APTs Way: Evading your EBNIDS

Ali Abbasi

Jos Wetzels

UNIVERSITY OF TWENTE.

Who we are?

- **Ali Abbasi:**
  - PhD student in Distributed and Embedded System Security Group at University of Twente. Researching on embedded systems security related to critical infrastructures. Got M.Sc. at Tsinghua University in China, and was working as head of vulnerability analysis and penetration testing group at Iran National CERT in Sharif University of Technology in Tehran.

- **Jos Wetzels**:
  - M.Sc. Student and a research assistant with the Services, Cyber security and Safety research group at the University of Twente. Currently working on projects aimed at on-the-fly detection and containment of unknown malware and Advanced Persistent Threats, where we focus on malware analysis, intrusion detection, and evasion techniques. Assisted teaching hands-on offensive security classes for graduate students at the Dutch Kerckhoffs Institute for several years.

Plan of Talk

- History of Exploitation and Shellcodes
- Intro to Emulation Based NIDS Approach
- Adaptation
- Detection Techniques and Heuristics
- Evasions
- Questions?

History

- Morris Worm 1988 used Buffer overflow on “finger” service on VAX systems.
- In 1990 first polymorphic virus designed by Washburn
- In 2001 K2 introduced ADMmutate a polymorphic engine to generate shellcodes
- In 2008 Conficker worm with one byte XORed shellcode

Morris fingerd shellcode

```
pushl $68732f '/sh\0'
pushl $6e69622f '/bin'
movl sp, r10
pushl $0
pushl $0
pushl r10
pushl $3
movl sp,ap
chmk $3b
```

Signature Based IDS

- Typical Exploit Code:

JUNK	Overwrite RET	Padding Instruction (NOP)	Shellcode	Exploit the System

Detection based on:

- Return Addresses
- NOP Instructions (\x90)
- Shellcode signatures
- Detecting polymorphic encoder signatures

Limitations of Signature based NIDS

- Attackers change a byte of the payload and evade detection.

- Polymorphic shellcodes with custom encoders/decoders will evade detection.

Decryptor	Encrypted Payload

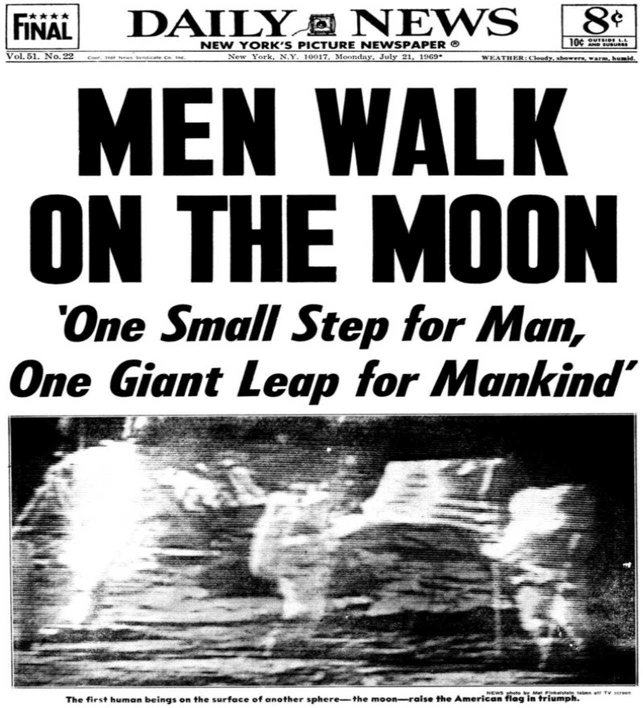
- You must always update and maintain your signatures.

Emulation-Based NIDS, a Giant Leap

- Emulation-Based NIDSes emulate suspicious payloads.

- Meant to solve the problem of detecting polymorphic shellcodes.

- Emulation-Based NIDSes are a great step forward:
  - Detect polymorphic shellcodes regardless of which type of encoding technique is used.
  - Can detect 0-day exploits.
  - Do not rely on any specific vulnerability (signatures).
  - Uses heuristics, a behavior black listing technique.



FINAL ★★★★

DAILY NEWS

NEW YORK'S PICTURE NEWSPAPER ®

8¢

10¢ OUTSIDE L.I. AND SUBURBS

Vol. 51. No. 22 New York, N.Y. 10017, Moonday, July 21, 1969* WEATHER: Cloudy, showers, warm, humid.

