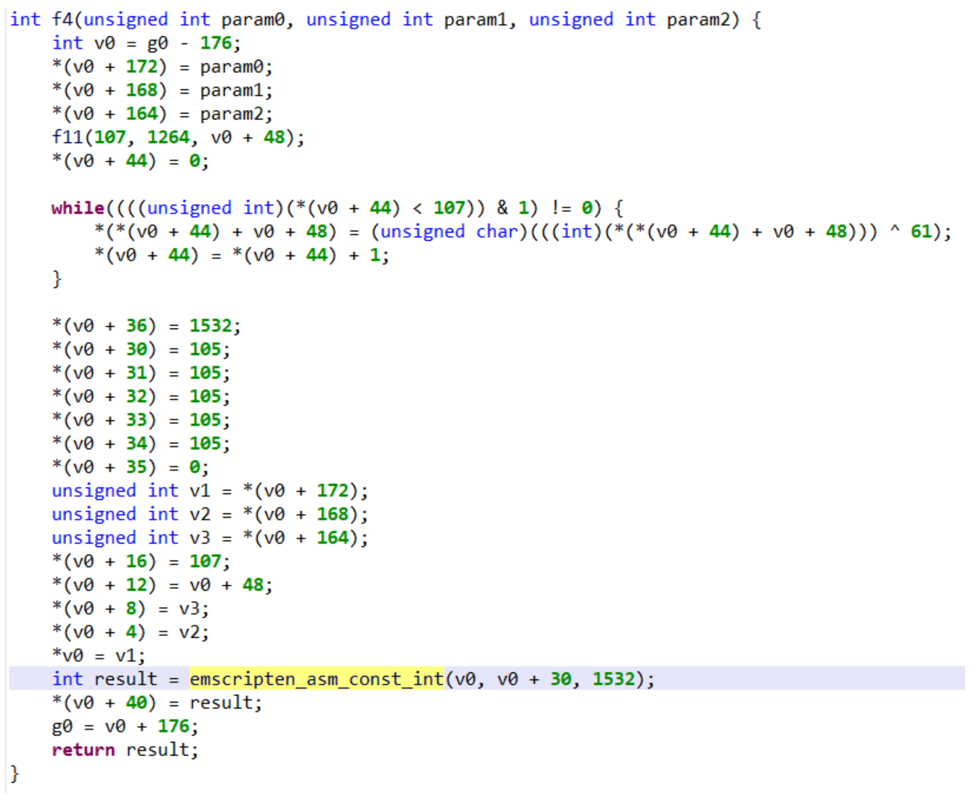
Moving on, we have 3 function calls to “f4”.



```
else {
    int v3 = f4(233, 5, ((int)(*2908)));

    if(!v3) {
        *(v0 + 12) = 0;
    }
    else {
        int v4 = f4(178, 3, ((int)(*2895)));

        if(!v4) {
            *(v0 + 12) = 0;
        }
        else {
            int v5 = f4(155, 7, ((int)(*2890)));
```

Fig11

This function takes 3 arguments and passes them along with other 2 arguments to "emscripten_asm_const_int".



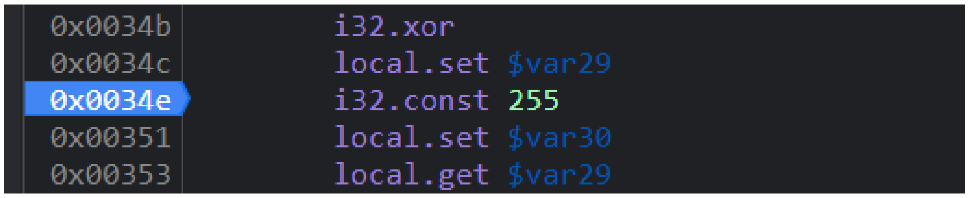
```
int f4(unsigned int param0, unsigned int param1, unsigned int param2) {
    int v0 = g0 - 176;
    *(v0 + 172) = param0;
    *(v0 + 168) = param1;
    *(v0 + 164) = param2;
    f11(107, 1264, v0 + 48);
    *(v0 + 44) = 0;

    while((((unsigned int)(*(v0 + 44) < 107)) & 1) != 0) {
        *(*(v0 + 44) + v0 + 48) = (unsigned char)(((int)(*(*(v0 + 44) + v0 + 48))) ^ 61);
        *(v0 + 44) = *(v0 + 44) + 1;
    }

    *(v0 + 36) = 1532;
    *(v0 + 30) = 105;
    *(v0 + 31) = 105;
    *(v0 + 32) = 105;
    *(v0 + 33) = 105;
    *(v0 + 34) = 105;
    *(v0 + 35) = 0;
    unsigned int v1 = *(v0 + 172);
    unsigned int v2 = *(v0 + 168);
    unsigned int v3 = *(v0 + 164);
    *(v0 + 16) = 107;
    *(v0 + 12) = v0 + 48;
    *(v0 + 8) = v3;
    *(v0 + 4) = v2;
    *v0 = v1;
    int result = emscripten_asm_const_int(v0, v0 + 30, 1532);
    *(v0 + 40) = result;
    g0 = v0 + 176;
    return result;
}
```

Fig12

The first of these 2 arguments is (v0+48) which seems to be a byte array by looking at the while loop at the start of the function. It loops from 0 to 107 and xors the contents of (v0+48) with the key 61.

