UM6P
University
Mohammed VI
Polytechnic

College of
Computing

Static Analysis of MIPS ELF Executable

Lab 2 Part 2

EL HANI Mohammed- Rida

Day 2 - Journey Through Embedded Chaos

Contents

1 Introduction

Alright, so after surviving the Windows PE ransomware adventure, I thought it was time to give embedded malware a shot. Spoiler alert: MIPS binaries are a different kind of hell. This report documents the static analysis of a suspicious ELF executable with SHA256 hash `981907e3f5ed07062b33b3e992d1f3412a2f3352208e92c1b58ff3c2387d50ae`, which turned out to be a textbook example of IoT botnet malware targeting MIPS-based embedded devices.

The sample clocks in at 203,096 bytes before unpacking and 202,436 bytes after, which is already suspicious because who loses weight after unpacking? The initial file command revealed this beauty: an ELF 32-bit MSB executable for MIPS R3000, statically linked, and proudly announcing that it has no section header. Classic obfuscation move, or just lazy compilation? Turns out it was packed with UPX, the malware author's favorite compression tool since 1996.

The goal here was simple: figure out what this thing does, where it came from, and who it's phoning home to. Armed with strings, rabin2, radare2, Ghidra, and an unhealthy amount of caffeine, I dove headfirst into the world of MIPS assembly, botnet C2 infrastructure, and the joys of analyzing stripped static binaries. What I found was an IoT botnet framework that checks for honeypots, spoofs HTTP headers, downloads additional payloads, and even drops a cryptocurrency miner called xmrig. Because why stop at DDoS when you can also mine Bitcoin on someone's router?

The analysis revealed hardcoded C2 IP addresses, anti-analysis tricks involving proc filesystem checks, remote shell execution capabilities, and enough command strings to suggest this is part of a larger botnet operation. The self-identification as iranbot combined with the xmrig component points to a specific threat actor ecosystem targeting MIPS-based IoT devices across the Middle East and beyond. But I'm getting ahead of myself. Let's break down what I found, byte by suspicious byte.

2 Sample Properties and Initial Fingerprinting

Running `file` on the sample gave us the first clue: this is an ELF 32-bit MSB (Most Significant Byte first, aka big-endian) executable compiled for MIPS R3000 architecture, version 1 of the SYSV ABI. The fact that it's statically linked immediately raised eyebrows because static linking in embedded malware usually means the author wants this thing to run on as many minimal Linux systems as possible without worrying about missing shared libraries. It's the malware equivalent of bringing your own food to a party because you don't trust the host's cooking.

The file header also noted that there's no section header, which is suspicious as hell and typically indicates either packing or intentional obfuscation. Section headers are used by loaders and debuggers to understand the binary's structure, so stripping them is a classic anti-analysis technique. However, a quick dive into the strings output revealed something hilarious: the very last strings in the binary proudly announce `This file is packed with the UPX executable packer` and multiple `UPX!` markers. So much for stealth. It's like wearing a disguise but leaving your ID badge on.

Using `rabin2 -I` to extract detailed binary information confirmed what we already

suspected. The architecture is MIPS with CPU type MIPS1, base address at `0x400000` (standard for ELF executables), and a binary size of 202,436 bytes after unpacking. The binary type is ELF, 32 bits, with big-endian byte order, which is typical for older MIPS-based embedded devices like routers and IoT cameras. The machine type is listed as MIPS R3000, an ancient architecture that's still widely deployed in cheap IoT hardware because manufacturers apparently think security updates are optional.

Security-wise, this thing is a disaster. The rabin2 output shows no stack canaries, no NX (Non-Executable stack) protection, no RELRO (Relocation Read-Only), and no PIE (Position Independent Executable). Basically, every modern exploit mitigation is turned off, which makes sense for IoT malware because these devices often run ancient toolchains that don't support modern hardening features. The binary is also stripped, meaning all symbols and debugging information have been removed, leaving us with nothing but raw addresses and function pointers to work with.

