Hacking WebAssembly Games with Binary Instrumentation


WEBASSEMBLY 101

WASM 101

- Developers have done (and continue to do) incredible work speeding up Javascript
- However, the dynamic nature of Javascript will always be a roadblock
- WebAssembly provides a static, pre-compiled binary format for performance intensive applications

WASM 101

- WebAssembly "defines an instruction set and binary format for an assembly-like architecture"
- WebAssembly is built to be targetable by existing compilers and languages
  - Finally we can write web applications in C!

WASM USES

- WebAssembly video games are becoming very common
  - Look at any browser game website (Newgrounds, Kongregate, etc)
- Unity3D and Unreal Engine 4 can now both target WebAssembly
- This means there’s a lot of targets without a lot of tools

WASM USES

- WebAssembly is used for a lot of types of applications, not just video games
  - Retargeted desktop applications
  - 3D applications
  - Crypto miners
  - ...etc
- These techniques are not video game specific — video games are just the most fun target

WASM REVERSING

- With WebAssembly, web RE has started to feel more like “traditional” binary RE
  - Back to the disassembler!
- A few tools support WebAssembly (mostly static analysis)
  - radare2
  - JEB decompiler
  - wabt (WebAssemby Binary Toolkit)

WASM REVERSING

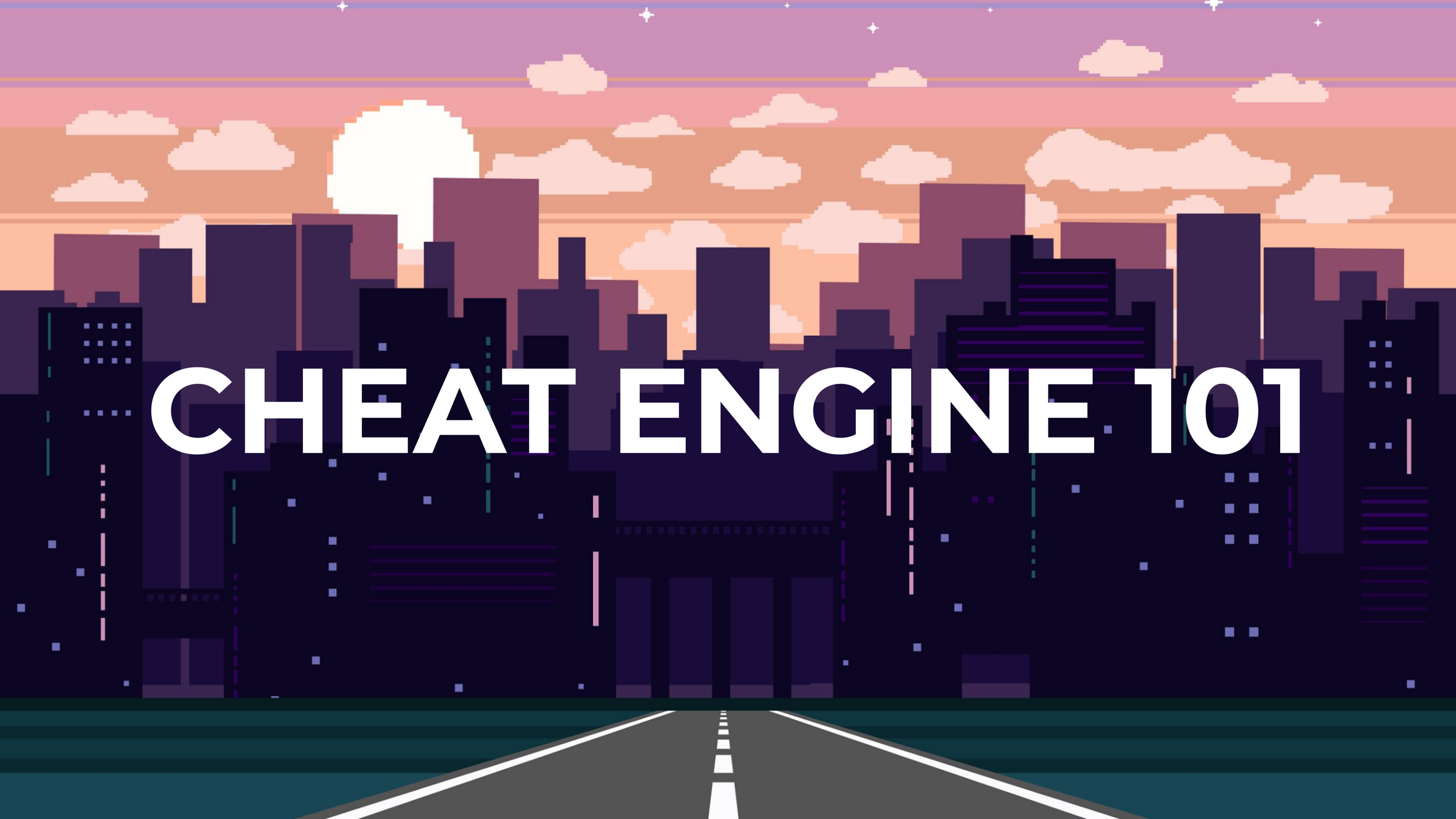
- Browser debugging capabilities for WASM are pretty lacking
  - No watchpoints
  - No conditional breakpoints
  - Lots of bugs