And the last argument is 107 which is the length of (v0+48) byte array.

We can use the debugger to get the final value of (v0+48). The best place to set a breakpoint would be after the xor operation.



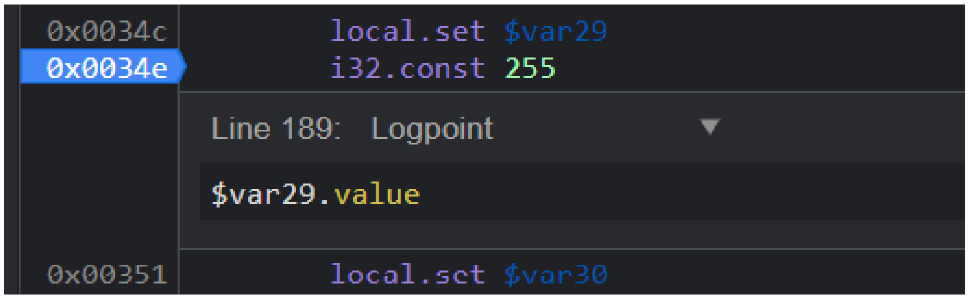
```
0x0034b         i32.xor
0x0034c         local.set $var29
0x0034e         i32.const 255
0x00351         local.set $var30
0x00353         local.get $var29
```

Fig13

The xor result will be stored in “$var29”, but we can’t just stop after every hit of the breakpoint and print that value (107 times).

What we need is a “Logpoint” instead of the regular breakpoint, this allows us to log a specific value every time the breakpoint is hit.

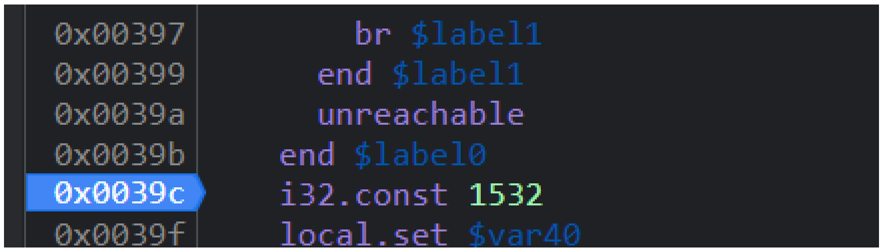
We can double click on the breakpoint and make it log “$var29.value”.



```
0x0034c         local.set $var29
0x0034e         i32.const 255
           Line 189:  Logpoint
           $var29.value
0x00351         local.set $var30
```

Fig14

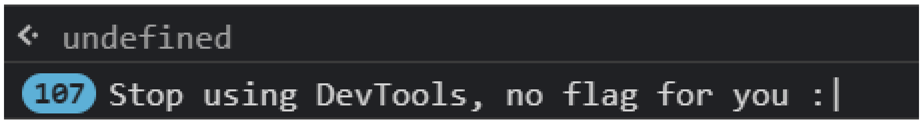
We also need to set another breakpoint at the end of the loop (as “f4” function is called 3 times and we just need to print the array once).



```
0x00397           br $label1
0x00399         end $label1
0x0039a         unreachable
0x0039b       end $label0
0x0039c       i32.const 1532
0x0039f       local.set $var40
```

Fig15

If we entered a random passcode and run the debugger, we will get this message.

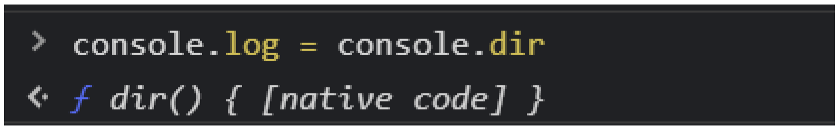


```
undefined
107 Stop using DevTools, no flag for you :|
```

Fig16

Well, looks like “console.log” is hooked (anti-debugging trick).

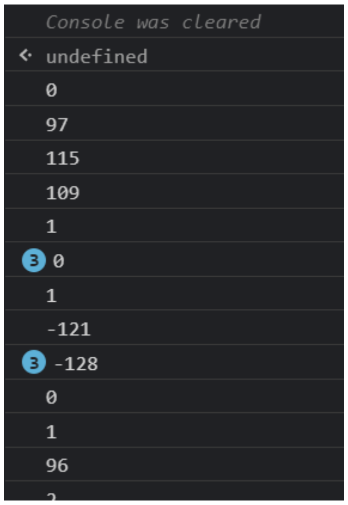
A work around for this is to replace “console.log” with another similar function, here I will use “console.dir”.



```
> console.log = console.dir
<· ƒ dir() { [native code] }
```

Fig17

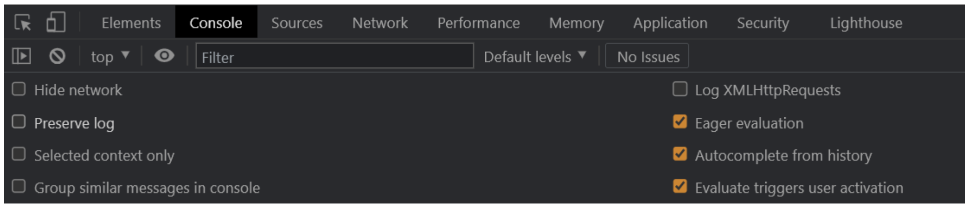
Now let’s clear the console and run the debugger again.



```
  Console was cleared
<· undefined
  0
  97
  115
  109
  1
3 0
  1
  -121
3 -128
  0
  1
  96
```