The language is listed as C, which tracks with what we see in the strings and structure. This isn't some exotic malware written in Rust or Go; it's good old-fashioned C code compiled with a basic MIPS cross-compiler. The compilation setup likely involved a simple embedded Linux toolchain, probably something like Buildroot or OpenWrt, targeting the o32 ABI (the 32-bit MIPS calling convention). The fact that it's statically linked and stripped suggests the author cared more about compatibility and anti-analysis than about binary size or performance.

3 Unpacking the UPX-Packed Binary

The first major hurdle was dealing with the UPX packing. UPX (Ultimate Packer for eXecutables) is a free, open-source executable packer that's beloved by both legitimate developers and malware authors. It compresses the binary and adds a small decompression stub that unpacks everything at runtime. The advantage for malware authors is that it makes static analysis harder and reduces the binary's signature footprint, though modern antivirus tools have no problem detecting UPX-packed files.

Unpacking was straightforward. I ran the following command to decompress the binary:

```
upx -d 981907e3f5ed07062b33b3e992d1f3412a2f3352208e92c1b58ff3c2387d50ae.elf.defanged
```

UPX spit out some warnings about bad b_info and recovery issues, which is typical for malformed or manually modified UPX-packed binaries. Some malware authors intentionally corrupt the UPX headers to break automated unpacking tools, but in this case, UPX managed to decompress it anyway. The resulting unpacked binary was only slightly smaller than the original, which suggests the compression ratio wasn't great. This makes sense because the binary is mostly code and strings rather than highly compressible data structures.

After unpacking, the section headers magically reappeared, revealing a standard ELF structure with .text, .rodata, .data, .bss, and several other sections. The fact that section headers were restored after unpacking confirms that they were deliberately hidden by UPX, not manually stripped by the malware author. This is a key distinction because manually stripped binaries are much harder to analyze than UPX-packed ones that can

be easily decompressed.

4 ELF Header Analysis and Section Layout

With the binary unpacked, it was time to examine the ELF header structure. Using readelf and various Ghidra views, I extracted the full ELF header details. The magic bytes are `7f 45 4c 46 01 02 01 00`, which translates to ELF, 32-bit, 2's complement big-endian, version 1, UNIX System V ABI. The ELF class is ELF32, data encoding is big-endian, and the version is current, meaning this is a valid, standards-compliant ELF file.

The ELF type is `EXEC`, which means this is an executable file rather than a shared object or relocatable file. The machine type is confirmed as MIPS R3000 with version `0x1`, and the entry point address is `0x400260`. This entry point is located very early in the .text section, which is normal for executables and suggests a straightforward startup routine rather than a convoluted loader stub like we saw in the Windows PE sample.

The program header table starts at byte 52 into the file, with each entry being 32 bytes long, and there are 3 program headers total. The section header table starts much later at byte offset 202,536, near the end of the file, with 14 section headers each 40 bytes long. The section header string table index is 13, which points to the section containing section names. The ELF header size is 52 bytes, which is standard for 32-bit ELF files.

The e_flags field shows `0x1007`, which decodes to several MIPS-specific flags: noreorder (tells the assembler not to reorder instructions for delay slot optimization), pic (position-independent code conventions), cpic (call position-independent code), and o32 (the 32-bit MIPS ABI). The presence of PIC-related flags is interesting because this is marked as an EXEC binary, not a shared object, but MIPS toolchains often generate PIC-style code even for executables to maximize compatibility.

Looking at the program headers, we see three segments: two LOAD segments and one GNU_STACK segment. The first LOAD segment maps the executable code starting at offset 0 in the file to virtual address `0x400000` with read and execute permissions. This segment has a file size of 0x2c340 bytes and a memory size of 0x2c340 bytes, meaning there's no additional memory allocation beyond what's in the file. The second LOAD segment maps the data sections starting at file offset 0x2c344 to virtual address `0x46c344` with read and write permissions. This segment has a file size of 0x5380 bytes but a memory size of 0x10a2cc bytes, meaning a large chunk of memory is allocated for .bss and .sbss sections that don't exist in the file but are initialized to zero at load time.

The GNU_STACK segment has all permissions set to RWE (read, write, execute), which confirms what rabin2 told us: there's no NX protection on the stack. This is a major security weakness because it means an attacker can execute code from the stack, making buffer overflow exploits trivial. Of course, this malware is the attacker, so it probably doesn't care about its own security posture.