VIDEO GAME REVERSING

- Video games are a unique challenge when it comes to RE
- Video game binaries are typically much larger and more complex than other applications
- Video games are more performance intensive, and performance impacts are more noticeable
  - No one wants to play a game at 5 FPS
- With this in mind, I was looking for a tool like Cheat Engine for WASM

CHEAT ENGINE

- Cheat Engine (made by Dark Byte) is effectively a specialized debugger for hacking video games
- Cheat Engine can:
  - Search memory
  - Modify and “freeze” memory
  - Set watchpoints
  - Inject/patch code


CHEAT ENGINE 101

We want to make our character invincible. We know our character currently has 5 health

We start by searching the game’s memory for the value 5

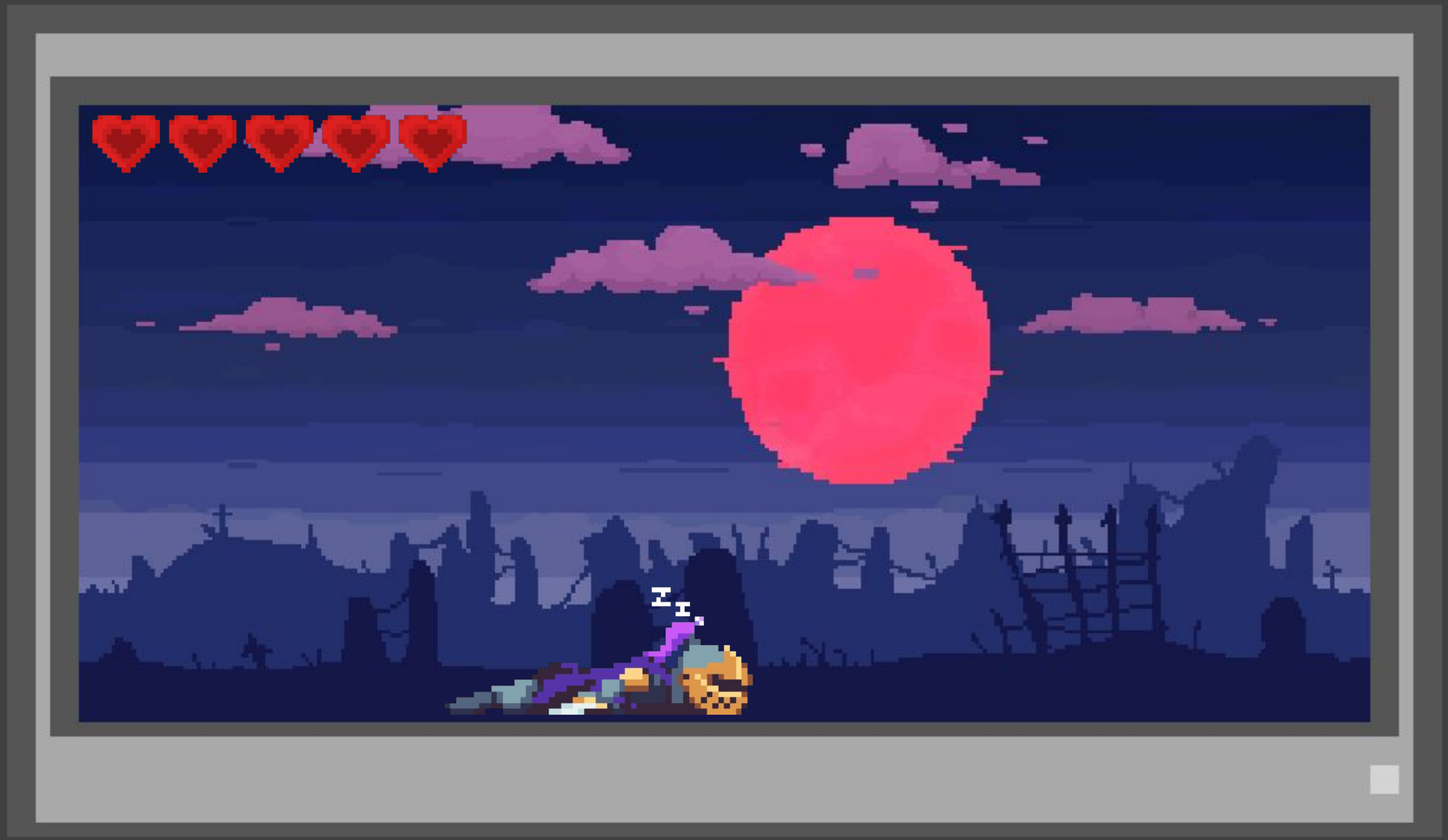
Then we cause the value to change and search for the new value...

