Towards the Higher Level Debugging with QEMU

Pavel Dovgalyuk, ISP RAS

About us

- Ivannikov Institute for System Programming of the RAS
- Emulation-related projects
- Full system record/replay in mainline QEMU
- VM introspection and instrumentation
- Stealth WinDbg stub for QEMU
- Reverse debugging patches ready for 4.3 (or 5.0?)

- https://github.com/ispras/swat

Plan

- Approaches to system-wide debugging
- Problems of system-wide debugging
- New ideas and proposals

Debugging with QEMU/KVM

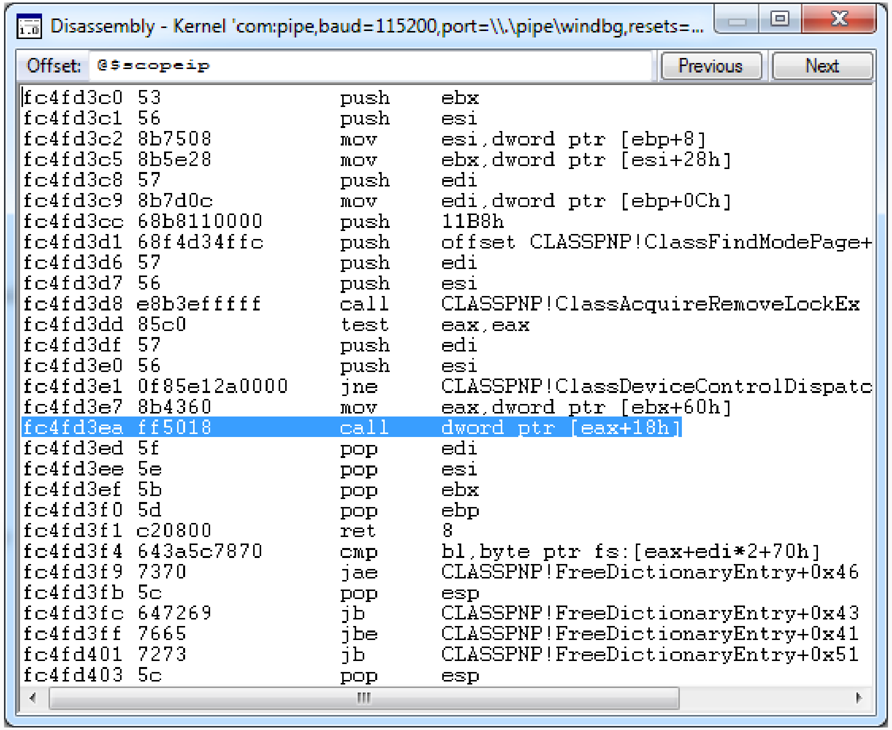
- QEMU/KVM
  - Debugging OS/drivers/BIOS
  - Malware analysis
- QEMU only
  - Execution recording (time travel debugging)
  - Cross-platform debugging

Debugger functions

- Processes
  - Pages
  - Threads/Fibers
  - Process switches
- Executables
  - Memory areas
  - Function names
  - Variable names
  - Call stack
- Breakpoints
  - Memory access
  - Register access
- Events
  - Exceptions
  - Interrupts
  - System calls
  - I/O

Full system debugging with WinDbg

- OS debug mode has to be enabled
- Has complete kernel information
- Can debug separate processes
- Unofficial stub for QEMU
- Windows only



