Virtdbg

Using virtualization features for debugging the Windows 7 kernel

Damien Aumaitre

Recon 2011

Roadmap

How it began

Study of Windows 7 x64 internals

- PatchGuard
- Code integrity
- DRM

But how?

- Static analysis: IDA, metasm
- Dynamic analysis: need a kernel debugger, Which one?

Dynamic analysis: kernel debugger

Plenty of kernel debuggers

- Local: SoftICE, Syser, HyperDbg...
- Remote: WinDbg, Gdb, ...

Which one to choose?

- Need x64 support $\Leftarrow$ gdb and windbg handle x64
- Need to work without /DEBUG $\Leftarrow$ gdb inside virtual machine?

Dynamic analysis: virtualisation

Lots of candidates

- XEN, qemu, vmware, ...
- But kernel debugging not easy with gdb stub (too process-centric), need to develop extensions to gdb protocol
- Big softwares, hard to tinker with
- Emulated devices may interfere with Windows 7 internals (not sure if DRM stack will load under vmware)

Need a debugger!

- Which handles x64
- And work without /DEBUG boot flag
- Using real hardware

What kind of debugger?

Local debuggers

Pros

- Need only one box
- Easy to setup

Cons

- Hard to extend
- Lack basic features (copy/paste ?)
- Hard dependencies on hardware like framebuffer, keyboard for the GUI

Remote debuggers

Pros

- Extendable easily with plugins
- Lot of code is deported, easier to develop
- Copy/paste :)

Cons

- Requires two boxes
- Need a remote stub for handling communications

How do they work?

They catch processor interrupts (often by hooking the IDT).

Example: IA-32

- IDT (Interruption Descriptor Table) stores interruption vectors
- INT1 is used by hardware breakpoints (DR registers) and singlestep
- INT3 is used by software breakpoints
- INT14 (page fault) is used for catching memory breakpoints

How do they work?

Two modes of operations

- Debug mode: debugger waits for user interactions, other processors are halted
- Normal mode: the operating system runs as usual

Interactions

- We switch to debug mode when receiving a breakin request:
  - To inspect processor registers
  - To inspect memory (virtual, physical, io)
  - To set breakpoints
- Switch to normal mode with a continue request

Debugging Windows 7 x64

Several problems arise

- How to execute kernel code when signed drivers are mandatory?
- How to gain control of the interruption vectors when PatchGuard deny modifications of the IDT?
- Patching the kernel is not possible with PatchGuard (bye bye software breakpoints)
- How to communicate with the debugger stub when the entire OS is frozen?

Bypassing PatchGuard

PatchGuard

PatchGuard protects the kernel from malicious activities:

- It prevents kernel code modification
- Also prevents the modification of major structures (IDT, SSDT, ...)
- Causes a blue screen if a modification is detected

Bypassing PatchGuard control of IDT

- By using hardware assisted virtualization
- We can control very precisely the execution of the target system with a hypervisor (VMM)
- We take control whenever an interruption occurs
- Bonus: very stealth by design (cf. BluePill)

Communication with the hypervisor

Communication between debugger/debuggee

- Usually kernel debuggers use a serial interface to communicate; it works but it's really slow :(
- WinDbg uses FireWire for highspeed debugging, it's convenient and easy to setup

Using FireWire?

- Speed is good (around 10Mb/s)
- Widely deployed
- But I don't know FireWire...
- Except for doing DMA attacks :-)

DMA

Theory

- Historically, all I/O came through the CPU. It's slow.
- DMA instead goes through a fast memory controller
- Implemented as part of the PCI specification
- Any device on the PCI / PCI Express bus can issue a read/write DMA

A flawed idea?

