GO WASM
CVE-2021-38297

Paras Saxena, Zhejia Yang, Gopala Krishnan,

Anisha Nilakantan, Shubham Kulkarni

Overview

- Introduction
- Background (wasm)
- What's the Vulnerability?
- Impact (of the Vulnerability)
- Our Exploit Explained
- Live Demo
- Fix
- Fixed Demo
- Reflections

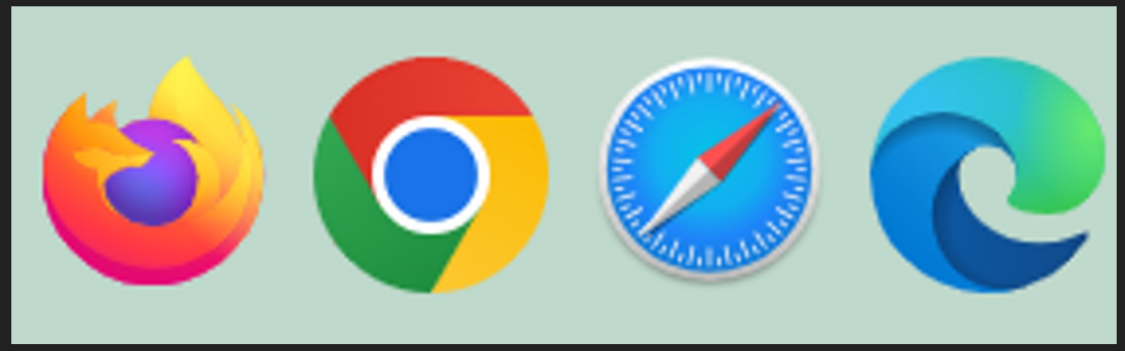
CVE

CVSS Base Score:9.8 (10/18/2021)

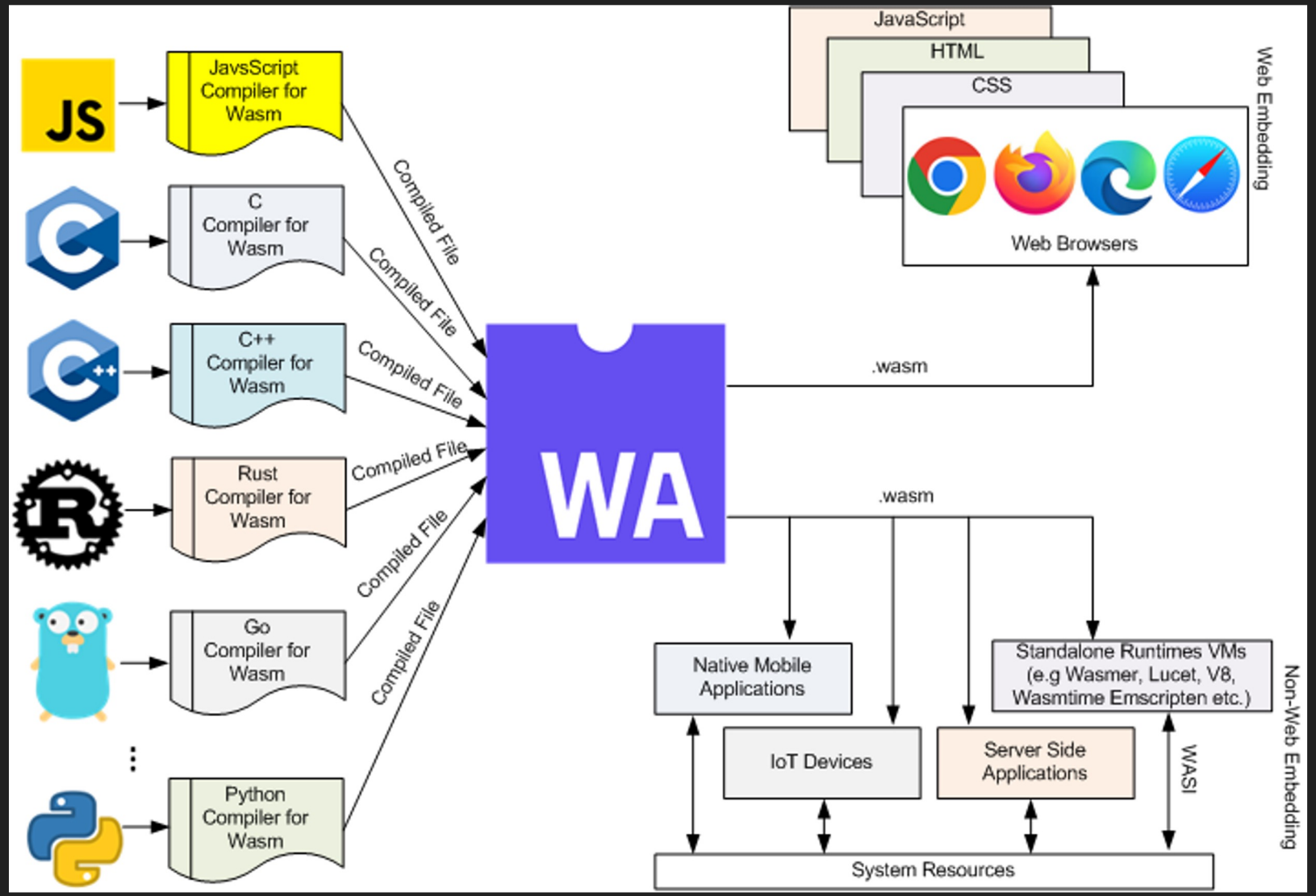
CVE-2021-38297 Go before 1.16.9 and 1.17.x before 1.17.2 has a **Buffer Overflow** via large arguments in a function invocation from a WASM module, when GOARCH=wasm GOOS=js is used.