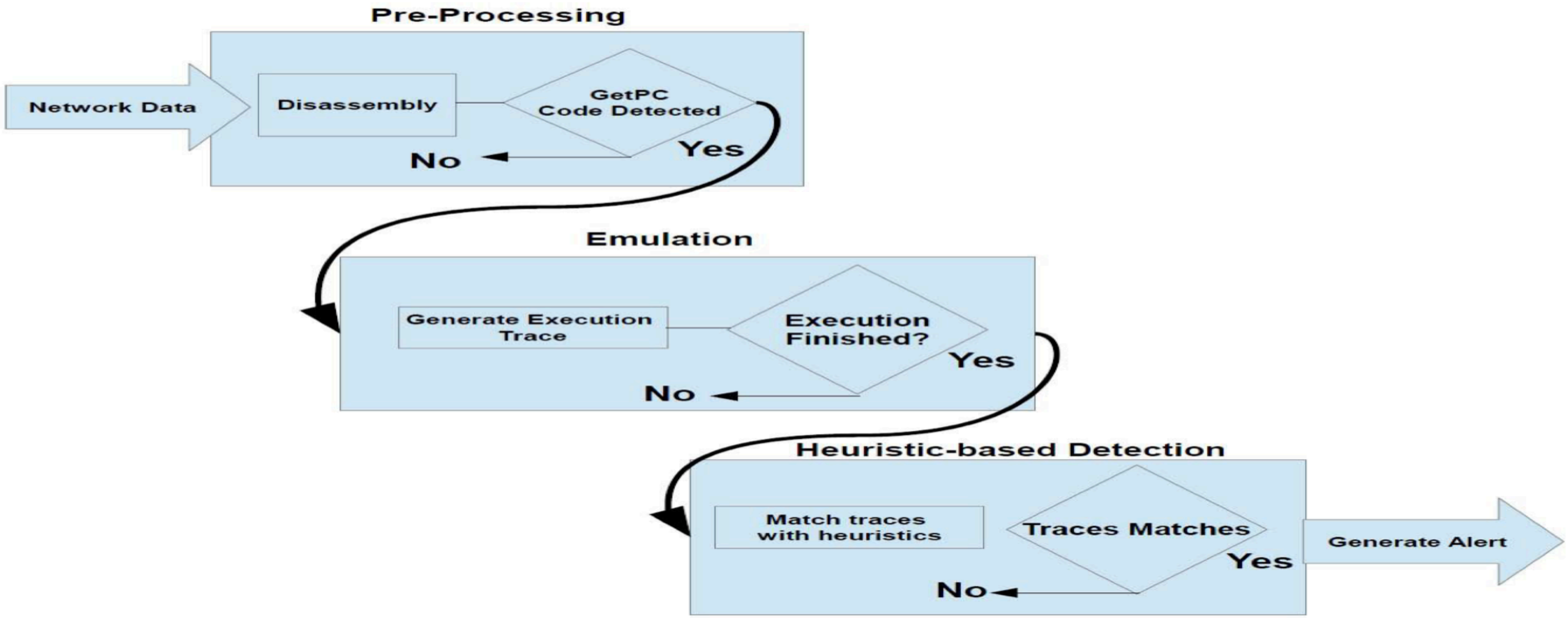
MEN WALK ON THE MOON

'One Small Step for Man, One Giant Leap for Mankind'

NEWS photo by Mel Finkelstein taken off TV screen

The first human beings on the surface of another sphere—the moon—raise the American flag in triumph.

How Emulation Based NIDS Works?



black hat EUROPE 2014

Emulation Based Technique Adopted

FireEye

SGNET

i-code

libemu
x86 emu

Honeynet
Project

NEMU

NoAH

dionaea
catches bugs

Amun

Emulation Based NIDS

- Nemu:
  - The state of the art in emulation based network intrusion detection because of its broad range of heuristics.

- Libemu:
  - A simple shellcode detection engine (used in several Honeynet projects).

Pre-Processing

- Looking for GetPC seeding instruction.
  - **Call instructions**

```
jmp startup
Getpc:
  mov (%esp), %eax
  ret
startup:
call getpc
```

```
if (inst_trace[x].getpc == 1) {
 /* getPC write */
fprintf(trace_fp, "\033[1;31m w \033[0m");
    }else if (inst_trace[x].getpc == 2) {
 /* getPC read */
fprintf(trace_fp, "\033[1;31m r \033[0m");
    }
```

```
/* emulate.c Heuristic detection trigger*/
if ((tc[prev_PC].inst.type == INSTRUCTION_TYPE_CAL

        (tc[prev_PC].inst.type ==
INSTRUCTION_TYPE_FSTENV)) {
        has_getpc = 1;
        EXECTRACE_CMD(inst_trace[num_exec].getpc
= 1);}
```

  - **FPU Instructions**

```
00C67000 D9 EE          fldz
00C67002 D9 74 24 F4    fnstenv   [esp-0Ch]
00C67006 5B             pop       ebx
```

```
/* 1 if call/fstenv, 2 if PC read, 0 if none */
if ((tc[prev_PC].inst.type ==INSTRUCTION_TYPE_CALL) ||
(tc[prev_PC].inst.type == INSTRUCTION_TYPE_FSTENV))
{
has_getpc = 1;
EXECTRACE_CMD(inst_trace[num_exec].getpc = 1);}
```

Emulation

- Create possibility to track the behavior of the emulated CPU during execution
- Emulate X86 instruction sets
- Emulate FPU Instructions
- make a generic memory image for some local variables

Basic Heuristics Detection

- **GetPC Code:**
  - detect invoking CALL or FPU instructions and check if the emulator started from the seeding GetPC code.

- **Payload Read**:
  - detect polymorphic shellcode by observing in an execution trace some form of GetPC code followed by a number of unique memory reads exceeding so-called PRT.

- **Write-Execute Instructions**:
  - Check in the areas that emulator performed write instructions how many executed X instructions get emulated. If this X instructions pass certain value then the payload will be flagged as Non-self-contained shellcode.

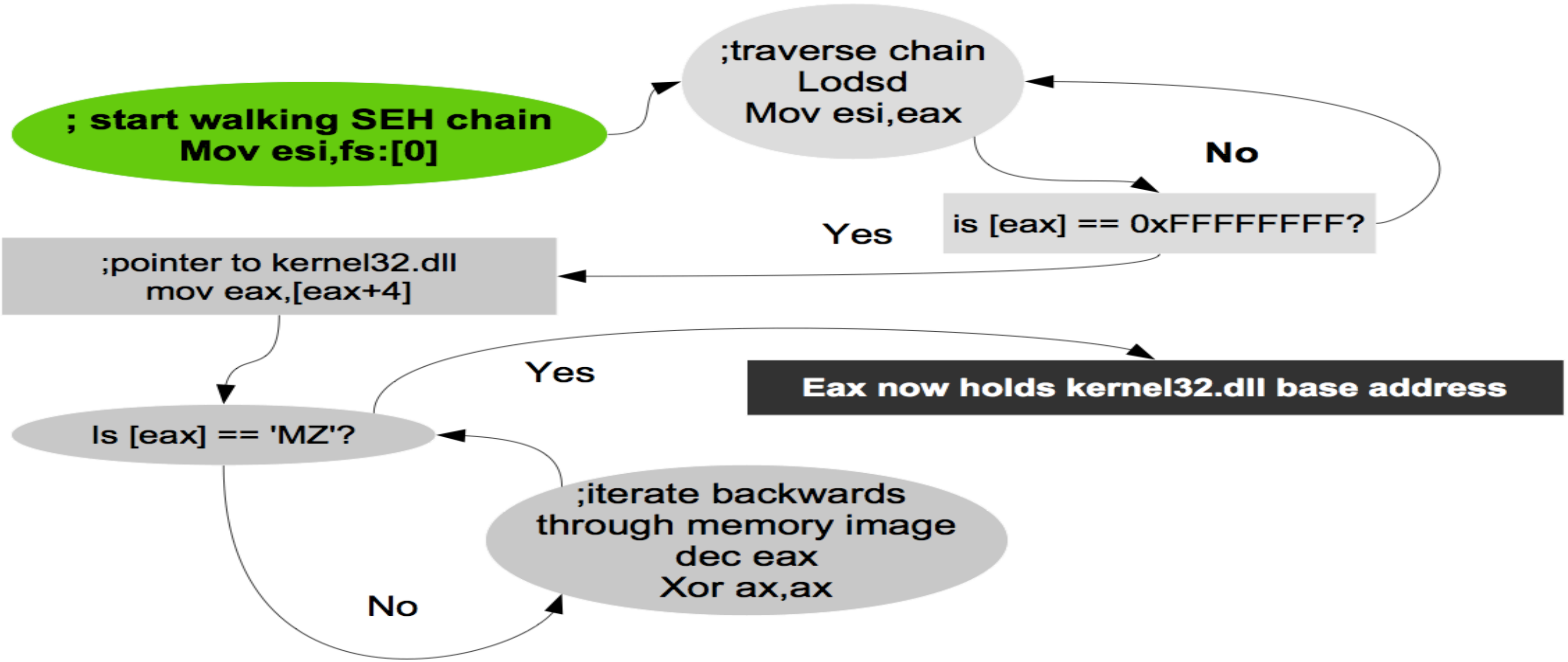
Additional Heuristics

- Kernel32.dll based address resolution
- SEH-based GetPC code
- Process Memory Scanning

