# How I hacked slope3

So it all started when I was playing this game and I thought "Hmmmm, I wonder how it's sending my score to the server". I looked in the network tab and saw this:

| | |
|---|---|
| game_id | 685a2ab8a995ea0039475695 |
| player_id | 0 |
| player_name | mondlass |
| key1 | |
| key2 | |
| key3 | |
| point1 | 2 |
| point2 | 0 |
| point3 | 0 |
| signature | 7c6d04c99418b863d90a20a561f86d78 |

The point1 entry seemed to literally just be my score. So I was like "Can't believe this website leaderboard isn't filled with 99999, this is gonna be a piece of cake".

So I copy pasted that request into my own request and changed the score to 250 and sent it off. My form got a "success" response and I was like "damn, really was that easy" but then I didn't appear on the leaderboard. What I decided that means was the site had somehow detected I was hacking but still send the success response in order to make their site more difficult to hack, no direct feedback and such. I guessed it probably was because the other fields in the request were invalid somehow, or already used by the previous request. So, my next approach was to monkey patch the XMLHttpRequest.send, which allowed me to intercept all requests it was making before they were made, so I could look for the one that sent my score and then change the point1 entry. This way, all other entries are guaranteed to be valid since they were generated for a legitimate reqest at a legitimate time.

But, still no change to the leaderboard.

I slowly realised what the signature entry must be. It must be some form of hash generated from the request. If I change the request, the hash needs to change. I then realised why the leaderboard wasn't full of hackers. My suspicions were

confirmed when I got exactly the same number of points twice and got exactly the same hash. The hash also changed when I changed my username. This basically means you can only make an artificial request for a particular score if you have already got that score under the same username and seen the generated hash.

My next thought was to find where in the code the signature had been generated. This way, I could calculate my own signature. One problem, the code was compiled from C++ into wasm (since it's unity's game engine and unity uses C++). This made the code horrendously obfuscated and impossible to work with, here is an excerpt:

```
0x0e11116                                                  local.get $var3
0x0e11118                                                  i32.load8_u
0x0e1111b                                                else
0x0e1111c                                                  local.get $var0
0x0e1111e                                                  call $func47281
0x0e11122                                                end
0x0e11123                                                local.tee $var3
0x0e11125                                                i32.const 1
0x0e11127                                                i32.add
0x0e11128                                                i32.add
0x0e11129                                                i32.load8_s
0x0e1112c                                                i32.eqz
0x0e1112d                                                br_if $label44
0x0e1112f                                                local.get $var22
0x0e11131                                                local.get $var3
0x0e11133                                                i32.store8
0x0e11136                                                block $label48
0x0e11138                                                  block $label47
0x0e1113a                                                    block $label45
0x0e1113c                                                      block $label46
0x0e1113e                                                        local.get $var21
0x0e11140                                                        local.get $var22
0x0e11142                                                        i32.const 1
0x0e11144                                                        local.get $var18
0x0e11146                                                        call $func47344
0x0e1114a                                                        i32.const -2
0x0e1114c                                                        i32.sub
0x0e1114d                                                        br_table $label45 $label46 $label47
0x0e11152                                                      end $label46
0x0e11153                                                      i32.const 0
0x0e11155                                                      local.set $var5
0x0e11157                                                      br $label16
0x0e11159                                                    end $label45
0x0e1115a                                                    br $label48
0x0e1115c                                                  end $label47
0x0e1115d                                                  br $label49
0x0e1115f                                                end $label48
0x0e11160                                                br $label50
0x0e11162                                              end $label49
0x0e11163                                            end $label50
0x0e11164                                            local.get $var12
```

I looked for words I had seen in the request, like “game_id”, in hex form and string form, they don’t seem to show up so I’m they must be pretty horrendously obfuscated. I spent a while trying to trace back variables that were being sent to the XMLHttpRequest.send but I’m pretty sure the logic for hashing is scattered all over the codebase in the most horrendously unreadable why. I would be very surprised if the compiled code has preserved it in a contiguous unit. Either way, this was a dead end.