What is Web Assembly ?

WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.
WebAssembly makes it possible to run **high-performance** applications on web pages.
Relatively new standard across web: Introduced in 2017
Right Now it's by default shipped with 4 major browser engine



Background


JS
JavsScript Compiler for Wasm
C Compiler for Wasm
C++ Compiler for Wasm
Rust Compiler for Wasm
Go Compiler for Wasm
Python Compiler for Wasm
Compiled File
Compiled File
Compiled File
Compiled File
Compiled File
Compiled File
WA
JavaScript
HTML
CSS
Web Browsers
Web Embedding
.wasm
.wasm
Native Mobile Applications
IoT Devices
Server Side Applications
Standalone Runtimes VMs (e.g Wasmer, Lucet, V8, Wasmtime Emscripten etc.)
WASI
System Resources
Non-Web Embedding

How WebAssembly Works

- **Compilation**: Source code in languages like C, C++, GO or Rust is compiled into WebAssembly bytecode (.wasm).
- **Browser Execution**: Browsers have a Wasm engine that interprets or compiles .wasm files.
- **Execution Environment**: Wasm runs in a virtual machine, providing a consistent and secure environment.

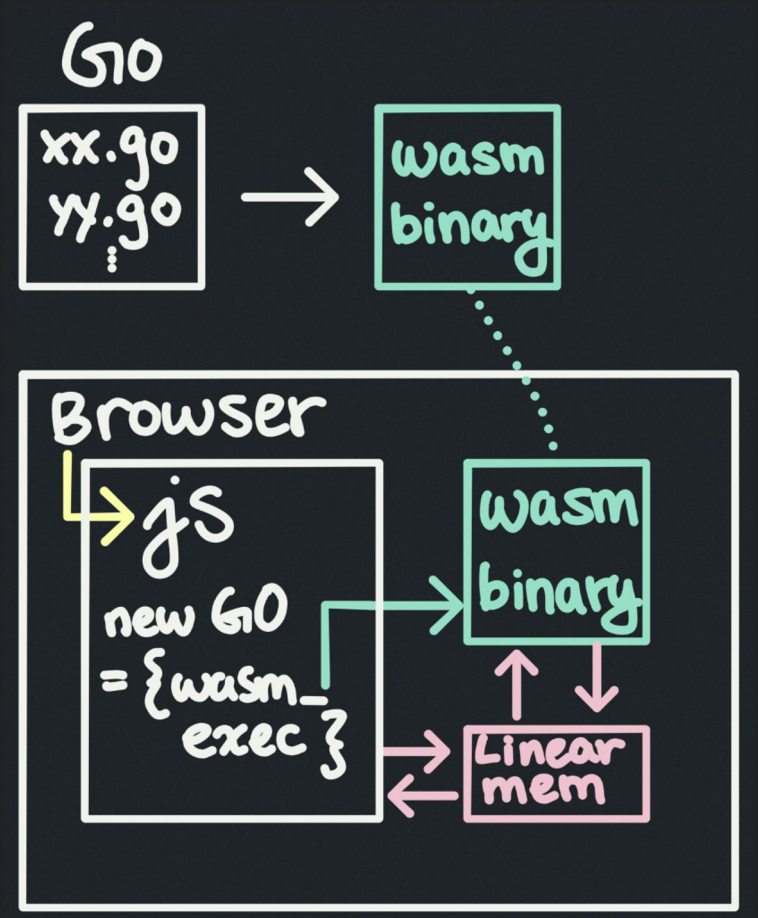
GO

- **Go (Golang):** A statically-typed, compiled language known for simplicity and efficiency.
- **Objective:** Use Go to write code that runs in the browser through WebAssembly.

Integration of Go with WebAssembly

1> Introduction of GOARCH=wasm: Direct compilation of Go code to WebAssembly.
2> Use `GOARCH=wasm go build -o output.wasm` to compile Go code to WebAssembly.
3> Generates a .wasm file that can be executed in the browser.

Interacting with JavaScript



[WebAssembly Interface Types: Interoperate with All the Things! - Mozilla Hacks - the Web developer blog]

1> Invoking JavaScript Functions from Go

```go
// +build js,wasm

package main

import (
  "syscall/js"
)

func main() {
  // Get the global JavaScript 'console' object
  js.Global().Get("console").Call("log", "Hello from Go!")
}
```

1> Handling JavaScript Callbacks in Go

```go
// +build js,wasm

package main

import (
  "syscall/js"
)

func main() {
  // Define a Go function to be called from JavaScript
  callback := js.FuncOf(func(this js.Value, p []js.Value) interface{} {
    // Handle the callback logic
    return "Callback invoked with: " + p[0].String()
  })

  // Assign the Go function to a JavaScript variable
  js.Global().Set("myGoFunction", callback)

  // Call a JavaScript function that will invoke the Go callback
  js.Global().Call("invokeCallback", callback)
}
```

