Resource Hog & Monitoring Extension

CYBR 525: Advanced Computer and Information Security Project Report

Eastern Washington University
School of Electrical Engineering and Computer Science

Team Members: Christopher Davisson
Instructor: Sanmeet Kaur
June 2025

abstract

Modern browsers, by use of WebAssembly (WASM), can execute near-native code. This enables the development of powerful applications but also introduces new attack vectors. This project demonstrates the real-world feasibility of a WASM-based resource exhaustion attack targeting CPU, memory, and network while exploring the browser's ability to detect and defend against such threats.

- A custom WASM resource hog compiled from C, simulating attacks directly in-browser.
- A Chrome extension designed to monitor and visualize per-tab resource usage, exposing the browser's internal blind spots and limitations.

Acknowledgement

I would like to express my sincere gratitude to Professor Sanmeet Kaur for her guidance, feedback, and support throughout this project.

Thanks also to my classmates in CYBR 525 for contributing to a collaborative and challenging environment that inspired deeper inquiry.

This project was made possible using open source tools, including Emscripten, Chrome DevTools, and the Chrome Extension API. Lastly, I appreciate the encouragement and patience of my dog Izzy,

who waited with me and always kept my lap warm.

Contents

List of Figures

List of Abbreviations

WASM WebAssembly

CPU Central processing unit

JS JavaScript

DoS Denial of Service attack

1. Introduction

1.1 Overview of the Project

This is a example project meant to demonstrate potential shortcomings in web browser security.

1.2 Problem Statement

WASM has enabled browsers to run high performance, low-level code much more efficently than ever before. However, this hace come with a significant risk, malicious WASM modules can silently exhaust system resources, causing degraded performance or facilitating browser-based crypto minind and DoS attacks. Browser users, web developers and extension developers have limited access to real time, per-tab resource usage making detection and response sluggish and difficult.

1.3 Objectives and Scope

Create both a web extension that can measure per-tab resource usage and a WASM based testbed to simulate attacks.

1.4 Importance and Potential Impact

Expansion of the testbed and extension could provide useful third-party alternatives to the standard methods.

2. Literature Review

3. Architecture and Methodology

3.1 System Architecture

3.1.1 A. WASM Resource Hog

- C code is compiled to WASM using `Emscripten (emcc)`.
- Exposes functions for:
  - CPU stress (tight floating-point loops)
  - Memory allocation (dynamic arrays)
  - Simulated network activity
- JavaScript glue code triggers these functions and tracks their execution.

3.1.2 B. Chrome Monitoring Extension

- Manifest v3 extension using a sidebar UI.
- Attempts to poll and visualize per-tab CPU, memory, and network metrics using the (limited) Chrome APIs.
- Maps tabs by ID, displays resource consumption, and (planned) color-coding and kill-switch for runaway processes.

4. System Implementation

4.1 Testbed Environment

- Localhost: WASM hog webpage running on local server port 8000
- Browser: Google Chrome Canary - v. 139.0.7222.0
- Extension: Custom sidebar extension loaded in developer mode

4.2 Implementation

- WASM Hog:
  - C code: Funcitons for CPU and memory abuse
  - Comptiled with Emscripten
  - JavaScript front-end triggers C functions usig ccall
- Extension:
  - Manifest v3, sidebar
  - chrome.tabs, chrome.processes API's
  - (Planned) Attach PID to tab titles and impliment 'kill' switch

5. Results and Analysis

5.1 Demo

- Launch WASM hog web page
  - Demonstrate each of the resources being attacked from multiple angles
- Open sidebar extension
- Demonstrate current limitations - non-functioning kill switch, lack of tab names

5.2 Results

- WASM can attack CPU and memory resources
- Browser defenses only function properly some of the time
- Partial Success
  - OS level details are not exposed by the chrome api
  - PID and tab ID are not reliably mapped
  - Network usage is reduced when done though WASM as opposed to JS

6. Discussion

This project really serves as a potential jumping off point. It lays the groundwork for the deeperm more comprehensive and real world testing. From a user perspective these attacks won't matter too much as most modern PC's come with CPUs that have multiple cores and the sandboxed environment of WASM means that only one core and one section of memory are effect. But this still serves as a potential threat for crypto jacking, some malicious developer using your electricity and processing power to generate wealth for themselves. I would also have like to find and impliment better defenses, but the closed off nature of Chrome made that hard. Perhaps with further research I can find a way to get around it so as to expand upon this.

7. Conclusion and Future Work

7.1 Conclusion

- WASM based resource exhaustion is a credible and reproducible browser level threat
- Chrome (and other browsers) do not provide sufficent, consistent APIs for extensions or users to reliably detect and mitigate these attacks
- Current browser level defense is reactive and incomplete

7.2 Future Scope

- Technical Enhancements
  - Enable kill switch and auto sorting
  - Integrate helper apps for native messaging and deeper OS telemetry
  - Demonstrate potential for cryptomining
- Security Research
  - Test against real-world WASM malware and cryptominers
  - Develop browser or OS hooks for user alerts to automate defenses

A. Appendix

WASM hog functions

```
static unsigned char *persistent_memory[MAX_MEMORY_MB] = {0};

void cpu_hog_for_seconds(double seconds) {
    double start = emscripten_get_now();
    volatile double x = 0.0001;
    while ((emscripten_get_now() - start) < seconds * 1000.0) {
        x += x * 1.0000001;
    }
}
EMSCRIPTEN_KEEPALIVE
void memory_hog(int megabytes) {
  for (int i = 0; i < megabytes; ++i) {
    if (!persistent_memory[i]) {
      persistent_memory[i] = (unsigned char*)malloc(1024*1024);
      if (!persistent_memory[i]) return;
      memset(persistent_memory[i], 0xFF, 1024*1024);
    }
  }
}
```

emcc build command

```
emcc hog.c -O3 -s WASM=1 -s ALLOW_MEMORY_GROWTH=1 \
-s INITIAL_MEMORY=64MB -s MAXIMUM_MEMORY=1024MB \
-s EXPORTED_FUNCTIONS="['_memory_hog', '_cpu_hog_for_seconds']" \
-s EXPORTED_RUNTIME_METHODS="['ccall']" -o hog.js
```