Fig18

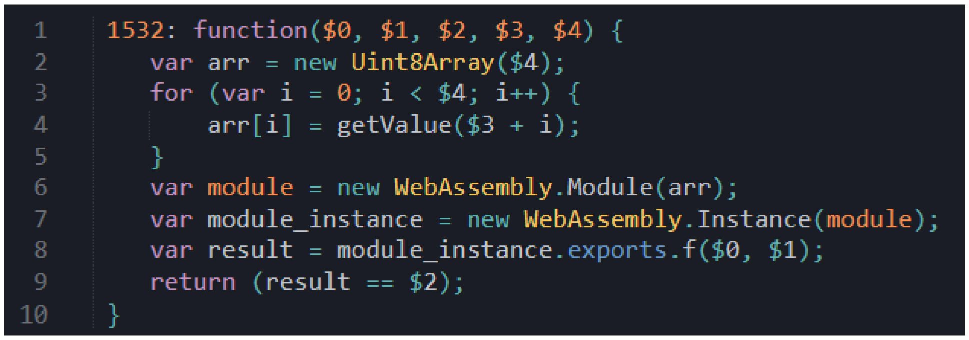
Nice! Here is our lovely array, we can disable similar messages grouping in console to get the full list of values.



```
☐ Hide network                      ☐ Log XMLHttpRequests
☐ Preserve log                      ☑ Eager evaluation
☐ Selected context only             ☑ Autocomplete from history
☐ Group similar messages in console ☑ Evaluate triggers user activation
```

Fig19

Looking back to Fig12, we see that the arguments are passed to the function with index 1532 (remember the ASM_CONSTS map).

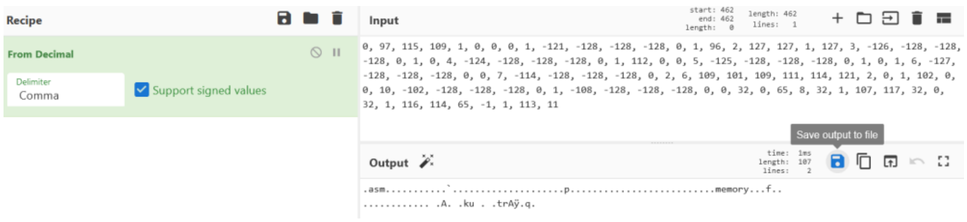


```
1532: function($0, $1, $2, $3, $4) {
    var arr = new Uint8Array($4);
    for (var i = 0; i < $4; i++) {
        arr[i] = getValue($3 + i);
    }
    var module = new WebAssembly.Module(arr);
    var module_instance = new WebAssembly.Instance(module);
    var result = module_instance.exports.f($0, $1);
    return (result == $2);
}
```

Fig20

This function treats the third argument (the array we dumped earlier) as wasm instructions, so we just save that array as a wasm file and throw it into JEB.

We can use CyberChef to handle the negative values for us.

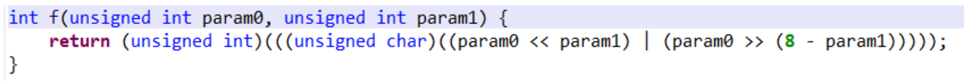


```
0, 97, 115, 109, 1, 0, 0, 0, 1, -121, -128, -128, -128, 0, 1, 96, 2, 127, 127, 1, 127, 3, -126, -128, -128,
-128, 0, 1, 0, 4, -124, -128, -128, -128, 0, 1, 112, 0, 0, 5, -125, -128, -128, -128, 0, 1, 0, 1, 6, -127,
-128, -128, -128, 0, 0, 7, -114, -128, -128, -128, 0, 2, 6, 109, 101, 109, 111, 114, 121, 2, 0, 1, 102, 0,
0, 10, -102, -128, -128, -128, 0, 1, -108, -128, -128, -128, 0, 0, 32, 0, 65, 8, 32, 1, 107, 117, 32, 0,
32, 1, 116, 114, 65, -1, 1, 113, 11
```

```
.asm..........`....................p.........................memory...f..
............ .A. .ku . .trAÿ.q.
```

Fig21

Here is the result.



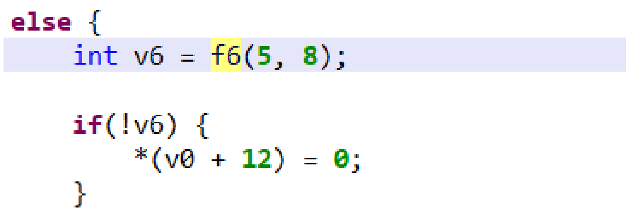
```
int f(unsigned int param0, unsigned int param1) {
    return (unsigned int)(((unsigned char)((param0 << param1) | (param0 >> (8 - param1)))));
}
```

Fig22

This is actually the implementation of “Rotate Left” operation, so it’s rotating “param0” to left by “param1” steps.