...and continue this process until we’ve found our health value in memory

Now we can manipulate our health to heal ourselves

...or give ourselves more health

...or “freeze” our health so we can’t get hurt



This process can take a while and needs to be redone every time we play. Ideally we want to permanently patch the game

We set a watchpoint on our health address, then trigger it

Now we know where health is decremented when we get hurt

...and we can patch it out

CHEAT ENGINE USES

- Cheat Engine not only helps us hack games, it can also be significant help in RE
- Using watchpoints we can associate a value in memory to the code that affects it
- This is an invaluable time saver reverse engineering large applications like video games

CHEAT ENGINE

> Since WASM doesn’t have watchpoints we can’t directly implement Cheat Engine features
> Can we emulate watchpoint behavior without “real” watchpoints?

EMULATING WATCH POINTS

- First attempt: Using the browser debugger
- Place a breakpoint at each load/store instruction and check if the access affects our “watched” address

EMULATING WATCH POINTS

- First attempt: Using the browser debugger
- Place a breakpoint at each load/store instruction and check if the access affects our “watched” address
- Way too slow — browser becomes unusable

EMULATING WATCH POINTS

- To emulate watchpoints, we want to inject code into the binary at each memory load/store instruction
- Injected code will check if this access affects the memory area we are “watching”
- If so, trigger our breakpoint code
- To do all this, we need to employ some form of binary instrumentation


BINARY
INSTRUMENTATION

BINARY INSTRUMENTATION 101

- In a nutshell, binary instrumentation is the process of manipulating an application binary to aid in analysis
- A lot of cool binary instrumentation tools exist for other types of binaries
  - Frida
  - DynamoRIO
  - ...others I haven’t used
- There’s even some existing tools for WASM!

OTHER TOOLS

- Wasabi (by Daniel Lehmann) is a very cool instrumentation and analysis tool for WASM
- However, it does not exactly fit our needs
  - Wasabi is written in Rust and intended to be run from the terminal
  - Wasabi does its analysis by injecting Javascript
- If we want to run a game at any decent FPS, we need to call Javascript as infrequently as possible

OTHER TOOLS

- WABT (WebAssembly Binary Toolkit) can parse and (to some extent) modify WASM binaries
- It’s even been compiled to Javascript!
- Unfortunately, WABT’s parsing takes too long/too much memory for most video game binaries

OTHER TOOLS

- What we want is a tool that:
  - Can instrument binaries from within the browser
  - Can handle large (40MB+) WASM binaries quickly and without running out of memory

WAIL

- The “WebAssembly Instrumentation Library” (WAIL) is my attempt at a solution to this problem
- WAIL is a Javascript library focused on making targeted modifications to WASM binaries
  - Can add entries to any specification-defined section
  - Can edit existing entries of sections
  - Can add/remove sections

WAIL PARSING

- WAIL uses a couple of tricks to modify binaries significantly faster with less memory usage than other libraries
  - WAIL only parses sections/elements that are necessary to perform the defined modifications
  - WAIL parses binaries as a “stream”

STREAM PARSING

- The normal way of parsing a binary involves creating a “map” of all pieces that make up the binary
- Once this map is created, you modify the pieces as needed and stick everything back together
- This is convenient, but also slow and memory intensive

STREAM PARSING

- WAIL parses binaries as a “stream” — handling and modifying each element as soon as it is read
- This is more efficient because we don’t need to save each element of the entire binary
- Rather, we act on a single element at a time and then “forget” about it and move to the next

PARSING GOTCHAS

- There are a few downsides to this approach:
- The first is that the parser can never go “backwards”
  - Once we finish parsing a particular element, we cannot go back and make changes to it
- To deal with this, we must define all our modifications before we start parsing