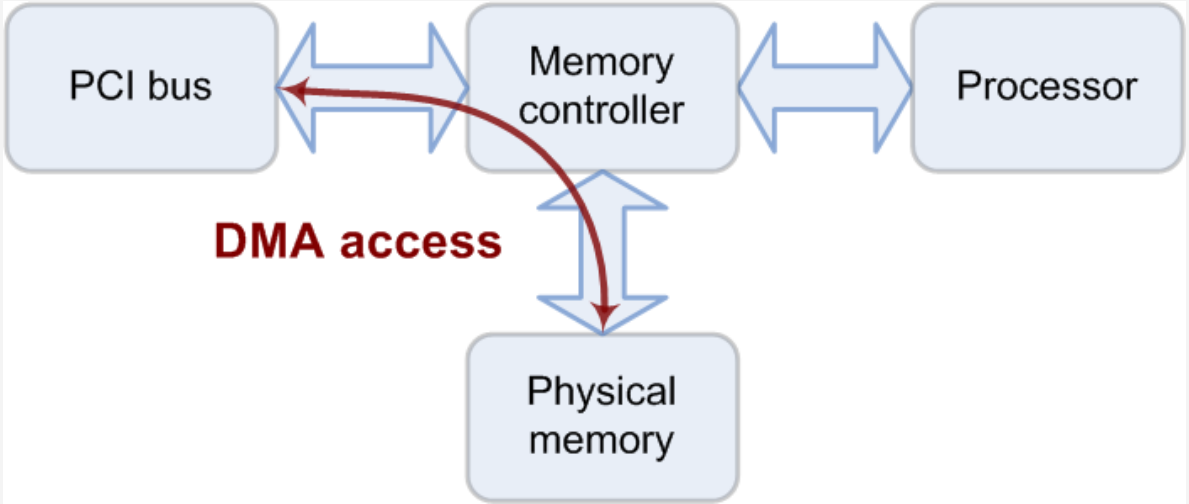
- The CPU and thus OS are entirely bypassed, cannot prevent malicious DMA requests

DMA

Consequences

- Any device may read/write the physical memory
- Operating system's code and internal data can be modified
- Security mechanisms rendered useless

Over-simplified example of DMA access



Communication with the hypervisor

Using DMA over FireWire?

- No hardware specific code (only read/write in shared buffers)
- `libforensic1394` library allows easy development

```
bus = Bus.new()
bus.enable_sbp2
puts "waiting for dma access..."
sleep 5
devices = bus.devices
dev = devices[0]
dev.open
mem = FireWireMem.new(dev)
puts mem.hexdump(0x8000, 0x100)
```

- Bonus: use DMA attacks to load unsigned kernel drivers

Bypassing driver integrity verification

Signed drivers

- Since Vista, drivers need to be signed on x64 editions of Windows
- This can be disabled but there are side effects (booting in test signing mode)

Solution: Loading a driver using FireWire

- Do not use the kernel code for loading drivers
- We implement our own loader thus skipping the verification process

Roadmap

1. Preamble
2. Loader
   - Reconstructing the virtual memory
   - Rerouting execution flow
   - Loader
3. Debugger

Loading the driver code

"Live" loader

- Parses physical memory and reconstructs virtual/physical layer
- Injects driver code in two stages (more on that later)
- Loads driver code without using kernel API but custom routines (hence bypassing signatures)
  1. Copy binary sections, resolve imports and apply relocations
  2. Redirect execution to driver entrypoint