# Constraint10: RotateLeft((S+2908), 5) == 233
# Constraint11: RotateLeft((S+2895), 3) == 178
# Constraint12: RotateLeft((S+2890), 7) == 155

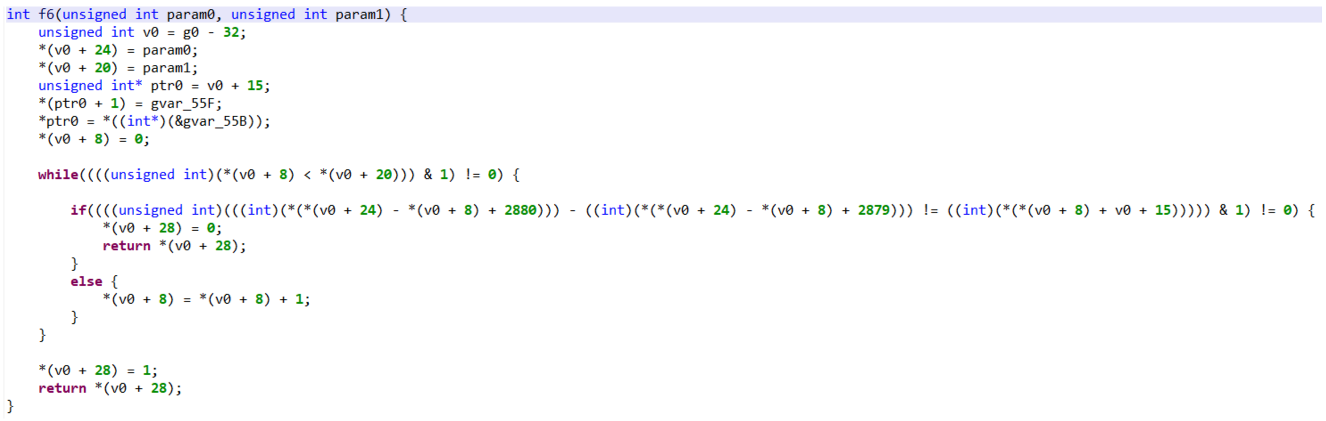
The next function is “f6” which takes 2 arguments.



```
else {
    int v6 = f6(5, 8);

    if(!v6) {
        *(v0 + 12) = 0;
    }
```

Fig23

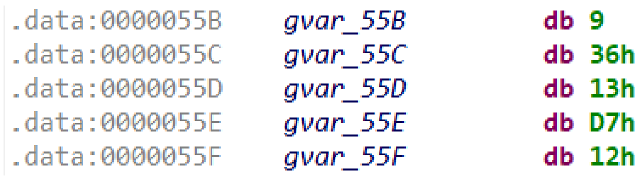
This function looks a little bit tricky.



```
int f6(unsigned int param0, unsigned int param1) {
    unsigned int v0 = g0 - 32;
    *(v0 + 24) = param0;
    *(v0 + 20) = param1;
    unsigned int* ptr0 = v0 + 15;
    *(ptr0 + 1) = gvar_55F;
    *ptr0 = *((int*)(&gvar_55B));
    *(v0 + 8) = 0;

    while((((unsigned int)(*(v0 + 8) < *(v0 + 20))) & 1) != 0) {

        if((((unsigned int)(((int)(*(*(v0 + 24) - *(v0 + 8) + 2880))) - ((int)(*(*(v0 + 24) - *(v0 + 8) + 2879))) != ((int)(*(*(v0 + 8) + v0 + 15))))) & 1) != 0) {
            *(v0 + 28) = 0;
            return *(v0 + 28);
        }
        else {
            *(v0 + 8) = *(v0 + 8) + 1;
        }
    }

    *(v0 + 28) = 1;
    return *(v0 + 28);
}
```

Fig24

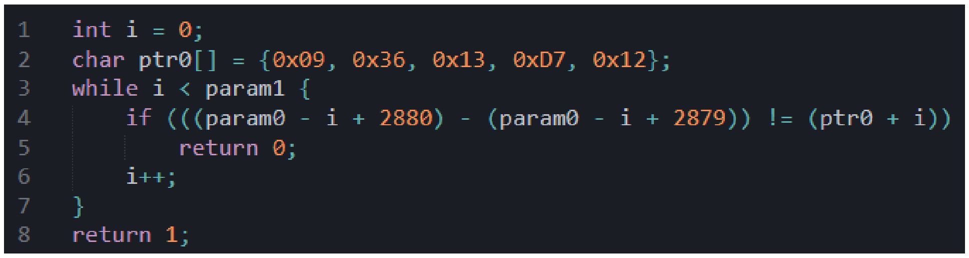
First it sets (v0+24) to param0 and (v0+20) to param1, then creates a pointer "ptr0" which points to (v0+15) and copies 5 bytes from address 0x55B to the pointer memory.



```
.data:0000055B     gvar_55B          db 9
.data:0000055C     gvar_55C          db 36h
.data:0000055D     gvar_55D          db 13h
.data:0000055E     gvar_55E          db D7h
.data:0000055F     gvar_55F          db 12h
```

Fig25

We can rewrite this function in C and rename those (v0+offset) variables to make it more clear.



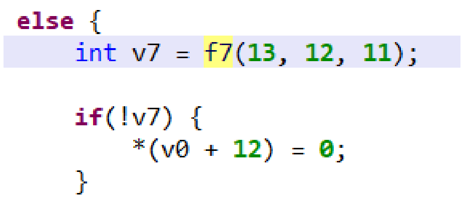
```
int i = 0;
char ptr0[] = {0x09, 0x36, 0x13, 0xD7, 0x12};
while i < param1 {
    if (((param0 - i + 2880) - (param0 - i + 2879)) != (ptr0 + i))
        return 0;
    i++;
}
return 1;
```

Fig26

That's definitely better.