5 Section Structure and Suspicious Indicators

The section table revealed 14 sections, which is pretty standard for a statically linked binary. Let's walk through the interesting ones. The .text section starts at virtual address

`0x400120` with a size of 0x29ef0 bytes (approximately 168 KB), and it has AX permissions (allocated and executable). This is where the bulk of the malware's code lives. The size makes sense for a statically linked binary that includes all the C library functions inline rather than relying on shared libraries.

The .rodata section sits at `0x42a070` with a size of 0x22d0 bytes (about 8.7 KB) and has read-only permissions. This section contains all the constant strings and read-only data structures, which is where we found all those juicy IoC strings during the strings analysis. The fact that this section is relatively small suggests the malware doesn't have extensive configuration data or large lookup tables embedded.

The .data section is at `0x46c460` with a size of 0x48e0 bytes (approximately 18.6 KB) and has read-write permissions. This section contains initialized global and static variables that need to be writable at runtime. The presence of writable data sections is normal, but the size suggests there's a fair amount of runtime state being tracked, which is consistent with a botnet that needs to maintain socket connections, command queues, and target lists.

Now for the suspicious stuff. The .ctors and .dtors sections are present at addresses `0x46c344` and `0x46c350` respectively, both with sizes of 0xc bytes (12 bytes) and read-write permissions. These sections contain constructor and destructor function pointers that are called before main() and after main() returns. Malware loves to hide initialization code in constructors because many analysts focus on main() and miss the early setup logic. I checked the contents of .ctors using radare2, and it showed the classic terminator values `0xFFFFFFFF` and `0x00000000`, which means the constructor table is empty. So no hidden constructor tricks here, just standard compiler-generated boilerplate.

The .data.rel.ro section at `0x46c35c` is another interesting one. This section contains data that needs to be relocated at load time but should be read-only after relocation. It's often used for function pointer tables and other security-sensitive data structures. The presence of this section in a non-PIE executable is a bit unusual but not unheard of in statically linked MIPS binaries.

The .bss and .sbss sections (at `0x471700` and `0x4716c4` respectively) are NOBITS sections, meaning they don't occupy space in the file but are allocated in memory at load time and initialized to zero. The .bss section is huge at 0x104f10 bytes (approximately 1 MB), which is a massive amount of uninitialized data. This strongly suggests the malware allocates large buffers at runtime, probably for network traffic, payload staging, or tracking compromised devices. The .sbss section (small bss) is much smaller at 0x38 bytes and follows the MIPS convention of separating small global variables into their own section for faster access via the global pointer.

Finally, there's an .mdebug.abi32 section, which is a MIPS-specific debug section containing ABI information. The presence of this section is a fingerprint of the MIPS toolchain used to compile the binary and can help with attribution if we ever find similar samples compiled with the same toolchain.

6 Strings Analysis and Behavioral Indicators

This is where things got really interesting. Running strings on the unpacked binary produced a goldmine of behavioral indicators, configuration data, and even author commentary. The strings can be grouped into several categories: proc filesystem interactions,

network and HTTP functionality, file operations, command and control, anti-analysis, and payload staging.

The proc filesystem strings are extensive and include paths like `/proc/%s/*`, `/proc/self/exe`, `/proc/%d/cmdline`, `/proc/%d/stat`, and `/proc/%d/status`. These indicate the malware is heavily invested in examining the process environment, checking running processes, and potentially hiding itself or detecting analysis tools. There's even an explicit anti-honeypot string that says something about exiting if proc is missing, which is a dead giveaway that this malware checks for virtualized or sandboxed environments before proceeding with its payload.

The networking strings are what you'd expect from botnet malware. There are complete HTTP request templates including `GET %s HTTP/1.1`, `POST %s HTTP/1.1`, and `HEAD %s HTTP/1.1`, along with HTTP headers like `Host:`, `User-Agent:`, `Connection: keep-alive`, and `Accept: */*`. The User-Agent string is particularly funny: `Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029` which is the malware pretending to be Chrome on Windows 10. Because nothing says "legitimate MIPS IoT device" like claiming to be a Windows desktop browser.

