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
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
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 mentioned 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.
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
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
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??
mfmsr r0
abwiarch r3,r0,16+1,32-1,31
rlwinm r0,r0,0,17,15
SYNC
mtmsr r0
blr

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.

Clear Interrupt Flag - Clearing the IF flag causes the processor to ignore maskable external interrupts
PART I
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
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 (tolower, toupper, isascii...)
Polymorphism – How it works?

Generally, divided in two pieces:
- The decoding loop
- The GetEIP trick

```
call decoder
shellcode
decoder
jmp shellcode
```
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 inversion
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 */
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)
/*
 * 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:
  xorl $0x00,(%eax) /* <-- number to xor */
  incl %eax
  loop label1

/* assembly code (shellcode) goes here */
Target-based decoders

- Keyed encoders have the keying information available or deducted 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)
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
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 */
Context-keyed decoders

- I)ruid’s idea: [http://www.uninformed.org/?v=9&a=3&t=txt](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...
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
Syscall Proxying

- When a process needs any resource, it must perform a system call in order to ask the operating system for the needed resource.

- Syscall interfaces 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.
System Call – How does it work?

User Mode

Kernel Mode

Process

System Call Handler
System Call – Reading a File...

Reader

open

read

close

Linux Kernel

Disk
System Call – strace output

# strace cat /etc/passwd

exeave("/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
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
The idea is to split the default syscall functionality in two steps:

- A client stub
  Receives the requests for resources from the programs
  Prepare 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
System Call Proxying – Reading a File...

Reader → Client Stub → Syscall Server → Linux Kernel

- h_open → sys_send → open
- h_read → sys_send → read
- h_close → sys_send → close

Disk
System Call Proxying – Packing

Client Request
- filename (pointer)
- flags
- mode

DATA

Server Stack
- The request
- local vars
- saved ESP
- ESP
- EBP
- 0xbffffff

©2009 COSEINC. All rights reserved. CONFIDENTIAL
A small demo

Host Machine
192.168.10.10

Attacker Machine
192.168.20.128

Firewall

Eth0 – VMNet5
.254

BT Target Machine
192.168.10.128

Win2K Target Machine
192.168.10.101

192.168.20.128

192.168.10.128

192.168.10.101

Eth2 – VMNet4
.254

Eth1 – VMNet1
.254
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
PART II
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
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
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.
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)

- SC Morphism help (no new releases since 2004!!)

```
-A <nopsize in bytes> --> Gen. random alphanumeric junks only (only print the junks 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 Z, 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)
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?
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
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?
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
Architecture Overview – Splitting the problem in layers

Automatic Debuggers
- Smart Disassembler
  - Well-known Return Address (loading and library addresses)
  - Well-known Dangerous Sequences
  - Dumb Disassembly
    - Target-aware information
  - Streaming
    - Second inspection

Vuln. Research Center

X86/pa-risc/sparc

Still need to be implemented

Aceleration Layer (Vuln. Research Center)
A real traffic...

Vuln. Research Center

X86/pa-risc/sparc

X86/pa-risc/sparc

X86/pa-risc/sparc

x86  pa-risc  sparc

Still need to be implemented

Aceleration Layer (Vuln. Research Center)

Well known threats or bad packets

New techniques, well-known false positives, automatic debuggers

Attack detected

Follow the instructions and create state meta-information

Attack detected

Looking for valid Linux return addresses (stack, heap, text library)

Attack detected

Looking for cpuid, call $+4, jmp/call/pop, fnstenv

Attack detected

Counting valid x86 instructions

Attack detected

Target run linux on intel, inspect x86

The actual payload: \x90\x90\x90\x90...

Packet Reassembly/Protocol Inspection

Packet 1

Packet 2

Packet 3

©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 automatically 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
‘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:

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>6AOF</td>
<td>push 0x7F</td>
<td>0000</td>
<td>6AOF</td>
<td>push 0x7F</td>
</tr>
<tr>
<td>0002</td>
<td>59</td>
<td>pop ecx</td>
<td>0002</td>
<td>59</td>
<td>pop ecx</td>
</tr>
<tr>
<td>0003</td>
<td>E8FFFFFFF</td>
<td>call 0x7</td>
<td>0003</td>
<td>E8FFFFFFF</td>
<td>call 0x7</td>
</tr>
<tr>
<td>0008</td>
<td>C15E304C</td>
<td>rcr [esi+0x30],0x4C</td>
<td>0007</td>
<td>FFC1</td>
<td>inc ecx</td>
</tr>
<tr>
<td>000C</td>
<td>0E</td>
<td>push cs</td>
<td>0009</td>
<td>5E</td>
<td>pop esi</td>
</tr>
<tr>
<td>000D</td>
<td>07</td>
<td>pop es</td>
<td>000A</td>
<td>304C0E07</td>
<td>xor [esi+ecx+0x7],cl</td>
</tr>
<tr>
<td>000E</td>
<td>E2FA</td>
<td>loop 0xA</td>
<td>0010</td>
<td>E2FA</td>
<td>loop 0xA</td>
</tr>
</tbody>
</table>

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

Before call instruction

After call instruction

EIP points here

EIP stored in EAX
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 */
‘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 spotted 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
‘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
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.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>55</td>
<td>F4</td>
<td>4B</td>
<td>90</td>
<td>33</td>
<td>C0</td>
<td>EB</td>
<td>19</td>
<td>5E</td>
</tr>
<tr>
<td>10:</td>
<td>31</td>
<td>A</td>
<td>81</td>
<td>E9</td>
<td>89</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>81</td>
</tr>
<tr>
<td>20:</td>
<td>80</td>
<td>BF</td>
<td>94</td>
<td>81</td>
<td>EE</td>
<td>FC</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>30:</td>
<td>E2</td>
<td>F2</td>
<td>EB</td>
<td>E8</td>
<td>E2</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>03</td>
</tr>
</tbody>
</table>

Input Stream Of Bytes
### Spiders in action

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

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