Disassembly - Kernel 'com:pipe,baud=115200,port=\\.\pipe\windbg,resets=...

```
fc4fd3c0 53              push    ebx
fc4fd3c1 56              push    esi
fc4fd3c2 8b7508          mov     esi,dword ptr [ebp+8]
fc4fd3c5 8b5e28          mov     ebx,dword ptr [esi+28h]
fc4fd3c8 57              push    edi
fc4fd3c9 8b7d0c          mov     edi,dword ptr [ebp+0Ch]
fc4fd3cc 68b8110000      push    11B8h
fc4fd3d1 68f4d34ffc      push    offset CLASSPNP!ClassFindModePage+
fc4fd3d6 57              push    edi
fc4fd3d7 56              push    esi
fc4fd3d8 e8b3efffff      call    CLASSPNP!ClassAcquireRemoveLockEx
fc4fd3dd 85c0            test    eax,eax
fc4fd3df 57              push    edi
fc4fd3e0 56              push    esi
fc4fd3e1 0f85e12a0000    jne     CLASSPNP!ClassDeviceControlDispatc
fc4fd3e7 8b4360          mov     eax,dword ptr [ebx+60h]
fc4fd3ea ff5018          call    dword ptr [eax+18h]
fc4fd3ed 5f              pop     edi
fc4fd3ee 5e              pop     esi
fc4fd3ef 5b              pop     ebx
fc4fd3f0 5d              pop     ebp
fc4fd3f1 c20800          ret     8
fc4fd3f4 643a5c7870      cmp     bl,byte ptr fs:[eax+edi*2+70h]
fc4fd3f9 7370            jae     CLASSPNP!FreeDictionaryEntry+0x46
fc4fd3fb 5c              pop     esp
fc4fd3fc 647269          jb      CLASSPNP!FreeDictionaryEntry+0x43
fc4fd3ff 7665            jbe     CLASSPNP!FreeDictionaryEntry+0x41
fc4fd401 7273            jb      CLASSPNP!FreeDictionaryEntry+0x51
fc4fd403 5c              pop     esp
```

Developer’s view to the debugging

- Run gdb server in the guest
- Run gdb client on the host
- Attach to guest process
- Load symbols
- Debug the program

- Run gdb client
- Load kernel symbols
- Connect to guest/emulator gdb server
- Debug the kernel

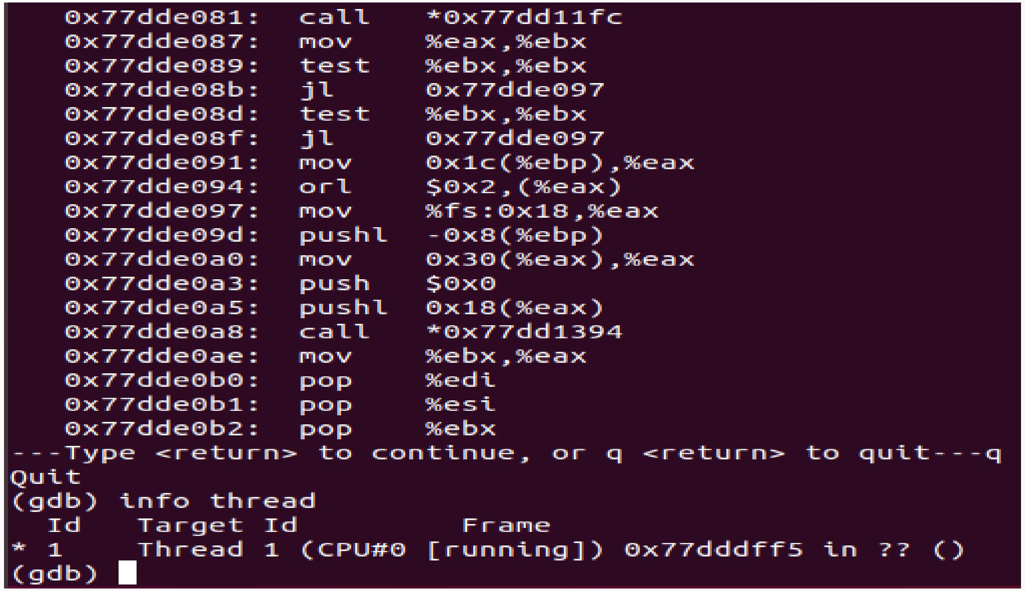
Reverser's view to the debugging

- ~~Run gdb server in the guest~~
- Run gdb client on the host
- Attach to guest process
- Load symbols
- Debug the program

- Run gdb client
- ~~Load kernel symbols~~
- Connect to guest/emulator gdb server
- Debug the kernel

Full system debugging with GDB

- Need to figure out the address for loading symbols from the binaries
- Not usable for Windows
- Can’t distinguish the processes even when having the symbols



```
   0x77dde081:  call    *0x77dd11fc
   0x77dde087:  mov     %eax,%ebx
   0x77dde089:  test    %ebx,%ebx
   0x77dde08b:  jl      0x77dde097
   0x77dde08d:  test    %ebx,%ebx
   0x77dde08f:  jl      0x77dde097
   0x77dde091:  mov     0x1c(%ebp),%eax
   0x77dde094:  orl     $0x2,(%eax)
   0x77dde097:  mov     %fs:0x18,%eax
   0x77dde09d:  pushl   -0x8(%ebp)
   0x77dde0a0:  mov     0x30(%eax),%eax
   0x77dde0a3:  push    $0x0
   0x77dde0a5:  pushl   0x18(%eax)
   0x77dde0a8:  call    *0x77dd1394
   0x77dde0ae:  mov     %ebx,%eax
   0x77dde0b0:  pop     %edi
   0x77dde0b1:  pop     %esi
   0x77dde0b2:  pop     %ebx
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) info thread
  Id   Target Id         Frame
* 1    Thread 1 (CPU#0 [running]) 0x77dddff5 in ?? ()
(gdb) 
```