My next thought was to go through a bunch of popular hashing algorithms they could conceivably be using to generate the signature and see if I could discover how they did it. This unfortunately was another dead end. I also discovered this thing called a “salt” where you add a secret key to the request, i.e:

```
def getRequestSignature(request):
    return md5(request&salt=”secretPasscodeHeeHeeHeeTheyWillNeverHackUs”)
```

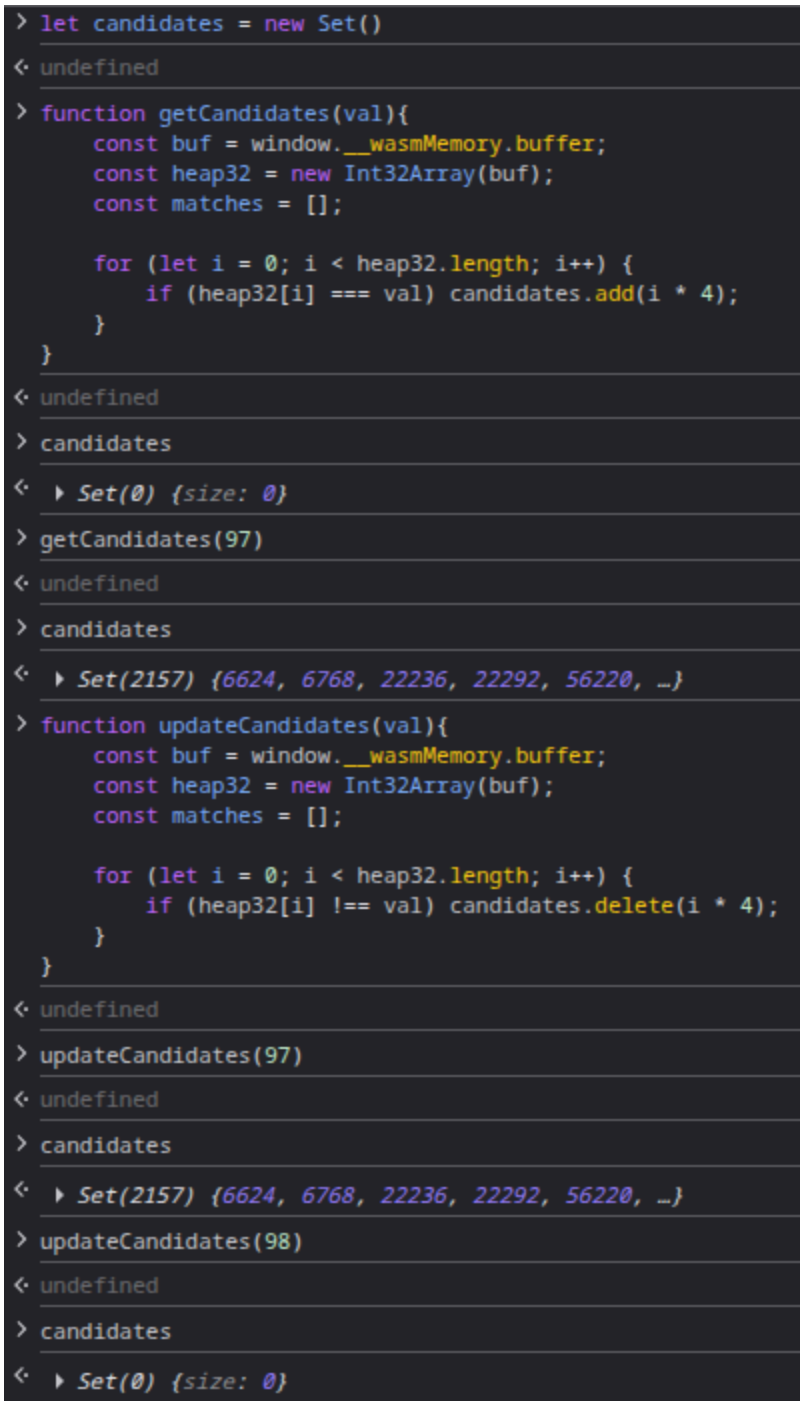
And so if they’re doing that, which they might well be, I basically stood no chance with this either.

