Pitfalls and Limits of Dynamic Malware Analysis

Dr. Tamas K Lengyel
tamas@tklengyel.com

# whoami

Senior Security Researcher @ Intel

Chief Research Officer @ Honeynet.org

Maintainer of

- Xen
- LibVMI
- DRAKVUF

Disclaimer

I don’t speak for my employer. All opinions and information here are my personal views.

Agenda

1. Why Dynamic Analysis
2. Assorted Problems and Pitfalls
3. Food for Thought

The Problem

New malware binaries are easy to create

- Same malware, different binary = can’t be fingerprinted
- Unpacked code itself is heavily obfuscated
- Writing “unpackers” is cumbersome and very fragile
- Polymorphic & metamorphic malware
- Not all packed binaries are malware

For more details read “SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers”

Dynamic Analysis v1

Observation: malware *must run*

- Code that doesn’t execute is not malicious
- Let’s execute the packed binary
- Dump its unpacked version from memory
- Fingerprint
- ...
- Profit!

Dynamic Analysis v2

Observation: malware *must run*

- Code that doesn’t execute is not malicious
- Let’s execute the packed binary
- Observe it’s execution
- Fingerprint it’s <u>*behavior*</u>
- *...*
- Profit!

Desired abilities

Isolation

Stealth

Interposition

Fidelity

Interpretation

Scalability

How to “observe”?

Debuggers

- strace/gdb/windbg/valgrind/etc..
- Not designed to be stealthy
- NO security boundary
- Very easy to detect
- Observer effect

How to “observe”?

Emulators

- QEMU/TEMU/PANDA/Binee/Qiling Framework
- Very high (instruction level) visibility
- NO security boundary
- Easy to detect
- Observer effect

A Better way to Observe?

Use kernel-mode code to observe

- Hardware protected execution mode
- Most modern kernels require signed kernel modules
- You can disable enforcement but that’s detectable
- The OS will limit what you can observe
- Windows doesn’t like it if you just randomly hook stuff
- Tedious to clean-up after each analysis

A Better way to Observe?

Windows Sandbox

- Built-in feature since Windows 10 18305
- Easy to deploy
- Light-weight copy-on-write clone of the main OS
- Strong hypervisor-based isolation (as long as you don’t enable GPU passthrough and writable shared folders)
- No API yet to monitor through the kernel/hypervisor

https://techcommunity.microsoft.com/t5/Windows-Kernel-Internals/Windows-Sandbox/ba-p/301849

Hypervisor Introspection

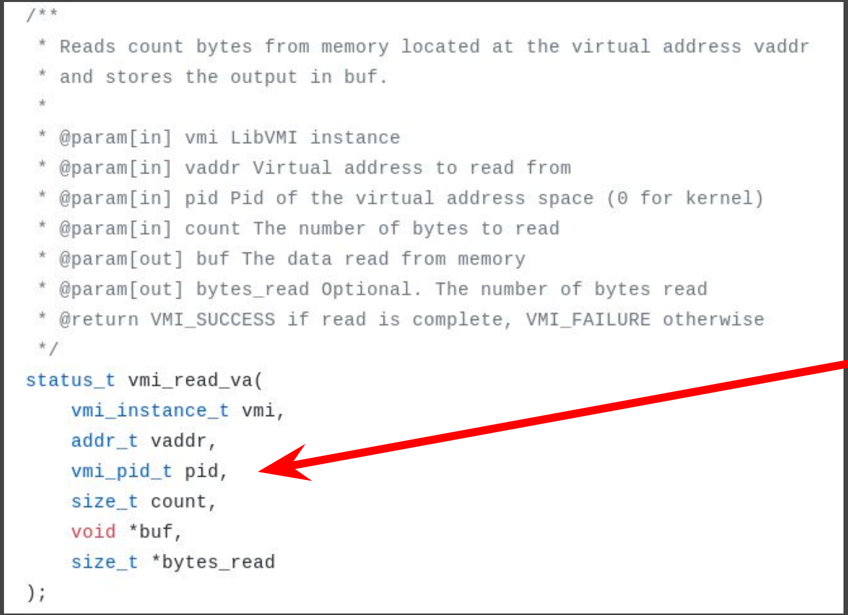
- Easy to detect virtualization
- Hard to detect introspection
- Strong isolation
- Good visibility
- Easy to cleanup after analysis
- Hard to develop
- Harder to develop it correctly