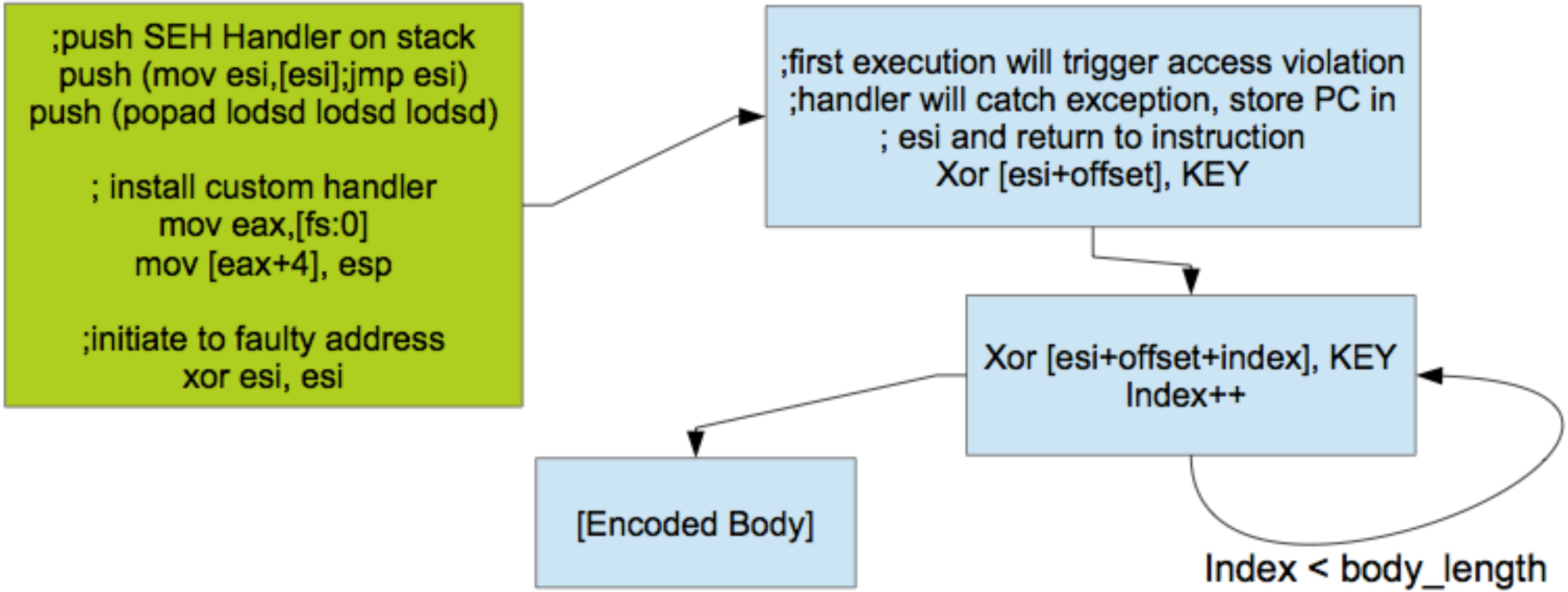
PEB Based Kernel32.dll Resolution

```
mov eax, fs:[0x30]; PEB
mov eax,[eax+0x0C]; LoaderData
mov eax,[eax+0x1C]; InInitializationOrderModulelist.flink
lodsd ; Get 2nd entry in list
mov eax,[eax+0x08] ; base address
```