# Constraint13: (S+2880+8-0) - (S+2879+8-0) == 0x09
# Constraint14: (S+2880+8-1) - (S+2879+8-1) == 0x36
# Constraint15: (S+2880+8-2) - (S+2879+8-2) == 0x13
# Constraint16: (S+2880+8-3) - (S+2879+8-3) == 0xD7
# Constraint17: (S+2880+8-4) - (S+2879+8-4) == 0x12

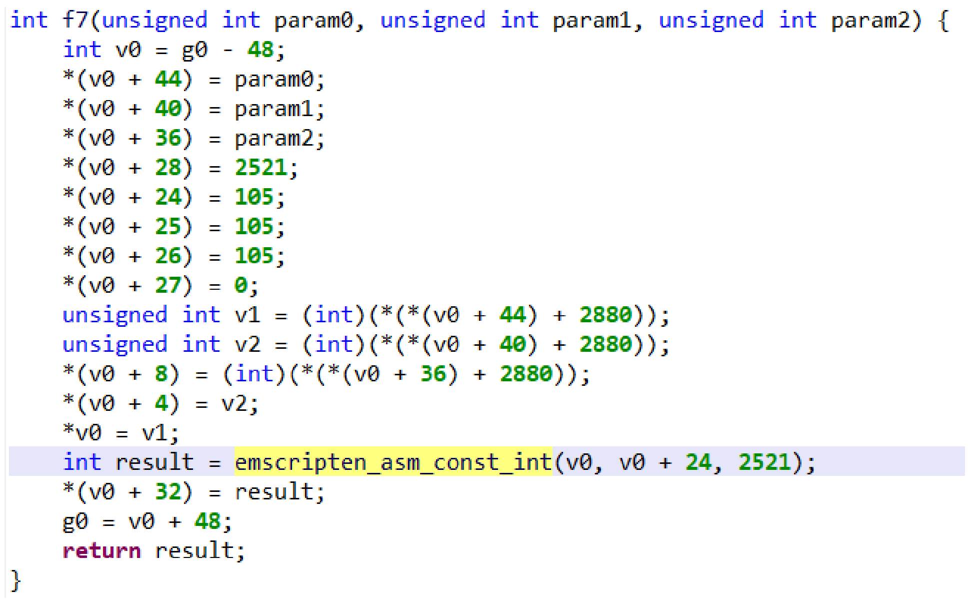
Let's keep moving, the next function is "f7".



```
else {
    int v7 = f7(13, 12, 11);

    if(!v7) {
        *(v0 + 12) = 0;
    }
```

Fig27

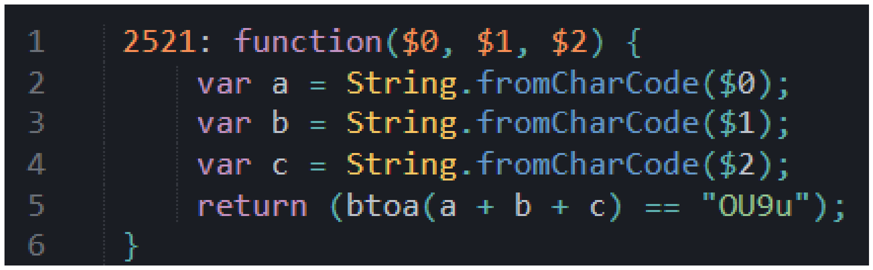
It takes 3 arguments, adds them to the passcode stack offset then passes them to "emscripten_asm_const_int".



```
int f7(unsigned int param0, unsigned int param1, unsigned int param2) {
    int v0 = g0 - 48;
    *(v0 + 44) = param0;
    *(v0 + 40) = param1;
    *(v0 + 36) = param2;
    *(v0 + 28) = 2521;
    *(v0 + 24) = 105;
    *(v0 + 25) = 105;
    *(v0 + 26) = 105;
    *(v0 + 27) = 0;
    unsigned int v1 = (int)(*(*(v0 + 44) + 2880));
    unsigned int v2 = (int)(*(*(v0 + 40) + 2880));
    *(v0 + 8) = (int)(*(*(v0 + 36) + 2880));
    *(v0 + 4) = v2;
    *v0 = v1;
    int result = emscripten_asm_const_int(v0, v0 + 24, 2521);
    *(v0 + 32) = result;
    g0 = v0 + 48;
    return result;
}
```

Fig28

The javascript function at index 2521 is quite simple.



```
2521: function($0, $1, $2) {
    var a = String.fromCharCode($0);
    var b = String.fromCharCode($1);
    var c = String.fromCharCode($2);
    return (btoa(a + b + c) == "OU9u");
}
```

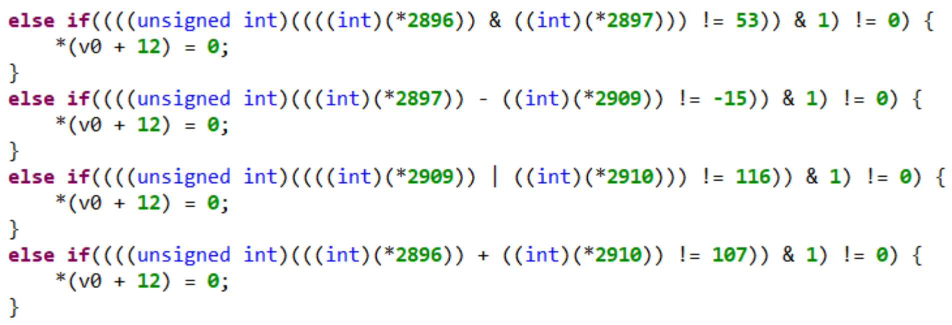
Fig29

It concatenates the 3 arguments then uses “btoa” function to base64-encode them and compares the result with the string “OU9u”.

