Advanced Payload Strategies: What is new, what works and what is hoax?

Rodrigo Rubira Branco (BSDaemon)
Senior Vulnerability Researcher
Vulnerability Research Labs (VRL) – COSEINC
rodrigo_branco *noSPAM* research.coseinc.com

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Who Am I?

- Rodrigo Rubira Branco aka BSDaemon;
- Senior Vulnerability Researcher/COSEINC
- Was Security Expert @Check Point & Linux Developer in the Advanced Linux Response Team of IBM;
- Mainteiner of many open-source projects;
- Some interesting researchs:
  - FreeBSD/NetBSD/TrustedBSD/DragonFlyBSD all version kernel integer overflow
  - FreeBSD 5.x Kernel Integer Overflow Vulnerability
  - Apple Mac OS X 10.4.x kernel memory corruption vulnerability
  - X11R6 XKEYBOARD extension Strcmp() buffer overflow vulnerability (Solaris all versions, including 10)
  - Remote exploit for Borland Interbase 7.1 SP 2 and lower
  - Remote root exploit for AppleFileServer
  - MacOSX DirectoryService local root exploit
  - Halflife <= 1.1.1.0 , 3.1.1.1c1 and 4.1.1.1a remote exploit
  - Mac OS X v10.3.8, Mac OS X Server v10.3.8 env overflow
  - 2 security bugs reported to Microsoft (affects ISA Server)
  - Phrack Article about SMM rootkits
- RISE Security member
- SANS Instructor: Mastering Packet Analysis, Cutting Edge Hacking Techniques, Reverse Engineering Malwares
- Member of the GIAC Board for the Reverse Engineering Malwares Certification
- **Organizer: H2HC Conference (http://www.h2hc.com.br/en)**

©2009 COSEINC. All rights reserved. CONFIDENTIAL

DISCLAIMER

- Altought I'm a company employee and I'm using my work time to come here, everything that I'm presenting was completely created by me and are not supported, reviewed, guaranteed or whatever by my employer
  - **The protection part of this presentation is my master thesis and was started many years ago**

- Some technologies analysed in this work are patented so if you wish to use, expand or whatever the ideas mentionated here it's a good idea to contact me or the companies who are holding the patents first

- I'm using whenever possible Check Point's terminology, since they hold a patent on the matter

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Agenda

- Objectives / Introduction

PART I

- Modern Payloads
  - Polymorphic Shellcodes
    - Context-keyed decoders
    - Target-based decoders
  - Camouflage – Bypassing context recognition
  - Syscall proxying and remote code interpreter/compiler

PART II

- How intrusion prevention/detection system works
- Actual limitations and proposals
  - Network traffic disassembly
  - Virtual execution challenges
- Future

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Objectives

- Show the added value of Hacking
- Demonstrate how prevention systems works, and why/ when they are useful (or not)
- Explain what changed in the world of payloads without focusing in the assembly language because it became boring
- Most important: Start a discussion regarding possible solutions on how to detect this advanced payloads in a generic way, without caring about other problems we are actually suffering (like SSL sites for example) – All the live demonstrations are a master project which will be released together with a paper on this subject later on this year

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Introduction

- Evolution of exploitation frameworks made possible for newbies to use advanced encoding techniques
- Assembly knowledge or advanced skills are not anymore a pre-req for the usage of advanced payloads (are you sure it was in the past?)
- There is a huge gap of what actually exists in those frameworks and what is been formaly documented (yeah, we are all guilt)
- Detection/Prevention systems have not evolved as well (they tried, but they are loosing miserably the competition)
- Old-school vulnerabilities (let’s say, system-level, low-level, or whatever that involves code injection) are still not generically prevented by those systems – can you expect them to prevent web 2.0 attacks??

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Survey

```
mfmsr   r0                      /* Get current interrupt state */
rlwinm  r3,r0,16+1,32-1,31      /* Extract old value of 'EE'   */
rlwinm  r0,r0,0,17,15           /* clear MSR_EE in r0          */
SYNC                            /* Some chip revs have
                                   problems here...            */
mtmsr   r0                      /* Update machine state        */
blr                             /* Done                        */
```

↓

cli

↓

CLear Interrupt Flag - Clearing the IF flag causes the processor to ignore maskable external *interrupts*

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Survey

This presentation will focus on the public that is used with the explanation approach:

CLear Interrupt Flag - Clearing the IF flag causes the processor to ignore maskable external *interrupts*

Whenever is possible I'll simplify the contents, but a good base on the matters of this presentation are required for a best understanding.

Ask your questions as soon as possible, since usually I don't leave any time in the end.

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

PART I

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Modern Payloads

- They try (or they do) to avoid detection (channel encryption, code encoding)
- Usually they are more advanced, which means, bigger, which means staged (they ‘download’ in someway more portions of their own code)
- The idea is not just have a remote ‘/bin/sh’, but provide a complete environment without leave any forensics evidences

©2009 COSEINC. All rights reserved. CONFIDENTIAL

What is a polymorphic shellcode?

- Is a code with the ability to automatically transform itself into a semantically equivalent variant, frustrating attempts to have a verifiable representation.
  - They avoid detection
  - They help to bypass application-specific filters (tollower, toupper, isascii...)

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Polymorphism – How it works?

Generally, divided in two pieces:

- The decoding loop
- The GetEIP trick

```
--------------------
call decoder
--------------------
shellcode
--------------------
decoder
--------------------
jmp shellcode
--------------------
```

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Polymorphism - How it works?

The decoder will invert the process used to encode the shellcode.

This process usually are a simple byte-to-byte loop + operations, like:

- ADD
- SUB
- XOR
- SHIFT
- Byte invertion

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Trampoline – No Null Bytes

```
/ *  the %ecx register contains the size of assembly code (shellcode).
 *
 *  pushl    $0x01
 *            ^^
 *            size of assembly code (shellcode)
 *
 *  addb    $0x02,(%esi)
 *            ^^
 *            number to add
 */
   jmp     label3
label1:
   popl    %esi
   pushl   $0x00  /* <-- size of assembly code (shellcode) */
   popl    %ecx
label2:
   addb    $0x00,(%esi) /* <-- number to add */
   incl    %esi
   loop    label2
   jmp     label4
label3:
   call    label1
label4:

/* assembly code (shellcode) goes here */
```

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Noir's trick: fnstenv

- *Execute an FPU instruction (fldz)*
  - *D9 EE FLDZ -> Push +0.0 onto the FPU register stack.*

- *The structure stored by fnstenv is defined as user_fpregs_struct in sys/user.h (tks to Aaron Adams) and is saved as so:*

  0 | Control Word
  4 | Status Word
  8 | Tag Word
  12 | FPU Instruction Pointer Offset
  ...

- *We can choose where this structure will be stored, so (Aaron modification of the Noir's trick):*

```
fldz
fnstenv -12(%esp)
popl %ecx
addb 10, %cl
nop
```

- *We have the EIP stored in ecx when we hit NOP. It's hard to debug this technique using debuggers (we see 0 instead of the instruction address)*

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Fnstenv

```
/*
 *  the %ecx register contains the size of assembly code (shellcode).
 *
 *  pushl    $0x00
 *            ^^
 *            size of assembly code (shellcode)
 *
 *  xorb     $0x00,(%eax)
 *            ^^
 *            number to xor
 */
    fldz
    fnstenv -12(%esp)
    popl    %eax

    pushl   $0x00  /* <-- size of assembly code (shellcode) */
    popl    %ecx
    addb    $0x13, %al  /* <-- size of the entire decoder */

label1:
    xorb    $0x00,(%eax) /* <-- number to xor */
    incl    %eax
    loop    label1

/* assembly code (shellcode) goes here */
```


©2009 COSEINC. All rights reserved. CONFIDENTIAL

Target-based decoders

- Keyed encoders have the keying information available or deductived from the decoder stub.