Vulnerability Explanation

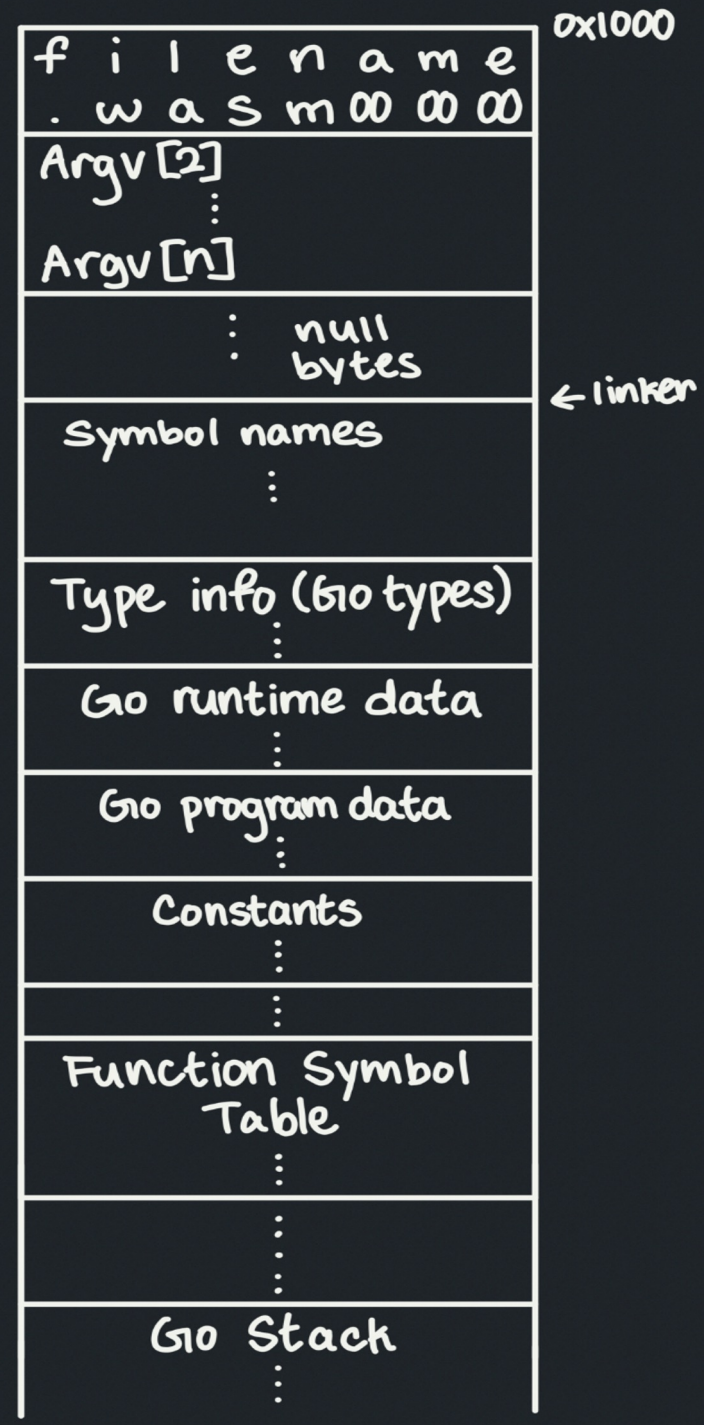
Memory Layout

- **Linear Memory:** Shared memory between JavaScript and GO. Arguments are populated in Linear memory by wasm_exec.js.

- **WebAssembly Memory Object (Browser side):** Maintains WASM stack and code.

Linear Memory Layout

- Linear memory starts with a zero page.
- Arguments are placed at 0x1000.
- Program specific data is loaded in linear memory.



Vulnerability

```
// Pass command line arguments and environment variables to WebAssembly by writing them to the linear memory.
let offset = 4096;

const strPtr = (str) => {
    // console.log(offset, str)
    const ptr = offset;
    const bytes = encoder.encode(str + "\0");
    new Uint8Array(this.mem.buffer, offset, bytes.length).set(bytes);
    offset += bytes.length;
    if (offset % 8 !== 0) {
        offset += 8 - (offset % 8);
    }
    return ptr;
};

const argc = this.argv.length;

const argvPtrs = [];
console.log("Argv list = ", this.argv)
this.argv.forEach((arg) => {
    argvPtrs.push(strPtr(arg));
});
argvPtrs.push(0);
```

Impact: Attack Vectors

- Overwrite type information to cause out of bound memory accesses.
- Overwriting shared variables in the data section.
- Overwrite entries in function symbol table.

