6 Detecting MIPS Emulation

by Craig Heffner

In this article, we'll look at some handy tricks for detecting the difference between real MIPS hardware and the Qemu emulator. First, in Section 6.1, we'll look at special function registers whose values in the emulator reveal the use of Qemu. Then, in Section 6.2, we'll intentionally run code which has a pending overwrite in the data cache to determine whether the instruction and data caches are synchronized with one other, as they are in Qemu but are not in real hardware. The techniques presented in this article were tested on Qemu v2.0.1.

6.1 Detection through hardware registers

Qemu can be identified with a reasonable level of certainty by examining discrepancies in the MIPS `CP0` (Coprocessor0) registers. The most obvious register to examine is the `PRId` (Processor ID) register, shown in Figure 4.

```
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                    |Company  Options|    Company ID   |      CPU ID     |     Revision    |
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
QEMU                |0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 1|1 0 0 0 0 0 1 1|0 0 0 0 0 0 0 0|
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Atheros AR7240 SoC  |0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 1|1 0 0 1 0 0 1 1|0 1 1 1 0 1 0 0|
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Ralink RT3352F SoC  |0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 1|1 0 0 1 0 1 1 0|0 1 0 0 1 1 0 0|
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Company Options	Reserved for use by the manufacturer.
Company ID	Uniquely identifies the manufacturer, but is set to 0 for older processors as it was not defined in the MIPS specification.
CPU ID	Identifies the specific MIPS CPU type. (MIPS 4KC, MIPS 24K, etc)
Revision	Used to specify the CPU core revision number.

Figure 4: Processor ID (`PRId`) Register

The `PRId` register can be read using the `mfc0` (move from coprocessor0) instruction.

```
mfc0 $t0, $15   ; Move CP0 register 15 (PRId) into general purpose register $t0
```

Figure 4 also shows the differences between Qemu and two common system-on-chip devices that are found in real hardware. Note in particular the differences in the `Revision` field. Qemu sets this field to all zeros regardless of which MIPS core is being emulated, but most real-world systems will have this field set to a non-zero value representing the major/minor/patch version of the MIPS core in use by that CPU.[9]

It is also useful to examine the `Config` register. Much like `PRId`, the `Config` register can be read using the `mfc0` instruction.

```
mfc0 $t0, $16   ; Move CP0 register 16 (Config) into general purpose register $t0
```

[9]Programming the MIPS32 24K Core Family, Section 2.2

```
                        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                        | |                               |B|   |     |     |     |V|     |
                        |M|              Impl             |E| AT| AR  |  MT |0 0 0|I|  K0 |
                        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Qemu                    |1|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1|0  0|0 0 0|0 1 1|0 0 0|0|0 1 0|
                        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Atheros 7240 SoC:       |1|0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0|1|0  0|0 0 1|0 0 1|0 0 0|0|0 1 1|
                        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Ralink RT3352F SoC:     |1|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1|0|0  0|0 0 1|0 0 1|0 0 0|0|0 1 1|
                        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

`M`	`1` if there is another config register. (`Config1`)
`Impl`	implementation specific.
`BE`	`1` if the processor is big endian, `0` for little endian.
`AT`	Specifies whether the processor supports MIPS32 (0) or MIPS64 (1 == MIPS32 address map; 2 == full address map) encoding.
`AR`	Architecture Revision level (0 == MIPS32/64 release 1; 1 == MIPS32/64 release 2).
`MT`	Specifies the MMU type.
`0 0 0`	Unused
`VI`	Set to 1 if the L1 instruction cache uses virtual tagging.
`K0`	Specifies the MIPS kseg0 memory region's caching behavior.

Figure 5: `Config` register

Again, we can find some general differences in register values between different CPUs, which are shown in Figure 5. Most notably, `Impl` is zero in Qemu, while the Atheros and Ralink chips have this field set to non-zero values. The PIC32 datasheet also notes that it uses these bits to store information regarding segment caching and the SRAM bus interface.[10]

These register variations are generally reliable, and are particularly applicable if you expect to only run on one particular CPU, such as an exploit for a specific target.

6.2 Detection in Linux user space

Examining CPU hardware registers requires execution in kernel mode. But, for many Linux based MIPS devices, you may be executing from Linux user space. Here, you may simply examine `/proc/cpuinfo`, which in Qemu typically looks something like the following:

```
root@qemu:~# cat /proc/cpuinfo
system type             : MIPS Malta
processor               : 0
cpu model               : MIPS 24Kc V0.0  FPU V0.0
BogoMIPS                : 2097.15
wait instruction        : yes
microsecond timers      : yes
tlb_entries             : 16
extra interrupt vector  : yes
hardware watchpoint     : yes, count: 1, address/irw mask: [0x0ff8]
ASEs implemented        : mips16
shadow register sets    : 1
core                    : 0
VCED exceptions         : not available
VCEI exceptions         : not available
```

[10] PIC32 Reference Manual, `61113E.pdf`

First, most real MIPS systems will set `system type` to reflect the SoC vendor, such as "Ralink SoC" or "Broadcom BCM5357 chip rev 2". It would be extremely unlikely to see MIPS Malta on a production system.