- That means, the static key is stored in the decoder stub

or

- The key information can be deduced from the encoding algorithm since it's known (of course we can not assume that we will know all the algorithms)

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

xoring against Intel x86 CPUID

- Itzik's idea: http://www.tty64.org
- Different systems will return different CPUID strings, which can be used as key if we previously know what is the target platform
- Important research that marked the beginning of target-based decoders, but easy to detect by the 'smart' disassembly – more on this later

©2009 COSEINC. All rights reserved. CONFIDENTIAL

xor-cpuid

```
/*  Coded by Rodrigo Rubira Branco rodrigo_branco@research.coseinc.com */
    xorl  %eax, %eax  /* EAX=0 -  Getting vendor ID */
    cpuid

    jmp     label3

label1:
    popl    %esi

    pushl   $0x00  /* <-- size of assembly code (shellcode) */
    popl    %ecx

label2:
    xorb    %bl, (%esi)
    incl    %esi
    loop    label2
    jmp     label4

label3:
    call    label1

label4:
/* assembly code (shellcode) goes here */
```

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Context-keyed decoders

- I)ruid's idea: http://www.uninformed.org/?v=9&a=3&t=txt
- Instead of use a fixed key, use an application-specific one:
  - Static Application Data (fixed portions of memory analysis)
  - Event and Supplied Data
  - Temporal Keys
- Already implemented in Metasploit...

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Camouflage – Bypassing context

- My big friend Itzik Kotler showed in Hackers 2 Hackers Conference III
- The idea is to create a shellcode that looks like a specific type of file (for example, a .zip file)
- This will bypass some systems, because they will identify it's a binary file and will not trigger an alert
  - Interesting is that some systems uses file identification to avoid false-positivies

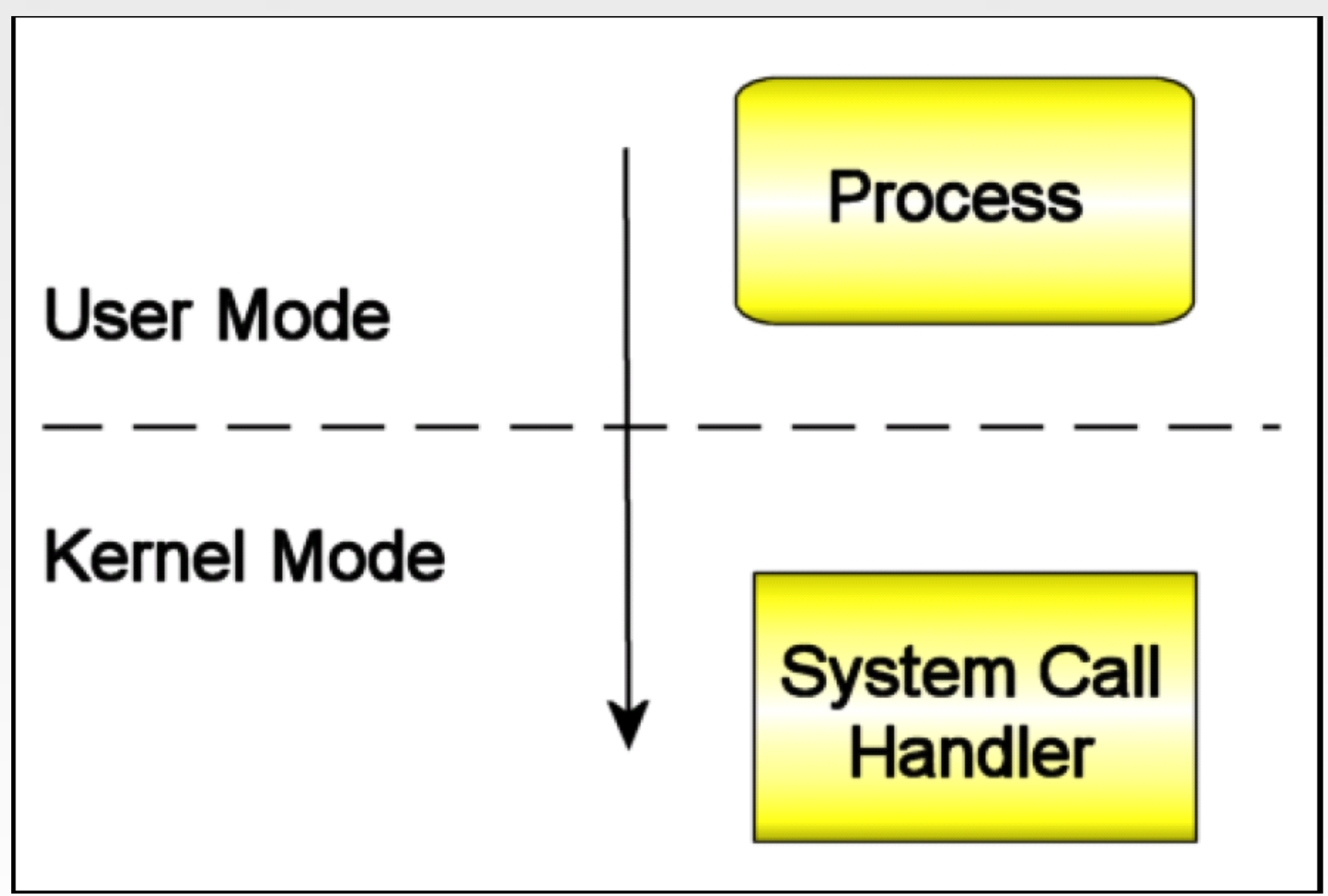
©2009 COSEINC. All rights reserved. CONFIDENTIAL

Syscall Proxying

- When a process need any resource it must perform a system call in order to ask the operating system for the needed resource.
- Syscall interface are generally offered by the libc (the programmer doesn't need to care about system calls)
- Syscall proxying under Linux environment will be shown, so some aspects must be understood:
  - Homogeneous way for calling syscalls (by number)
  - Arguments are passed via registers (or a pointer to the stack)
  - Little number of system calls exists.