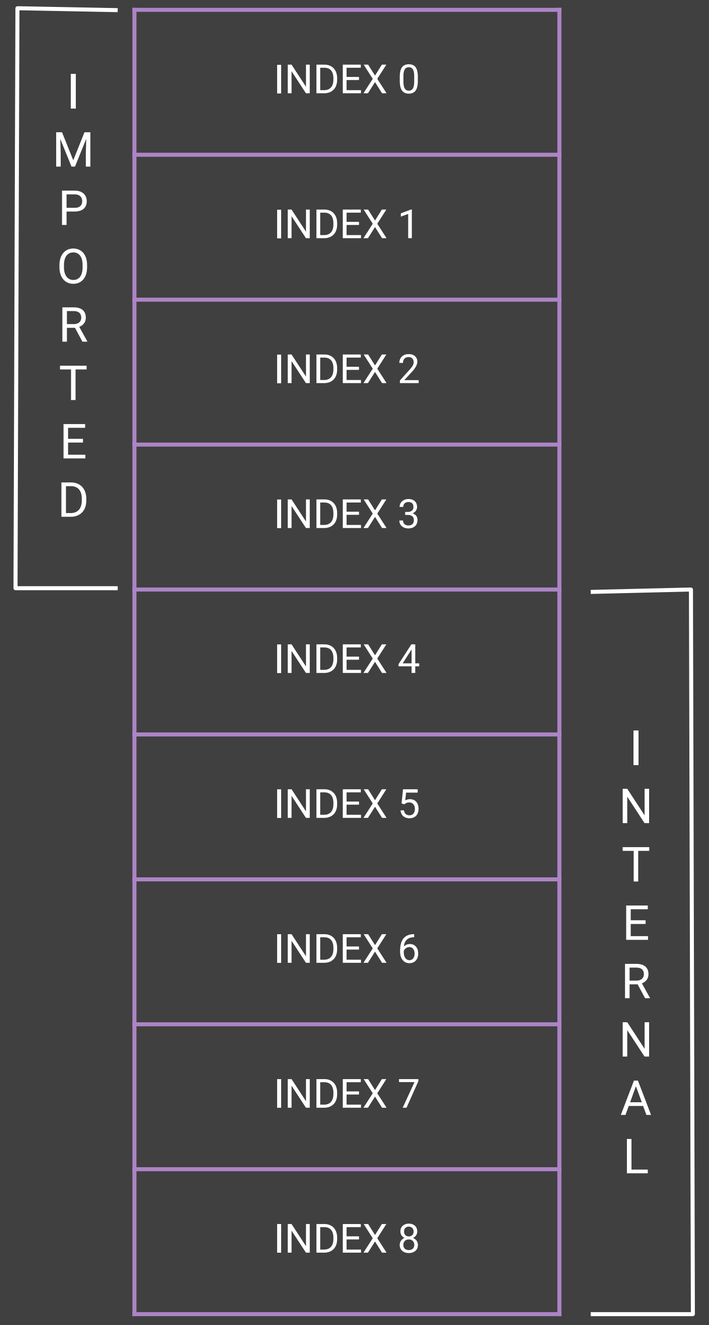
PARSING GOTCHAS

- In some cases, one addition to a binary will require knowledge of another
- For instance, to insert a new function into a WASM binary we must:
  - Add an element to the TYPE section
  - Add an element to the FUNCTION section that references the new TYPE element
  - Add an element to the CODE section corresponding to the new FUNCTION element