Jedi debugging

- Use the Force to figure out CR3
- break *0xdeadf00d if $cr3=0x1ee7

Debugging problems

- VM Introspection to extract OS-level information
  - Processes and threads
  - Call stack
  - Address spaces and page tables
  - Executed images and symbol/debug information
- Client which capable of full-system debugging
  - Process and thread support
  - Support for switching the address spaces

Introspection: guest agents

- Have full control to the guest data structures and API
- Require SDK inside the image
  - or debug mode for WinDbg
  - or running gdbserver
- Side effects
  - behavior change
  - can be detected by malware
  - can’t be recorded/replayed

Introspection: memory analysis

- Rekall/Volatility
- Parse memory dumps
- Include many OS profiles
- Hardly applicable for custom kernels and esoteric OSes
- Too slow for runtime monitoring

Introspection: event hooking

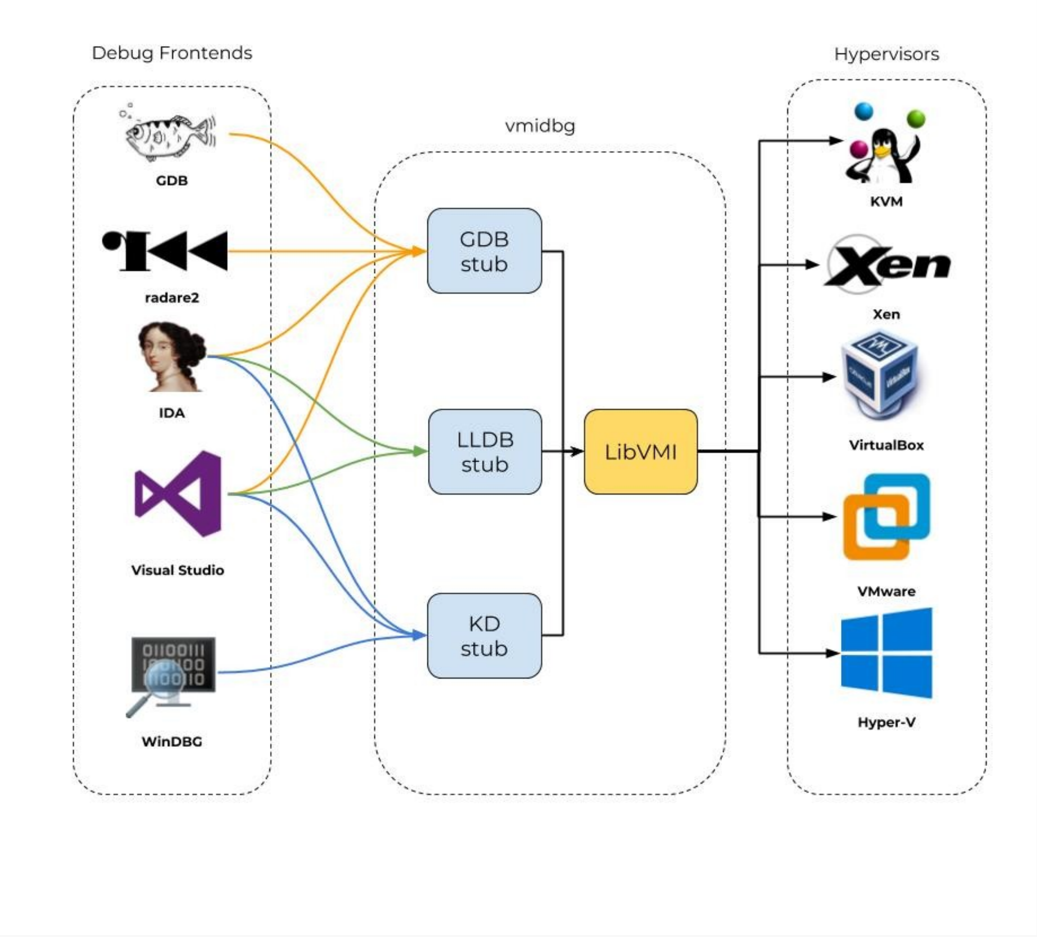
- Volatility-like profiles and event monitoring (PANDA)
  - Needs configuring for every kernel
  - Requires SDK for the guest
- Profile-less and agent-less event monitoring (SWAT)
  - Single config for all Linux kernels 2.x-4.x
  - Lacks some details of the kernel internals