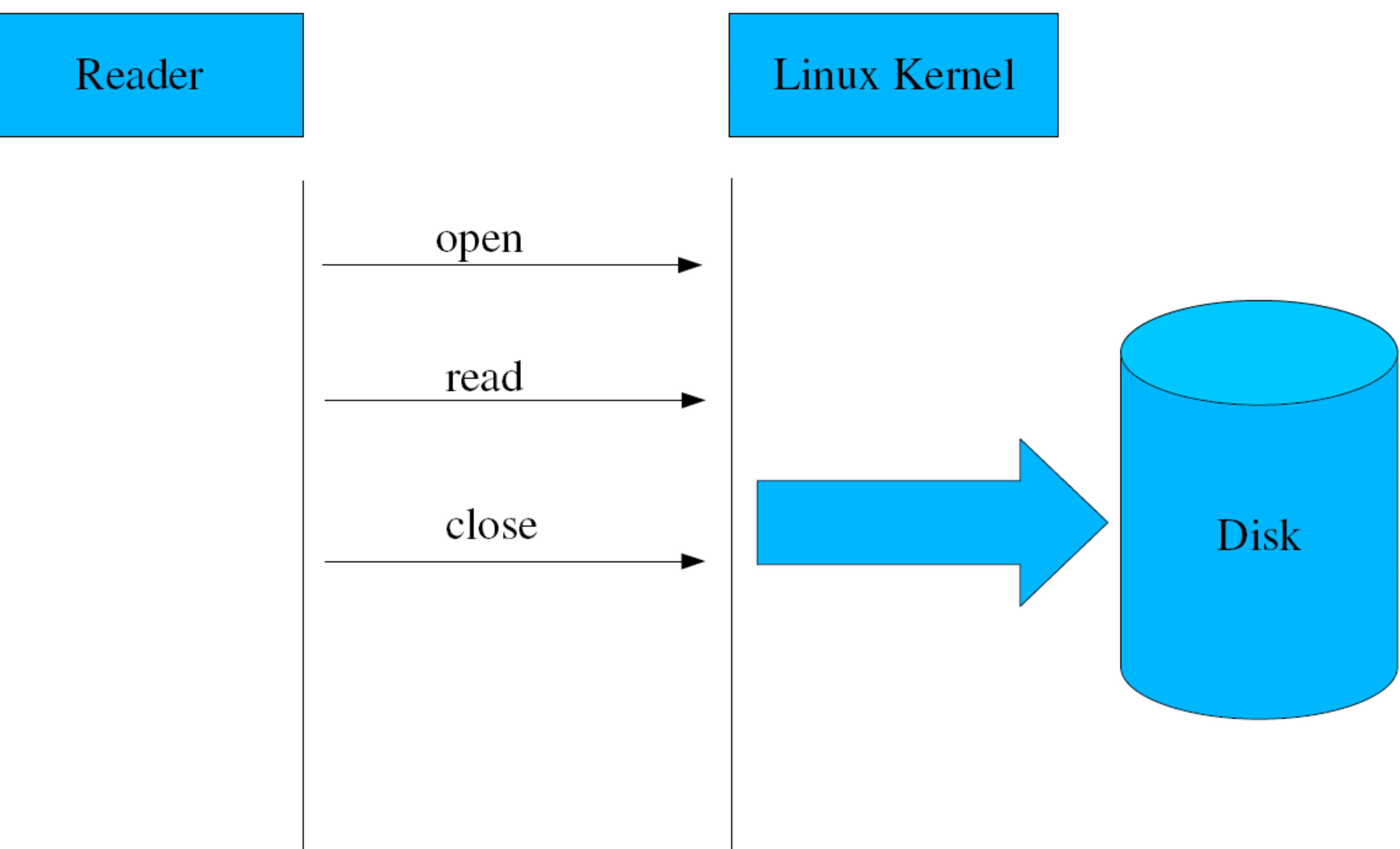
©2009 COSEINC. All rights reserved. CONFIDENTIAL

System Call – How does it works?



©2009 COSEINC. All rights reserved. CONFIDENTIAL

System Call – Reading a File...



©2009 COSEINC. All rights reserved. CONFIDENTIAL

System Call – strace output

```
# strace cat /etc/passwd


execve("/bin/cat", ["cat", "/etc/passwd"], [/* 17 vars */]) = 0
...
open("/etc/passwd", O_RDONLY|O_LARGEFILE) = 3
...
...
read(3, "root:x:0:0:root:/root:/bin/bash\n"..., 4096) = 1669
...
close(3)                              = 0
```

- As we can see using the strace program, even a simple command uses many syscalls to accomplish the task

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

System Call Arguments

- EAX holds the system call number
- EBX, ECX, EDX, ESI and EDI are the arguments (some system calls, like socket call do use the stack to pass arguments)
- Call int $0x80 (software interrupt)
- Value is returned in EAX

©2009 COSEINC. All rights reserved. CONFIDENTIAL

System Call Proxying

- The idea is to split the default syscall functionality in two steps:
  - A client stub

    Receives the requests for resources from the programs

    Prepair the requests to be sent to the server (marshalling)

    Send requests to the server

    Marshall back the answers

  - A syscall proxy server

    Handle requests from the clients

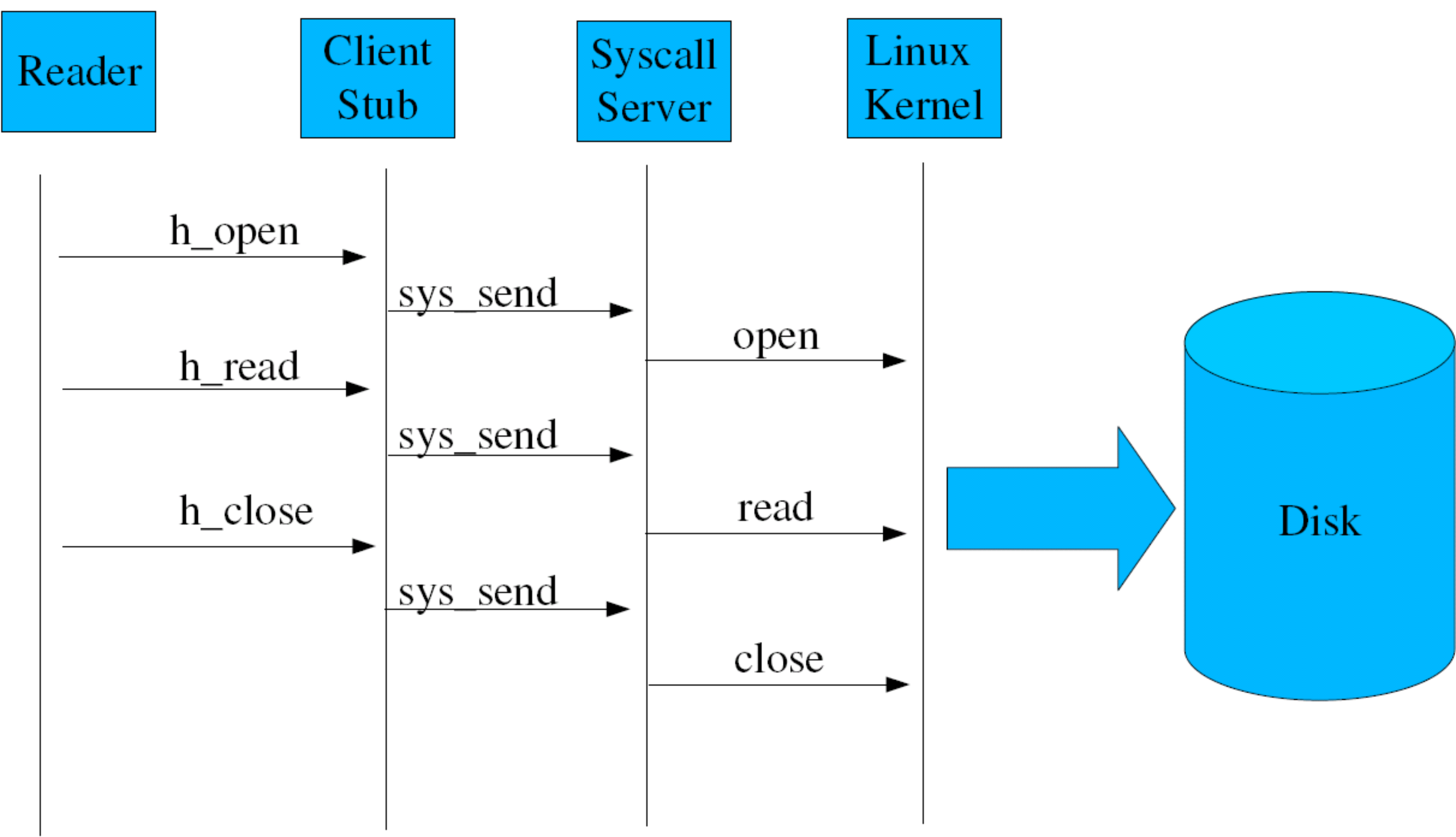
    Convert the request into the native form (Linux standard – but may support, for example, multi-architectures and mixed client/server OS)

