The Watness III

Writeup by hpp3
http://watness.pwni.ng/

The page contains a WebGL application, which is a recreation of the indie hit puzzle game The Witness. Just like in The Witness, there is a puzzle board on which we can draw a path. In The Witness, there were usually requirements that the path had to fulfill in order for a solve to be valid. Unfortunately we don't have any information about what path to use. Let's check the source code.

main.js

main.js contains a bunch of cryptography code (lines 1 to 400; line numbers are from the pretty printed version), a main function at line 459, and the main game event loop at line 514. This code snippet at line 591 also looks very interesting:

```
        t.decode = e=>{
                  const t = [17, 34, 51, 68, 85, 102, 119];
                  for (let r = 0; r < 18; r += 2)
                      t[r + 7] = 16 * e[r] + e[r];
                  const r = new Uint8Array(t)
                    , i = new Uint8Array([7, 99, 44, 37, 40, 240, 88, 3, 69, 107, 162,
242, 120, 37, 105, 17])
```

```
                , o =
a.utils.hex.toBytes("d00bdd332962b071daf3bd798cc52c860dc5720bcc3a9f79ff714ec4ba10df504
d0d21aec15aa521788da8933b24c970")
                , n = new a.ModeOfOperation.cbc(r,i).decrypt(o);
              return String.fromCharCode(...n)
          }
```

Let's see where "decode" is being used. Line 533-540:

```
            p = e.readPixels(0, 0, 200),
                !_ && p[150][0]) {
                    const e = p.slice(130, 148).map((([e])=>e))
                      , t = l.decode(e);
                    console.log(t),
                    alert(t),
                    _ = !0
                }
```

It looks like when some condition becomes true, then some data gets automatically used to decode the flag, which is then shown to the user. So what's the data? p comes from readPixels, which reads the pixel values at the given area. readPixels(0, 0, 200, i=1) means read from the bottom left corner (0, 0) an area that is 200 wide and 1 tall. It would be useful to see what values are already here. I used Chrome's Local Overrides feature to edit the JS and save the result locally. When the page is reloaded, rather than fetch that source file from the actual site it uses my local copy instead. This feature is extremely useful for debugging.

Right after the call to readPixel on 533, I inserted this line:

```
        console.log(p.slice(0, 151).map((x,i)=>(i) + ": " + x.toString())).join("; ")),
```