Injecting a driver with DMA requests

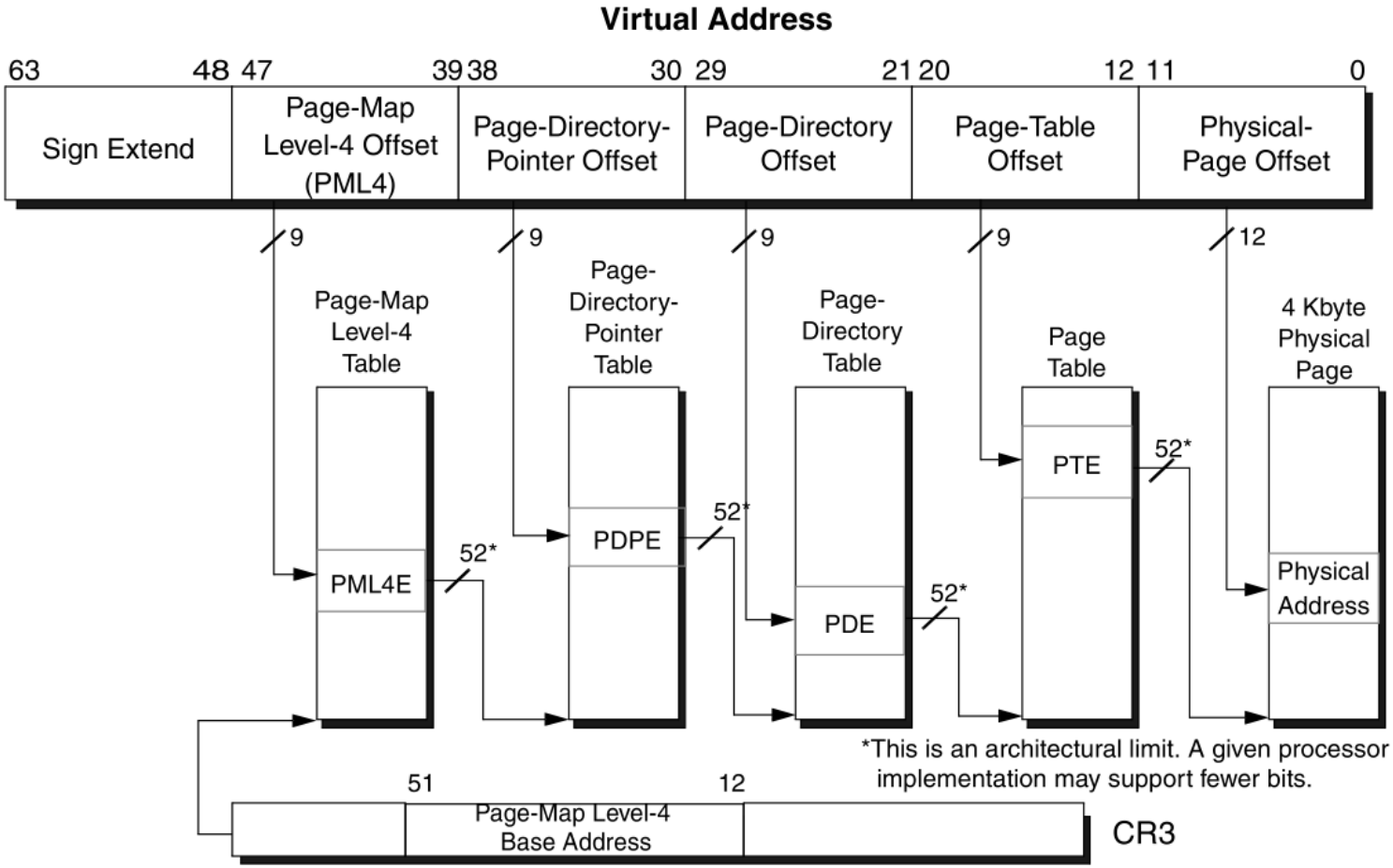
What do we need?

- Reconstructing the virtual memory mapping
- Space for storing our payload
- Finding a function pointer

Difficulties

- Can't use kernel routines for loading the image
- Need to not trigger PatchGuard

x64 Virtual address translation



Source: AMD64 Architecture Programmer´s Manual Volume 2 (System Programming)

Finding cr3

Classic method

- Searching for the beginning of an EPROCESS structure
- Use backup copy of cr3 in DirectoryTableBase field
- Take forever... (around several minutes)
- Can be at the end of the physical memory

```
struct _KPROCESS, 37 elements, 0x160 bytes
    +0x000 Header           : struct _DISPATCHER_HEADER, 29 elements, 0x18 bytes
    +0x018 ProfileListHead  : struct _LIST_ENTRY, 2 elements, 0x10 bytes
    +0x028 DirectoryTableBase : Uint8B
    ...
```

Finding cr3

Quicker method

- Searching for the kernel beginning
- Getting kernel symbols addresses using debug information (PDB)
- Get offset of the first KPCR structure (KiInitialPCR)
- We find the values of all control registers included cr3

```
struct _KPCR, 27 elements, 0x4e80 bytes
    +0x180 Prcb                    : struct _KPRCB, 242 elements, 0x4d00 bytes
       +0x040 ProcessorState          : struct _KPROCESSOR_STATE, 2 elements, 0x5b0 bytes
          +0x000 SpecialRegisters        : struct _KSPECIAL_REGISTERS, 27 elements, 0xd8 bytes
             +0x000 Cr0                     : Uint8B
             +0x008 Cr2                     : Uint8B
             +0x010 Cr3                     : Uint8B
             ...
```

What's next?

- We can interpret any address in virtual memory
- In order to execute arbitrary kernel code we need a pointer to overwrite

Which pointer?

- Can't touch IDT or SSDT or kernel code due to PatchGuard
- Need something stealthier, often called and not checked by PatchGuard!

