Integrity Verification of User Space Code

By

Andrew White, Bradley Schatz and Ernest Foo

Presented At

The Digital Forensic Research Conference

DFRWS 2013 USA  Monterey, CA (Aug 4th - 7th)

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment. As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and challenges to help drive the direction of research and development.

http://dfrws.org
Integrity Validation of User Space Code

Andrew White, Ernest Foo, Bradley Schatz
Queensland University of Technology
Brisbane Australia

DFRWS

7th August 2013
Goal

- Reduce amount of memory requiring manual analysis
- Highlight any memory that is potentially suspicious
  - e.g. malware
- Achieved by filtering out known code
Process Memory

- Each process given its own view of memory
- User Space
  - Lower half of virtual memory
  - 0x00000000 - 0x80000000 (2GB) on 32 bit
  - Where process code and data is stored
- User space memory used by the process described by the VAD Tree
Some memory is code, some memory is data
Code must have executable permissions
  Otherwise it will not run
Memory permissions can be used to distinguish code and data
  No Execute (NX) bit in Page Table Entry (PTE)
  VAD permissions do not matter
Code on Windows

- Portable Executable (PE)
  - Format used by Windows for programs and code
  - .exe, .dll, .drv etc
- Format same in memory and on disk
  - Layout is different
- Content between memory and disk not quite the same
  - Code requires updating to reflect environment
  - Relocations and imports
  - Changes not known till run time
Malware

- Common need to determine if malware is running on the system
- Numerous ways in which that malware could have been loaded
- Locating that malware can be complicated
Reducing memory requiring analysis

Andrew White

Integrity Validation of User Space Code
Example for explorer.exe on Win7
Every process on a Windows 7 system
Every process on a Windows 7 system
Proposed solution

- Build hashes of trusted code from on disk
  - e.g. a default Windows install
- Apply hashes to code in user space memory
  - Apply in a manner that takes into account imports, relocations etc.
- Remove code that passes validation from further analysis
- Reduce memory requiring analysis from whole memory image to only code that was not validated
Related Work

- **Malfind** [Ligh, 2009]
  - Uses VAD permissions to detect potentially injected code
  - Code capable of subverting detection exists [Keong, 2004]

- **System Virginity Verifier** [Rutkowska, 2005]
  - Compares contents of files on disk to contents of files in memory on a live system
  - Requires trusting contents of disk and memory simultaneously

- **Walters et al.** [2008]
  - Built hashes of code from on disk and applied to a memory image
  - Only able to if a page matches or not, not whether it should or should not
Building Hashes

- Parse PE files on disk
- Convert PE to virtual layout
- Normalise variable locations
  - relocations, imports, etc.
- Hash normalised page
- Output a hash, list of normalised locations and metadata for each page
- Similar to Walters et al. [2008] approach
### Sample Hashes

<table>
<thead>
<tr>
<th>Filename</th>
<th>Offset</th>
<th>Normalised Hash</th>
<th>Executable</th>
<th>To Normalise</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntdll.dll</td>
<td>0</td>
<td>721652da644c8b8be9c27909f76319ca1e2c6648</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ntdll.dll</td>
<td>32</td>
<td>0e04ac081fdd61f63a9efbf46154578da56d15cc</td>
<td>1</td>
<td>35d 4df d3a</td>
</tr>
<tr>
<td>ntdll.dll</td>
<td>45</td>
<td>d1d6e5357344dbb74957c0eec9c98cd703ab4222</td>
<td>1</td>
<td>0d2 141 190 1bd 1e7 233 24e 268 289 33a 34f 366 ... c7c c81 c88 c92 c97 caf cb9 cbe fa9 fb4 fde fe8 fed</td>
</tr>
<tr>
<td>ntdll.dll</td>
<td>5b</td>
<td>e6cc914ef3095a5a7e5f967a92a57c1c5779a806</td>
<td>1</td>
<td>fb5</td>
</tr>
</tbody>
</table>
Applying hashes

