Shellcode analysis using dynamic binary instrumentation

Daniel Radu & Bruce Dang
Microsoft Corporation
Agenda

- Introduction
- Demo
- Dynamic Binary Instrumentation
- Implementation & Features
- Future Work
- Conclusion
Motivation

- Increased # of targeted attacks using 0day exploits:
  - CVE-2010-0249 (Aurora)
  - CVE-2010-2883 (U.S. gov.)
  - CVE-2010-3962 (U.S. gov.)
  - CVE-2011-0609 (SWF embedded in .XLS)
  - CVE-2011-0611 (SWF embedded in .DOC)

* All these vulnerabilities are patched
Security response
- Identify attack
- Analyze infection vector
- Cleanup

Time is critical
- React fast
Introduction::React fast

- Shorten analysis step:
  - Develop a tool that can automatically detect and analyze the shellcode

- Main Tool Features:
  - Vulnerability Agnostic
  - Fast & Accurate
  - Detailed Output
  - Configurable
Introduction::The tool

SHAN : The shellcode analyzer

• Features:
  – dynamic analysis - plugin for PIN framework
  – All features discussed in prev. slide

• Handles
  – Classic shellcode (executable buffers)
  – ROP shellcode (gadgets)
Analysis - CVE-2010-2883
DEMO::Shellcode Stage 1

Rop Shellcode detected: first gadget @ 0x4a82a714 in module icucnv36.d11

<table>
<thead>
<tr>
<th>Stack Ptr.</th>
<th>Stack Value</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0c0c0c0c0c</td>
<td>0x4a8063a5</td>
<td>icucnv36.d11!uenum_count_3_6 + 0x001D</td>
</tr>
<tr>
<td>0x0c0c0c10</td>
<td>0x4a8a0000</td>
<td>icucnv36.d11!0x4A8A0000</td>
</tr>
<tr>
<td>0x0c0c0c14</td>
<td>0x4a802196</td>
<td>icucnv36.d11!ubidi_getClassCallback_3_6 + 0x0022</td>
</tr>
<tr>
<td>0x0c0c0c18</td>
<td>0x4a881f90</td>
<td>icucnv36.d11!ubidi_getDirection_3_6 + 0x0018</td>
</tr>
<tr>
<td>0x0c0c0c1c</td>
<td>0x4a84903c</td>
<td>icucnv36.d11!CreateFileA</td>
</tr>
<tr>
<td>0x0c0c0c20</td>
<td>0x4a80b692</td>
<td>icucnv36.d11!u_errorName_3_6 + 0x00DF</td>
</tr>
<tr>
<td>0x0c0c0c24</td>
<td>0x4a801064</td>
<td>icucnv36.d11!0x4A801064</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Stack Pointer</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4a82a714</td>
<td>pop esp</td>
</tr>
<tr>
<td>0x4a82a715</td>
<td>ret</td>
</tr>
<tr>
<td>0x4a8063a5</td>
<td>pop ecx</td>
</tr>
<tr>
<td>0x4a8063a6</td>
<td>ret</td>
</tr>
<tr>
<td>0x4a802196</td>
<td>mov dword ptr [ecx], eax</td>
</tr>
<tr>
<td>0x4a802198</td>
<td>ret</td>
</tr>
<tr>
<td>0x4a801f90</td>
<td>pop eax</td>
</tr>
<tr>
<td>0x4a801f91</td>
<td>ret</td>
</tr>
<tr>
<td>0x4a80b692</td>
<td>jmp dword ptr [eax] (kernel32.d11!CreateFileA)</td>
</tr>
<tr>
<td>0x4a8063a5</td>
<td>pop ecx</td>
</tr>
<tr>
<td>0x4a8063a6</td>
<td>ret</td>
</tr>
</tbody>
</table>
DEMO::Shellcode Stage 2