The smoking gun for C2 infrastructure is the hardcoded IP address `87.121.84.11`. This is the command and control server the malware communicates with, and it's a high-value IOC for threat intelligence and network monitoring. Additional networking strings include parameter names like `httpmode=`, `icmp`, `udpplain`, `port`, `srcport`, `psize`, and `domain`, which suggest the malware supports multiple attack modes: HTTP flooding, ICMP flooding, and UDP flooding. The presence of source port manipulation and packet size control indicates this is designed for DDoS attacks, not just C2 communication.

Command execution strings are plentiful: `/bin/sh`, `%s 2>&1`, `!shellcmd`, `!update`, and `!kill`. The exclamation mark prefix suggests these are command codes sent from the C2 server to control infected bots. The `!shellcmd` command probably executes arbitrary shell commands on the victim device, `!update` downloads and installs updated malware versions, and `!kill` terminates the bot or specific processes. The `%s 2>&1` format string indicates shell command output is being captured and presumably sent back to the C2 server.

File operation strings include writable paths like `/tmp`, `/var`, and `/dev/shm`, which are common staging directories for downloaded payloads. The presence of download tool strings like `wget`, `curl`, `tftp`, and `ftpget` confirms the malware can fetch additional payloads from remote servers. This is typical second-stage behavior where the initial infection downloads more sophisticated tools or lateral movement frameworks.

The attribution jackpot came in the form of the string `iranbot.xmrig`. This is the malware self-identifying as part of the iranbot family and indicating it includes or downloads the xmrig cryptocurrency miner. Xmrig is a popular open-source Monero miner that's been weaponized by countless botnet operations because it's efficient, well-maintained, and easy to configure. The iranbot family is known for targeting IoT devices across the Middle East, so this attribution makes sense given the MIPS architecture and the overall feature set.

7 Radare2 Analysis and Function Discovery

With strings giving us a roadmap of functionality, it was time to dig into the actual code structure using radare2. After loading the unpacked binary and running automated analysis with `aaa`, radare2 reported some interesting statistics: 7 functions, 7692 cross-references, 11 calls, 273 strings, 0 symbols, 0 imports, code coverage of 4316 bytes, total code size of 181056 bytes, and a coverage percentage of 2 percent. Let me translate: this binary is absolutely massive, almost entirely stripped, and radare2's automated analysis barely scratched the surface.

The function count of 7 is hilariously low for a binary this size. In a normal, unstripped executable, you'd expect hundreds or thousands of functions. The fact that radare2 only found 7 suggests that either the control flow is heavily obfuscated, or the binary uses jump tables and indirect calls that confuse static analysis tools. The cross-reference count of 7692 is extremely high, indicating there's a ton of internal string references and data accesses happening, which tracks with all the format strings and configuration data we saw in the strings output.

Looking at the function list with `afl`, we see entry0 at `0x400260`, a few tiny helper functions, and one massive function at `0x41c2d8` with 34 basic blocks and 928 bytes of code. This giant function is almost certainly the main event loop or command dispatcher, which is typical for botnet malware where all the logic is crammed into one massive state machine. The other functions are likely wrappers, setup routines, or utility functions for string manipulation and network I/O.

The entry point function at `0x400260` is named entry0 by radare2. Examining it shows a typical C runtime startup pattern: it sets up a stack frame, calls a larger initialization function (which is probably the MIPS equivalent of __libc_start_main), and then enters an infinite loop or noreturn state. This is consistent with daemon-style malware that's designed to run indefinitely in the background.

The monster function at `0x41c2d8` is where the real action happens. With 34 basic blocks, this function has complex control flow with multiple conditional branches, likely representing different command handlers for different C2 instructions. Without spending hours manually labeling and tracing this function, it's impossible to say exactly what each block does, but the size and complexity strongly suggest this is the command parsing and execution logic. This is the function that reads commands from the C2 server, parses the command type, and dispatches to the appropriate handler function.