VAD Entry
0x1000000 - 0x11000000
EXECUTE_WRITECOPY
explorer.exe

Executable Allocation

Executable Page
5468697320
697320736f
6d65206669
6c657220
7465787420
6f6e206120

Normalize
5468697320
000000006f
6d65206669
6c657220
7400000000
6f6e206120

Offset
Locations

Apply Hash
Result

Hash Set
Hash

Filename
Applying hashes

- Apply hashing process to every executable page in the user space of every process
- Use metadata to locate correct hash before hashing
- Categorise results
  - Verified - page matched stored hash
  - Failed - page did not match stored hash
  - Unknown - no stored hash available
  - Unverifiable - known problem Windows behaviour
## Sample Output

<table>
<thead>
<tr>
<th>PID</th>
<th>Verified</th>
<th>Failed</th>
<th>Unverifiable</th>
<th>Unknown</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00004</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>System</td>
</tr>
<tr>
<td>00268</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>smss.exe</td>
</tr>
<tr>
<td>00372</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>csrss.exe</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00764</td>
<td>85</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>svchost.exe</td>
</tr>
<tr>
<td>01110000</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>ole32.dll executable alloc (Unverifiable)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02376</td>
<td>100</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>wmpnetwk.exe</td>
</tr>
<tr>
<td>003a0000</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>ole32.dll executable alloc (Unverifiable)</td>
</tr>
<tr>
<td>6cd00000</td>
<td>47</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>msmpge2enc.dll (Executable Data)</td>
</tr>
<tr>
<td>6ced0000</td>
<td>103</td>
<td>0</td>
<td>26</td>
<td>0</td>
<td>blackbox.dll (Unverifiable / Executable Data)</td>
</tr>
<tr>
<td>6de80000</td>
<td>165</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>drmv2clt.dll (Executable Data)</td>
</tr>
<tr>
<td>6dfa0000</td>
<td>57</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>wmdrmdev.dll (Executable Data)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Totals**

- **Allocations**: 2076
- **Pages**: 38788

**Unverifiable Pages Breakdown**

- **59 Executable Data**
- **14 Default Windows Behaviour**
Complications

- Windows exhibits default behaviour that cannot be verified
  - Executable pages that are not predictable
- Windows XP - data marked executable
  - Read-Only Shared Heap
  - Desktop Heaps
  - Win32k.sys Allocation
  - Winlogon.exe Allocations
- Windows 7 - obfuscated and irregular PE loading
  - blackbox.dll
  - shell32.dll in searchindexer.exe
- Transition pages
  - Page Table Entries do not have correct permission value
  - Need to query Page Frame Number database to retrieve
  - Complicates determining if a page is executable
- See paper for more details
Potential For Subversion

- Hashing process normalises part of input
  - Can these normalised locations be modified to create malware?
- Redirect program flow to external code source
  - External code source would be detected under current approach
- Replace normalised locations with malicious code
  - Code would be broken into 4 byte chunks and interleaved with normal execution
  - Difficult to create useful behaviour in this manner
- Return Orientated Programing (ROP)
  - Technique used to bypass lack of executable permissions
  - Code only exists as stack frames (data)
  - Currently only used for single function calls, not entire programs
Implement in two parts

Hashbuild
- Python script to parse a filesystem for PE files and build hashes

Hashtest
- Volatility plugin to apply the hashes to a memory image

Time taken to build hashes
- Clean XP install - 1.5 min
- Clean Win7 install - 3.5 min

Time taken to test hashes against an image
- XP 256MB image - 30s
- Win7 1GB image - 2min
Experimental Setup

- Tested against Windows XP SP3 and Windows 7 SP1
- Tested against malware and application dataset for each OS
- Images created with virtual machines
  - Each malware sample examined to ensure it executed correctly
## Malware Results - XP