The output in the console looks like this:

```
0: 0,0,1,0; 1: 255,255,1,0; 2: 255,255,1,0; 3: 255,255,1,0; 4: 255,255,1,0; 5:
255,255,1,0; 6: 255,255,1,0; 7: 255,255,1,0; 8: 255,255,1,0; 9: 255,255,1,0; 10:
255,255,1,0; 11: 255,255,1,0; 12: 255,255,1,0; 13: 255,255,1,0; 14: 255,255,1,0; 15:
255,255,1,0; 16: 255,255,1,0; 17: 255,255,1,0; 18: 255,255,1,0; 19: 255,255,1,0; 20:
255,255,1,0; 21: 255,255,1,0; 22: 255,255,1,0; 23: 255,255,1,0; 24: 255,255,1,0; 25:
255,255,1,0; 26: 255,255,1,0; 27: 255,255,1,0; 28: 255,255,1,0; 29: 255,255,1,0; 30:
255,255,1,0; 31: 255,255,1,0; 32: 255,255,1,0; 33: 255,255,1,0; 34: 255,255,1,0; 35:
255,255,1,0; 36: 255,255,1,0; 37: 255,255,1,0; 38: 255,255,1,0; 39: 255,255,1,0; 40:
255,255,1,0; 41: 255,255,1,0; 42: 255,255,1,0; 43: 255,255,1,0; 44: 255,255,1,0; 45:
255,255,1,0; 46: 255,255,1,0; 47: 255,255,1,0; 48: 255,255,1,0; 49: 255,255,1,0; 50:
255,255,1,0; 51: 255,255,1,0; 52: 255,255,1,0; 53: 255,255,1,0; 54: 255,255,1,0; 55:
255,255,1,0; 56: 255,255,1,0; 57: 255,255,1,0; 58: 255,255,1,0; 59: 255,255,1,0; 60:
255,255,1,0; 61: 255,255,1,0; 62: 255,255,1,0; 63: 255,255,1,0; 64: 255,255,1,0; 65:
255,255,1,0; 66: 255,255,1,0; 67: 255,255,1,0; 68: 255,255,1,0; 69: 255,255,1,0; 70:
255,255,1,0; 71: 255,255,1,0; 72: 255,255,1,0; 73: 255,255,1,0; 74: 255,255,1,0; 75:
255,255,1,0; 76: 255,255,1,0; 77: 255,255,1,0; 78: 255,255,1,0; 79: 255,255,1,0; 80:
255,255,1,0; 81: 255,255,1,0; 82: 255,255,1,0; 83: 255,255,1,0; 84: 255,255,1,0; 85:
255,255,1,0; 86: 255,255,1,0; 87: 255,255,1,0; 88: 255,255,1,0; 89: 255,255,1,0; 90:
255,255,1,0; 91: 255,255,1,0; 92: 255,255,1,0; 93: 255,255,1,0; 94: 255,255,1,0; 95:
255,255,1,0; 96: 255,255,1,0; 97: 255,255,1,0; 98: 255,255,1,0; 99: 255,255,1,0; 100:
0,0,0,0; 101: 0,0,0,0; 102: 0,0,0,0; 103: 0,0,0,0; 104: 113,21,16,0; 105: 54,128,11,0;
106: 0,0,0,0; 107: 0,0,0,0; 108: 0,0,0,0; 109: 0,0,0,0; 110: 0,0,0,0; 111: 0,0,0,0;
112: 0,0,0,0; 113: 0,0,0,0; 114: 0,0,0,0; 115: 0,0,0,0; 116: 0,0,0,0; 117: 0,0,0,0;
118: 0,0,0,0; 119: 0,0,0,0; 120: 0,0,0,0; 121: 0,0,0,0; 122: 0,0,0,0; 123: 0,0,0,0;
```

```
124: 0,0,0,0; 125: 0,0,0,0; 126: 0,0,0,0; 127: 0,0,0,0; 128: 0,0,0,0; 129: 0,0,0,0;
130: 0,0,0,0; 131: 0,0,0,0; 132: 0,0,0,0; 133: 0,0,0,0; 134: 0,0,0,0; 135: 0,0,0,0;
136: 0,0,0,0; 137: 0,0,0,0; 138: 0,0,0,0; 139: 0,0,0,0; 140: 0,0,0,0; 141: 0,0,0,0;
142: 0,0,0,0; 143: 0,0,0,0; 144: 0,0,0,0; 145: 0,0,0,0; 146: 0,0,0,0; 147: 0,0,0,0;
148: 0,0,0,0; 149: 255,255,1,0; 150: 0,0,0,0
```

So for some reason, the bottom line of pixels in the canvas is not being rendered normally. Something is setting some weird pixel values. And if you play around in the game with this output running, you'll notice that some of the pixel values seem to correspond to the player's position or view angle. We need to set pixel 150 to 1 in order to trigger the decode function. But we can't just do that manually because the decode function relies on pixels 130-148 having the correct values too.

So we should take a look at the renderer code to see where these special pixel values are coming from.

Renderer Program

The source code is at line 1086, in a mostly obfuscated C-like language. I copied the code into my editor and made some replacements to make the code more readable. Then I went through and renamed macros, variables, and functions based on my best guess for their behavior. This my final (partially) deobfuscated program: https://pastebin.com/1NDr1aeF.

Let's examine the macros:

```
#define MIN(B, C)(B<C?B:C)
#define D(B, E, F)(min(max(B, E), F))
#define LOOPBACK_PACKED_XYZ(H, I)PACK(loopback[int(H)].xyz)*I
#define LOOPBACK_X(H)loopback[int(H)].x
#define LOOPBACK_XY(H)loopback[int(H)].xy
#define LOOPBACK_XYZ(H)loopback[int(H)].xyz
#define LOOPBACK_BOOL(H)(loopback[int(H)].x==0?false:true)
#define TWO_ADJACENT_LOOPBACK_PACKED_XYZ(H, I)vec2(LOOPBACK_PACKED_XYZ(H, I),
LOOPBACK_PACKED_XYZ(int(H)+1, I))
#define BOTTOM_ROW(H, exp)if (gl_FragCoord.y<=1. && gl_FragCoord.x>H &&
gl_FragCoord.x<=H+1.){
    gl_FragColor=exp;
    return;
}

#define SET_UNPACK_DIV(H, R, I) BOTTOM_ROW(H, vec4(UNPACK((R)/(I)), 0.))
#define SET_X_255(H, R) BOTTOM_ROW(H, vec4(float(R)/255., 0, 0, 0))
#define SET_XY_255(H, R) BOTTOM_ROW(H, vec4(float(R.xy)/255., 0, 0))
#define SET_XYZ_255(H, R) BOTTOM_ROW(H, vec4(float(R.x)/255., float(R.y)/255.,
float(R.z)/255., 0))
#define SET_IF(H, R) BOTTOM_ROW(H, vec4(R?1.:0., 0, 0, 0))
#define PI  3.1415
```

