Capable VMs

23 Apr 24 - CHERITech '24

Jacob Bramley, Andrei Lascu, Duncan Lowther, Dejice Jacob, Jeremy Singer, Laurie Tratt and Tom Wallis
Virtual machines (VMs, also known as managed language runtimes) are ubiquitous components in the modern software stack. They power the web, running in client-side browsers, server-side applications, and smartphone apps. In any ranking of popular programming languages, at least **half of the top ten languages run on VMs** (e.g. Python, Java, C#, JavaScript, PHP).

A key problem is that **VM security** has traditionally been a secondary concern relative to performance. Industrial strength VMs have large, complex code-bases, and large numbers of hand-crafted optimizations. Not only are they beyond any one person's ability to understand, but security has tended to be treated reactively: mature, widely used VMs such as HotSpot (the standard Java VM) regularly have **50-100 CVEs** per year.

The **CapableVMs** project hypothesises that **CHERI hardware-enforced capabilities** are the first realistic technique to **make VM security proactive**.
Why are virtual machines special?

- lots of low-level platform-specific systems code
- multiple interacting dynamic components
- intensive allocation and garbage collection
- runtime code generation
Why are virtual machines special?

- lots of low-level platform-specific systems code
- multiple interacting dynamic components
- intensive allocation and garbage collection
- runtime code generation
Why are virtual machines special?

- lots of low-level platform-specific systems code
- multiple interacting dynamic components
- intensive allocation and garbage collection
- runtime code generation
1. Low-level system-specific code

- CHERIfication
- Specific porting process
- Measure proportion of LoC altereded
- KDE: 0.026%
- higher for systems code
  - 0.18% for MicroPython
  - 1% for snmalloc
import dodgylib

tiny1 = bytearray(3)
tiny2 = bytearray(12)

# setup 'Oh' raw string
tiny1[0] = 0x4f  # O
tiny1[1] = 0x68  # h
tiny1[2] = 0x00  # \0

# setup 'Hello' raw string
tiny2[0] = 0x48  # H
tiny2[1] = 0x65  # e
tiny2[2] = 0x6c  # l
tiny2[3] = 0x6c  # l
tiny2[4] = 0x6f  # o
tiny2[5] = 0x00  # \0

dodgylib.dodgy(tiny1)

print(tiny1.decode('utf-8'))

print(tiny2.decode('utf-8'))

root@amarena:~ # ./.micropython-hybrid.exploit.py

Oh

HACK!!
import uctypes as uct

def dodgy(x):
    ptr = uct.addressof(x)
    unsafe = uct.bytearray_at(ptr, 2000)
    i = 0
    while unsafe[i] != 0x65 or unsafe[i+1] != 0x6c:
        i += 1
        if i > 2000:
            break
    unsafe[i] = 0x41
    unsafe[i+1] = 0x43
    unsafe[i+2] = 0x4b
    unsafe[i+3] = 0x21
    unsafe[i+4] = 0x21
    unsafe[i+5] = 0x00
    return

dodgylib.dodgy(tiny1)
dodgylib.dodgy(tiny2)

root@amarena:~ # ./micropython-purecap exploit.py
Oh
In-address space security exception (core dumped)
CVE-2023-7158 Description

A vulnerability was found in MicroPython up to 1.21.0. It has been classified as critical. Affected is the function slice_indices of the file objslice.c. The manipulation leads to heap-based buffer overflow that may be used. The exploit has been disclosed to the public and may be used. Upgrading to version 1.22.0 is able to be used. The patch is identified as 8b24aa36ba978eafc6114b6798b477bfecdca26. It is recommended to apply a patch to fix this issue. VDB-249158 is the identifier assigned to this vulnerability.

CVE-2023-7152 Description

A vulnerability, which was classified as critical, has been found in MicroPython 1.21.0/1.22.0-preview. Affected by this issue is the function poll_set_add_fd of the file extmod/modselect.c. The manipulation leads to use after free. The exploit has been disclosed to the public and may be used. The patch is identified as 8b24aa36ba978eafc6114b6798b477bfecdca26. It is recommended to apply a patch to fix this issue. VDB-249158 is the identifier assigned to this vulnerability.
Other findings from MicroPython

• Pointer size assumptions
  • don't affect correctness only
  • they also have an impact on performance

• Porting to a variety of platforms
  • Morello: [github.com/glasgowPLI/micropython](https://github.com/glasgowPLI/micropython)
  • working on CHERIoT RISC-V Ibex core
Why are virtual machines special?

- lots of low-level platform-specific systems code
- intensive allocation and garbage collection
- runtime code generation
- multiple interacting dynamic components
2. Multiple interacting components

- compartmentalization (c18n) is **lightweight isolation**
- hybrid code enables DDC-based isolation
- need a **compartment switcher**
- need a **libc** per compartment
- need clever tricks to handle dynamic loading
- overhead - how small should each compartment be?
  - compartment per *function*
  - compartment per *shared object*
  - alternative compartment boundaries?
Alternative c18n strategy

• For purecap MicroPython code
• We isolate at FFI boundaries
  • e.g. calls to external C libraries
  • (work in progress)
Why are virtual machines special?

- lots of low-level platform-specific systems code
- multiple interacting dynamic components
- intensive allocation and garbage collection
- runtime code generation
3. Malloc and GC

Complications include:
• finding and tracing the root set
• scanning the full heap
• moving objects

• We have studied **BDWGC**
• Morello & RISC-V: [github.com/capablevms/bdwgc](https://github.com/capablevms/bdwgc)
Observations about purecap GC

• Can't afford to lose capability tags
• conservative -> precise
• overhead reduction!
• issue with sealed caps in userspace code
• issue with coalescing
int main() {
    int i, size, *buffer;

    srand(SEED);

    for (i=0; i<NUM_ALLOCS; i++) {
        size = rand() % 1024;
        // printf("alloc buffer size \%d\n", size);
        buffer = (int *)malloc(size * sizeof(int));

        // confuse conservative collector?
        savebuffer[i%(NUM_ALLOCS>>5)] = (long int)buffer;
    }

    return 0;
}

Why are virtual machines special?

- lots of low-level platform-specific systems code
- multiple interacting dynamic components
- intensive allocation and garbage collection
- runtime code generation
4. Runtime code generation

• Several baseline interpreters ported to Morello purecap: WARDuino, MicroPython, JSC

• Some investigations on runtime code generation: JSC (& v8)

• This is work-in-progress
Summary

• Our Capable VMs project has demonstrated that

CHERI does provide defence-in-depth against VM-based exploits

Challenges include:

1. how to quantify additional defence?
2. how to measure performance?
3. how to encourage adoption?