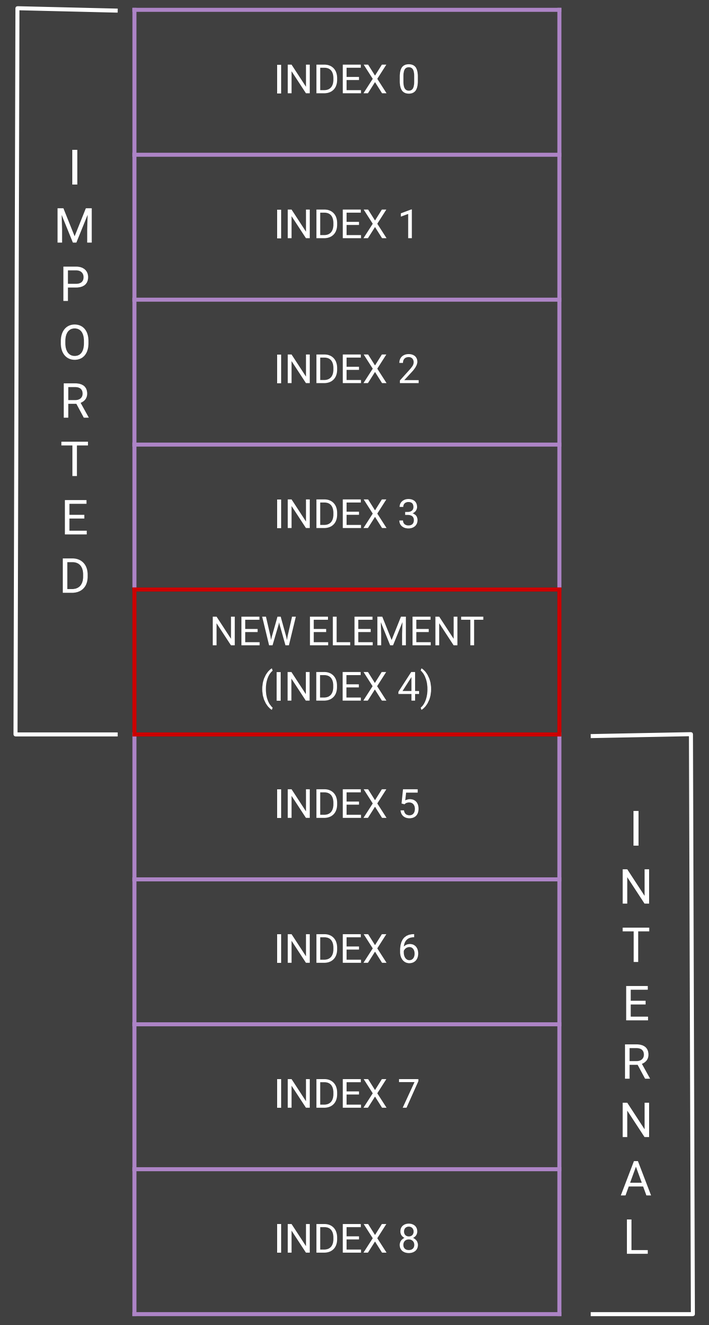
PARSING GOTCHAS

- WAIL uses a special grammar to deal with these cases
- Each addition we make returns a “handle” to a value that will be resolved when parsing
- This handle can be used in subsequent modifications
- This allows us to perform complex modifications to binaries while still defining everything up front

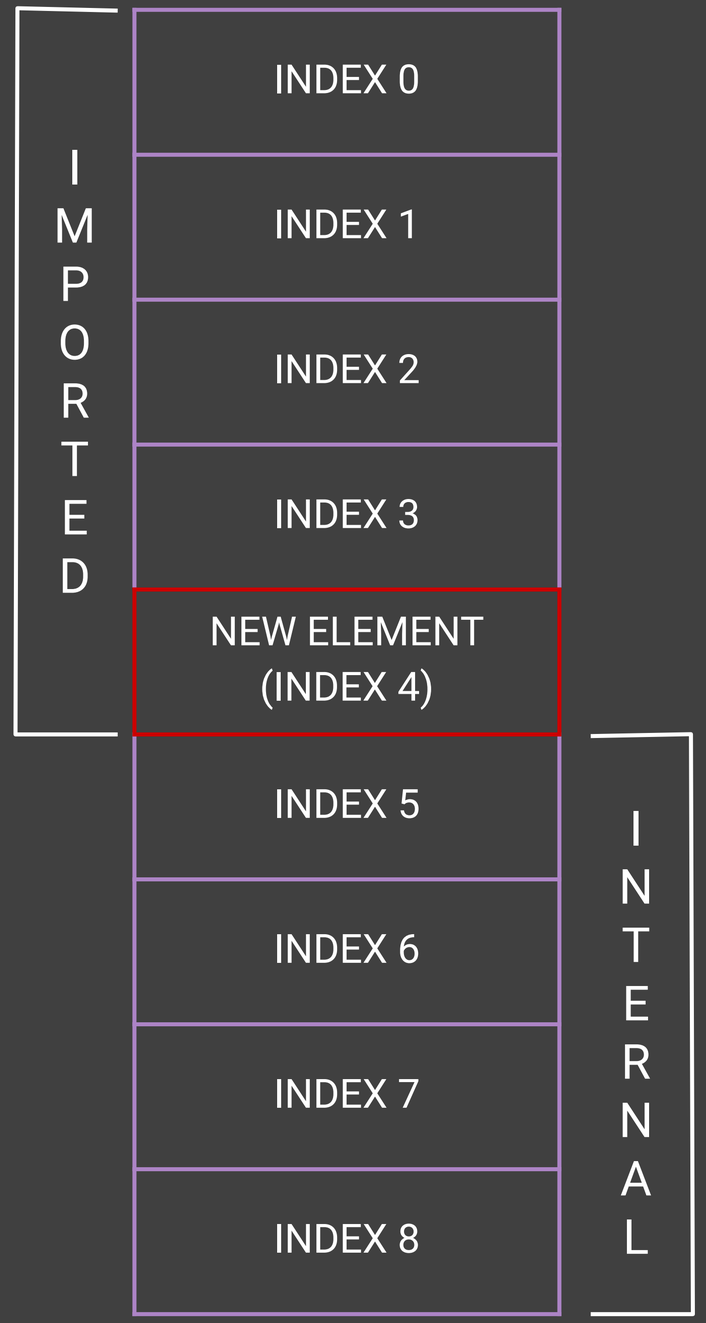
- The next gotcha: the function and global variable tables
- Functions and globals are referenced by index into the respective table
- The function table is built by taking all imported functions, then appending all internal functions
- The same goes for imported and internal globals