So we just need to base64-decode this string to get the correct values.

# Constraint18: (S+2880+11) == '9'
# Constraint19: (S+2880+12) == 'O'
# Constraint20: (S+2880+13) == 'n'

Next we have some bits operations.

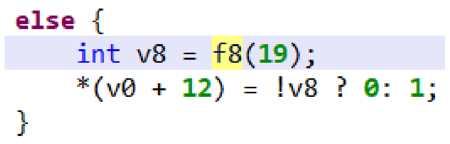


```
else if((((unsigned int)((((int)(*2896)) & ((int)(*2897))) != 53)) & 1) != 0) {
    *(v0 + 12) = 0;
}
else if((((unsigned int)(((int)(*2897)) - ((int)(*2909)) != -15)) & 1) != 0) {
    *(v0 + 12) = 0;
}
else if((((unsigned int)((((int)(*2909)) | ((int)(*2910))) != 116)) & 1) != 0) {
    *(v0 + 12) = 0;
}
else if((((unsigned int)(((int)(*2896)) + ((int)(*2910)) != 107)) & 1) != 0) {
    *(v0 + 12) = 0;
}
```

Fig30

# Constraint21: ((S+2896) & (S+2897)) == 53
# Constraint22: ((S+2897) - (S+2909)) == -15
# Constraint23: ((S+2909) | (S+2910)) == 116
# Constraint24: ((S+2896) + (S+2910)) == 107

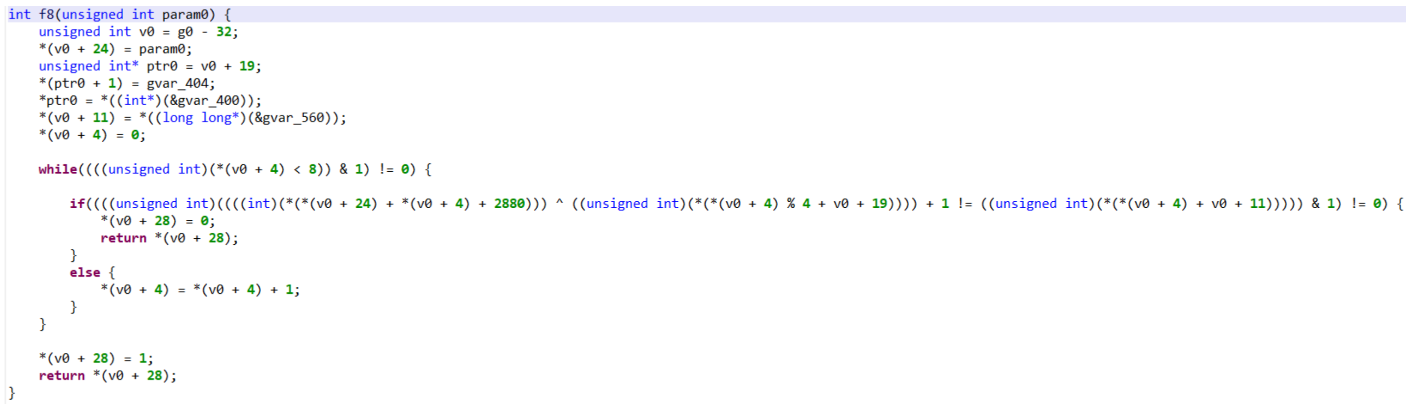
The last checking function is “f8”.



```
else {
    int v8 = f8(19);
    *(v0 + 12) = !v8 ? 0: 1;
}
```

Fig31

This function looks a lot like “f6”.

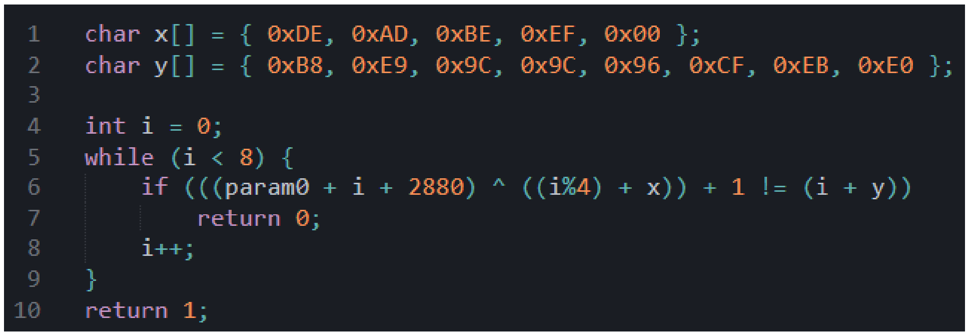


```
int f8(unsigned int param0) {
    unsigned int v0 = g0 - 32;
    *(v0 + 24) = param0;
    unsigned int* ptr0 = v0 + 19;
    *(ptr0 + 1) = gvar_404;
    *ptr0 = *((int*)(&gvar_400));
    *(v0 + 11) = *((long long*)(&gvar_560));
    *(v0 + 4) = 0;

    while((((unsigned int)(*(v0 + 4) < 8)) & 1) != 0) {

        if((((unsigned int)((((int)(*(*(v0 + 24) + *(v0 + 4) + 2880))) ^ ((unsigned int)(*(*(v0 + 4) % 4 + v0 + 19)))) + 1 != ((unsigned int)(*(*(v0 + 4) + v0 + 11))))) & 1) != 0) {
            *(v0 + 28) = 0;
            return *(v0 + 28);
        }
        else {
            *(v0 + 4) = *(v0 + 4) + 1;
        }
    }

    *(v0 + 28) = 1;
    return *(v0 + 28);
}
```