Of note is the macro which I called BOTTOM_ROW, which looks like it's responsible for writing values to the bottom row (which is what we're interested in). This macro is used by a bunch of macros that I called "SET_...", which set a bottom-row pixel at a given index to data of various types.
The other group of macros read from the loopback variable. Jumping back to the main.js, I see on line 531 t.set("loopback", p.concat(o)). Recall that p is the result of reading the bottom row of pixels. So each iteration passes the data-containing pixel values back to the rendering program. Thus the SET_... and LOOPBACK_... macros are how this program reads from and writes to the bottom row pixels, which is also how the program takes input, provides output, and stores execution state.

Data Format

The next step is to figure out what data is stored at each index. I noticed that indices 0-99 change when I draw on the puzzle board; specifically, (x,y,1,0) at index i means that the ith node visited in the puzzle path is at (x, y), and (255,255,1,0) is a special value that represents an unused index (i.e. the path isn't that long and is already over). Index 100 and 101 change when you mouse over the board while in draw mode, these represent the mouse coords while in draw mode. Index 102 and 103 are the x, y coordinate of the player within the room, and 104 and 105 are the left-right and up-down angles of the viewport respectively.

Index 120 is 0 while not in draw mode and 255 when in draw mode; this represents whether we're currently drawing on the board. The following indices can be determined by looking at the "o" variable in main.js (515-530):

200: LEFT
201: RIGHT
202: UP
203: DOWN
204: CLICK
205: RIGHTCLICK
210: mouse X
211: mouse Y

Recall that index 150 is our goal. It looks like it depends on the index 121, which we'll call STATE. STATE depends on Cs (CharacterState).AB, which is set by the functions Cl, Ce, and CN.

Puzzle 1

Let's look at Cl:

```
PAIR Cl(vec2 BOARD_POS){
    bool ONBOARD=LOOPBACK_BOOL(ENUM_ONBOARD.);
    bool Bp=Bj(8);
...
```

```
return PAIR(ONBOARD, Bp);
}
```

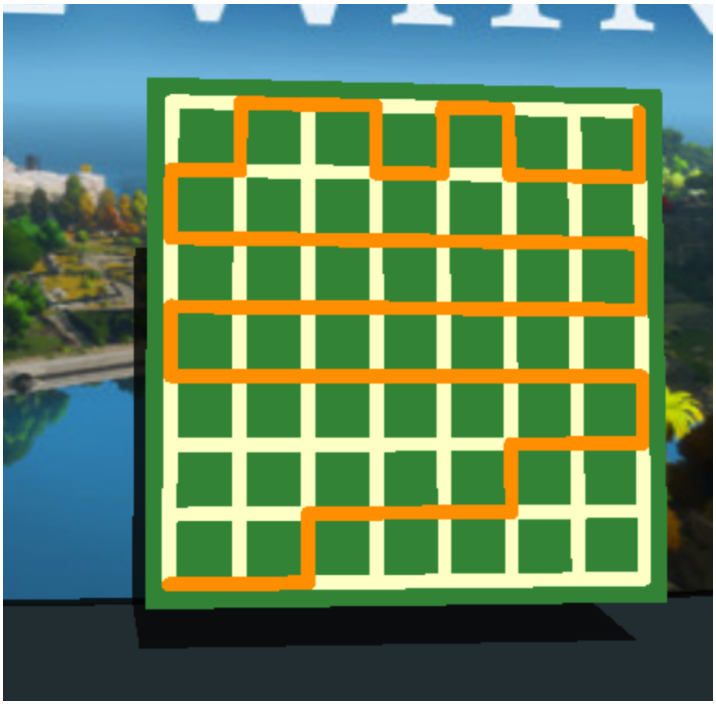
Bp (which lets us advance Cs.AB) is set by Bj. This looks to be the validation logic for the first puzzle. By looking at the code of Bj, we should be able to figure out what path we need to draw to solve the puzzle.

```
if (Aj==ivec3(board_size-1, board_size-1, 1)){
    return true;
}
```

So the goal is to get to the top right corner.

```
float Bk=(float(Aj.x)+float(Ag.x))/(2.*float(board_size));
float Bl=(float(Aj.y)+float(Ag.y))/(2.*float(board_size));
vec4 Bm=texture2D(introImage, vec2(Bk, Bl));
if (Bm.a>=1.){
    return false;
}
```

But if this condition occurs then the path is ruled invalid. Unfortunately, I don't actually understand what's happening here. It seems like there should be transparency set in the intro image (The Witness background) but I couldn't find any (update: there apparently are some pixels with 254 opacity in that image, I just didn't look hard enough). So instead, I modified Bj to return the coordinates of the first node that caused the path to be invalid, and I use a SET macro to write that to index 149. Then I copied my patched source code into line 1086 of main.js and reloaded the page. Now I can simply try a path and if I see index 149 change, I know that the last move was incorrect. This is like how lockpickers try to set pins so each tumbler can be brute forced independently.