Pitfalls of introspection

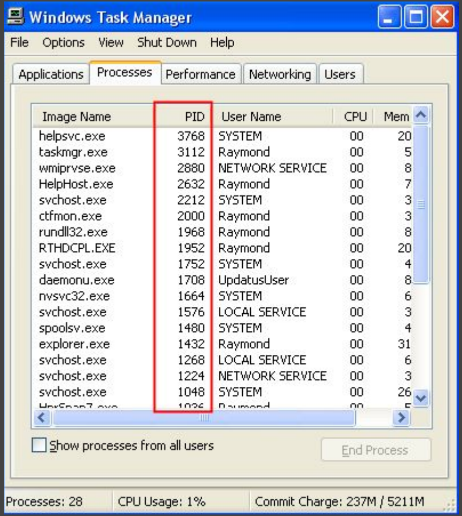
What part of the VM do you trust?

- In-guest kernel data-structures can be unreliable
- Race-condition on multi-vCPU guests
- Stale data
- Tricky and unusual code-sequences

LibVMI pitfall: PID



```
/**
 * Reads count bytes from memory located at the virtual address vaddr
 * and stores the output in buf.
 *
 * @param[in] vmi LibVMI instance
 * @param[in] vaddr Virtual address to read from
 * @param[in] pid Pid of the virtual address space (0 for kernel)
 * @param[in] count The number of bytes to read
 * @param[out] buf The data read from memory
 * @param[out] bytes_read Optional. The number of bytes read
 * @return VMI_SUCCESS if read is complete, VMI_FAILURE otherwise
 */
status_t vmi_read_va(
    vmi_instance_t vmi,
    addr_t vaddr,
    vmi_pid_t pid,
    size_t count,
    void *buf,
    size_t *bytes_read
);
```



Windows Task Manager
File Options View Shut Down Help
Applications Processes Performance Networking Users
Show processes from all users
End Process

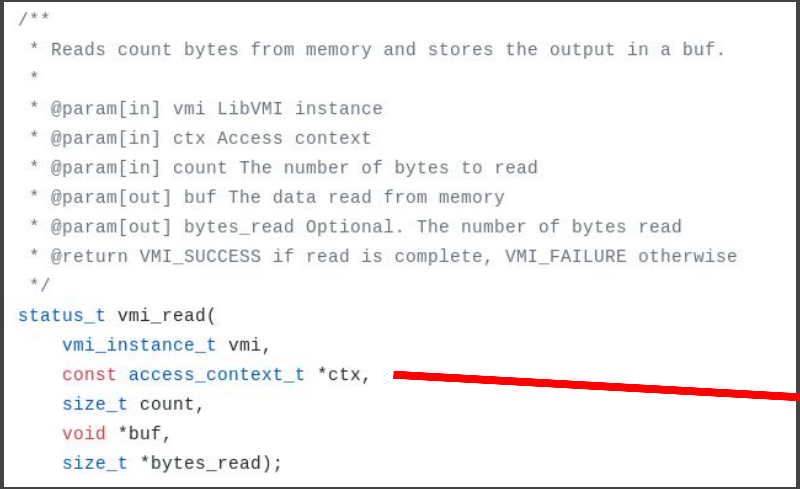
Image Name	PID	User Name	CPU	Mem
helpsvc.exe	3768	SYSTEM	00	20
taskmgr.exe	3112	Raymond	00	5
wmiprvse.exe	2880	NETWORK SERVICE	00	8
HelpHost.exe	2632	Raymond	00	7
svchost.exe	2212	SYSTEM	00	3
ctfmon.exe	2000	Raymond	00	3
rundll32.exe	1968	Raymond	00	8
RTHDCPL.EXE	1952	Raymond	00	20
svchost.exe	1752	SYSTEM	00	4
daemonu.exe	1708	UpdatusUser	00	8
nvsvc32.exe	1664	SYSTEM	00	6
svchost.exe	1576	LOCAL SERVICE	00	3
spoolsv.exe	1480	SYSTEM	00	4
explorer.exe	1432	Raymond	00	31
svchost.exe	1268	LOCAL SERVICE	00	6
svchost.exe	1224	NETWORK SERVICE	00	3
svchost.exe	1048	SYSTEM	00	26