pyvmidbg

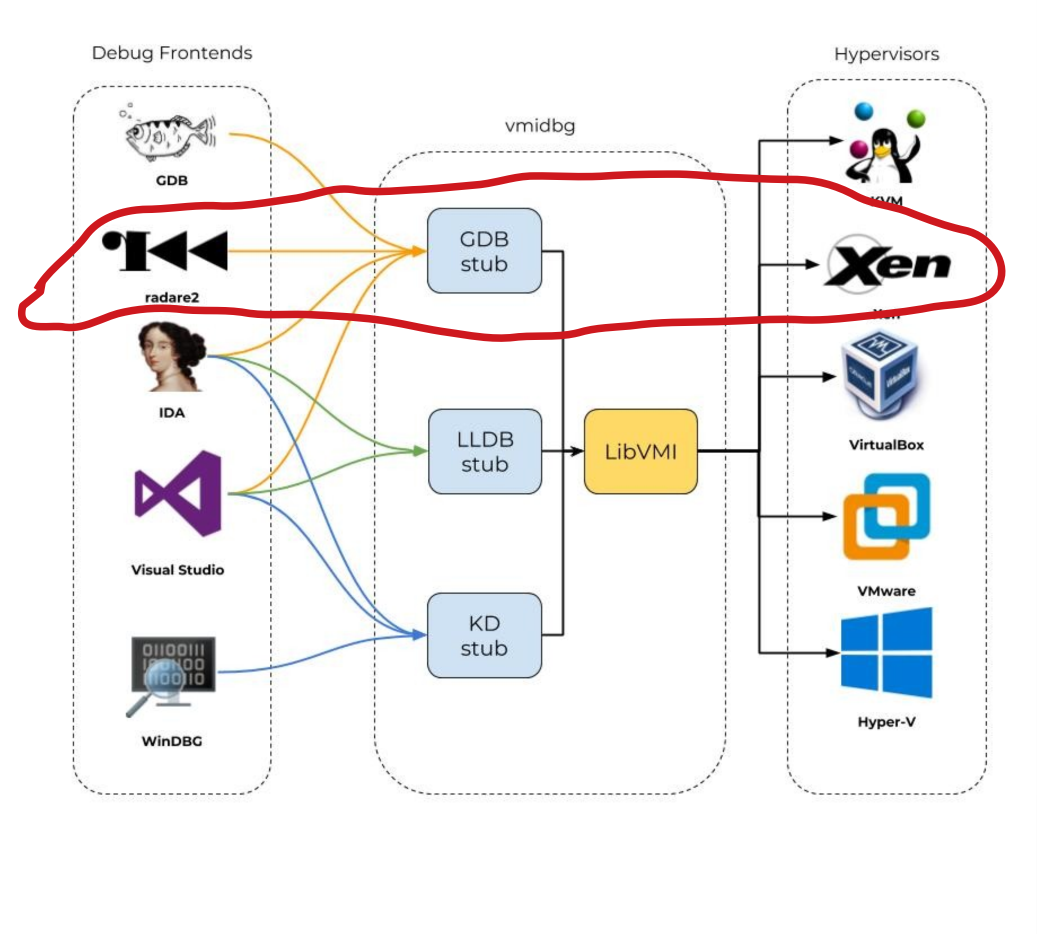
- OS-agnostic debug interface
- Uses Rekall for introspection
- Intended to support
  - Linux and Windows
  - all debuggers

- https://github.com/Wenzel/pyvmidbg

pyvmidbg


Debug Frontends
GDB
radare2
IDA
Visual Studio
WinDBG
vmidbg
GDB stub
LLDB stub
KD stub
LibVMI
Hypervisors
KVM
Xen
VirtualBox
VMware
Hyper-V

pyvmidbg


Debug Frontends
vmidbg
Hypervisors
GDB
radare2
IDA
Visual Studio
WinDBG
GDB stub
LLDB stub
KD stub
LibVMI
KVM
Xen
VirtualBox
VMware
Hyper-V