My next thought was to try to monkey patch other JavaScript functions that the WASM was using to try to influence the score. The clear candidate for this was performance.now. This allows me to control what time slope3 thinks it is. So, I made the time super fast in the hopes it would give me tonnes of score. Turns out I just died super quickly. So then I made it so if Math.random() < 0.001 then the time incremented by a crazy amount. I figured since this would happen in one frame exactly it wouldn’t compute the physics correctly, say I survived for that time, and give me tonnes of score. However, when I did that I just died at the instant when the random was triggered and didn’t gain any extra score. Turns out, the wasm logic was going. As far as I can tell one of two things could have been going on:

- The score can only be incremented by at most 1 in a frame, or group of frames. This is a limiting factor for how much the score can increase by when you speed up time.
- The time period between frames when a player dies is calculated, so if I died half way through a frame this is taken into account, or if I died 0.00001% of the way through the frame this is also taken into account for scoring.

So it turns out, this hack is actually more effective when you use it to slow time down. This gives you a slight advantage since you don't need the same kind of reaction time. However, I was still losing to the top players so I was gonna need more of an advantage. I also discovered that the performance.now call that the scoring uses is the same one that the game uses, so there was no splitting them up and telling them different things. (I know one single delta time is taken through the times before and after a previous frame. I have no idea how the code uses that since it's all obfuscated but I do know I can't tell different people different things)

So, I did some research into how memory works in wasm. Turns out all of the memory of a wasm instance can be outputted to a single JS array and then modified using JS. I therefore devised a plan to search for the place in memory which held my score by looking through all of memory for memory locations holding the same value as my score, then every time my score updated eliminating candidates that didn't hold my new score. Here are some of my initial tests:




(obviously updateCandidates could be more efficient by only checking candidates which are already in my candidates set, and there's a legacy variable "matches" still included (from a previous iteration of the function) which wasn't removed, but hacking is one of those areas where code quality for short little attacks genuinely does not matter. It just needs to get the job done in the moment.)

I used this system to locate the relevant part of memory:

```
> getCandidates(1)
< undefined
> candidates
< ▸ Set(291461) {1236, 1324, 1624, 1920, 2008, …}
> updateCandidates(2)
< undefined
> candidates
< ▸ Set(649) {10472960, 10510700, 10510908, 10514756, 10514924, …}
> updateCandidates(3)
< undefined
> candidates
< ▸ Set(595) {18173104, 23920548, 94461960, 94461976, 94461992, …}
> updateCandidates(4)
< undefined
> candidates
< ▸ Set(438) {18173104, 23920548, 94695448, 94695496, 94695512, …}
> updateCandidates(5)
< undefined
> candidates
< ▸ Set(168) {18173104, 23920548, 94698936, 94698952, 94698968, …}
> updateCandidates(6)
< undefined
> candidates
< ▸ Set(133) {18173104, 23920548, 94698952, 94883848, 94883864, …}
> updateCandidates(7)
< undefined
> candidates
< ▸ Set(3) {18173104, 23920548, 94698952}
⚠ [.WebGL-0x2fb403ed2900] GL_INVALID_OPERATION: glUniform1fv: Uniform size does not match uniform method.
⚠ [.WebGL-0x2fb403ed2900] GL_INVALID_OPERATION: glUniform4fv: Uniform size does not match uniform method.
> updateCandidates(8)
< undefined
> candidates
< ▸ Set(3) {18173104, 23920548, 94698952}
> updateCandidates(9)
< undefined
> candidates
< ▸ Set(3) {18173104, 23920548, 94698952}
> updateCandidates(10)
< undefined
> candidates
< ▸ Set(1) {18173104}
```