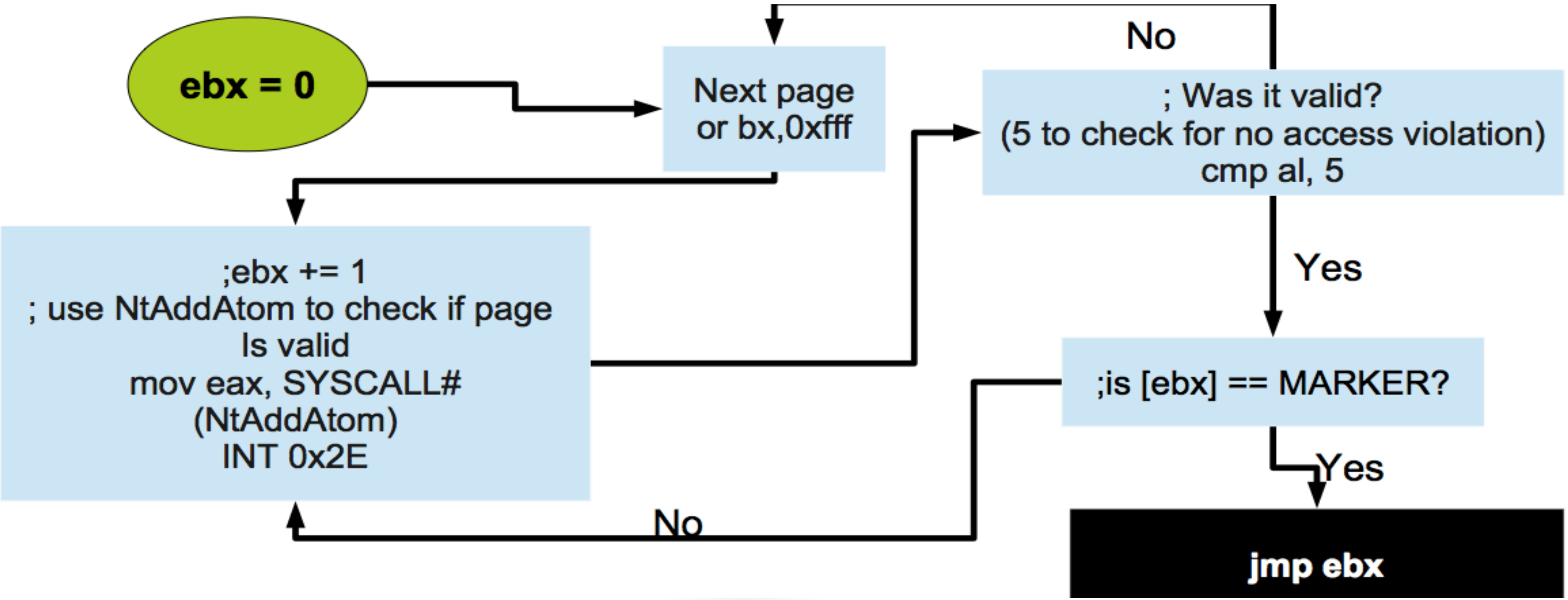
BCKWD Kernel32.dll Resolution



SEH GetPC


;push SEH Handler on stack
push (mov esi,[esi];jmp esi)
push (popad lodsd lodsd lodsd)
; install custom handler
mov eax,[fs:0]
mov [eax+4], esp
;initiate to faulty address
xor esi, esi
;first execution will trigger access violation
;handler will catch exception, store PC in
; esi and return to instruction
Xor [esi+offset], KEY
Xor [esi+offset+index], KEY
Index++
[Encoded Body]
Index < body_length

Syscall Process Memory Scanning



Evasions

	Implementation	Intrinsic
Pre-Processing	✗	
Emulation	✗	✗
Heuristics	✗	

Intrinsic Limitations

- Unavailable context data
  - Emulation-based NIDSes cannot have a complete memory image of all possible targets.
  - Context keying.
  - Non-self contained shellcodes.

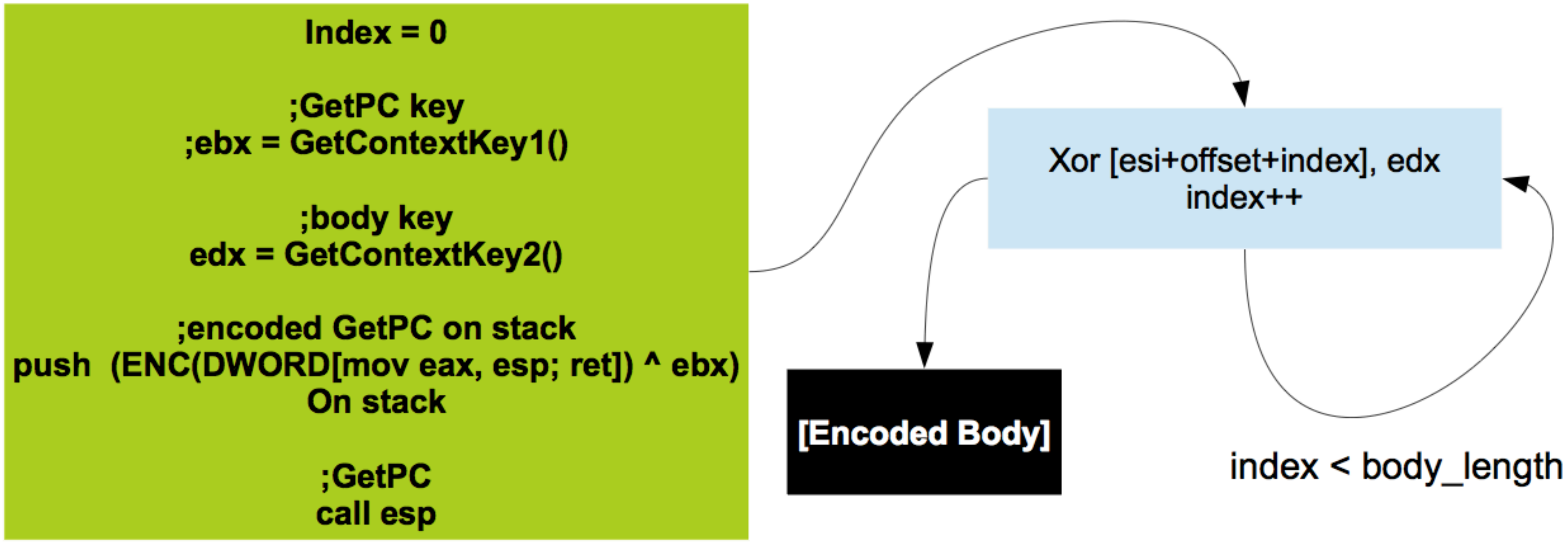
- Execution threshold
  - The emulator needs to stop at some point, the attacker can wait.

- Cannot deal with fragmented shellcode
  - Send the shellcode payload in multiple (non-consecutive) fragments.

Unavailable Context Data

- Non-self contained shellcodes
- Context Keying
  - CKPE
    - Using CPUID, values present at static memory addresses, system time or file information as a key.