LibVMI

- Extracts CPU and memory state from running VM
- Supports runtime events
  - Memory access, privileged registers access, debug events, …
- Suitable for GDB and WinDbg stubs
- Doesn’t support QEMU yet

- https://github.com/libvmi/libvmi

Instrumenting the code

- Debugger can’t parse call stack when frame pointer is omitted
- Break on specific opcode
  - syscall – ok for libvmi (exception)
  - call/ret – not ok for libvmi
- Break on register access
  - CR3 – ok for libvmi (privileged)
  - ESP – not ok for libvmi

- Impossible for HW hypervisors
- Possible with QEMU, but not implemented yet

More debugging problems

- Too dumb breakpoints
- Can’t inspect hardware state except the CPU registers

Breakpoints: emulator-side conditions

- Set breakpoint
- Run
- Stop at breakpoint
- Check condition
- Run
- Stop at breakpoint
- Check condition
- Run
- Stop at breakpoint
- Check condition
- Stop execution

- Set breakpoint
- Run
- Check condition
- Run
- Check condition
- Run
- Check condition
- Stop execution

More breakpoints

- I/O breakpoints
- Memory area (e.g. whole array) watchpoints
- Breakpoints at specific process
- Breakpoints at interrupts and exceptions

- Need to extend QEMU and the debugger

Device introspection and debugging

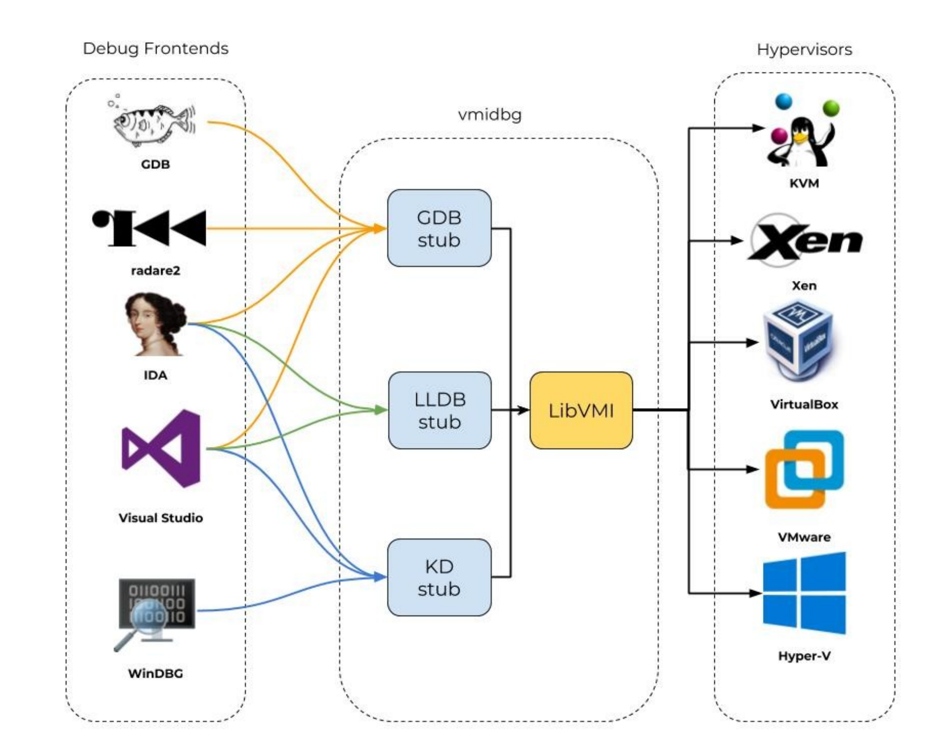
- Hardware-software codesign
- Driver debugging
- Emulator debugging

- Not very handy approaches
  - Debug logs in QEMU
  - Running two debuggers

Conclusion

- Only WinDbg supports system-wide view
- LibVMI is not enough for extracting all the details
- Need synchronized QEMU-GDB efforts to extend the protocol

- Solutions
  - use only Windows as a guest
  - create new debugger (maybe based on the existing one)



NewDbg

LibVMII

QEMU+

system-wide debugging

introspection and instrumentation

and maybe others