Processes: 28 CPU Usage: 1% Commit Charge: 237M / 5211M

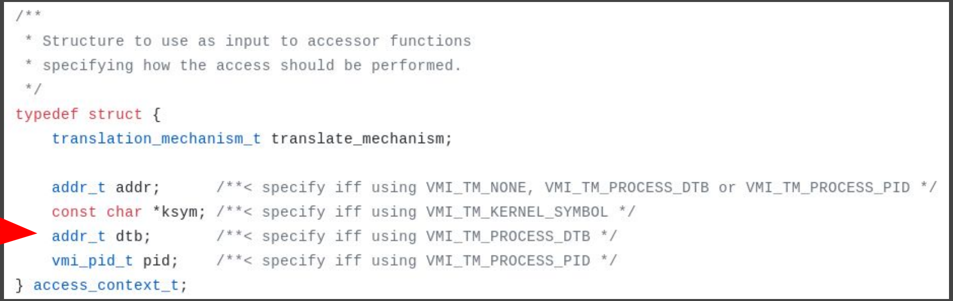
The PID is an OS construct!
Can be recycled
Associated PT can change
Can be missing (DKOM)!

LibVMI pitfall: PID

Don’t use the PID, use the pagetable address directly



```
/**
 * Reads count bytes from memory and stores the output in a buf.
 *
 * @param[in] vmi LibVMI instance
 * @param[in] ctx Access context
 * @param[in] count The number of bytes to read
 * @param[out] buf The data read from memory
 * @param[out] bytes_read Optional. The number of bytes read
 * @return VMI_SUCCESS if read is complete, VMI_FAILURE otherwise
 */
status_t vmi_read(
    vmi_instance_t vmi,
    const access_context_t *ctx,
    size_t count,
    void *buf,
    size_t *bytes_read);
```



```
/**
 * Structure to use as input to accessor functions
 * specifying how the access should be performed.
 */
typedef struct {
    translation_mechanism_t translate_mechanism;

    addr_t addr;      /**< specify iff using VMI_TM_NONE, VMI_TM_PROCESS_DTB or VMI_TM_PROCESS_PID */
    const char *ksym; /**< specify iff using VMI_TM_KERNEL_SYMBOL */
    addr_t dtb;       /**< specify iff using VMI_TM_PROCESS_DTB */
    vmi_pid_t pid;    /**< specify iff using VMI_TM_PROCESS_PID */
} access_context_t;
```

Introspection pitfall: pagetables

It’s under the control of the guest

- Guest may change it at any time
- Hardware caches translations in the TLB
- TLB invalidation is also the guest’s responsibility
- The pagetable may not be authoritative if TLB is split
- PCID and VPID keeps entries in the TLB longer
- No instruction to translate in context of VM (on x86)
- Can cause serious issues for both hw* and sw**

*See XSA-304/CVE-2018-12207 “x86: Machine Check Error on Page Size Change DoS”

**Read “DKSM: Subverting Virtual Machine Introspection for Fun and Profit”

LibVMI pitfall: software TLB

Virtual-to-physical translation is costly

- LibVMI has a software TLB
- Two level-hashtable lookup: dtb $\rightarrow$ vaddr
- *LibVMI doesn’t perform TLB flushes automatically*
- Cached translations can become stale
- Up to the user to implement flush policy

Introspection pitfalls: breakpoints

0xCC can be trapped to VMM

- Hypervisor-level debugging!
- External agent can patch-in the breakpoint
- Guest itself may be using them
- Can we make it stealthy?