Impact: Attack Vectors

- GO computes the hash of function symbol table at runtime and compares it with the compile time hash.
- Overwriting symbol table will result in a hash mismatch and will result in an exception.

```
for _, modulehash := range datap.modulehashes {
        if modulehash.linktimehash != *modulehash.runtimehash {
                println("abi mismatch detected between", datap.modulename, "and", modulehash.modulename)
                throw("abi mismatch")
        }
}
```

Impact: Affected Apps

Affected apps:

- Any server/application running go compiled wasm with user input into env vars or argv
- Any server that allows users to run go compiled wasm
- Example: IBM event streams

Potential impacts :

- wasm is relatively new… but it’s getting more widespread
- Many frontend heavy webapps that require live editing of images would likely migrate (Figma already uses webassembly, just not golang compiled wasm)

Our Exploit

Overwriting shared variables Using Buffer Overflow ArgV

The Vulnerable Application

2 ways to run wasm:

- *client side* in the web browser (containerized)
- server is running Go compiled wasm using Nodejs (less likely)

We chose the first scenario…

- Cannot break into user computer (without other vulnerabilities in the browser)
- Can possibly edit server contents through user input

Goal: Stored XSS attack (using our buffer overflow vulnerability)

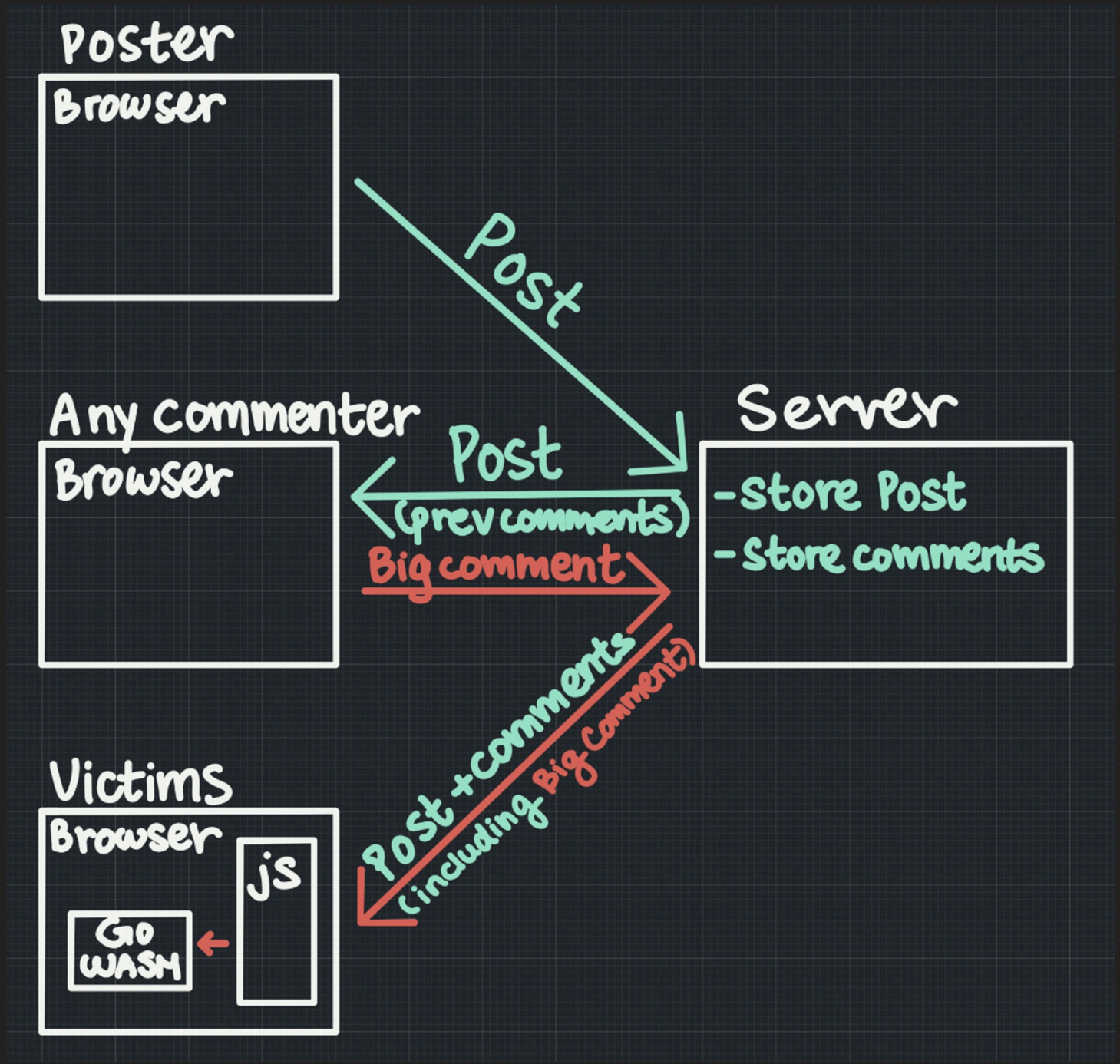
Requirements:

1) Server storing user input
2) Client side executes Go compiled wasm on user input (sent from server)

Our Scenario:

Social Networking Web App:

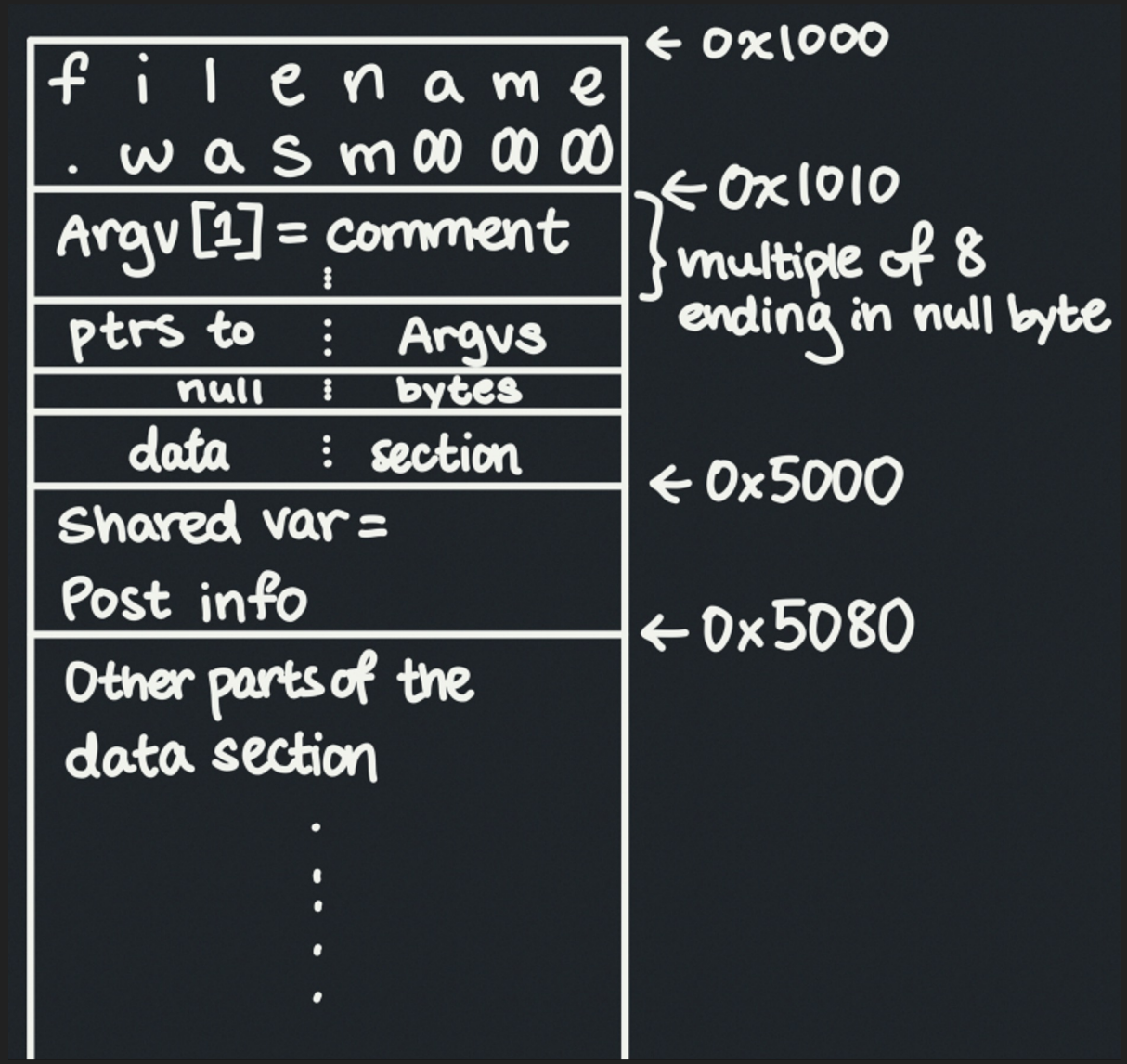
1) Stores user posts & comments
2) Render website using Go compiled wasm
    a) Specifically, **render each comment using Go compiled wasm**


Poster
Browser
Post
Any commenter
Browser
Post
(prev comments)
Big comment
Server
-Store Post
-Store comments
Post +comments
(including Big Comment)
Victims
Browser
JS
Go WASM

The Exploit

1) Find shared variable locations in vulnerable websites:
   - Inspect linear memory using browser inspect tools (built in)
   - Get objdump somehow (more difficult for wild servers)
   - We have a shared variable at linear memory location **0x5000**
   - Our Shared var = Post the comment is under
1) Find an argument that is controlled by user input
   - Our input: Go-wasm renders comments -> **Our comment**
   - Go compiled wasm argv starts at **0x1000**