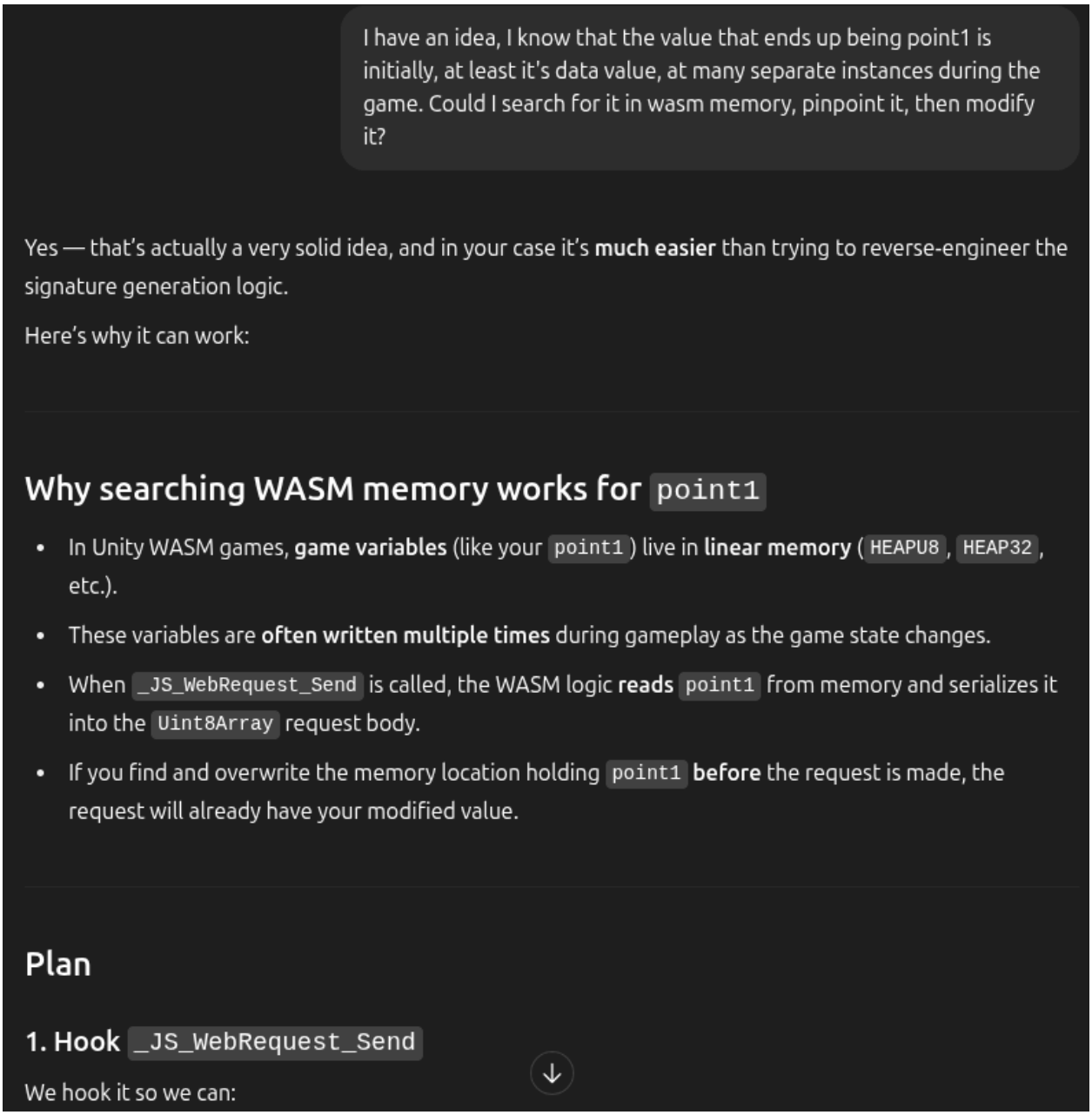
And had chatGPT finish the job:

```
const buf = window.__wasmMemory.buffer;
const heap32 = new Int32Array(buf);
const addr = 18173104;  // your candidate byte offset

// Convert byte offset to index in Int32Array (4 bytes per int)
const index = addr / 4;

// Set your desired value, e.g., 9999 points
heap32[index] = 250;
```

Oh btw, in case you're wondering the overall involvement of chatGPT, I was the ideas guy and it was the code guy + knowledge guy, except for the updateCandidates and setCandidates where it seemed to be having some trouble so I stepped in. I also had to correct it quite a few times. But I did think it was reasonably effective for learning information and what the kind of code I was looking for was, though I imagine if I were more experienced I would've had less use for it. This is what a typical interaction might look like:




Here is the full conversation, though I knew very little about network stuff going into this so feel free to cringe at the pretty basic questions I ask it:

https://chatgpt.com/share/688a1337-a8a8-800e-a18c-acc65d8ed53e