Context Keyed Payload Encoding



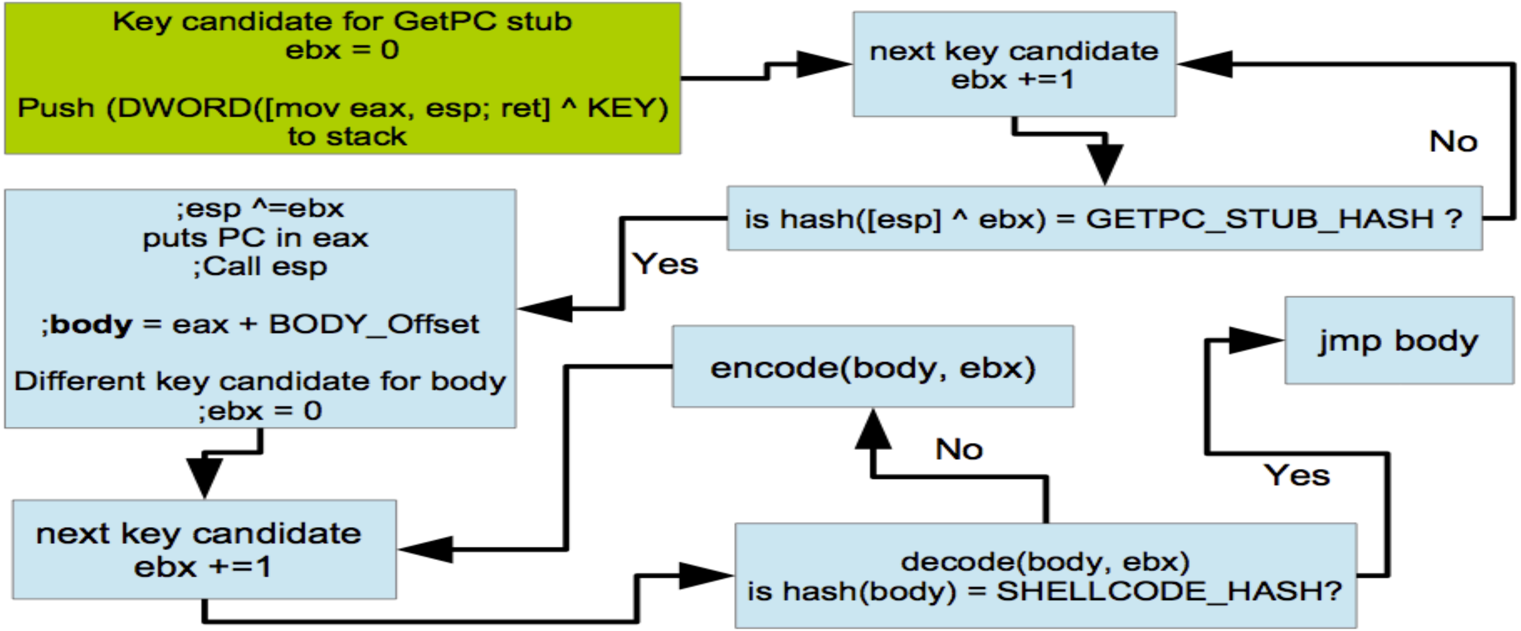
Execution Threshold

- Using time consuming loops to evade the threshold of execution

```
while (++num_exec < exec_threshold);
STATS_CMD(if (num_exec >= exec_threshold) stop_cond = S_THRESH);
```

	Opaque loop	Intensive loop	Integrated loop	RDA
Nemu	9/9	9/9	0/9	0/9
Libemu	0/1	0/1	0/1	0/1

Execution Threshold Random Decryption Algorithm (RDA)



Fragmentation

- Very rare condition
- Shellcode will be sent in two different instances.
- Shellcode have two stage but in one instance

Results

- Context keying
  - Modified version of the Context CPUID Metasploit key generator stub.
  - **Not detected.**
- Non-self contained shellcodes:
  - Dynamically built the entire GetPC code and the shellcode decoder out of ROP gadgets.
  - **Not detected.**
- Execution Threshold
  - Built shellcodes with four types of time-intensive loops.
  - Nemu could **detect half** of the shellcodes (loops were not taking enough time).
  - Libemu could **not detect any**.

Demo

- RDA (Exec Threshold)

- CKPE

Implementation Limitations

- Heuristics are kind of black listing
  - You have to list all possible shellcode behavior patterns, attackers can always find a missing one.
- Runtime difference (Emulator detection)
  - Shellcode can detect if it is being emulated.
- Unsupported instructions
- Detection relies on successful shellcode disassembly
  - Malware already applies anti-disassembly techniques to avoid analysis

Heuristics Evasion

- Kernel32.dll address resolution evasion.

- Evading Payload Read:
  - Use syscalls to execute read operations instead of reading directly in the payload shellcode.

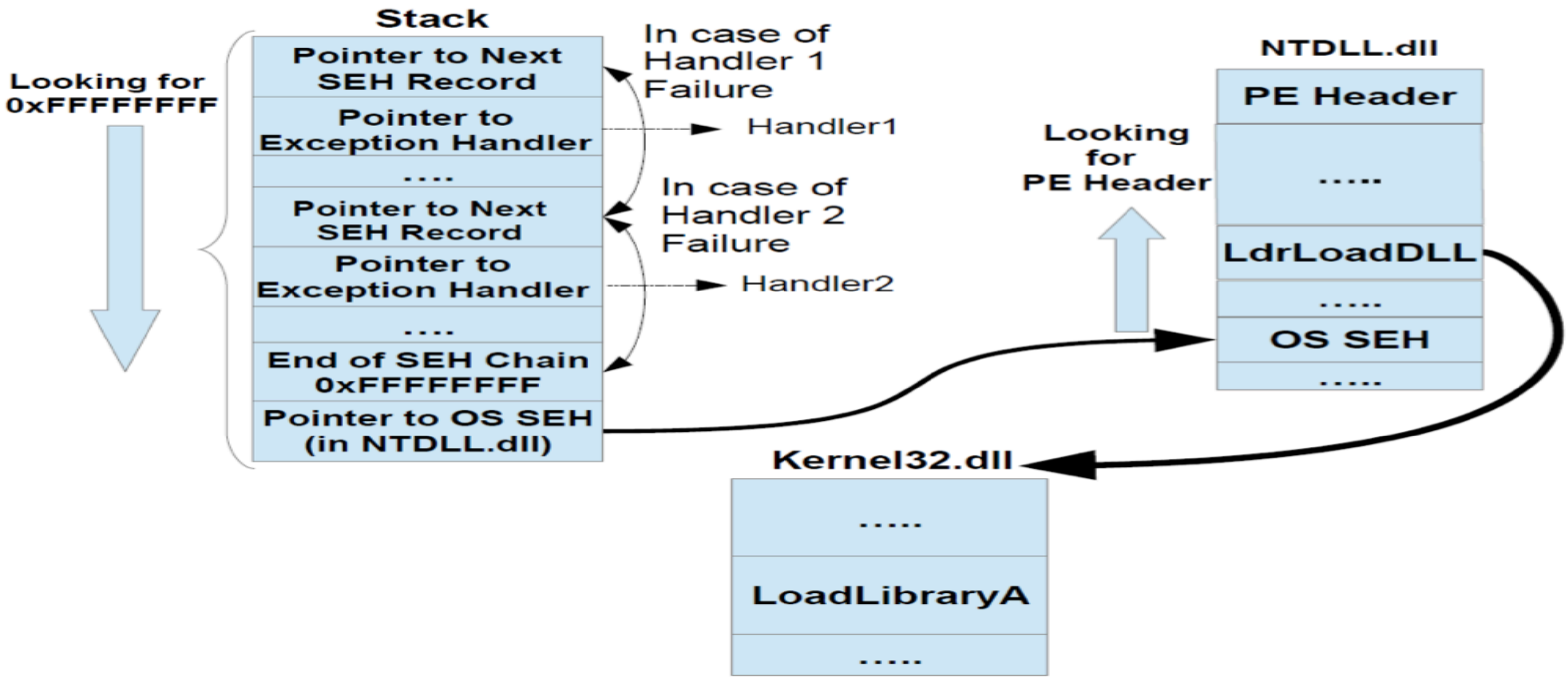
- Evading W-X Instruction:
  - Using Virtual Mapping