Linear Memory Diagram

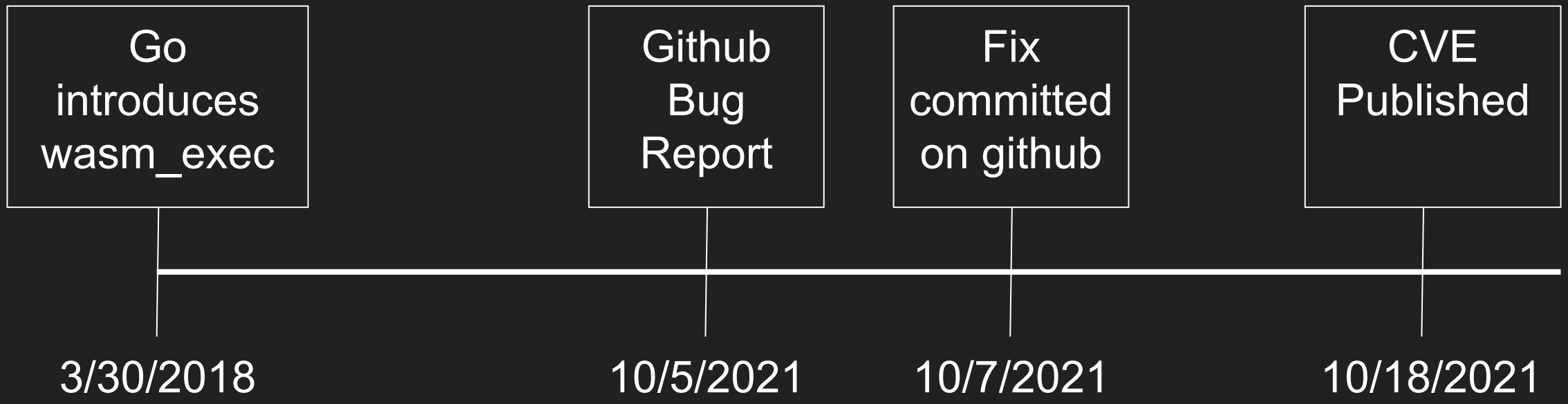


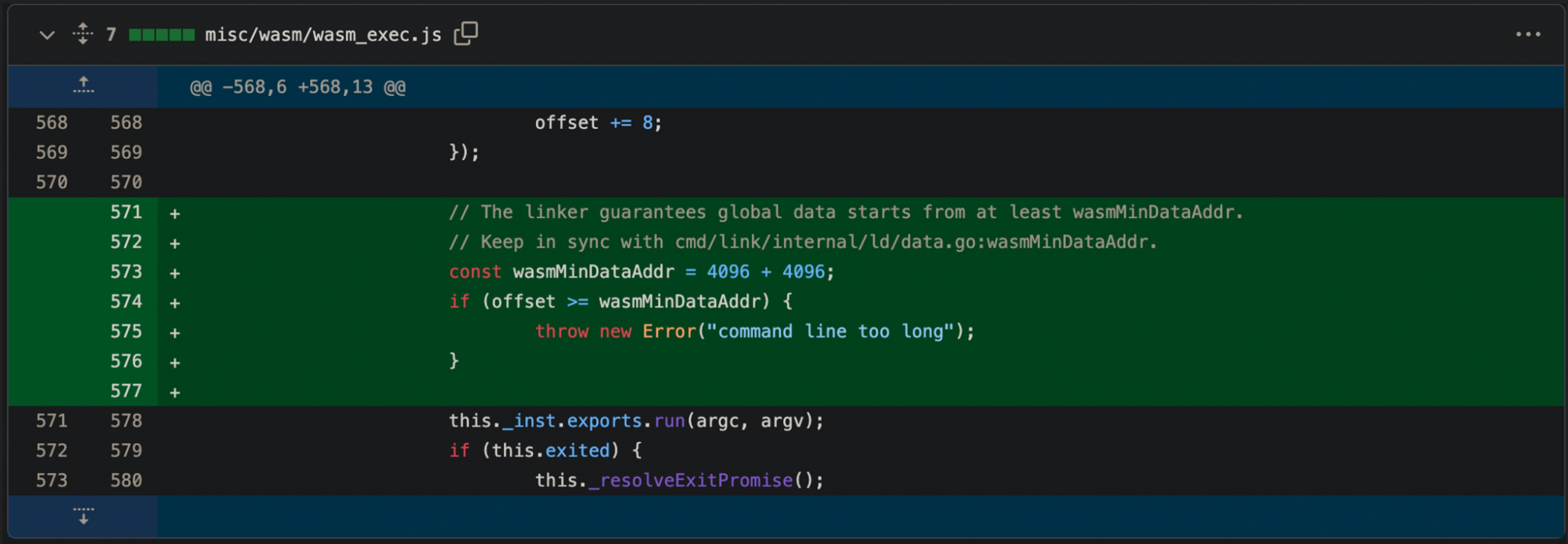
*we manipulated the linear memory so our shared var is towards the beginning of the data section

Live Demo

Fix

Timeline


Go introduces wasm_exec
Github Bug Report
Fix committed on github
CVE Published
3/30/2018
10/5/2021
10/7/2021
10/18/2021



https://github.com/golang/go/commit/77f2750f4398990eed972186706f160631d7dae4

- va = virtual address

```
      2445 + // On Wasm, we reserve 4096 bytes for zero page, then 4096 bytes for wasm_exec.js
      2446 + // to store command line args. Data sections starts from at least address 8192.
      2447 + // Keep in sync with wasm_exec.js.
      2448 + const wasmMinDataAddr = 4096 + 4096
      2449 +
2445  2450   // address assigns virtual addresses to all segments and sections and
2446  2451   // returns all segments in file order.
2447  2452   func (ctxt *Link) address() []*sym.Segment {
           @@ -2451,10 +2456,14 @@ func (ctxt *Link) address() []*sym.Segment {
2451  2456           order = append(order, &Segtext)
2452  2457           Segtext.Rwx = 05
2453  2458           Segtext.Vaddr = va
2454       -       for _, s := range Segtext.Sections {
      2459 +       for i, s := range Segtext.Sections {
2455  2460                   va = uint64(Rnd(int64(va), int64(s.Align)))
2456  2461                   s.Vaddr = va
2457  2462                   va += s.Length
      2463 +
      2464 +                 if ctxt.IsWasm() && i == 0 && va < wasmMinDataAddr {
      2465 +                         va = wasmMinDataAddr
      2466 +                 }
```