Check out “Building a {Flexible} Hypervisor-Level Debugger” by Mathieu Tarral

Introspection pitfalls: breakpoints

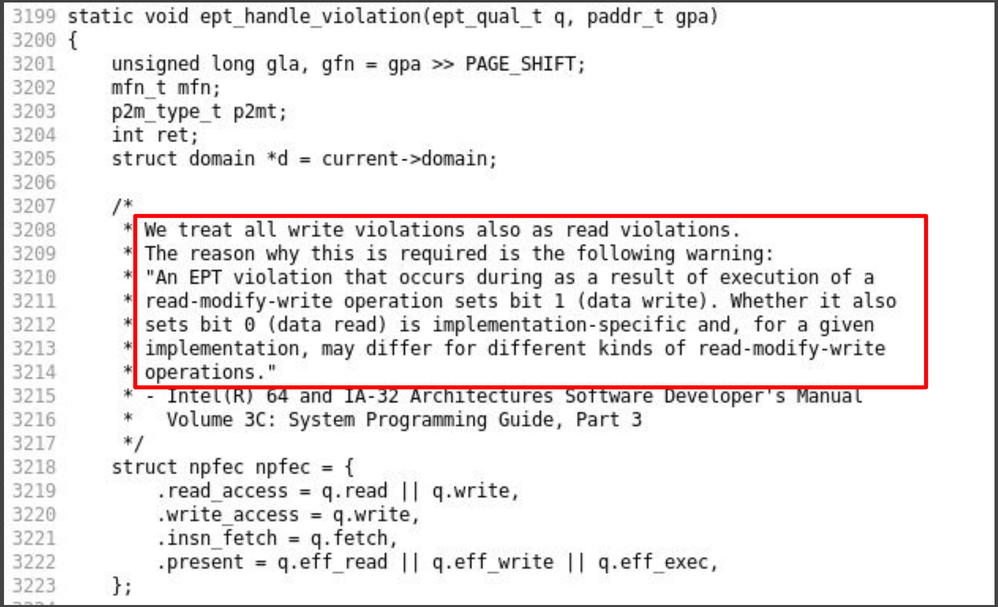
Hiding 0xCC

- Make page execute-only in EPT, lie about memory contents when something tries to read it
- Remove 0xCC → singlestep → replace 0xCC
- Racy on multi-vCPU systems
- Can emulate faulting instruction and lie about memory contents
- Shadow memory using EPT remapping!*

*More details: https://xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m

Introspection pitfalls: breakpoints

Read-modify-write instructions...