- Evasion of Process memory scanning :
  - SEH-walking to evade detection of SEH-based process memory scanning heuristic
  - API-based egg-hunting to evade SYSCALL-based memory scanning heuristic

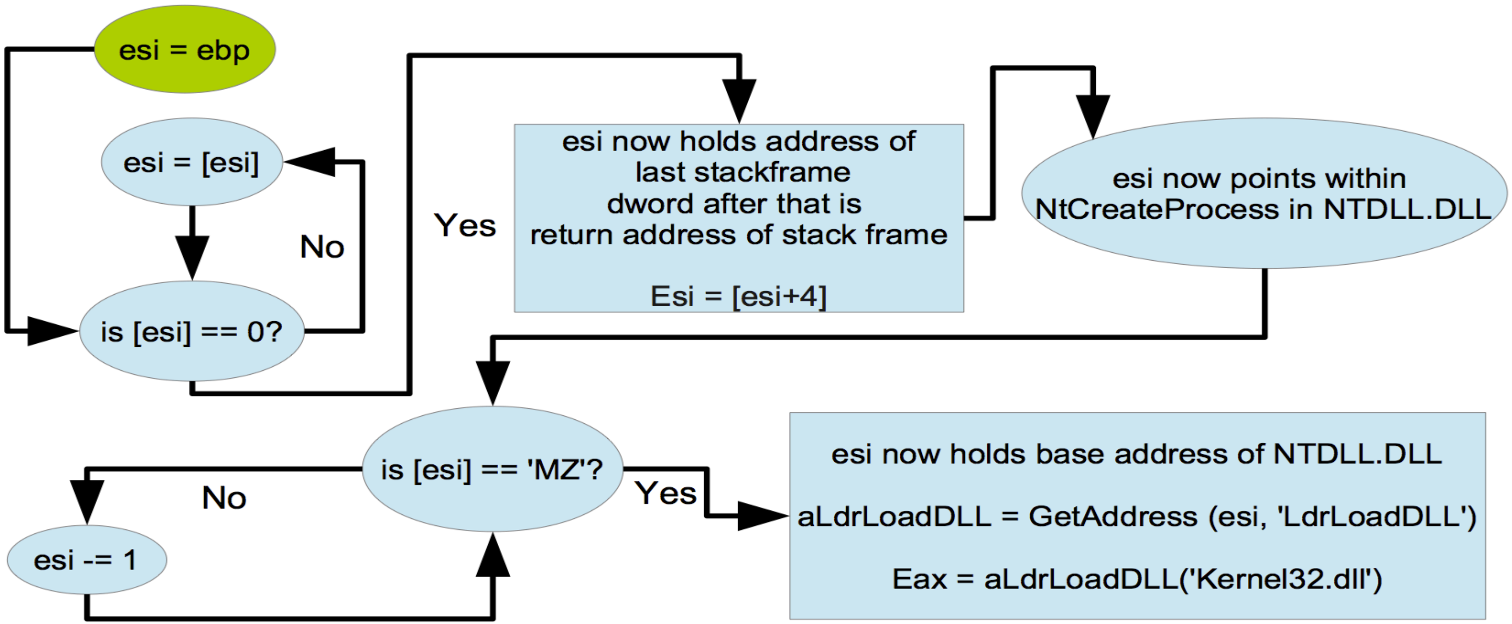
Kernel32.dll Resolution Heuristic Evasion

- Evading Kernel32.dll heuristic using SEH Chain.
- Evading Kernel32.dll heuristic using Stack Frame pointers (using NtcreateProcess API)