- Therefore, if we add a new imported function or global, we've thrown off all references to internal functions/globals



- WAIL fixes this automatically by changing affected entries in the following sections:
  - EXPORT
  - ELEMENT
  - CODE
  - START



EMULATING WATCH POINTS

- First we create two new global variables
- One for the address we are watching
- One will hold two different “flags”
  - Is watchpoint enabled?
  - Size of value being watched

EMULATING WATCH POINTS

- Next, we add an IMPORT entry for a Javascript function
- This function will only be called when our watchpoint is triggered
  - This makes performance impact minimal

EMULATING WATCH POINTS

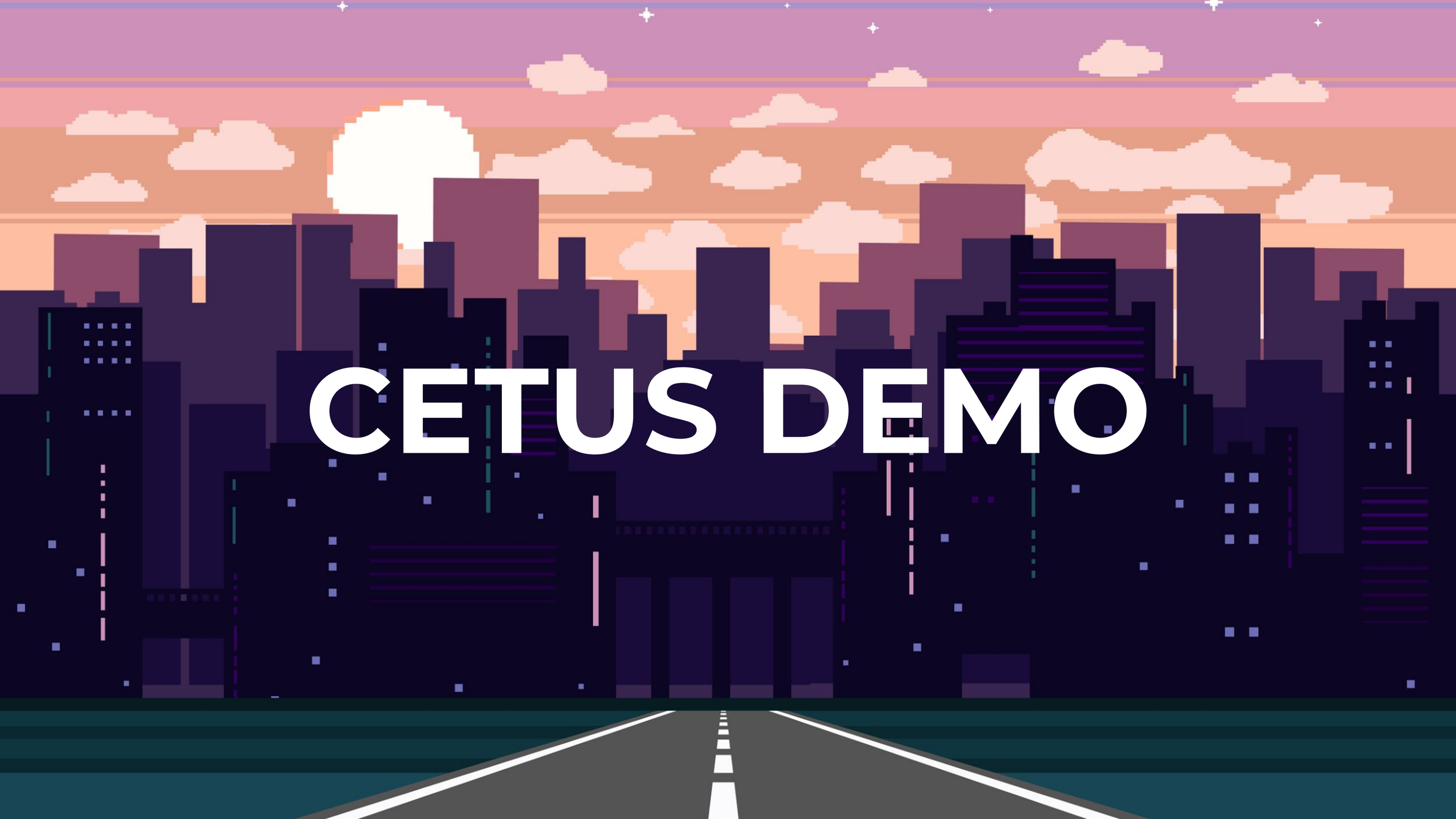
- Next we create a new internal function
  - As mentioned earlier, this requires adding to the TYPE, FUNCTION, and CODE elements
- This new function will perform the actual logic of our watchpoints
  - Check if an access overlaps with our “watched” address
  - If it does, call the “trigger” function