https://github.com/golang/go/commit/77f2750f4398990eed972186706f160631d7dae4

Live Demo Fixed

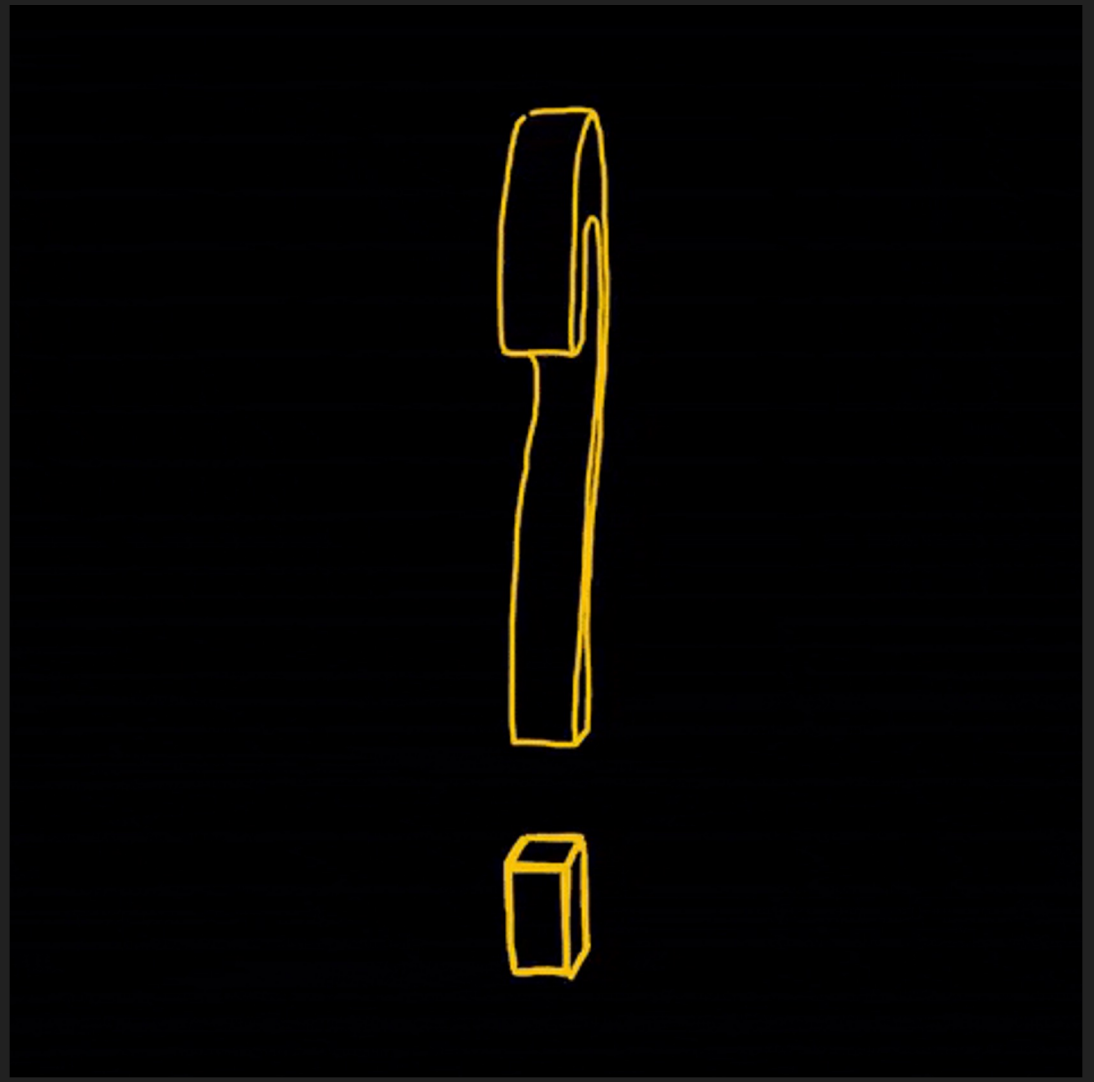
Reflection

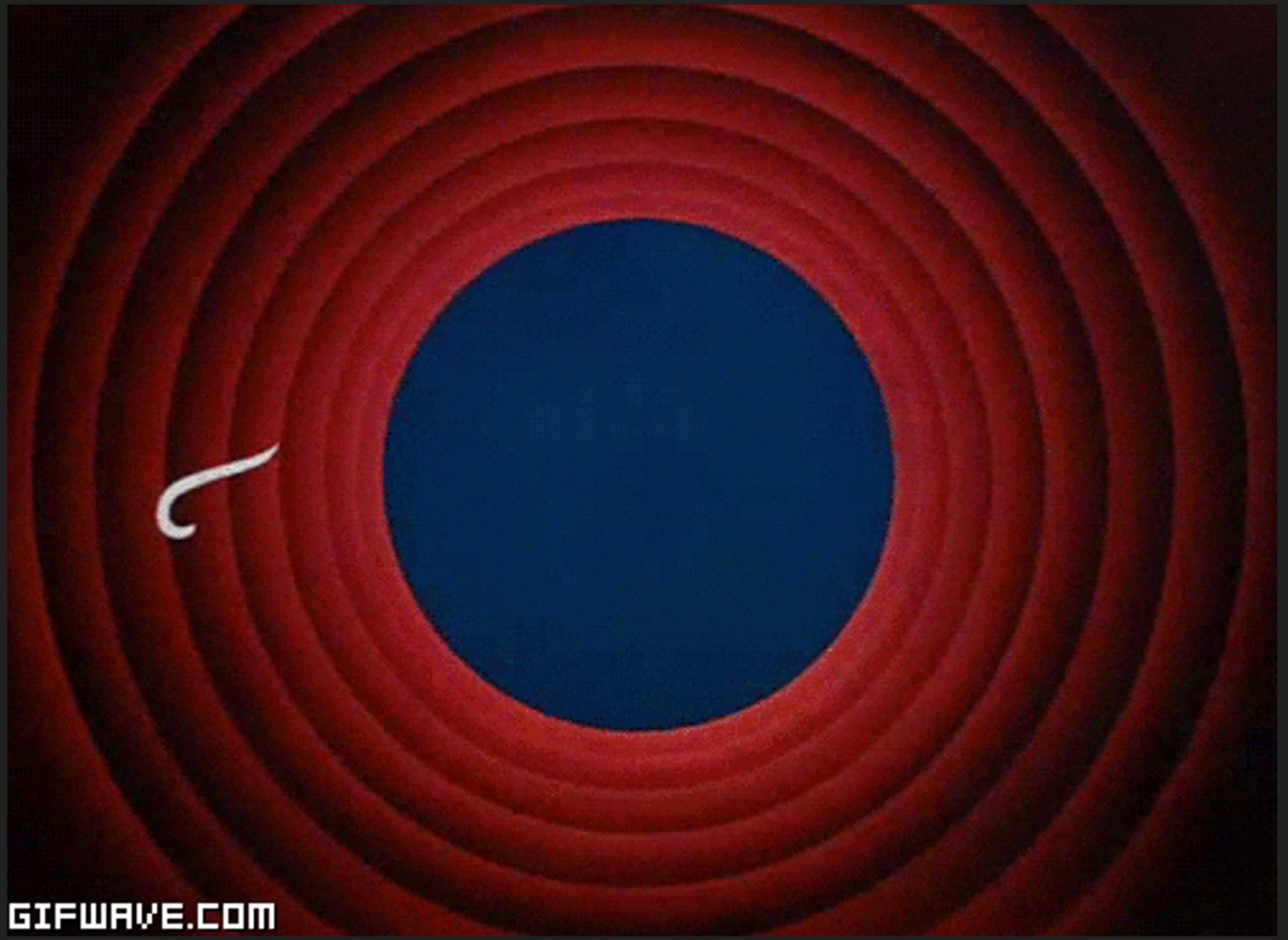
- Debugging tools help hackers 50000000%
  - wasmtime… doesn’t work for go compiled asm
  - wabt-tools - gives wasm->wat & objdump
    - no live debugging
- wasm adds attack vector to websites:
  - traditional website attack vectors: SQL injection, XSS
  - Adds (brings dangers of C into your website):
    - wasm memory management
    - language integration can introduce bugs (wasm is new & how to be safe hasn’t been ironed out)
- Due to Constraints: no existing exploits
- Article Misleading: CVE-2021-38297 - Go Web Assembly Vulnerability (jfrog.com)

When a large enough command-line parameter (or environment variable) is passed to the Wasm module – it will override the data of the actual Wasm module passed to the process.

In the exploitation, the attacker needs to overcome an obstacle – the UTF-8 encoding. During the copying of `argv` and the environment variables, `wasm_exec.js` will first encode them as UTF-8 strings. That means any input byte above 0x7F will be encoded in the output by two or more bytes. However, the attacker doesn't necessarily need to override the Wasm