Evading Kernel32.dll Heuristic using SEH Chain

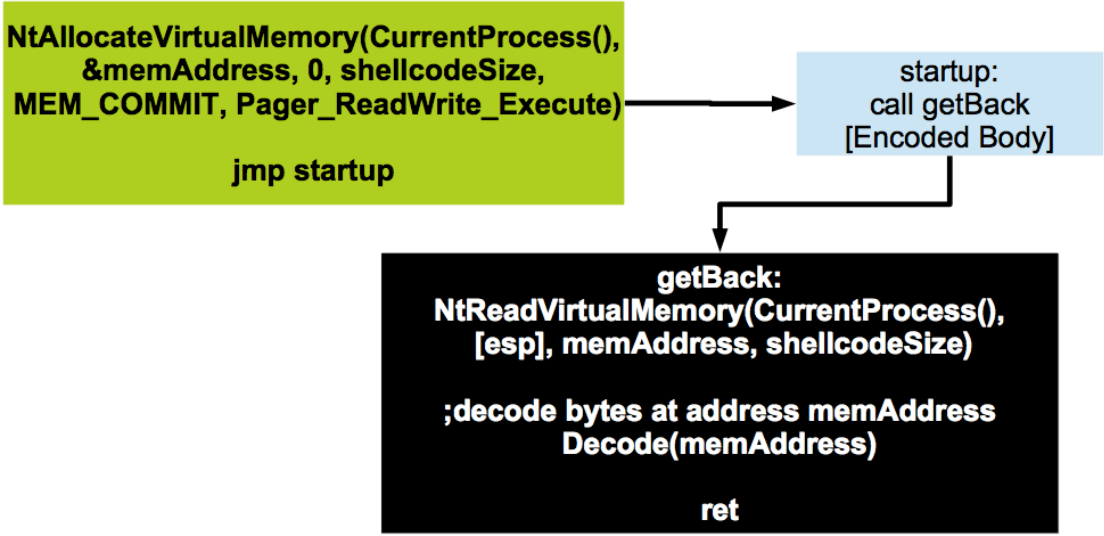


Kernel32.dll Heuristic Evasion using Stack Frame Walking



Payload Read Threshold Heuristic Evasion

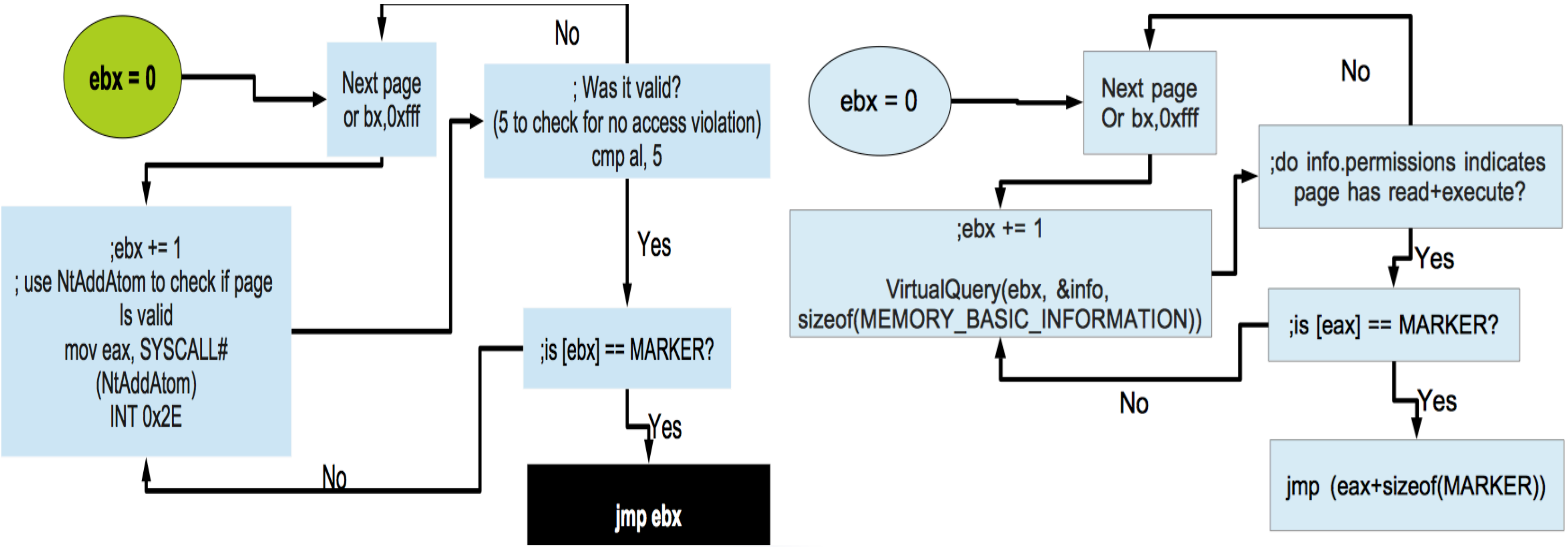
SYSCALL-based relocation



Stack Constructing Shellcode GetPC+PRT evasion

Push [shellcode[n-4:n] ^ KEY]
....
Push [shellcode[4:8] ^ KEY]
Push [shellcode[0:4] ^ KEY]
;decode bytes at esp
jmp esp

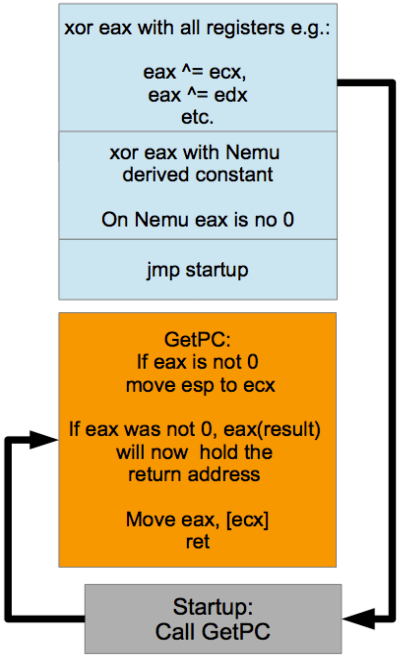
Egg Hunt (Using API)



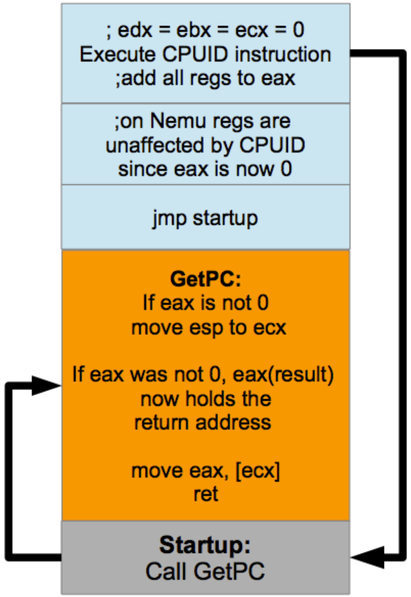
Heuristics Evasion Demo

- PRT Evasion
- Kernel32 Evasion(Both Techniques)
- Process Memory Scanning Evasion