    Calls the asked system call

    Sends back the response

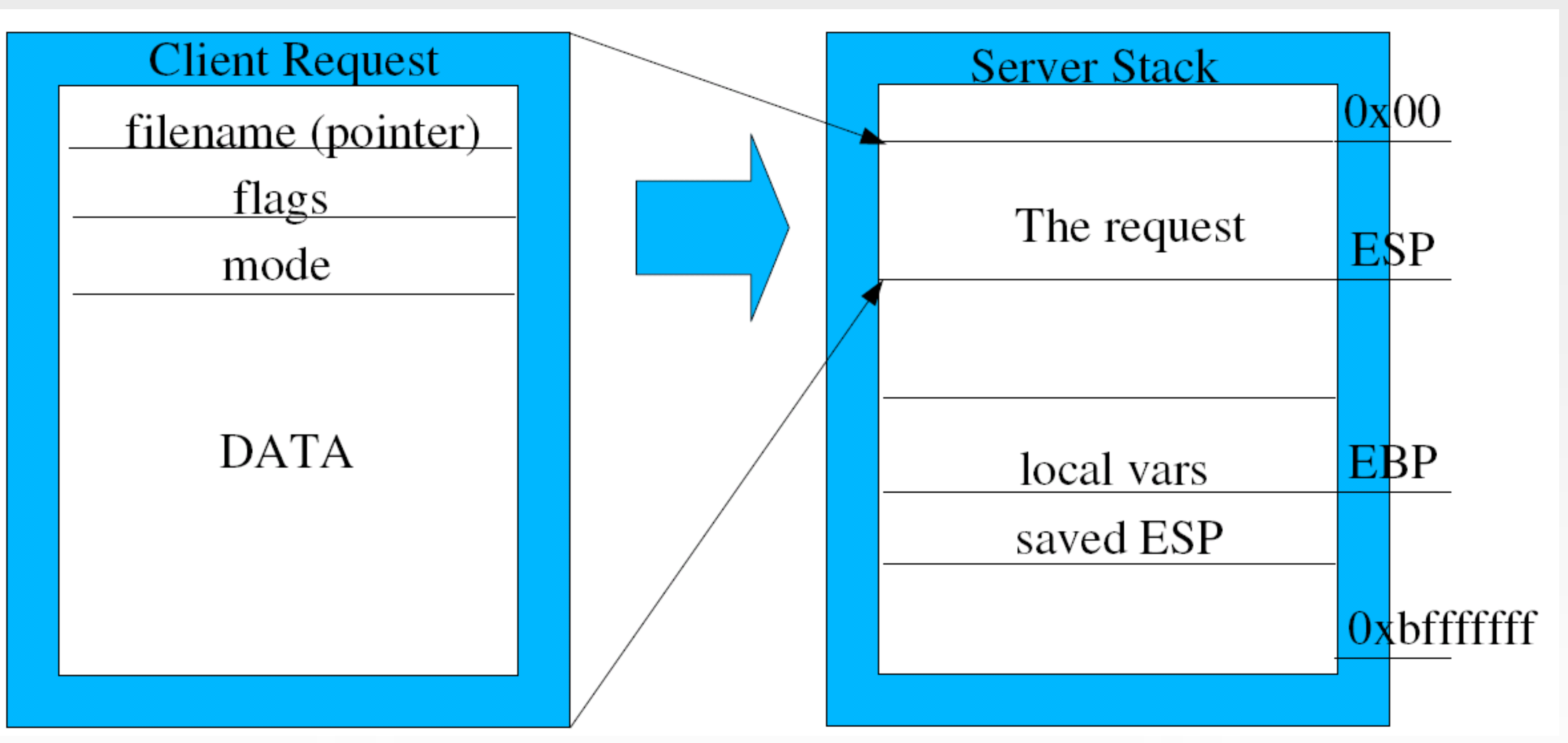
©2009 COSEINC. All rights reserved. CONFIDENTIAL

System Call Proxying – Reading a File...



©2009 COSEINC. All rights reserved. CONFIDENTIAL

System Call Proxying – Packing



©2009 COSEINC. All rights reserved. CONFIDENTIAL

MOSDEF

- MOSDEF (mose-def) is short for “Most Definately”
- MOSDEF is a retargetable, position independent code, C compiler that supports dynamic remote code linking written in pure python
- In short, after you’ve overflowed a process you can compile programs to run inside that process and report back to you
  - » Source: http://www.immunityinc.com/downloads/MOSDEF.ppt

©2009 COSEINC. All rights reserved. CONFIDENTIAL

PART II

©2009 COSEINC. All rights reserved. CONFIDENTIAL

How IDS/IPS works

- Capture the traffic
- Normalize it (session/fragment reassembly)
- Inspect
  - Pattern matching
  - Protocol validation (some does just basic protocol validation, like ip, tcp and udp only, some others are doing more advanced validations, like RPC implementations, SMB, DNS, HTTP... But that really does not matter here)
  - Payload verification -> Here we are interested in

©2009 COSEINC. All rights reserved. CONFIDENTIAL

0day protection

- Every vendor in the market claims 0day protection
- Every vendor in the market claims polymorphic shellcode detection
- Every vendor in the market are lieing?
  - → **THIS IS A JOKE**

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Methods for detecting malicious code

- Signatures/Patterns
  - Reactive – can only detect known attacks.
  - Require analysis of each vulnerability/exploit.
  - Vulnerable to obfuscation & polymorphic attacks.

- Anomaly Detection
  - Baseline profiles need to be accumulated over time
    - Protocols, Destinations, Applications, etc.
  - High maintenance costs
    - Need highly experienced personnel to analyze logs
  - If the exploit looks like normal traffic – it will go undetected.

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Patterns on the decoder...

- Detect the fixed portion of the code: The decoder
- It does not work, because the decoder itself can be mutated to avoid pattern matching:
  - Trash code (jumped)
  - Do nothing code (replacing NOPs)
  - Self-constructing decoders (shikata ga nai)
- SCMorphism help (no new releases since 2004!!)

```
-A <nopsize in bytes> --> Gen. random alphanumeric junks only (only print the ju
nks by now - see TODO)
-j <nopsize> --> Size of NOP include in the shellcode
-T <type> --> Type of NOPs:
        1-) In the begin of shellcode (default)
        2-) In the middle of shellcode
        3-) Both (by now, the -j option will be multi by 2, see TODO
-L <location> --> Location of NOPs:
        1-) In the original shellcode (default)
        2-) In the encoded shellcode
        3-) In both
-v --> Version and Greetz information
-h --> Help stuff

 Choose your option, please
bsdaemon:/Projetos/SCMorphism# _
```

Shikata ga nai

- Created by spoonm for Metasploit
- Uses FPU GetEIP trick:
  - 102 FPU instructions available + fnstenv
  - 4 clear ECX instructions (ECX used as counter)
  - 1 pop EBX
  - 1 move key
  - 6 loop blocks
  - 1 loop instruction
- No-interation between some portions permits then to be randomly exchangeable (difficult to find patterns)

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Actual limitations and proposals

- The truth is: It's impossible to detect this kind of shellcode just using pattern matching
  - I'm not saying that it is useful in anyway

- What about behavioural analysis? Network traffic disassembly? Code emulation?
  - Assuming the perfect world, where the computational power is unlimited maybe it is easy... But in the real world, is it possible?