Classic shellcode detected starting 0x01530000 referenced from MSVCR80.dll!0x781451aa

```
0x01530000 2b c9  sub ecx, ecx
0x01530002 b1 33  mov cl, 0x33
0x01530004 b8 9f 63 ab b0  mov eax, 0xb0ab639f
0x01530009 db cc  fcmovne st0, st4
0x0153000b d9 74 24 f4  fnstenv ptr [esp-0xc]
0x0153000f 5a  pop edx
0x01530010 31 42 0e  xor dword ptr [edx+0xe], eax
0x01530013 03 42 0e  add eax, dword ptr [edx+0xe]
0x01530016 83 ea fc  sub edx, 0xffffffff
0x01530019 e2 f5  loop 0x1530010
...
0x0153001b fc  cld
0x0153001c e8 89 00 00 00  call 0x15300aa
0x0153001e 5d  pop ebp
0x01530020 6a 01  push 0x1
0x01530022 8d 85 b9 00 00 00  lea eax, ptr [ebp+0xb9]
0x01530026 50  push eax
0x01530028 66 31 8b 6f 87  push 0x876f8b31
0x0153002c ff d5  call ebp
0x0153002e 66  x pushad
0x01530030 89 e5  mov ebp, esp
0x01530032 89 e5  x xor edx, edx
0x01530034 66 8b 52 30  mov edx, dword ptr fs:[edx+0x30]
0x01530038 8b 52 0c  mov edx, dword ptr [edx+0xc]
0x0153003c 8b 52 14  mov edx, dword ptr [edx+0x14]
0x0153003f 8b 72 28  mov esi, dword ptr [edx+0x28]
0x01530042 0f b7 4a 26  movzx ecx, word ptr [edx+0x26]
```
Dynamic Binary Instrumentation (DBI)

- Developed by Intel
- Instrument code just before it runs (JIT)
  - No need to modify/recompile a binary before analyzing it
  - Discovers code at runtime
  - Can handle dynamic/self-modifying code
- Rich set of API’s
  - Powerful
  - Easy to use
Implementation::PIN
Implementation::Details

- SHAN Detection Modules
  - Classic shellcode
  - ROP Shellcode
Skeleton detection algorithm

for_each ins in Program:
    If IndirectBranchOrCall(ins):
        If BranchDest(ins) is OutsideLoadedModules():
            ShellCodeDetected()

Works well in practice

- Heuristic algorithms implemented to remove FPs on packed binaries
Weak points

- Packed/Protected binaries
- JIT Code
  - Java
  - Flash

Solutions

- Use heuristics
  - Detect packer/protector code
  - Detect JIT code
Two detection algorithms implemented

- “Shadow Stack” algorithm
- In-house algorithm
  - Based on Basic Block Analysis
Implementation::ROP Shellcode

General idea:
- Detect returns where return address wasn’t pushed on the stack by a previous call

Skeleton Shadow Stack Alg:

```python
for_each ins in Program:
    if Is_Call(ins):
        ShadowStack.push(return_addr(ins))
    if Is_Ret(ins):
        if return_addr(ins) == ShadowStack.top():
            ShadowStack.pop()
        else:
            ShellcodeDetected()
```
Weak points
- Packer/Protector code
- Exceptions

Solutions
- Use heuristics to detect protector code
- Pop from stack until values match
Implementation:: Shadow Stack

Advantages:
- Easy to implement
- Relatively Small # of FPs, in general
- Fast enough

Disadvantages
- FPs on packed binaries, in particular
- Fails to detect ROP shellcode without Ret
Pre-processing step

```cpp
for_each image in process:
    for_each bb in image:
        Bblist.push_back(BBInfo(bb))
```

Detection step

```cpp
for_each ins in Program:
    if IsIndirectBranchOrCall(ins)
        if BranchDest(ins) is InsideLoadedModules():
            if BranchDest(ins) not in Bblist.Instructions():
                ShellcodeDetected()
```
Advantages
- Detects ROP gadgets without ret
- Small number of FPs

Disadvantages
- Slower than “shadow stack” approach
SHAN Features

- Dumps executed gadgets (ROP shellcode)
- Allows the researcher to debug the program once the shellcode (ROP or classic) has been reached
- Supports debugging symbols
- API logging
Limitations

- **Pin:**
  - No kernel mode exploits

- **SHAN:**
  - FP’s for very complex packers/protectors
  - Worst case scenarios may take hours to analyze
  - Not fully automated (if exploit requires interaction)
Future Work

- Increase speed
- Improve detection heuristics
- Improve packer heuristics
- Implement new detection algorithms
The end

QUESTIONS?
Contact

Daniel Radu: dradu@microsoft.com
Bruce Dang: bda@microsoft.com

Beta testers, please email us.