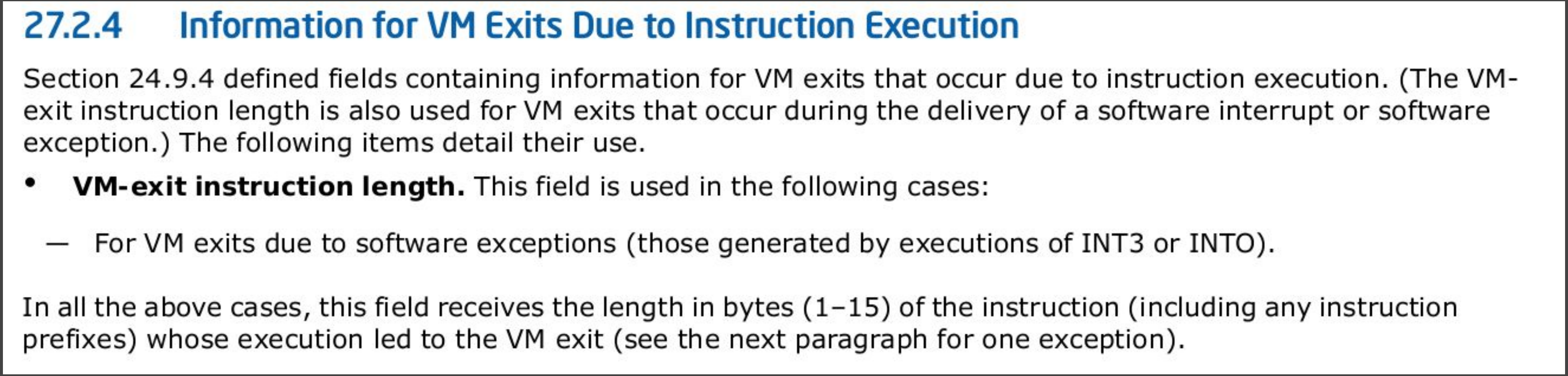


```
static void ept_handle_violation(ept_qual_t q, paddr_t gpa)
{
    unsigned long gla, gfn = gpa >> PAGE_SHIFT;
    mfn_t mfn;
    p2m_type_t p2mt;
    int ret;
    struct domain *d = current->domain;

    /*
     * We treat all write violations also as read violations.
     * The reason why this is required is the following warning:
     * "An EPT violation that occurs during as a result of execution of a
     * read-modify-write operation sets bit 1 (data write). Whether it also
     * sets bit 0 (data read) is implementation-specific and, for a given
     * implementation, may differ for different kinds of read-modify-write
     * operations."
     * - Intel(R) 64 and IA-32 Architectures Software Developer's Manual
     *   Volume 3C: System Programming Guide, Part 3
     */
    struct npfec npfec = {
        .read_access = q.read || q.write,
        .write_access = q.write,
        .insn_fetch = q.fetch,
        .present = q.eff_read || q.eff_write || q.eff_exec,
    };
```

https://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/arch/x86/hvm/vmx/vmx.c;h=535e0384fe170450fe82442a500fab3b088e31cf;hb=HEAD

Introspection pitfalls: breakpoints

What is the 0xCC instruction's length?



27.2.4 Information for VM Exits Due to Instruction Execution

Section 24.9.4 defined fields containing information for VM exits that occur due to instruction execution. (The VM-exit instruction length is also used for VM exits that occur during the delivery of a software interrupt or software exception.) The following items detail their use.

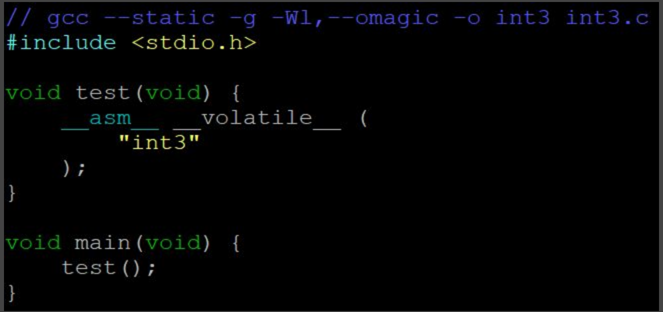
- **VM-exit instruction length.** This field is used in the following cases:
  - For VM exits due to software exceptions (those generated by executions of INT3 or INTO).

In all the above cases, this field receives the length in bytes (1–15) of the instruction (including any instruction prefixes) whose execution led to the VM exit (see the next paragraph for one exception).

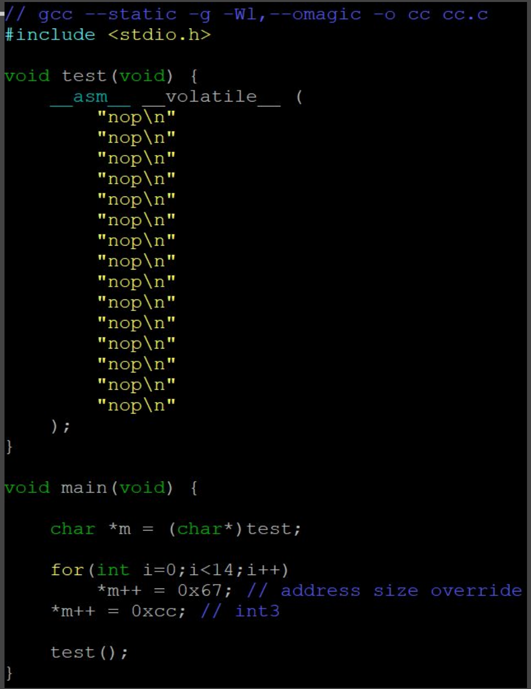
- … (including any instruction prefixes) ...