EMULATING WATCH POINTS

- Finally, we place calls to our watchpoint function before each memory load or store instruction
- As long as we’re careful about performance, we can apply watchpoints to games without noticeable drop in FPS

CETUS

- Cetus is a browser extension that implements features of Cheat Engine for WASM
- Comes from the Latin word for “sea monster”
- Cetus intercepts and instruments WASM binaries on the fly
  - Adds read/write watchpoints
  - Adds “freezing” functionality
  - Can apply user-defined patches

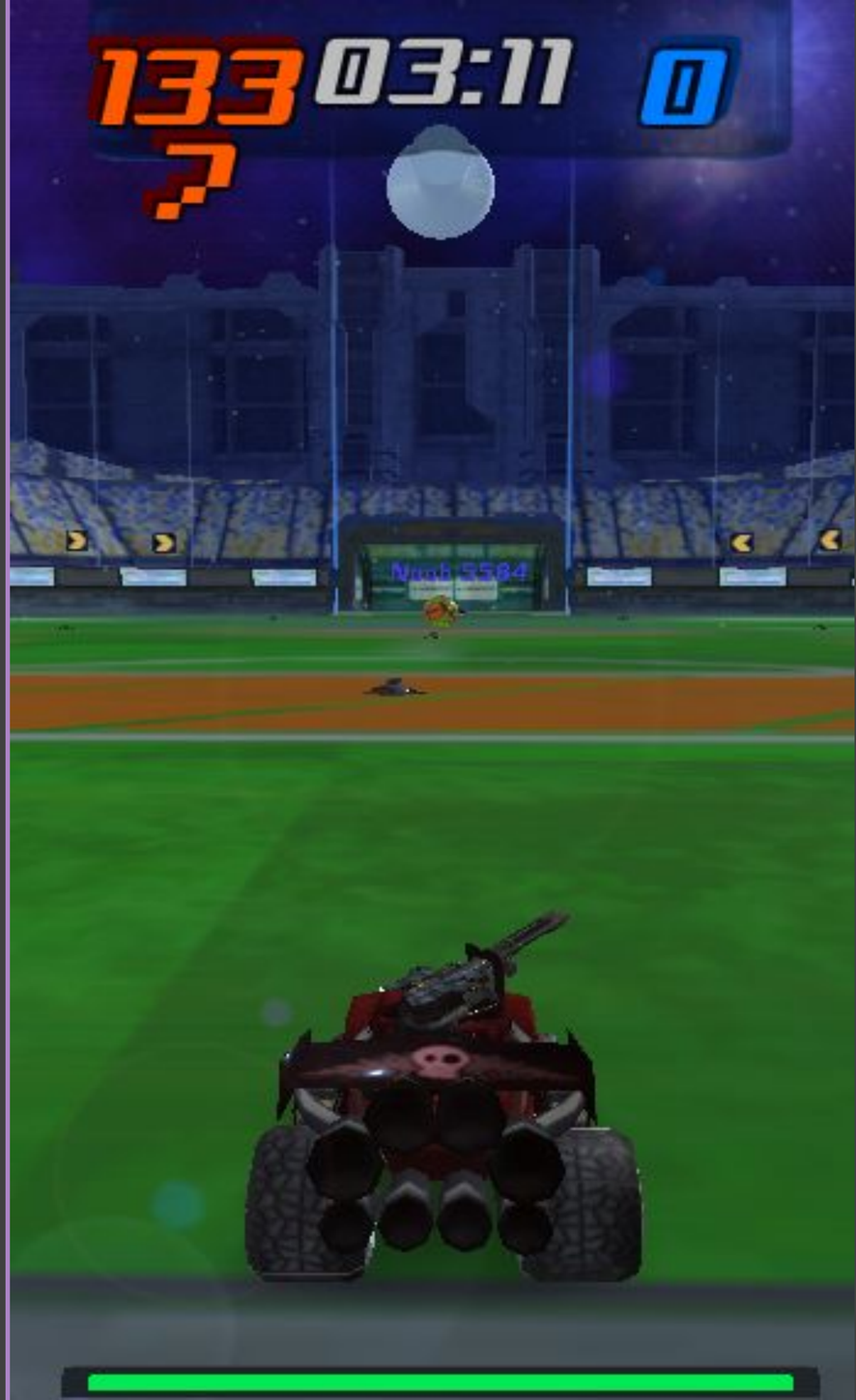

CETUS DEMO

MORE CETUS

- Cetus can also do “differential” searching
  - Used to find values when an exact starting value is not known
- Cetus also comes with a built-in speed hack
  - Works by replacing performance.now() and Date.now()