Solution

Using function pointers located in `OBJECT_TYPE_INITIALIZER` structure [a]

[a] Must read: Skape and Skywing, "A catalog of Windows Local Kernel-mode Backdoor Techniques", 2007, Uninformed Vol. 8

OBJECT_TYPE_INITIALIZER

- Each object is categorized by an object type represented by a OBJECT_TYPE structure

```
struct _OBJECT_TYPE, 12 elements, 0xd0 bytes
   +0x000 TypeList         : struct _LIST_ENTRY, 2 elements, 0x10 bytes
   +0x010 Name             : struct _UNICODE_STRING, 3 elements, 0x10 bytes
   +0x020 DefaultObject    : Ptr64 to Void
   +0x028 Index            : UChar
   ...
   +0x040 TypeInfo         : struct _OBJECT_TYPE_INITIALIZER, 25 elements, 0x70 bytes
   ...
```

- OBJECT_TYPE structure contains a nested structure named OBJECT_TYPE_INITIALIZER

OBJECT_TYPE_INITIALIZER

- Several fields of the OBJECT_TYPE_INITIALIZER structure are functions pointers

```
struct _OBJECT_TYPE_INITIALIZER, 25 elements, 0x70 bytes
    ...
   +0x030 DumpProcedure    : Ptr64 to     void
   +0x038 OpenProcedure    : Ptr64 to     long
   +0x040 CloseProcedure   : Ptr64 to     void
   +0x048 DeleteProcedure  : Ptr64 to     void
   +0x050 ParseProcedure   : Ptr64 to     long
   +0x058 SecurityProcedure : Ptr64 to     long
   +0x060 QueryNameProcedure : Ptr64 to     long
   +0x068 OkayToCloseProcedure : Ptr64 to     unsigned char
```

- For example, OpenProcedure will point to nt!PspOpenProcess for a Process

Finding OBJECT_TYPE structures

Finding non-exported kernel symbols

- Several non-exported kernel symbols are present in the KDDEBUGGER_DATA64 structure [a] [b]
- defined in the Debugging Tools For Windows SDK header file wdbgexts.h
- This structure is pointed by the KdDebuggerDataBlock kernel global variable
- ```
  typedef struct _KDDEBUGGER_DATA64 {
      DBGKD_DEBUG_DATA_HEADER64 Header;
      ULONG64   KernBase;
      ...
      ULONG64   PsLoadedModuleList;
      ...
      ULONG64   ObpTypeObjectType;
      ...
  ```