Introspection pitfalls: breakpoints

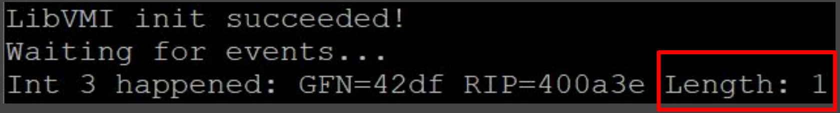
Redundant prefixes



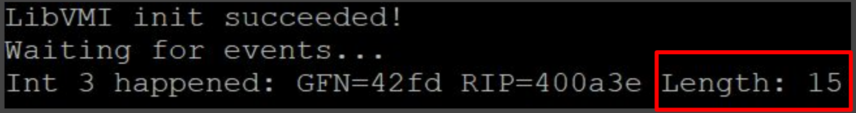
```c
// gcc --static -g -Wl,--omagic -o int3 int3.c
#include <stdio.h>

void test(void) {
    __asm__ __volatile__ (
        "int3"
    );
}

void main(void) {
    test();
}
```



```c
// gcc --static -g -Wl,--omagic -o cc cc.c
#include <stdio.h>

void test(void) {
    __asm__ __volatile__ (
        "nop\n"
        "nop\n"
        "nop\n"
        "nop\n"
        "nop\n"
        "nop\n"
        "nop\n"
        "nop\n"
        "nop\n"
        "nop\n"
        "nop\n"
        "nop\n"
        "nop\n"
        "nop\n"
        "nop\n"
    );
}

void main(void) {

    char *m = (char*)test;

    for(int i=0;i<14;i++)
        *m++ = 0x67; // address size override
    *m++ = 0xcc; // int3

    test();
}
```



```
LibVMI init succeeded!
Waiting for events...
Int 3 happened: GFN=42df RIP=400a3e Length: 1
```



```
LibVMI init succeeded!
Waiting for events...
Int 3 happened: GFN=42fd RIP=400a3e Length: 15
```

Introspection pitfalls: breakpoints

Length was assumed to be always 1

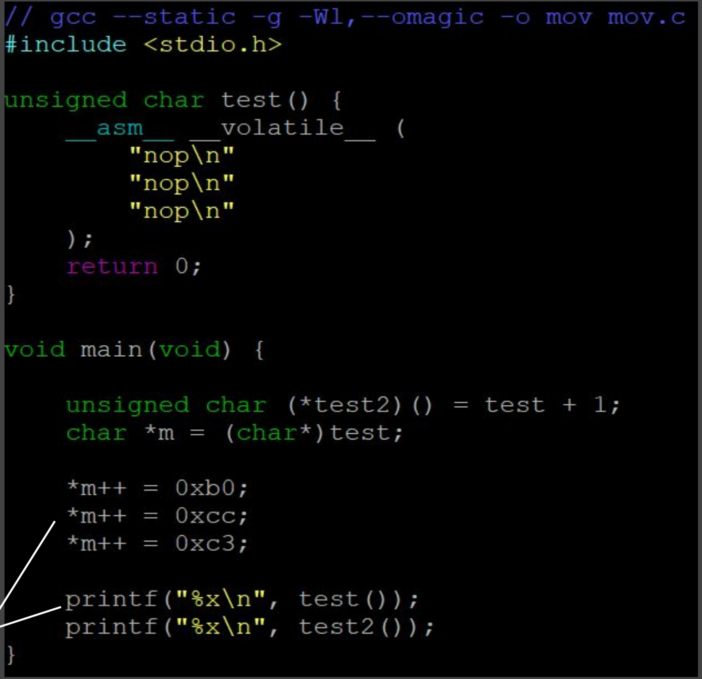
- The CPU reports the correct length to Xen
- Xen just didn’t send it to introspection apps
- Malicious code could use it to detect introspection
- Breakpoint would execute 15x within VM when incorrect length is used for reinjection
- Fixed starting in Xen 4.8
- Found with the awesome Xen Test Framework