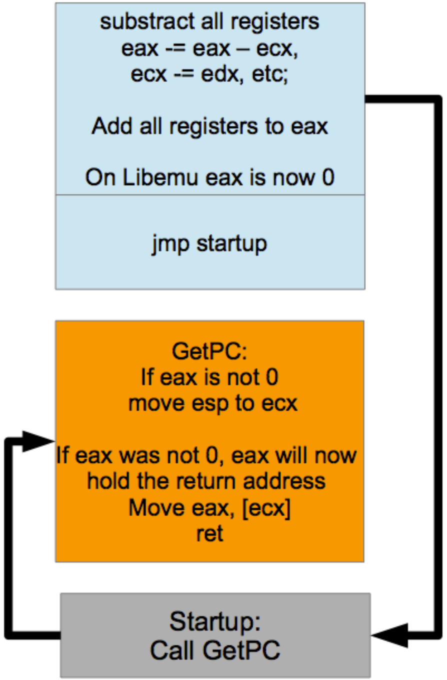
Emulator Detection



Nemu GP Register detection

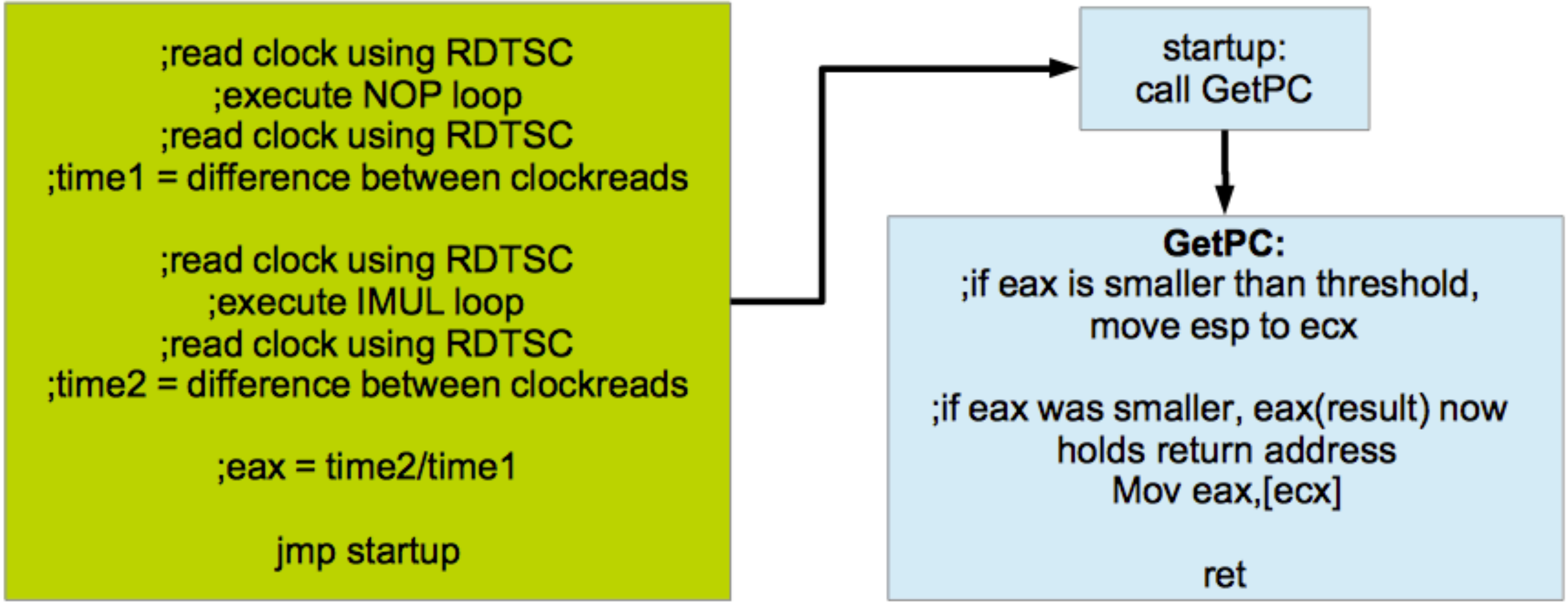


Nemu CPUID



Libemu

Timing


;read clock using RDTSC
;execute NOP loop
;read clock using RDTSC
;time1 = difference between clockreads
;read clock using RDTSC
;execute IMUL loop
;read clock using RDTSC
;time2 = difference between clockreads
;eax = time2/time1
jmp startup
startup:
call GetPC
GetPC:
;if eax is smaller than threshold,
move esp to ecx
;if eax was smaller, eax(result) now
holds return address
Mov eax,[ecx]
ret

Emulator Detection Demo

- Demo

Anti-Disassembly

- Using garbage bytes and opaque predicates
- Flow redirection to the middle of an instruction
- Push/pop-math stack-constructed shellcode
- Code transposition

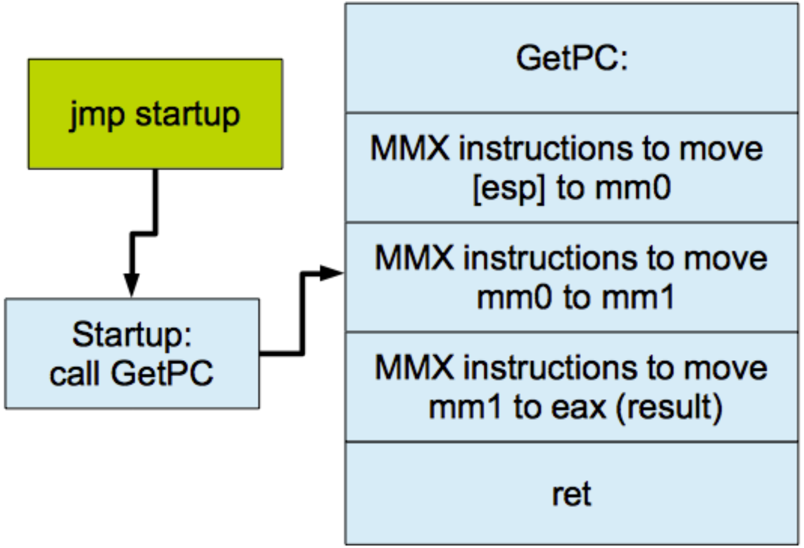
	Garbage Byte	**Flow Redirect**	**Push/Pop Math**	**Code Transposition**
Nemu	9/9	9/9	8/9	8/9
Libemu	0/1	1/1	0/1	1/1

Unsupported Instructions

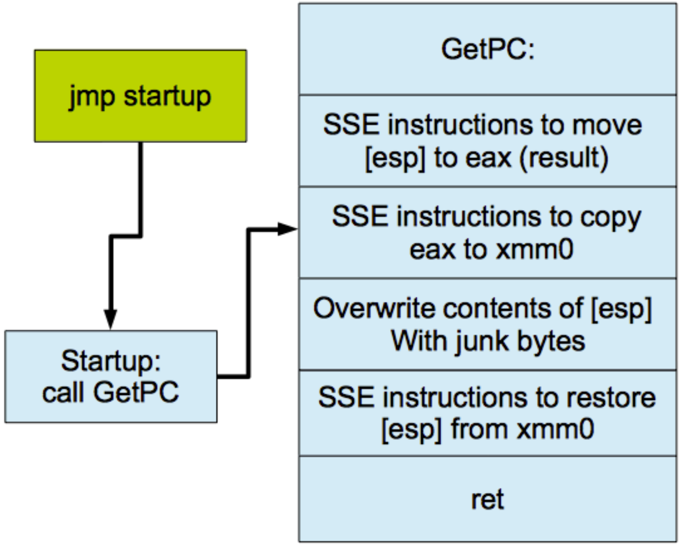
- Unsupported Instructions:
  - FPU Instructions (FNSTENV, FNSAVE))
  - MMX Instructions
  - SSE Instructions
  - Obsolete instructions (salc or xlatb)

	FPU (FNSTENV)	FPU (FNSAVE)	MMX	SSE	OBSOL
Nemu	9/9	0/9	0/9	0/9	0/9
Libemu	1/1	0/1	0/1	0/1	0/1

Unsupported Instructions



MMX



SSE

Question?

Everything that has a beginning has an end

The Matrix Revolution

.

Contact Us:

Ali Abbasi: a.abbasi@utwente.nl

Jos Wetzels: a.l.g.m.wetzels@student.utwente.nl