[a] http://uninformed.org/index.cgi?v=4&a=2&p=5

[b] http://web.archive.org/web/20061110120809/http://www.rootkit.com/newsread.php?newsid=153
  ```

Loader

- The hypervisor code is implemented as a WDK driver
- We inject the hypervisor with several steps:
  1. We inject a stager responsible for allocating the memory used for storing the code of the hypervisor (for example, first memory page of a already loaded driver or at the end of the `KUSER_SHARED_DATA` structure)
  2. Once the memory is allocated, we map the driver section by section
  3. Then we resolve imports and apply relocations

Signed drivers

Effectively bypassing signed driver mechanism

What's next?

- We can load an arbitrary driver in the target OS
- What kind of driver?
  - Hypervisor
  - Virtualize the operating system without rebooting (ala BluePill)
  - Requires Intel VMX hardware virtualization features (no AMD support)

Roadmap

1. Preamble
2. Loader
3. Debugger
   - Hypervisor
   - Using an hypervisor for debugging
   - Communication
   - Implementation

Specifications

Constraints

Minimal impact on the system (don't use OS functions (if possible) and no modifications of kernel structures

Features

- Stealth by design
- Heavy treatment deported on the client
- Basic primitives (for the moment):
  - Read/write memory
  - Read/write registers
  - Singlestepping execution
  - Stopping and resuming execution

VMX (Virtual Machines eXtensions)

Two operating modes

VMX root hypervisor mode

VMX non-root guest mode

Remarks

- By design we can't tell which mode is active
- Switching between these modes is called VM-Entry (VMM $\Rightarrow$ Guest) and VM-Exit (Guest $\Rightarrow$ VMM)
- Processor behavior is modified in VMX non-root mode: some events cause transitions (access to control or debug registers, exceptions etc.)

VMX (Virtual Machines eXtensions)

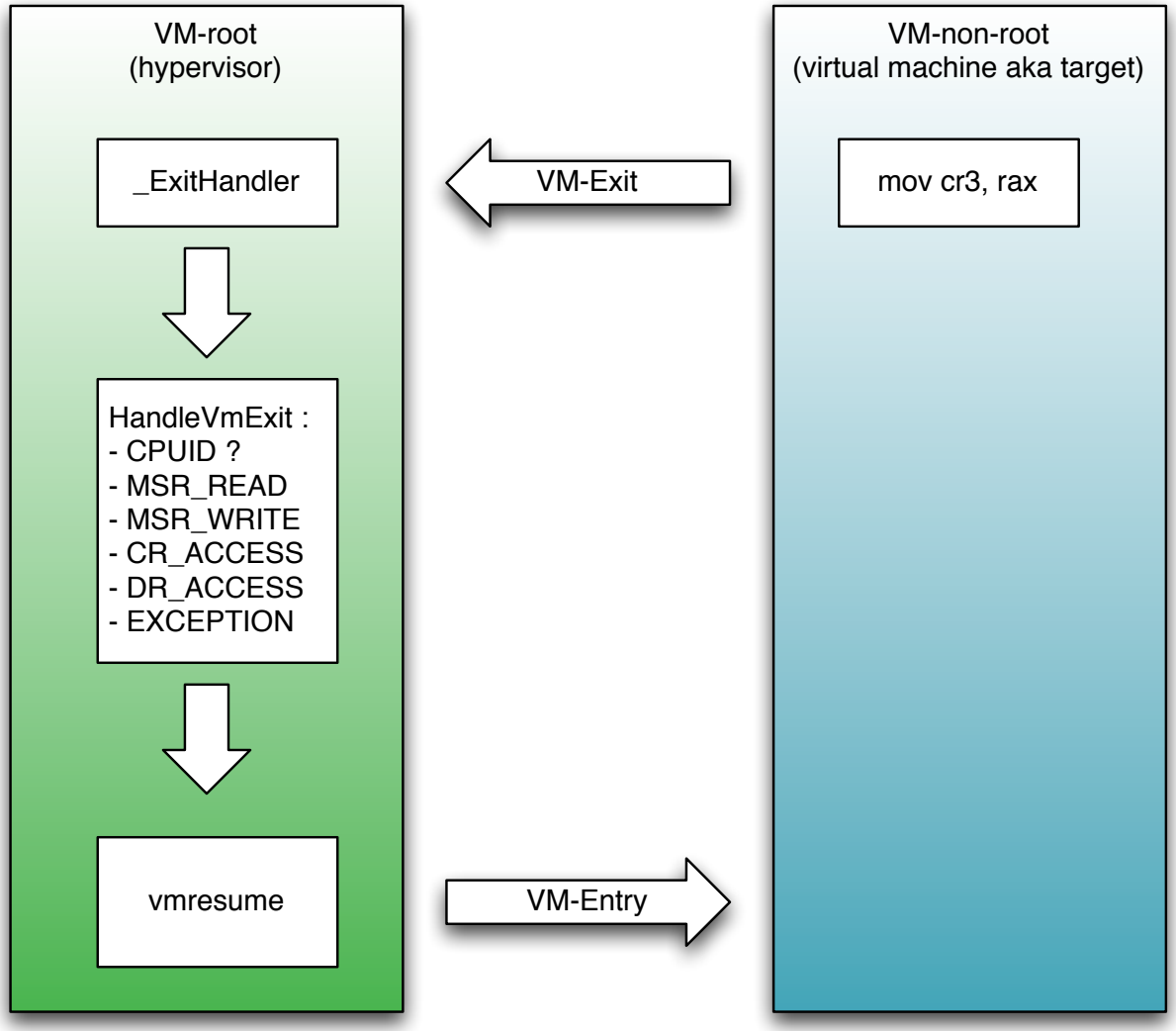
Virtualization setup

- VMX mode is activated by the `VMXON` instruction
- Virtual machines are launched with the `VMLAUNCH` instruction
- Each VM-Exit calls the hypervisor handler
- The hypervisor resume guest execution with the `VMRESUME` instruction
- VMX mode is deactivated by the `VMXOFF` instruction

VMCS (Virtual Machine Control Structure)

- Controls transitions between VMX root and VMX non-root
- Manipulated by new instructions: (`VMPTRST`, `VMPTRLD`, `VMREAD`, `VMWRITE` and `VMCLEAR`)

Transitions



Using an hypervisor for debugging

Debugger primitives

- Break or continue target execution
- Inspect registers
- Inspect memory
- Singlestep through code

Using an hypervisor for debugging

Break/Continue

VM-Exit occurs when context is changed (`mov cr3, rax` for example) hence providing us with a regular callback for knowing if the user asked for system interactions

- If we need to switch to debug mode:
  - Hypervisor stops the other processors