After putting in the correct path, the board changed colors and a door appeared. Going through the door led to another room, with another puzzle board. In total, there are 3 puzzles. Similarly, we can find the validation logic for those puzzles in the code. Bi is the validation logic for the second puzzle and BZ is the logic for the third puzzle.

Fortunately the second and third puzzles were easier to understand. For both puzzle 2 and 3, I used depth first search to find a valid path to the top right corner that fulfilled the requirements. The code and output are attached, but I'll leave the gameplay as an exercise for the reader.

Puzzle 2

```
def ak(af, al, am):
    ai = af % am
    an = 1
    for i in range(8):
        if al % 2 == 1:
            an = an * ai % am
        ai = ai * ai % am
        al = al // 2
    return an

def ap(r, aq):
    ar = ak(1021, r + 12, 4093)
    return ar * aq // 4093

legal = set()
for i in range(8):
    for j in range(8):
        if ap(i*8+j, 2):  legal.add((i, j))

seen = set()
def neighbors(i, j):
    res = []
    if i != 0:
        res.append((i-1, j))
    if j != 0:
        res.append((i, j-1))
    if i != 7:
        res.append((i+1, j))
    if j != 7:
        res.append((i, j+1))
    return res

work = [((0,0), [])]
while work:
    cur, path = work.pop()
    seen.add(cur)
    if cur == (7, 7):
        print([(i, j) for j, i in path])
        break
    path = path[:] + [cur]
    for n in neighbors(*cur):
        if n in legal and n not in seen:
            work.append((n, path)
```

Output: [(0, 0), (1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (2, 2), (2, 3), (2, 4), (3, 4), (3, 5), (3, 6), (4, 6), (5, 6), (5, 5), (6, 5), (7, 5), (7, 6)]

Puzzle 3

```
import numpy

class ivec2:
 def __init__(self, a, b):
   self.value = [a,b]

Ba = list(range(14))
Ba[0]=ivec2(2, 6);
Ba[1]=ivec2(2, 4);
Ba[2]=ivec2(1, 3);
Ba[3]=ivec2(1, 1);
Ba[4]=ivec2(4, 1);
Ba[5]=ivec2(6, 8);
Ba[6]=ivec2(6, 6);
Ba[7]=ivec2(5, 5);
Ba[8]=ivec2(5, 3);
Ba[9]=ivec2(8, 3);
Ba[10]=ivec2(9, 10);
Ba[11]=ivec2(9, 4);
Ba[12]=ivec2(10, 4);
Ba[13]=ivec2(11, 7);
Bb = list(range(14))
Bb[0]=ivec2(0, 7);
Bb[1]=ivec2(2, 7);
Bb[2]=ivec2(3, 6);
Bb[3]=ivec2(3, 4);
Bb[4]=ivec2(2, 3);
Bb[5]=ivec2(4, 9);
Bb[6]=ivec2(6, 9);
Bb[7]=ivec2(7, 8);
Bb[8]=ivec2(7, 6);
Bb[9]=ivec2(6, 5);
Bb[10]=ivec2(8, 11);
Bb[11]=ivec2(10, 11);
Bb[12]=ivec2(10, 10);
Bb[13]=ivec2(10, 7);

def neighbors(i, j):
   res = []
   if i != 0:
       res.append((i-1, j))
   if j != 0:
       res.append((i, j-1))
   if i != 11:
       res.append((i+1, j))
   if j != 11:
       res.append((i, j+1))
   return res

work = [((1,0), (0,0), 0, 0, [(0,0)]), ((0,1), (0,0), 0, 0, [(0,0)])]
while work:
   cur, last, bc, bd, path = work.pop()
   if cur == (11, 11):
       if bd == 14 and bc == 14:
         print("Found it", path)
         break
   else:
     path = path[:] + [cur]
     for n in neighbors(*cur):
       if n not in path:
```

```
        be, aj, ag = ((vec[0], vec[1], 1) for vec in (last, cur, n))
        bf = np.subtract(ag, aj)
        bg = np.subtract(be, aj)
        bh = np.cross(bf, bg)
        acted = False
        if bh[2] != 0:
          if bh[2] < -0.5:
            acted = True
            if bd >= 14:
              continue
            if np.equal(Bb[bd].value, aj[:2]).all():
              work.append((n, cur, bc, bd+1, path))
          if bh[2] > 0.5:
            acted = True
            if bc >= 14:
              continue
            if np.equal(Ba[bc].value, aj[:2]).all():
              work.append((n, cur, bc + 1, bd, path))
        if not acted:
          work.append((n, cur, bc, bd, path)
```