Fig32

First it sets (v0+24) to param0, stores from bytes from address 0x400 in (v0+19) and stores 8 bytes from address 0x560 tin (v0+11).

Let’s rewrite it in C as we did with “f6”.

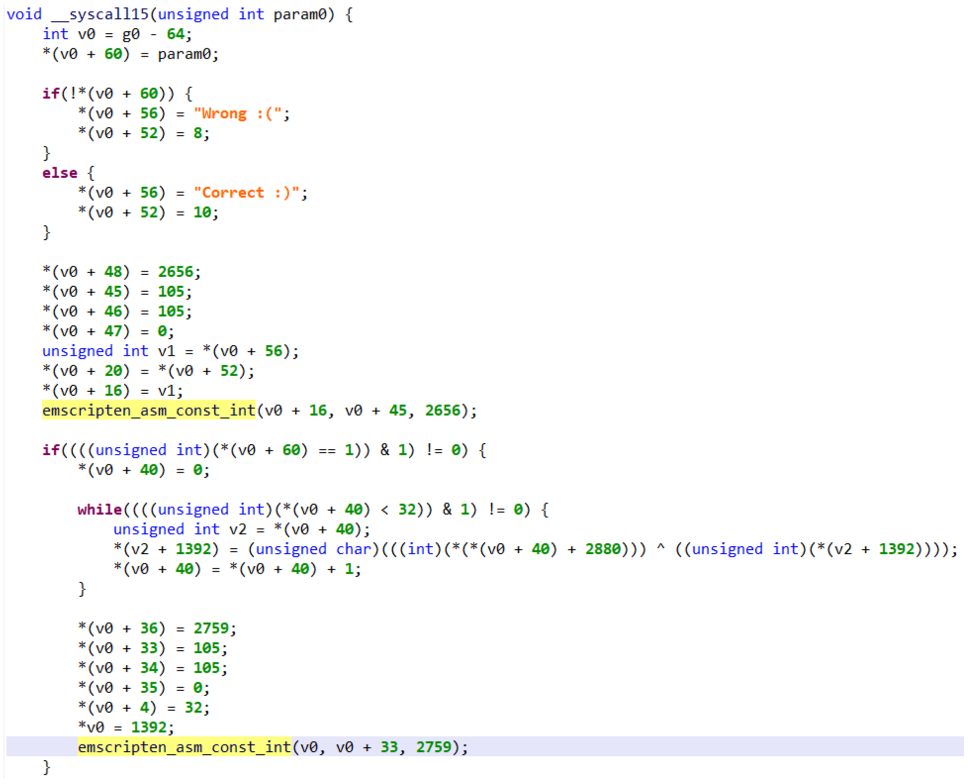


```
char x[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0x00 };
char y[] = { 0xB8, 0xE9, 0x9C, 0x9C, 0x96, 0xCF, 0xEB, 0xE0 };

int i = 0;
while (i < 8) {
    if (((param0 + i + 2880) ^ ((i%4) + x)) + 1 != (i + y))
        return 0;
    i++;
}
return 1;
```

Fig33

# Constraint25: ((S+2880+19) ^ 0xDE) + 1 == 0xB8
# Constraint26: ((S+2880+20) ^ 0xAD) + 1 == 0xE9
# Constraint27: ((S+2880+21) ^ 0xBE) + 1 == 0x9C
# Constraint28: ((S+2880+22) ^ 0xEF) + 1 == 0x9C
# Constraint29: ((S+2880+23) ^ 0xDE) + 1 == 0x96
# Constraint30: ((S+2880+24) ^ 0xAD) + 1 == 0xCF
# Constraint31: ((S+2880+25) ^ 0xBE) + 1 == 0xEB
# Constraint32: ((S+2880+26) ^ 0xEF) + 1 == 0xE0