©2009 COSEINC. All rights reserved. CONFIDENTIAL

So, how it can be detected?

- Disassembling of the network traffic
  - Lots of false positives
  - Are you sure you are really analysing the payload?
    - What if the vuln. affects the underlying protocol layer?
    - What about session reassembly?
    - What if......... -> I DON'T CARE, anyway an IPS need to know about that ☺
- To avoid the false positives we need a 'simulator' to follow the actual code logic:
  - Support to multi-architectures

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Malicious Code Protector

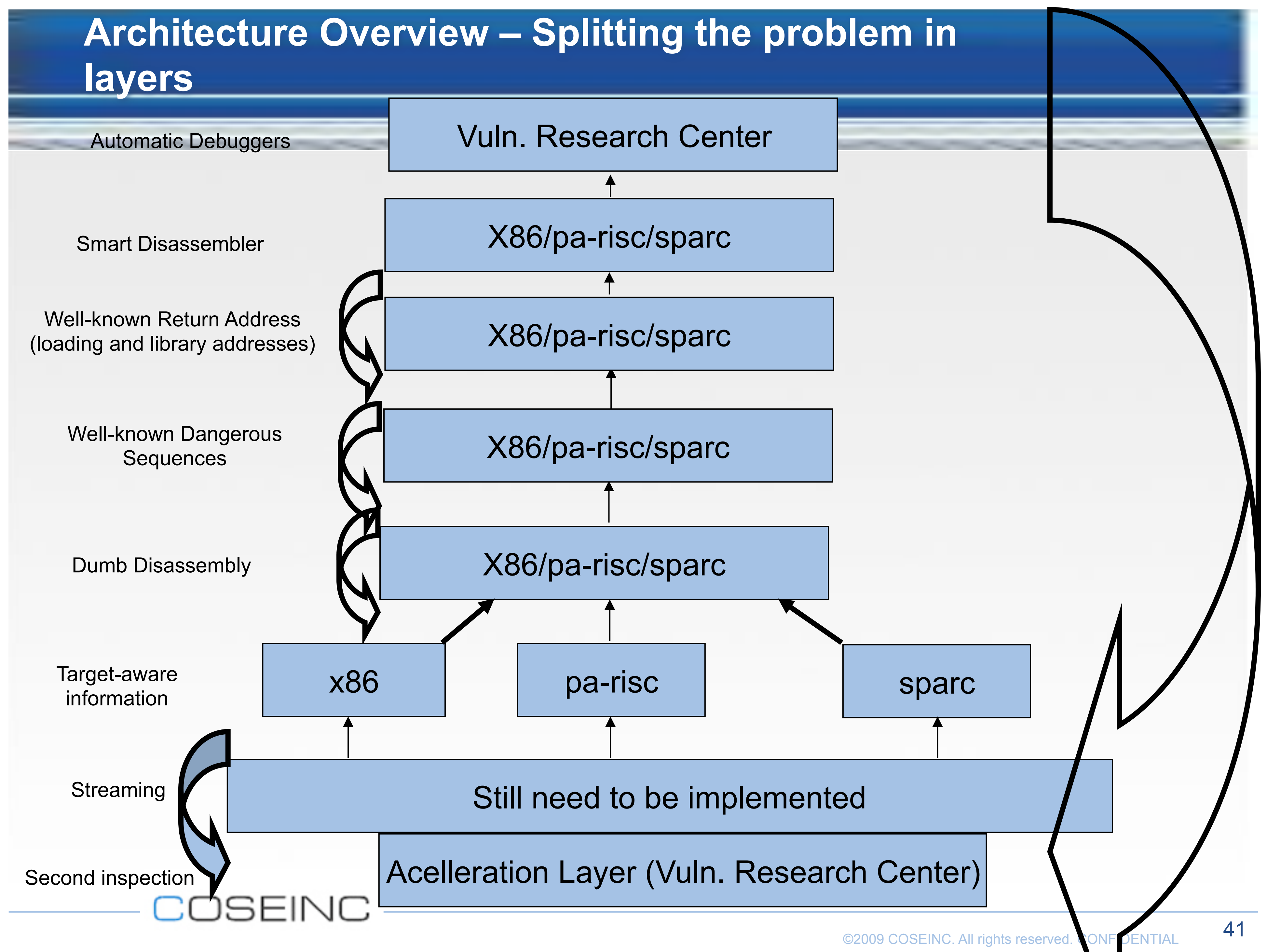
- Check Point Patent (US Patent 20070089171)
- Disassembly of the network traffic
  - » Intelligent Disassembler
  - » CPU Emulation
  - » Meta Instructions
  - » Heuristic decision function
- If it's a shellcode (probably a false positive, i.e.: a gif image), try to 'follow' it
  - – Disassembler just works with x86 and SPARC code
  - – High rate of false positivies
  - – Performance-penalti!
  - – Still the best option, but... What improvements are needed?

©2009 COSEINC. All rights reserved. CONFIDENTIAL

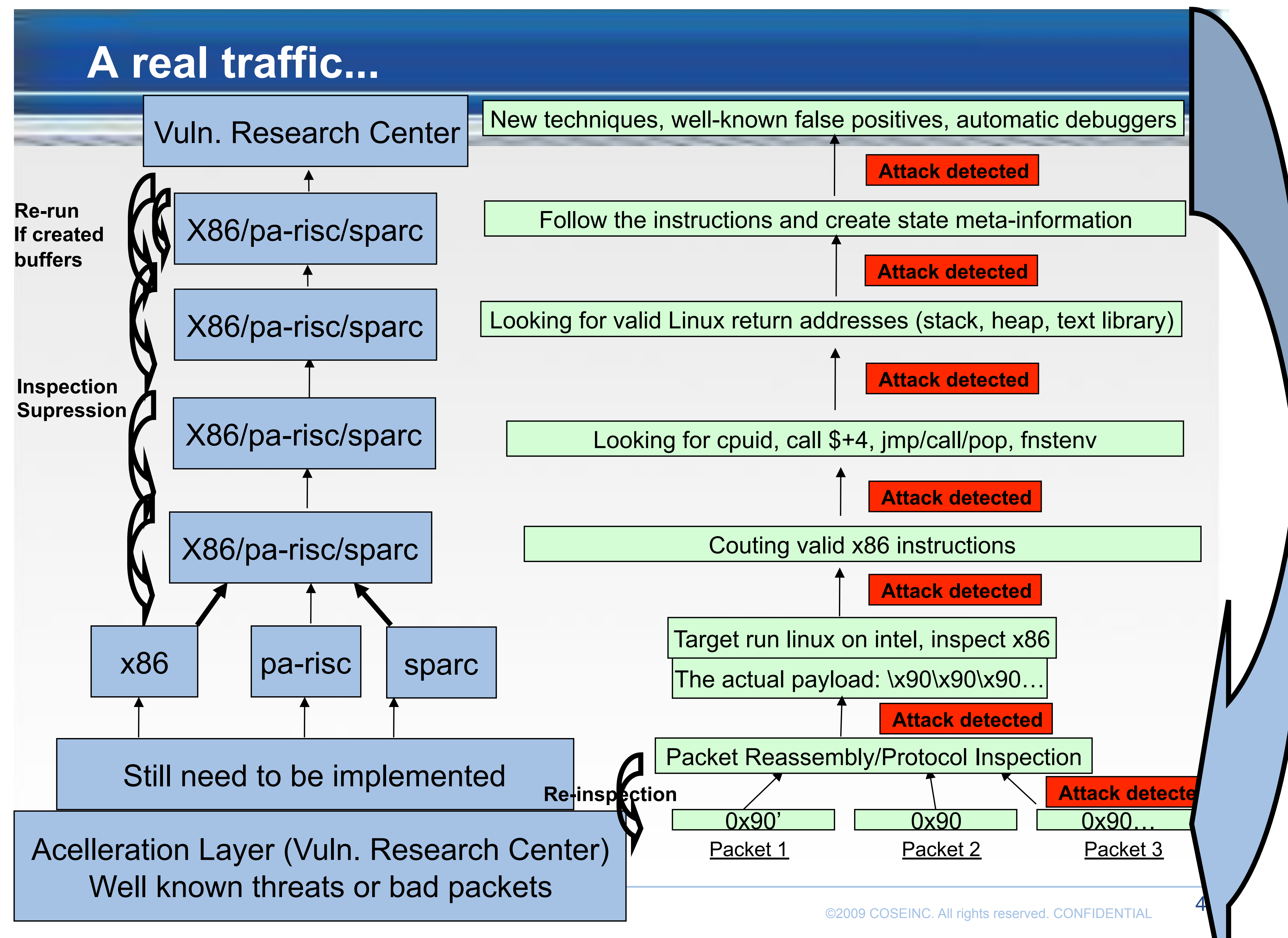
What to do?