Output:
Found it [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (1, 7), (2, 7), (2, 6), (3, 6), (3, 5), (3, 4), (2, 4), (2, 3), (1, 3), (1, 2), (1, 1), (2, 1), (3, 1), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6), (4, 7), (4, 8), (4, 9), (5, 9), (6, 9), (6, 8), (7, 8), (7, 7), (7, 6), (6, 6), (6, 5), (5, 5), (5, 4), (5, 3), (6, 3), (7, 3), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (8, 8), (8, 9), (8, 10), (8, 11), (9, 11), (10, 11), (10, 10), (9, 10), (9, 9), (9, 8), (9, 7), (9, 6), (9, 5), (9, 4), (10, 4), (10, 5), (10, 6), (10, 7), (11, 7), (11, 8), (11, 9), (11, 10)]

The third puzzle has an additional wrinkle because the board is placed in such a way that it's impossible to actually see what's on it while operating it. There is a reflective pool below the board, but it's impossible to both see the pool and trigger the path drawing on the board. I tried to input the path blind and reversed, but it was as annoying as it sounds.

But since we know how the data is stored, we can just pass it through JavaScript. I put a breakpoint right before p is submitted to loopback and pasted the following in the console, tricking the program into thinking that we already input the correct path:

```
p[0] = [0,0,1,0]
p[1] = [0,1,1,0]
p[2] = [0,2,1,0]
p[3] = [0,3,1,0]
p[4] = [0,4,1,0]
p[5] = [0,5,1,0]
p[6] = [0,6,1,0]
p[7] = [0,7,1,0]
p[8] = [1,7,1,0]
p[9] = [2,7,1,0]
p[10] = [2,6,1,0]
p[11] = [3,6,1,0]
p[12] = [3,5,1,0]
p[13] = [3,4,1,0]
p[14] = [2,4,1,0]
p[15] = [2,3,1,0]
p[16] = [1,3,1,0]
p[17] = [1,2,1,0]
p[18] = [1,1,1,0]
p[19] = [2,1,1,0]
```

```
p[20] = [3,1,1,0]
p[21] = [4,1,1,0]
p[22] = [4,2,1,0]
p[23] = [4,3,1,0]
p[24] = [4,4,1,0]
p[25] = [4,5,1,0]
p[26] = [4,6,1,0]
p[27] = [4,7,1,0]
p[28] = [4,8,1,0]
p[29] = [4,9,1,0]
p[30] = [5,9,1,0]
p[31] = [6,9,1,0]
p[32] = [6,8,1,0]
p[33] = [7,8,1,0]
p[34] = [7,7,1,0]
p[35] = [7,6,1,0]
p[36] = [6,6,1,0]
p[37] = [6,5,1,0]
p[38] = [5,5,1,0]
p[39] = [5,4,1,0]
p[40] = [5,3,1,0]
p[41] = [6,3,1,0]
p[42] = [7,3,1,0]
p[43] = [8,3,1,0]
p[44] = [8,4,1,0]
p[45] = [8,5,1,0]
p[46] = [8,6,1,0]
p[47] = [8,7,1,0]
p[48] = [8,8,1,0]
p[49] = [8,9,1,0]
p[50] = [8,10,1,0]
p[51] = [8,11,1,0]
p[52] = [9,11,1,0]
p[53] = [10,11,1,0]
p[54] = [10,10,1,0]
p[55] = [9,10,1,0]
p[56] = [9,9,1,0]
p[57] = [9,8,1,0]
p[58] = [9,7,1,0]
p[59] = [9,6,1,0]
p[60] = [9,5,1,0]
p[61] = [9,4,1,0]
p[62] = [10,4,1,0]
p[63] = [10,5,1,0]
p[64] = [10,6,1,0]
p[65] = [10,7,1,0]
p[66] = [11,7,1,0]
p[67] = [11,8,1,0]
p[68] = [11,9,1,0]
p[69] = [11,10,1,0]
p[70] = [11,11,1,0]
```

That solves the final puzzle and after going through the third door, the flag appears!