And with that we are done with "__syscall12", now let's check "__syscall15" to see if there's anything left.



```
void __syscall15(unsigned int param0) {
    int v0 = g0 - 64;
    *(v0 + 60) = param0;

    if(!*(v0 + 60)) {
        *(v0 + 56) = "Wrong :(";
        *(v0 + 52) = 8;
    }
    else {
        *(v0 + 56) = "Correct :)";
        *(v0 + 52) = 10;
    }

    *(v0 + 48) = 2656;
    *(v0 + 45) = 105;
    *(v0 + 46) = 105;
    *(v0 + 47) = 0;
    unsigned int v1 = *(v0 + 56);
    *(v0 + 20) = *(v0 + 52);
    *(v0 + 16) = v1;
    emscripten_asm_const_int(v0 + 16, v0 + 45, 2656);

    if((((unsigned int)(*(v0 + 60) == 1)) & 1) != 0) {
        *(v0 + 40) = 0;

        while((((unsigned int)(*(v0 + 40) < 32)) & 1) != 0) {
            unsigned int v2 = *(v0 + 40);
            *(v2 + 1392) = (unsigned char)(((int)(*(*(v0 + 40) + 2880))) ^ ((unsigned int)(*(v2 + 1392))));
            *(v0 + 40) = *(v0 + 40) + 1;
        }

        *(v0 + 36) = 2759;
        *(v0 + 33) = 105;
        *(v0 + 34) = 105;
        *(v0 + 35) = 0;
        *(v0 + 4) = 32;
        *v0 = 1392;
        emscripten_asm_const_int(v0, v0 + 33, 2759);
    }
```

Fig34

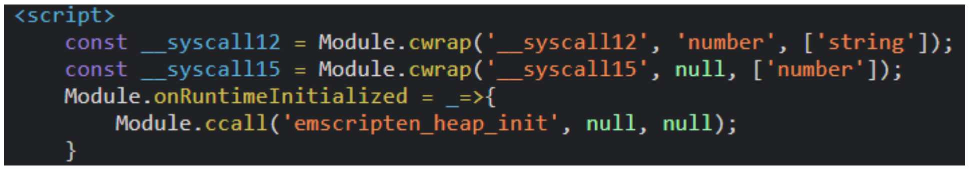
Cool, it just checks If param0 (the return value of "__syscall12") is equal to 1 (should be the correct passcode) and decrypts the flag using our passcode.

The javascript function at index 2656 takes one argument and calls "alert", while the function at index 2759 also takes one argument but calls "console.log" (remember that "console.log" is hooked as an anti-debugging trick).

To get the correct passcode using these constraints, we can use Z3 SMT Solver and give it our constraints.

BONUS:

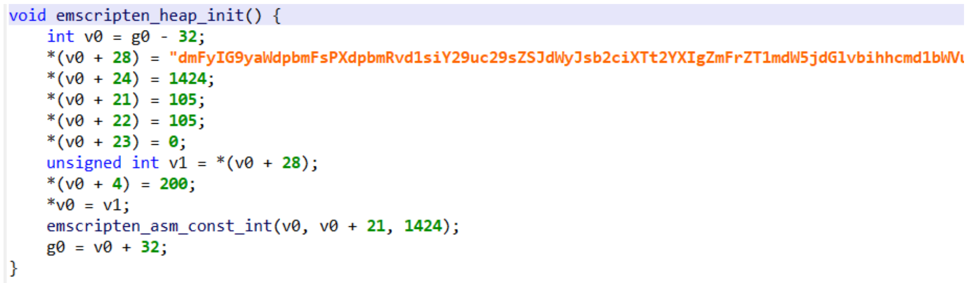
The function responsible for the anti-debugging trick (hooking "console.log") is called right after the page is loaded.



```
<script>
    const __syscall12 = Module.cwrap('__syscall12', 'number', ['string']);
    const __syscall15 = Module.cwrap('__syscall15', null, ['number']);
    Module.onRuntimeInitialized = _=>{
        Module.ccall('emscripten_heap_init', null, null);
    }
```

Fig35

It has a misleading name "emscripten_heap_init".

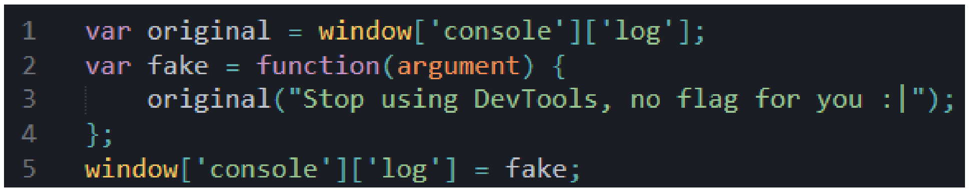


```
void emscripten_heap_init() {
    int v0 = g0 - 32;
    *(v0 + 28) = "dmFyIG9yaWdpbmFsPXdpbmRvd1siY29uc29sZSJdWyJsb2ciXTt2YXIgZmFrZT1mdW5jdGlvbihhcmd1bWVu
    *(v0 + 24) = 1424;
    *(v0 + 21) = 105;
    *(v0 + 22) = 105;
    *(v0 + 23) = 0;
    unsigned int v1 = *(v0 + 28);
    *(v0 + 4) = 200;
    *v0 = v1;
    emscripten_asm_const_int(v0, v0 + 21, 1424);
    g0 = v0 + 32;
}
```

Fig36

It contains a large base64-encoded strings that gets decoded and passed to "eval" function (this is implemented in the javascript function at index 1424).

This is how it looks after decoding.



```
var original = window['console']['log'];
var fake = function(argument) {
    original("Stop using DevTools, no flag for you :|");
};
window['console']['log'] = fake;
```

Fig37

Passcode: GF$8J!4jsf79OnrV75riE%tKcT0fOD4R

Flag: S3D{w3b_4ss3mb1y_1s_c00l_r1ght?}

Abdallah Elshinbary (@NightW0o0lf) from The Crafters (@CTFCreators)