module with another "full" compiled Wasm module. It is possible to write a shellcode in Wasm instructions, which provides a useful payload to the attacker (such as running an arbitrary shell command) while only using Wasm binary opcodes in the allowed range of 0x00 – 0x7F.

Questions




GIFWAVE.COM

Sources

- Github Exploit (by current team)
- Security Bulletin: IBM Event Streams affected by potential buffer overflow in Golang (CVE-2021-38297)
- CVE-2021-38297 Go prior to 1.16.9 and 1.17.x prior to 1.17.2 h... (vulmon.com)
- pedromarquez.dev/blog/2023/2/node_golang_wasm
- NVD - CVE-2021-38297 (nist.gov)
- security: fix CVE-2021-38297 · Issue #48797 · golang/go (github.com)
- misc/wasm: add scripts for running WebAssembly binaries · golang/go@f632502 (github.com)
- CVE-2021-38297 - Go Web Assembly Vulnerability (jfrog.com)
- emscripten - Why is WebAssembly safe and what is linear memory model - Stack Overflow
- WebAssembly
- https://hacks.mozilla.org/2019/08/webassembly-interface-types/
- C wasm buffer-overflow
- Everything Old is New Again Binary Security of Webassembly
- Blog on GO Webasm ABI

Backup

- Exploit recording
- Link: https://cmu.zoom.us/rec/share/NQ0U_P5EQbbFMI8CrJox61BnY2X7_nbhRTwAlAxTUSXX5LiGAIBDGjgrvox8_7E.6UrnG01WMEBJmaUd
- Passcode: U9hbF%Lu