- Disassemble input
  - Translate bytes into assembly instructions
  - Follow branching instructions (jumps & calls)
- Determine non-code probability
  - Invalid instructions (e.g. HLT)
  - Uncommon instructions (e.g. LAHF)
  - Invalid memory access (e.g. use of un-initialized registers) -> DANGEROUS
- Emulate execution
  - Assembly level "Stateful Inspection"
  - Keep track of CPU registers & stack
  - Identify code logic (**Meta Instructions**)
- Heuristic decision function
  - Evaluate the confidence level and decide if input is malicious or not

©2009 COSEINC. All rights reserved. CONFIDENTIAL



©2009 COSEINC. All rights reserved. CONFIDENTIAL



©2009 COSEINC. All rights reserved. CONFIDENTIAL

Worst Case Scenarios

- ASM Arch Identifier
  - An attacker sends a crafted packet with many different arch opcodes on the payload (trying to force multiple layers of inspection)
  - Even valid shellcodes maybe coded as multi-arch ones
    - Architecture Spanning Shellcode – Phrack Magazine
  - To avoid that, when we detect multiple architectures opcodes (more than 7 bytes each) we automaticly block the traffic and alert for that condition or (configuration option) we just inspect for the target platform
- Spider loops
  - An attacker may send a crafted packet to force as many as possible spiders to be created
  - To optimize that, we do return address lookup (searching for valid return address in windows dlls, binaries mappings, pool address for the .text, others)
  - Jmps to jmps receive higher scores – the suppression layers will learn and block
- Inspection suppression
  - Optimization in each layer to avoid go to high layers for an already-seen traffic

©2009 COSEINC. All rights reserved. CONFIDENTIAL

'Smart' Disassembly

- Plugin system, permitting the addition of architectures (x86 32 and 64 bits, power, sparc, pa-risc)
- Detect 'dangerous' instructions – avoid instruction mis-alignments:

```
0000   6A0F         push 0x7F                 0000   6A0F         push 0x7F
0002   59           pop ecx                   0002   59           pop ecx
0003   E8FFFFFFFF   call 0x7                  0003   E8FFFFFFFF   call 0x7
0008   C15E304C     rcr [esi+0x30],0x4C       0007   FFC1         inc ecx
000C   0E           push cs                   0009   5E           pop esi
000D   07           pop es                    000A   304C0E07     xor [esi+ecx+0x7],cl
000E   E2FA         loop 0xA                  000E   E2FA         loop 0xA
0010                                          0010
```

- By the way: This is also a 'trick', by Gera to GetEIP

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Gera's method

```
00417000                    shellcode:
00417000
00417000 E8 FF FF FF FF call    near ptr shellcode+4
00417005 C3             retn
00417005                ; END OF FUNCTION CHUNK FOR
00417006                ; --------------------------
00417006 58             pop     eax
```

Before call instruction

```
00417000 E8          shellcode db 0E8h
00417001 FF          db 0FFh
00417002 FF          db 0FFh
00417003 FF          db 0FFh
00417004             ; --------------
00417004 FF C3       inc     ebx
00417004             ; END OF FUNCTION
00417006 58          pop     eax
```

After call instruction EIP points here

EIP stored in EAX

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Call4 decoder

```
/*
 *  the %ecx register contains the size of assembly code (shellcode).
 *
 *  pushl    $0x01
 *           ^^
 *           size of assembly code (shellcode)
 *
 *  xorb    $0x02,(%eax)
 *           ^^
 *           number to xor
 */
    call    .+4
    ret

    popl    %eax
    pushl   $0x00  /* <-- size of assembly code (shellcode) */
    popl    %ecx
    addb    $0xe, %al

label1:
    xorb    $0x00,(%eax) /* <-- number to xor */
    incl    %eax
    loop    label1

/* assembly code (shellcode) goes here */
```

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

‘Smart’ Disassembly

- We can make use of the inherent functionality of the decoder stub to decode the payload of the network traffic.
- This is possible, but not needed in this case, since we already spoted a valid code, marking it for further examination (to avoid false-positives)
- The ‘smart’ disassembly is also layered, each layer avoiding deeper inspection, and doing that, keeping the performance in a high-level (still need to be better tested in real world networks – volunteers?)
  - Emulator inspection supression -> IMPORTANT -> Each layer will identify attackers forcing the cpu-consumption paths

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

‘Smart’ Disassembly

- Fpu instruction + fnstenv + pop = Dangerous sequence = Detection in a lower-layer of the Shikata ga nai decoder
- Even if not (some changes in the Shikata ga nai decoder can avoid it), the Smart disassembly will:
  - Detect the meta-construction: fpu instruction + fnstenv + pop and know where is the EIP
  - Will follow the clear ecx + loop to know what is the block condition
  - Will see the loop and will re-inspect the generated buffer after decoding

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Detecting the beginning of the code

- Since we don't know where in the input the shellcode begins we disassemble from every byte offset.
- Each offset is disassembled only once, the instruction is cached in a look-up table.
- Input bytes are processed by a 'Spider'.
- We drop a Spider on every offset.
- Multiple spiders scan the input in parallel.

