Code Integration-Based Vulnerability Auditing

Black Hat USA 2006

William Kimball
wk@layereddefense.com
Outline

• What is the problem?
• Related Tools and Techniques
• Introduction to L.E.V.I.
  – Overall Design
  – Implementation
• Case Scenario
• Questions
What is the problem?
The tools used to automate searching for vulnerabilities in the Windows Environment do not search every possible execution path.
Related Tools

• APIS32, APISPY32, API Monitor
  - Import hooking only
  - No support for VAR_ARGS or VAR_LIST types

• BugScam – IDC scripts for IDA
  - Incomplete signature Database
  - No Runtime Monitoring
Fuzz-testing
What did you input!?

- **Advantages**
  - Results are fast with fewer false positives than manual auditing.

- **Disadvantages**
  - Problem is exponential
  - Incomplete code coverage
Introducing L.E.V.I.
(Low-Evel Vulnerability Interceptor)

- Command-line tool for the Windows NT Family which monitors imported and inline code constructs.
- Used for *Code Profiling* during the vulnerability discovery process.
L.E.V.I. Design

• Static Analysis
  – Imported Functions
    • Extensible Import Database
  – Inlined code constructs
    • Optional Buffer Size Check

• Runtime Analysis
  – Import Hooking
    • Supports VAR_ARGS and VAR_LIST types
  – Inline Hooking
    • Option to use EPO or Code Integration

• Supports Multi-byte and Unicode Char Sets
Static Analysis

ServerX.exe

ServerX.asm

ImportsDB.dat

ServerX_IMPORTS.dat

ServerX_INLINES.dat

disasm.dll
Runtime Analysis

ServerX → S

L
E
V
I

InterceptImports.dll
InterceptInlinesEPO.dll
InterceptInlinesCI.dll

S.exe
S_bk.exe (backup)
S_IMPORTS.dat
S_INLINES.dat
S_RUNTIME_IMPORTS.dat
S_RUNTIME_INLINES.dat
S.RUNTIME_IMPORTS.dat
S.RUNTIME_INLINES.dat
Implementation

- Import Hooking Technique
  - IAT, Merged Sections
- Inline Hooking Technique
  - Entry-Point Obscuring
    - "Move"-Point Obscuring
  - Code Integration
Pattern Matching (1)
What do most insecure functions have in common?

<table>
<thead>
<tr>
<th></th>
<th>strcpy()</th>
<th>sprintf()</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401501:</td>
<td>mov dl, byte ptr[edx]</td>
<td>00401018:</td>
</tr>
<tr>
<td></td>
<td>add ecx, 01h</td>
<td>mov cl, byte[eax]</td>
</tr>
<tr>
<td></td>
<td>test dl, dl</td>
<td>mov byte[edx+eax], cl</td>
</tr>
<tr>
<td></td>
<td>je 00401570</td>
<td>inc eax</td>
</tr>
<tr>
<td></td>
<td>mov byte ptr[edi], dl</td>
<td>test cl, cl</td>
</tr>
<tr>
<td></td>
<td>add edi, 01h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>test dword[ecx], 03h</td>
<td>jne 00401018</td>
</tr>
<tr>
<td></td>
<td>jne 00401501</td>
<td></td>
</tr>
</tbody>
</table>

Black Hat Briefings
Pattern Matching (2)

Must include non-library code!

```c
for(i = 0; src[i] != '\0'; i++)
    dst[i] = src[i];
```

00401033:

```asm
mov al, byte[ebx]
mov byte[esi+ebx], al
inc ebx
test al, al
jne 00401033
```

**Rule Set**

1. *Small cycle*
2. Conditional Branch (ZF)
3. Move from and to memory
4. etc…
Move-Point Obscuring (1)

Server X

- jmp logger()
- mov byte[eax], cl
- cmp cl, 00
- je 00404063
- mov cl, byte[edx+01]
- add edx, 002
- mov byte[eax+01], cl
- add eax, 002
- cmp cl, 00
- jne 00404049

Before execution

LEVI

- Logger() {
  1. Log buffers, addresses, etc.
  2. Fix MPO with original code
  3. Obscure local branches with PostObscure()}

Black Hat Briefings
Move-Point Obscuring (2)

**ServerX**

```assembly
mov cl, byte[edx]
mov byte[eax], cl
cmp cl, 00
je 00404063
mov cl, byte[edx+01]
add edx, 002
mov byte[eax+01], cl
add eax, 002
cmp cl, 00
jne 00404049
jmp PostObscure()
```

**LEVI**

```plaintext
PostObscure() {
1. Fix local branches with original code
2. MPO
}
```

*After the original code executes.*
Problem with MPO

If the size of the EPO jmp is greater than the instruction being overwritten then the next instruction will be modified and any jmp to it will cause unpredictable execution.

1. EPO modified cmp instruction.

:0043A313 EB09
:0043A315 8B45F8
:0043A318 83C001
:0043A31B E94040
:0043A31E 4040F812

2. Subsequent jmps to address 0043A31E cause unpredictable execution.

jmp 0043A31E
mov eax, dword[ebp-08]
add eax, 001
mov dword[ebp-08], eax
cmp dword[ebp-08], 012
Definition

A virus infection technique used to merge two separate code sections without the need to “obscure” or recompile the existing code sections.

First used in Z0mbie’s Mistfall Engine.
Code Integration

1. Disassemble
2. Prepare PE file
3. Merge code
4. Recalculate addresses
5. If address sizes increase then repeat until merge settles

```
test dl, dl
je 00408452
inc ebx
mov dl, byte[eax]
inc eax
mov byte[ebx], dl
test dl, dl
jne 0040843D
```
Case Scenario / Questions

```cmd
C:\LEVI> levi
usage: levi [options] PE_file
   -s Creates a static audit of the imports (defined in ImportsDB.dat) and inlines in the PE file. This is the default option.
   -d Creates a backup of the PE file appended with _bk. Modifies original PE file with intercept routines for imports and inlines. This option creates a static audit of the PE file.
   -b Intercepts code constructs which check the buffer size. By default, code constructs which check the buffer are not intercepted.
   -i Use code integration (CI) as the interception technique. By default, Entry-point obscuring (EPO) is used.
   -c Restores original PE file with its backup file.
      Deletes PE files _IMPORTS.dat, _RUNTIME_IMPORTS.dat, _INLINES.dat, _RUNTIME_INLINES.dat, ImportsDB.dat, ASM file, and backup file. If cleaning directory, other than LEVI’s current directory, then InterceptImports.dll, InterceptInlinesEPO.dll and InterceptInlinesCI.dll are deleted.
   -r Restores original PE file with its backup file.

C:\LEVI>_
```