Check out XTF at http://xenbits.xenproject.org/docs/xtf/

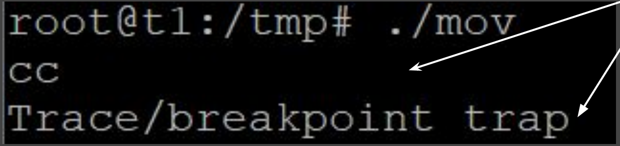
Introspection pitfalls: breakpoints

MOV instructions..

- Can read execute only memory!
- Copy "constant" to register
- Can leak stealth breakpoints!



```c
// gcc --static -g -Wl,--omagic -o mov mov.c
#include <stdio.h>

unsigned char test() {
    __asm__ __volatile__ (
        "nop\n"
        "nop\n"
        "nop\n"
    );
    return 0;
}

void main(void) {

    unsigned char (*test2)() = test + 1;
    char *m = (char*)test;

    *m++ = 0xb0;
    *m++ = 0xcc;
    *m++ = 0xc3;

    printf("%x\n", test());
    printf("%x\n", test2());
}
```



```
root@t1:/tmp# ./mov
cc
Trace/breakpoint trap
```

DRAKVUF pitfall: shadow pages

Hiding shadow pages with a sink page

- Shadow pages are at the very end of the VM physmap
- DRAKVUF read-protects and remaps them to sink page
- The sink page was just a page filled with 0’s
- That turned out to be a wrong guess
- Xen actually returns 1’s when accessing unmapped memory
- Trivia: What happens if you access high physical memory addresses that are not physically present?

Introspection pitfall: hardware limits

100% stealth not feasible

- Overhead of the monitoring is observable
- Yes, you can manipulate RDTSC but that’s not enough
- Thread-racing, network time, etc.
- Accessing guest memory introduces CPU cache side-effects that are observable!

See more details in “Who Watches The Watcher? Detecting Hypervisor Introspection from Unprivileged Guests”

Sandbox pitfall: is it realistic?

We don’t know what malware checks for

- We can guess
- We can observe empirically and deploy counter-measures
- “Malware can determine that a system is an artificial environment and not a real user device with an accuracy of 92.86%”*

*Recommended read: “Spotless Sandboxes: Evading Malware Analysis Systems using Wear-and-Tear Artifacts”

Sandbox pitfall: timeouts

Our assumption was that malware will run

- Well, it doesn’t have to run *right away*
- sleep(3 days)
- OK, we hook sleep and return in 1s
- There are infinite ways you can make your malware “sleep”
- Thread racing, API hammering, CPU cache profiling*..

*See “ExSpectre: Hiding Malware in Speculative Execution”

Sandbox pitfall: false negatives

Positive detections are great

- But no detection doesn’t mean no malware
- You just trust that the sandbox author is smarter then the malware author

- Is that really the best we can do?

Sandbox pitfall: Turing machines

Halting problem & Rice's theorem

- Non-trivial properties of programs are undecidable
- "There isn't (and never will be) a general test to decide whether a piece of software contains malicious code".

  IEEE Security & Privacy magazine, 2005

- Do we really have Turing machines?

Highly recommended read "Detecting Traditional Packers, Decisively"

No end in sight

Cat-and-mouse game to continue

- As long as we have a diverse detection ecosystem malware can’t hide from everything or risk getting extremely bloated
- There are also some new angles being proposed
- Requiring all code to be signed
- Micro Execution based testing*
- Fuzzing**

*https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/microx.pdf

**https://www.youtube.com/watch?v=bzc44WPxoxE

Thanks! Questions?

Dr. Tamas K Lengyel

tamas@tklengyel.com
https://tklengyel.com
@tklengyel

Shout out to

- Polish CERT: Michał Leszczyński, Krzysztof Stopczański
- Sebastian Poeplau, Mathieu Tarral, Sergej Proskurin