Input Stream Of Bytes

0:	[illegible]	55	F4	4B	90	33	C0	EB	19	5E
10:	31	[illegible]	81	E9	89	FF	FF	FF	81	36
20:	80	BF	[illegible]	94	81	EE	FC	FF	FF	FF
30:	E2	F2	EB	[illegible]	E8	E2	FF	FF	FF	03

©2009 COSEINC. All rights reserved. CONFIDENTIAL

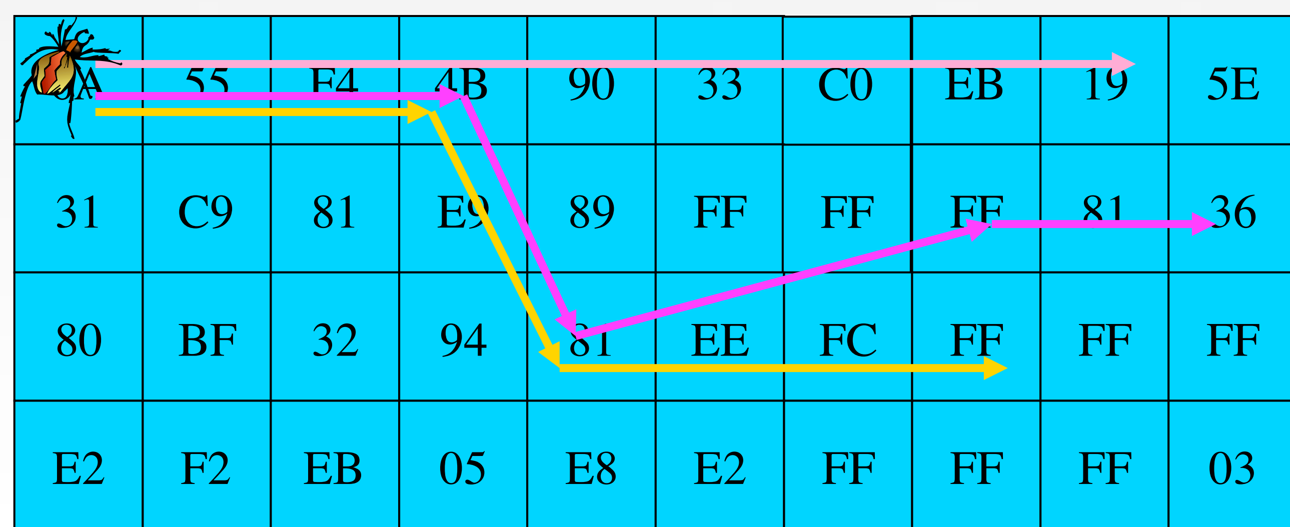
Spiders in action

- Since spiders follow branching instructions (calls & jumps) –
  A <u>single spider may travel in several paths</u> across the input buffer.

- Each of these paths is called a ***Flow***.

Input Stream Of Bytes

0:
10:
20:
30:



©2009 COSEINC. All rights reserved. CONFIDENTIAL

Meta Instructions

- Process each instruction in the context of previous instructions.
- Identify code logic common to malicious code:
  - Decryption Loop
  - EIP Calculation
  - PEB Access
  - SEH Access
- Also, target-OS aware
  - Interrupts
    - ‘INT 0x80’: Linux System Call
    - Invalid in Windows

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Detecting target-based decoders

- Detect jmps/calls to .text area looking for the loading address space (even in ASLR-enabled systems, those addresses are fixed in a range)
- Also, matching valid return addresses try to locate the start offset in the input buffer
- Detection of FPU instructions followed by fnstenv, CPUID instructions and others like the misalignment constructions in earlier slides
- VM detection code constructions:
  - sidt, sgdt and sldt and comparisons of returned values (in Linux IDT is at 0xc0ffffff (kernel-mode memory), in Windows (0x80ffffff)). In VMware (0xffXXXXXX) and in VirtualPC (0xe8XXXXXX). So, if the returned value by sidt is greater than 0xd0 it's in a VM, if its lower, its in the real OS). GDT is also in the same range in Windows and Linux. LDT is 0x0000 in the real OS
  - There is also VMWare specific instructions, like the following construction:

    mov EAX, 564D5868 -> VMXh

    xor EBX, EBX -> Of course if the attacker knows it's already zeroed, he may skip that (that's why is dangerous to assume the use of uninitialized registers as low-risk)

    mov ECX, 0A -> Guest-to-host communication

    mov EDX, 5658 -> VX

    in EAX, DX -> Check if VMWARE is active

    cmp EBX, 564D5868 -> The VMXh is a magic value which will be moved to EBX if VMWare do exist, otherwise it will be 0.

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Confidence indexing

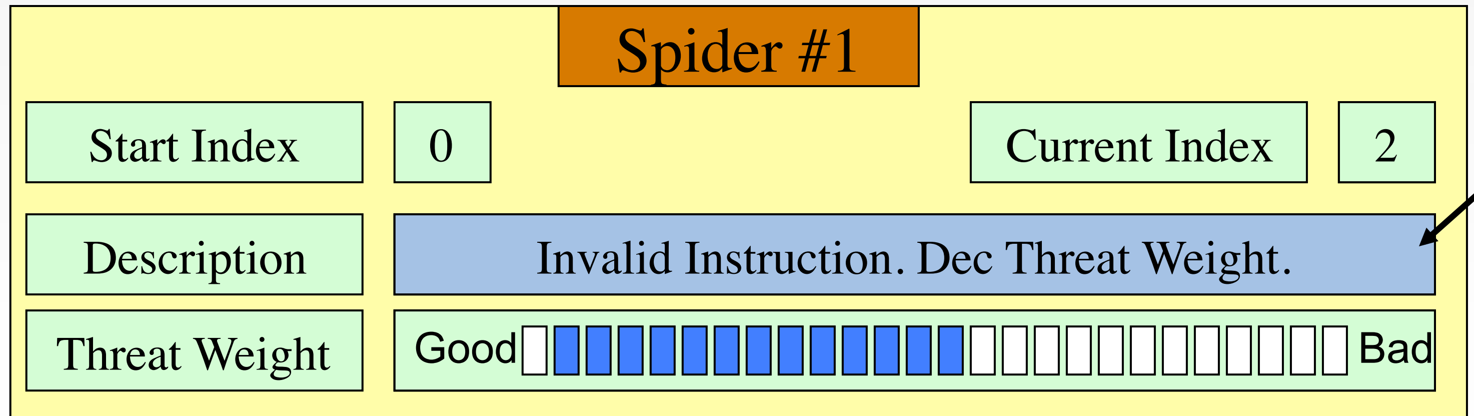
- Configured in a per-rule, per-protection way, extended to the disassembler
  - Per instruction
  - Per meta-construction
- If the ‘dumb’ disassembler detects a valid instruction number (configured by the user) it will add for example, 10% to the chances of this being an attack
  - **This value is proportional to the size of the payload itself (smaller payloads smaller the changes to have valid instructions) -> Tks to Julio Auto for the idea**
- If the ‘smart’ disassembler detects a dangerous construction forcing misaligment for example, it will add 70% to the chances of this being an attack (so the total now is 80%)
- Let’s assume a company who defined that, for the company to be considered an attack, we need to be 90% sure of that... It’s still not an attack
- A fragmented packet may receive 5%... It’s still not an attack

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Innocent portion of a packet been analyzed