```plaintext
<table>
<thead>
<tr>
<th>0:</th>
<th>A</th>
<th>55</th>
<th>F4</th>
<th>4B</th>
<th>90</th>
<th>33</th>
<th>C0</th>
<th>EB</th>
<th>19</th>
<th>5E</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:</td>
<td>31</td>
<td>C9</td>
<td>81</td>
<td>E9</td>
<td>89</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>81</td>
<td>36</td>
</tr>
<tr>
<td>20:</td>
<td>80</td>
<td>BF</td>
<td>32</td>
<td>94</td>
<td>81</td>
<td>EE</td>
<td>FC</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>30:</td>
<td>E2</td>
<td>F2</td>
<td>EB</td>
<td>05</td>
<td>E8</td>
<td>E2</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>03</td>
</tr>
</tbody>
</table>
```

**Input Stream Of Bytes**
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
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 misalignment 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
Innocent portion of a packet been analyzed

<table>
<thead>
<tr>
<th>Start Index</th>
<th>Current Index</th>
<th>Description</th>
<th>Threat Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>Invalid Instruction. Dec Threat Weight.</td>
<td>Good</td>
</tr>
</tbody>
</table>

Spider #1

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>6A F4 6E</td>
<td>PUSH ECX XOR EAX, EAX PUSH EBX</td>
</tr>
<tr>
<td>10:</td>
<td>ADD AL, 0x66</td>
<td>CINT 0x80 INT 0x80</td>
</tr>
<tr>
<td>20:</td>
<td>80 BF 32 94</td>
<td>81 EE FC FF FF FF FF BF</td>
</tr>
<tr>
<td>30:</td>
<td>E2 F2 EB 05</td>
<td>E8 E2 FF FF FF 03 F2</td>
</tr>
</tbody>
</table>

### Spider #2

<table>
<thead>
<tr>
<th>Start Index</th>
<th>Current Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt 0x80 Meta Instruction. Inc Threat Weight.</td>
</tr>
</tbody>
</table>

©2009 COSEINC. All rights reserved. CONFIDENTIAL
Spider #13

Start Index: 15
Current Index: 25
Description: Ready
Valid Instruction. Inc Threat Weight.
Threat Weight: [High]

Decoder analyzed

<table>
<thead>
<tr>
<th></th>
<th>6F</th>
<th>6F</th>
<th>6F</th>
<th>6F</th>
<th>6F</th>
<th>6F</th>
<th>6F</th>
<th>42</th>
<th>42</th>
<th>42</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:</td>
<td>8B</td>
<td>89</td>
<td>E8</td>
<td>77</td>
<td>EJMP +26</td>
<td>POP EAX</td>
<td>PUSH EBX</td>
<td>68</td>
<td>PUSH</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>20:</td>
<td>0x7801AAAD</td>
<td>POP EAX</td>
<td>CALL EAX</td>
<td>31</td>
<td>C9</td>
<td>B1</td>
<td>11</td>
<td>58</td>
<td>E2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:</td>
<td>FD</td>
<td>31</td>
<td>C0</td>
<td>48</td>
<td>C3</td>
<td>E8</td>
<td>E6</td>
<td>CALL -26</td>
<td>FF</td>
<td>FF</td>
<td>73</td>
</tr>
</tbody>
</table>
Create a distributed analysing machines for each architecture used in the company seems interesting to really debug the payload execution
  – Can be offered as a service, avoiding false-positives 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 penalty, since the smart disassembly will guarantee that just a small portion of the traffic will trigger this inspection level
  – Emulator inspection suppression -> 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
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)
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
End! Really !?

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