  - And enters a loop waiting for orders
- If we don't: just resume target execution

Inspecting registers

- A part of the context is saved in the VMCS
- We save the rest before entering in VMX root mode
- We restore the context before executing the `VMRESUME` instruction

Using an hypervisor for debugging

Inspecting memory

- Must be careful, no page fault can occur in the VMX-root handler!
- We check the PTE (Page Table Entry) in order to validate the address provided by the client
- Manipulating physical memory (no copy-on-write :-s)

Single Step

- Easy because we control the `RFLAGS` register and can intercept INT1

Communication

Shared physical memory

- Use `libforensic1394` [a] for handling DMA setup
- Contains the necessary information for synchronising and exchanging data between the hypervisor and the client
- Two areas are for exchanging requests ("uplink" and "downlink")

[a] https://freddie.witherden.org/tools/libforensic1394/

Virtdbg

What is it?

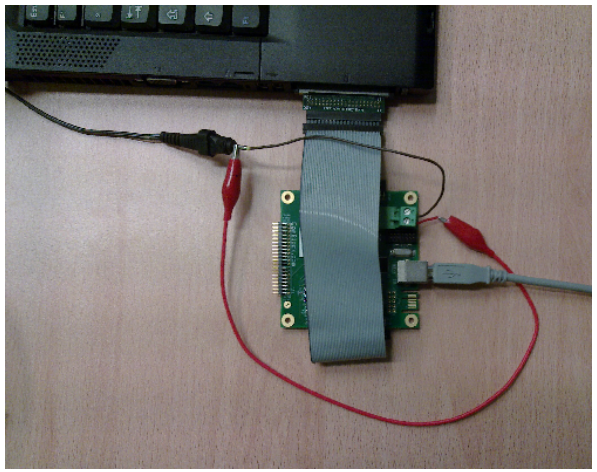
- Hypervisor (like BluePill) to control the target
- Uses FireWire DMA requests for loading the VMM and giving orders to the debugger
- Client GUI leveraging the Metasm framework
- Designed to debug Windows 7 x64 "on the fly" (i.e. without booting with /DEBUG)

Uses

- Analyze hardware specific software like DRM
- Malware analysis (TLD4)
- Debugging Windows internals (PatchGuard, code integrity)

First implementation (march 2010)

- 2 FPGA: one for handling DMA requests (Cardbus), the other for handling client requests (USB)
- Custom DMA engine in VHDL
- Driver for USB FPGA works only on Windows XP :(



Second implementation

- Drops FPGA and uses only a FireWire cable for communication
- Leverages the metasm framework

DEMO

Conclusion

Why this project?

- No real kernel debuggers for Windows 7 x64 (except WinDbg)
- They are very intrusive and need OS cooperation

Future work

- Porting the hypervisor to support 32-bit Windows
- Support of AMD processors (Phenom)
- Add more primitives and events (leveraging all possibilities of hardware virtualization features)
- Increase furtivity and anti-debug resilience
- Reduce operating system dependancy
- Support for VT-d Intel family of processors (currently only VT-x)
- Check out `ramooflax`[a]: preboot hypervisor

[a] https://github.com/sduverger/ramooflax

Thanks and credits

- Joanna Rutkowska for releasing BluePill (great source of information on hypervisors)
- Freddie Witherden [1], author of libforensic1394 (much better than previous bindings)
- Yoann Guillot for the Metasm framework [2]
- My colleagues at SOGETI/ESEC Lab (especially Christophe Devine)

[1] https://freddie.witherden.org/tools/libforensic1394/
[2] http://metasm.cr0.org

Q&A

- Thank you for your attention!
- Give it a try! http://code.google.com/p/virtdbg
- Questions?
- Contact: damien (at) security-labs.org