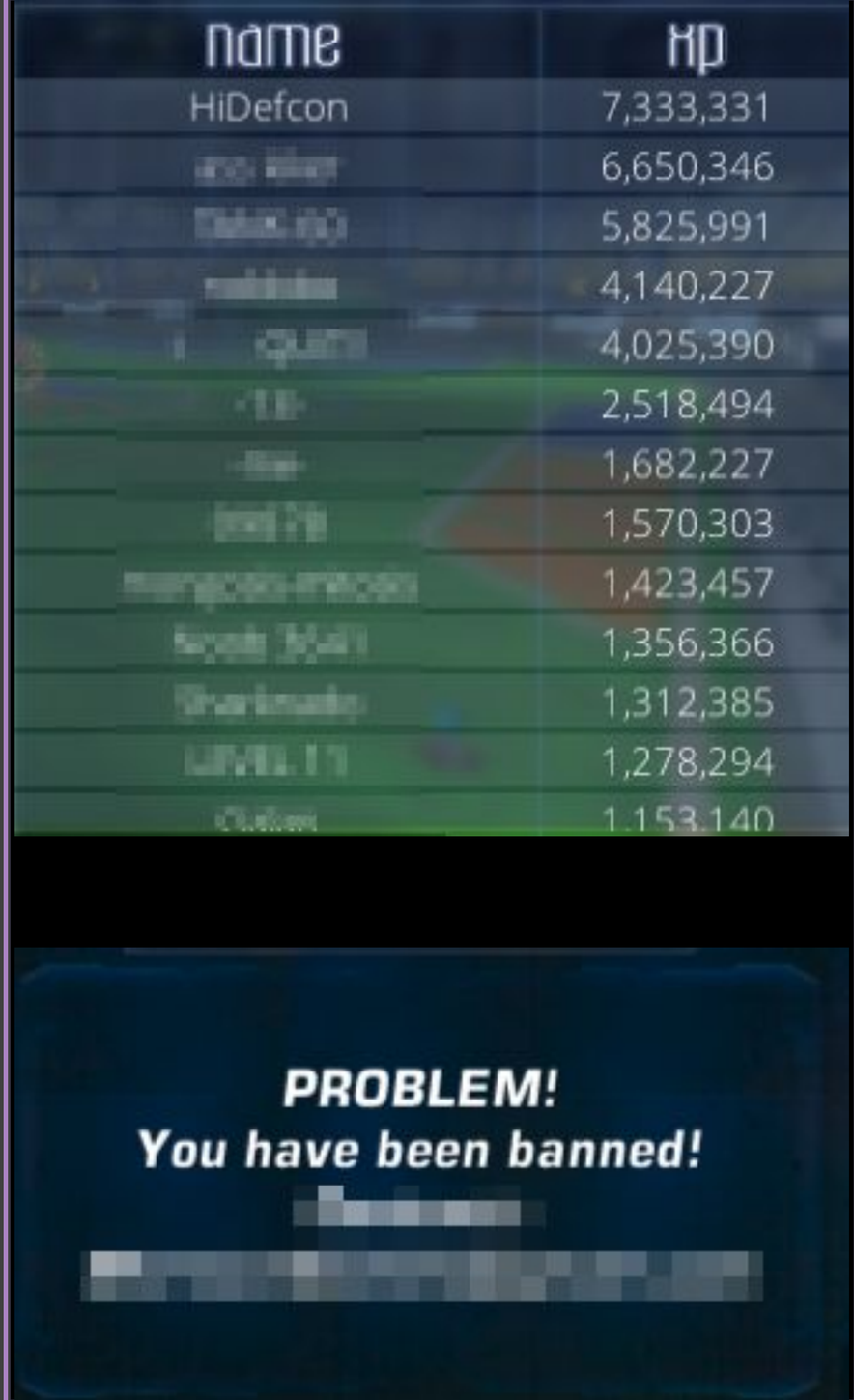

OTHER EXAMPLES

- WAIL can also be used to trace function calls by placing code at the beginning of each function
- This is slow, but still fairly useful



- WAIL can also replace a function entirely by swapping out all references to it
- For instance, we can take a WASM function and replace it with an imported Javascript function
- This way we can effectively patch WASM binaries using Javascript

- WAIL can take “internal” functions of a binary and export them
- This allows us to call the internal function on command with arbitrary arguments



ADDING SYMBOLS

- Using WAIL we can add our own symbols to a binary
- There are two ways this can be done:
  - Add a “name” section to the binary with our symbols
  - Add an export entry for each function we want to name

github.com/qwokka/wail

github.com/qwokka/cetus


GAME HACKING
MONTAGE