8 Ghidra Analysis and Decompilation Attempts

After hitting the limits of radare2's automated analysis, I loaded the binary into Ghidra hoping for better results. Ghidra recognized it as MIPS 32-bit big-endian (MIPS:BE:32) and correctly identified the image base as `0x400000`. It created 14 memory blocks matching the ELF sections we saw earlier, and reported approximately 1.27 MB of loaded memory footprint when including the NOBITS sections like .bss and .sbss.

The import summary showed the harsh reality: 0 instructions analyzed initially, only 1 function discovered before analysis, and the binary marked as non-relocatable. This is what happens when you feed Ghidra a stripped, statically linked MIPS binary. There

are no imports to use as analysis anchors, no symbols to guide the disassembler, and no relocations to help identify function pointers. Ghidra's auto-analysis is powerful, but it needs something to work with.

After running full analysis, Ghidra managed to identify more functions than radare2, but we're still talking about dozens rather than hundreds. I attempted to locate specific behavior by searching for the suspicious strings we identified earlier, hoping to find xrefs that would lead me to the relevant code sections. Unfortunately, almost all the strings were just sitting in the .rodata section with no clear xrefs to specific functions, which suggests they're being accessed through computed addresses or string tables rather than direct references.

I spent some time decompiling the entry point and that massive function at `0x41c2d8`, and the decompiled C code confirmed my suspicions. The entry point function (which Ghidra called processEntry) sets up a small stack frame and calls `FUN_0041c2d8` with what looks like a pointer to the real main function and an initialization callback. After that call, it drops into an infinite loop, which is exactly what you'd expect for embedded malware that runs as a daemon.

The massive function turned out to be a C runtime startup wrapper, essentially the static MIPS equivalent of __libc_start_main. It walks through the argv array, computes the envp pointer, iterates through the auxiliary vector (auxv) copying low-numbered entries, performs secure mode checks by comparing UID and EUID, initializes global pointers for the program name and error handling, and finally calls the real program entry point. So this function is just startup glue, not the malware logic.

This is where I hit the wall. The binary is so large, so stripped, and so devoid of useful symbols that attempting to reverse engineer it function by function would be a suicide mission. Without dynamic analysis to observe runtime behavior, identify which functions actually execute, and trace the call graph, static analysis can only take us so far. The malware author achieved their goal: make static analysis so painful that most analysts give up. But we still extracted enough behavioral indicators and IOCs to understand what this malware does and how to detect it.

9 Anti-Analysis and Honeypot Detection

One of the most interesting aspects of this malware is its anti-analysis and environment detection capabilities. The extensive proc filesystem checks aren't just for process enumeration; they're also used to detect honeypots and analysis environments. The explicit string about exiting if proc is missing is a dead giveaway. Many malware sandboxes and honeypots use lightweight virtualization or containers that don't fully implement the proc filesystem, or they implement it incorrectly. By checking for the presence and validity of proc entries, the malware can detect these fake environments and refuse to execute.

The proc checks also serve another purpose: process discovery and self-defense. By reading `/proc/%d/cmdline` and `/proc/%d/status`, the malware can identify other running processes, check for security tools or competing malware, and potentially kill them. The ability to read `/proc/self/exe` allows the malware to determine its own path on the filesystem, which is useful for persistence mechanisms and for restarting itself if it crashes.

The lack of standard exploit mitigations (no NX, no ASLR, no canaries) isn't just lazy compilation; it's intentional targeting of IoT devices that don't support these features.

Most embedded Linux systems use ancient toolchains and kernels that predate modern security features, so building a hardened binary would be pointless. The malware is designed to run on the lowest common denominator of embedded devices, which means embracing the security posture of 2005-era Linux.

10 Command and Control Infrastructure

The hardcoded IP `87.121.84.11` is the key to understanding this malware's C2 infrastructure. This IP likely hosts the command server that infected bots connect to for instructions. The presence of multiple attack mode strings (HTTP, ICMP, UDP) suggests the C2 server can send commands like "launch HTTP flood against this target" or "switch to UDP flood mode with these parameters". The port, srcport, and psize parameters give the botmaster fine-grained control over attack traffic characteristics.