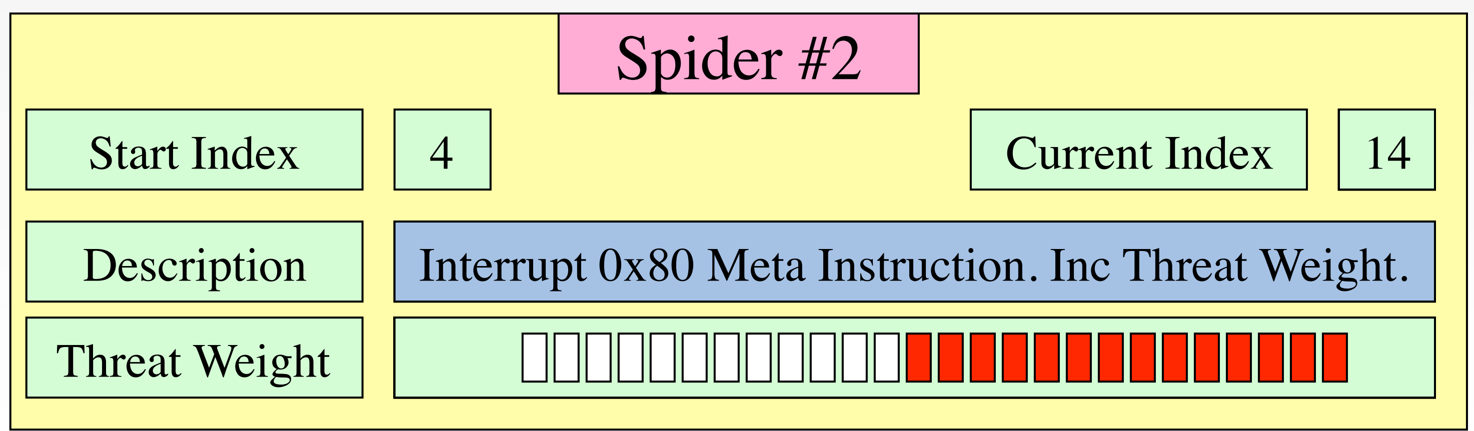
0:	PUSH 55		HLT	4B	90	33	C0	EB	19	5E	55
10:	31	C9	81	E9	89	FF	FF	FF	81	36	C9
20:	80	BF	32	94	81	EE	FC	FF	FF	FF	BF
30:	E2	F2	EB	05	E8	E2	FF	FF	FF	03	F2



©2009 COSEINC. All rights reserved. CONFIDENTIAL

Malicious portion of a packet been analyzed

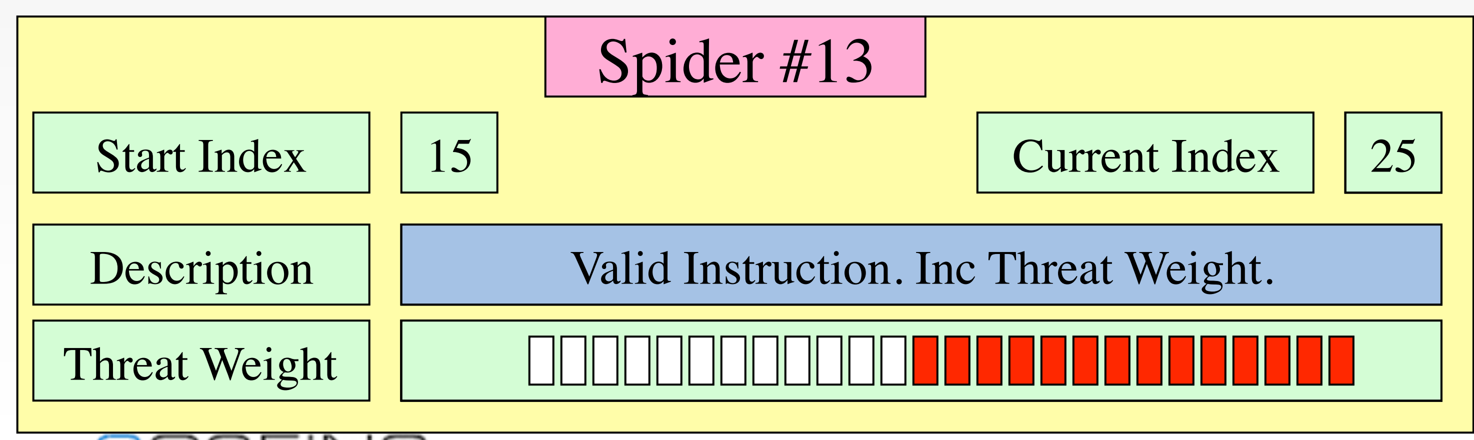
0:	6A	F4	6E	PUSH ECX	XOR EAX, EAX		PUSH EBX	PUSH 2		MOV ECX	
10:	ADD AL, 0x66		INT 0x80		89	FF	FF	FF	81	36	C9
20:	80	BF	32	94	81	EE	FC	FF	FF	FF	BF
30:	E2	F2	EB	05	E8	E2	FF	FF	FF	03	F2



©2009 COSEINC. All rights reserved. CONFIDENTIAL

Decoder analyzed

0:	6F	6F	6F	6F	6F	6F	6F	42	42	42	42
10:	8B	89	E8	77	JMP +26		POP EBX	PUSH EBX	68	PUSH	AA
20:	0x7801AAAD		POP EAX	CALL EAX		31	C9	B1	11	58	E2
30:	FD	31	C0	48	C3	E8	E6	CALL -26	FF	FF	73



©2009 COSEINC. All rights reserved. CONFIDENTIAL

Vulnerability Research Center

- Create a distributed analysing machines for each architecture used in the company seens interesting to really debug the payload execution
  - Can be offered as a service, avoiding false-positivies and new exploiting mechanisms

- Easy to do further automated investigation to validate shellcodes and detecting new wide-spreeding malwares, encoding techniques and false positives
  - No performance penalti, since the smart disassembly will guarantee that just a small portion of the traffic will trigger this inspection level
  - Emulator inspection supression -> IMPORTANT! -> REMEMBER that in the previous slides? It's because otherwise an attacker can just generate code that will force a lot of traffic to go to the vulnerability research center

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Implementation: Cell Architecture

- Powerful hybrid multi-core technology
- 128 registers files of 128 bits each:
  - Since each SPU register can hold multiple fixed (or floating) point values of different sizes, GDB offers to us a data structure that can be accessed with different formats:

```
(gdb) ptype $r70
type = union __gdb_builtin_type_vec128 {
int128_t uint128;
float v4_float[4];
int32_t v4_int32[4];
int16_t v8_int16[8];
int8_t v16_int8[16];
}
```

  - So, specifying the field in the data structure, we can update it:

```
(gdb) p $r70.uint128
$1 = 0x00018ff000018ff000018ff000018ff0
(gdb) set $r70.v4_int32[2]=0xdeadbeef
(gdb) p $r70.uint128
$2 = 0x00018ff000018ff0deadbeef00018ff0
```

- 256KB Local Storage -> Mainly used for log suppression and caching (avoiding calls to the PPU)
- Threads managed by the PPU, which handles the traffic and chooses the SPU to process it (the spiders) -> Resident threads to avoid the thread creation overhead
- **Thread abstraction – Easy to port (here I'm using a x86 VM instead of a Cell simulator for instance)**

©2009 COSEINC. All rights reserved. CONFIDENTIAL

Future

- I can't foresee the future!
- My guess is this kind of technology will be improved, mainly after some disasters:
  - Conficker worm was really successful even exploiting an already patched vulnerability (for which most vendors had signatures too)
  - This worm used a piece of payload taken from a public tool (Metasploit unreliable remote way to differentiate between XP SP1 and SP2)
- We all are aware that this kind of protection will not prevent everything, but will give a good level of protection against well-known payload strategies
- Still missing performance numbers, since all the Cell-related stuff are being developed in a Playstation3 (I don't have high-performance network cards for testing)
- Need to define the confidence level defaults

©2009 COSEINC. All rights reserved. CONFIDENTIAL

End! Really !?

Rodrigo Rubira Branco (BSDaemon)
Senior Vulnerability Researcher
Vulnerability Research Labs (VRL) – COSEINC
rodrigo_branco *noSPAM* research.coseinc.com

COSEINC

©2009 COSEINC. All rights reserved. CONFIDENTIAL