More importantly, `BogoMIPS` as reported in Qemu is a reflection of the *host machine's* CPU speed. 2,097 BogoMIPS would be insane for a real MIPS processor, which typically clocks in around 400MHz. More realistic `BogoMIPS` values for MIPS CPUs would be in the 200-300 range.

6.3 Execution-based detection

While the above detection methods are useful, they could easily be changed or patched, either by an end user or in future Qemu releases. A far more reliable method of detection is through the use of fundamental architecture features that are not properly emulated by Qemu and not easily implemented.

Qemu can be reliably detected by exploiting cache incoherency, which is inherent in MIPS CPUs but absent from Qemu.[11]

The MIPS cache is divided into two sections: one for instructions, and one for data. When data is written to memory, that data is first stored in the data cache, and is eventually written back to main memory at a later time. Instructions, as you may well guess, are cached in the instruction cache.

This is a common issue during MIPS exploitation. Let's say that we write some shellcode to a buffer; that shellcode is treated as data, and cached in the data cache. If we try to jump into that shellcode, however, the CPU will go looking for it in the instruction cache; since it is not cached there, the CPU then fetches the instructions from main memory. But the shellcode isn't in main memory, it's in the data cache!

This problem is typically mitigated by first flushing the data cache back to main memory before jumping into the buffer containing the shellcode. Cache flushes can be performed explicitly in MIPS through the `synci` or `cache` instructions, or by simply waiting a bit (e.g., `sleep`(1)) and letting the kernel do a cache flush, which will typically need to happen periodically anyway.

Qemu does not even try to emulate this cache behavior, and we can use that to our advantage by
1) writing a block of code to an address in memory,
2) executing `synci` to make sure the code is written back from the data cache to main memory,
3) writing a second block of code to the same address in memory, and then
4) immediately jumping to the memory address.

When running on MIPS hardware, the second code block is still sitting in the data cache, and the *first* block of code will be fetched from main memory and executed. However, in Qemu, since caching is not emulated, the second code block will overwrite the first, and the *second* block of code will be executed.

Thus, we can execute two completely different sets of code from the same memory address; one piece of code will be executed when running in Qemu, and the other piece of code will be executed when running on real MIPS hardware:

```
1  /*
   * PoC code which executes different pieces of code from the same address
3  * in Qemu vs real MIPS hardware.
   *
5  * On real MIPS hardware, main should return 1.
   * In Qemu, main should return 2.
7  *
   * Tested against Qemu 2.0.1 and a Broadcom BCM5357 (MIPS 74K) SoC.
9  *
   * Requires a MIPS32r2 compliant compiler.
11 */

13 #include <stdio.h>
   #include <stdlib.h>
15 #include <string.h>

17 #define CODE_SIZE 8
```

[11] Linux MIPS Wiki, Qemu Processor

```
/*
 * ret1 contains a MIPS function that returns 1.
 * ret2 contains a MIPS function that returns 2.
 */

/*
 * Big endian
 *
char ret1[CODE_SIZE] =
     "\x03\xe0\x00\x08"  // jr $ra
     "\x24\x02\x00\x01"; // li $v0,1
char ret2[CODE_SIZE] =
     "\x03\xe0\x00\x08"  // jr $ra
     "\x24\x02\x00\x02"; // li $v0,2
*/

/* Little endian */
char ret1[CODE_SIZE] =
         "\x08\x00\xe0\x03"  // jr $ra
         "\x01\x00\x02\x24"; // li $v0,1
char ret2[CODE_SIZE] =
         "\x08\x00\xe0\x03"  // jr $ra
         "\x02\x00\x02\x24"; // li $v0,2

int main(void) {
    int(*s)(void);
    int retval = 0;
    char buf[CODE_SIZE] = { 0 };

    /* The s function pointer points to buf */
    s = (void *) &buf;

    /* 1. Copy ret1 into buf (ret1 is now in the data cache)
     * 2. Execute the synci instruction to flush the data cache (ret1 is now in main memory)
     * 3. Copy ret2 into buf (ret2 is now in the data cache)
     * 4. Call the function located in buf (should fetch and execute ret1 from main memory)
     */
    memcpy(buf, ret1, sizeof(buf));
    asm ("synci 0(%0)": : "r" (buf));
    memcpy(buf, ret2, sizeof(buf));
    retval = s();

    printf("retval = %d\n", retval);
    return retval;
}
```

Because `synci` is not a privileged instruction, this method can be used in both user and kernel space. The only downside is that `synci` was not introduced until MIPS32r2, so older MIPS processors don't support that particular instruction. Since MIPS32r2 was introduced in 2003, it's unlikely that this will be an issue unless you're dealing with an older processor; in such an event, you'll need to use some alternate method of flushing the cache. This can be done in kernel space with the `cache` instruction, or in Linux user space, you can simply replace `synci` with a call to `sleep`(1).

It's worth noting that in theory, the second block of code (`ret2`) could be executed when running on real MIPS hardware if the kernel flushed the cache behind your back in between the time that `ret2` is copied into `buf` and the time that you actually call into `buf`. However, this would be a very unlucky edge case which I have yet to encounter in practice, provided the time between the second `memcpy` to `buf` and the call to `buf` is minimized. `ret1` is never executed in Qemu.