The command strings with exclamation mark prefixes (!shellcmd, !update, !kill, plus others like !ping and !pong) indicate a simple command protocol. The bot probably maintains a persistent connection to the C2 server, receives command messages, parses the command type by looking for these string prefixes, and dispatches to the appropriate handler. The ping/pong mechanism is likely a keepalive to ensure bots are still alive and responsive.

The update mechanism is particularly concerning because it means this malware can evolve over time. The !update command probably downloads a new binary from a specified URL, saves it to a writable directory like /tmp, makes it executable, and restarts itself with the new version. This allows the botmaster to patch bugs, add new features, or change C2 infrastructure without losing control of infected devices.

11 Cryptocurrency Mining Component

The iranbot.xmrig string reveals that this botnet has a dual purpose: DDoS attacks and cryptocurrency mining. Xmrig is one of the most popular Monero mining tools, and it's been integrated into countless malware families because Monero mining is relatively CPU-efficient and provides untraceable payments. The mining component is probably downloaded as a second-stage payload after initial infection, or it might be embedded in the binary and extracted at runtime.

Mining on IoT devices makes sense from an attacker's perspective because these devices are always on, have decent CPUs (by embedded standards), and their owners rarely monitor CPU usage. A router mining Monero at low intensity might go unnoticed for months or years, generating passive income for the botmaster. The combination of DDoS-for-hire and mining creates two revenue streams: the botnet can be rented out for DDoS attacks, and idle bots can mine cryptocurrency.

12 Conclusions and the Reality Check

After hours of static analysis, strings extraction, disassembly review, and decompilation attempts, here's the bottom line: this is a sophisticated IoT botnet framework targeting

MIPS-based embedded devices, specifically routers, IP cameras, DVRs, and other always-on network appliances. The malware identifies itself as part of the IranBot family and incorporates the xmrig cryptocurrency miner, indicating a dual monetization strategy combining DDoS-for-hire services with passive cryptocurrency mining.

The technical sophistication is moderate. The use of UPX packing is trivial to bypass, the hardcoded C2 IP makes infrastructure tracking easy, and the lack of code obfuscation (beyond stripping symbols) means the binary is analyzable with standard tools. However, the static linking and targeting of ancient MIPS architecture shows strategic thinking: the authors prioritized broad compatibility and reliability over stealth.

What we couldn't determine from static analysis alone includes the exact encryption algorithms used for C2 communication (if any), the specific mining pool configurations for xmrig, the full command protocol specification, the persistence mechanisms used to survive reboots, and the lateral movement or propagation techniques. These questions would require dynamic analysis in a controlled environment with proper MIPS emulation or actual target hardware.

The defensive recommendations are straightforward: block the C2 IP `87.121.84.11` at the network perimeter, monitor for outbound connections to suspicious IPs from IoT devices, look for xmrig mining traffic patterns, implement proper network segmentation to isolate IoT devices, and for the love of all that is holy, change default credentials and update firmware on embedded devices.

From an attribution perspective, the IranBot family name combined with MIPS targeting and Middle Eastern infrastructure (based on IP geolocation) suggests threat actors operating in or targeting that region. The integration of xmrig indicates the actors are financially motivated rather than purely focused on disruption. The relatively weak OPSEC (hardcoded IPs, easily unpacked, clear self-identification) suggests this is mid-tier threat actor work, not nation-state level sophistication.

The next logical step would be setting up a MIPS emulation environment using QEMU, executing the malware under observation with network traffic capture, monitoring the proc filesystem checks to understand the honeypot detection logic, capturing the actual C2 protocol communications, and extracting the xmrig configuration to identify mining pools and wallets. But that's a project for another day when I have the energy to build a proper MIPS analysis sandbox. For now, I'm comfortable saying we extracted maximum value from static analysis alone.

The malware author succeeded in making deep static analysis painful enough that most analysts would give up, but they failed at operational security by leaving clear attribution markers and using trivially bypassed packing. It's the malware equivalent of wearing a disguise but carrying your driver's license in your pocket. Good enough to fool automated systems, not good enough to fool humans with time and motivation (and large amount of caffeine).