<table>
<thead>
<tr>
<th>Malware</th>
<th>Executable Pages</th>
<th>Pages Failed</th>
<th>Pages Verified</th>
<th>Executable PE Data</th>
<th>Unverifiable Allocations</th>
<th>Unknown Allocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Sample</td>
<td>18701</td>
<td>0</td>
<td>100.00%</td>
<td>0</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>Cridex.B</td>
<td>18808</td>
<td>38</td>
<td>99.80%</td>
<td>0</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>Cridex.E</td>
<td>16964</td>
<td>28</td>
<td>99.83%</td>
<td>0</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>Dexter</td>
<td>37506</td>
<td>0</td>
<td>100.00%</td>
<td>0</td>
<td>25</td>
<td>2</td>
</tr>
<tr>
<td>NGRBot</td>
<td>19700</td>
<td>332</td>
<td>98.31%</td>
<td>0</td>
<td>25</td>
<td>44</td>
</tr>
<tr>
<td>Shylock</td>
<td>19583</td>
<td>30</td>
<td>99.85%</td>
<td>0</td>
<td>25</td>
<td>7</td>
</tr>
<tr>
<td>Spyeye</td>
<td>18564</td>
<td>107</td>
<td>99.42%</td>
<td>0</td>
<td>25</td>
<td>23</td>
</tr>
<tr>
<td>TDL3</td>
<td>19719</td>
<td>14</td>
<td>99.93%</td>
<td>0</td>
<td>25</td>
<td>49</td>
</tr>
<tr>
<td>TDL4</td>
<td>19911</td>
<td>14</td>
<td>99.93%</td>
<td>0</td>
<td>25</td>
<td>52</td>
</tr>
<tr>
<td>Vobfus</td>
<td>18322</td>
<td>0</td>
<td>100.00%</td>
<td>0</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>ZeroAccess</td>
<td>19644</td>
<td>0</td>
<td>100.00%</td>
<td>0</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>Program</td>
<td>Executable Pages</td>
<td>Pages Failed</td>
<td>Pages Verified</td>
<td>Executable PE Data</td>
<td>Unverifiable Allocations</td>
<td>Unknown Allocations</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------</td>
<td>--------------</td>
<td>----------------</td>
<td>---------------------</td>
<td>--------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>7zip</td>
<td>583</td>
<td>0</td>
<td>100.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Adobe Reader</td>
<td>3478</td>
<td>42</td>
<td>98.79%</td>
<td>0</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>Chrome</td>
<td>10867</td>
<td>9</td>
<td>99.92%</td>
<td>32</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Excel</td>
<td>2419</td>
<td>6</td>
<td>99.75%</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Firefox</td>
<td>4480</td>
<td>5</td>
<td>99.89%</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Google Talk</td>
<td>2951</td>
<td>0</td>
<td>100.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Internet Explorer</td>
<td>3794</td>
<td>27</td>
<td>99.29%</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>iTunes</td>
<td>5991</td>
<td>0</td>
<td>100.00%</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Notepad++</td>
<td>1651</td>
<td>0</td>
<td>100.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Outlook</td>
<td>6981</td>
<td>11</td>
<td>99.84%</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Pidgin</td>
<td>2720</td>
<td>0</td>
<td>100.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Powerpoint</td>
<td>3558</td>
<td>2023</td>
<td>43.14%</td>
<td>972</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Skype</td>
<td>7320</td>
<td>4216</td>
<td>42.40%</td>
<td>262</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Thunderbird</td>
<td>4247</td>
<td>5</td>
<td>99.88%</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>VLC</td>
<td>2073</td>
<td>0</td>
<td>100.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Winamp</td>
<td>3810</td>
<td>0</td>
<td>100.00%</td>
<td>0</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>Windows Media Player</td>
<td>3160</td>
<td>1</td>
<td>99.97%</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Winrar</td>
<td>1457</td>
<td>0</td>
<td>100.00%</td>
<td>11</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Wordpad</td>
<td>1545</td>
<td>0</td>
<td>100.00%</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Word</td>
<td>3403</td>
<td>9</td>
<td>99.74%</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Results

- Introduction of malware detected in all samples
  - Each introduced unknown allocations
  - Some changed existing pages
- Detected unknown code not found using Malfind
  - Executable pages in non-executable allocations
- Significant reduction in memory requiring analysis
  - $\sim$39,000 pages down to $\sim$75 on default Windows 7 system
Limitations

- Many applications introduced noise into this process
  - Some applications introduced unknown allocations
  - Packed application performance poor
- Does not take into account interpreted / JIT code
Conclusion

- Approach for validating the integrity of code in user space memory
  - Allows the reduction of memory requiring manual analysis
- Analysis of default Windows behaviour
- Implementation as a Volatility plugin
Future Work

- Other Windows versions
  - x64 / ARM
  - Vista and 8
- Kernel memory
  - Conversion of techniques for kernel memory
- Alternative hash building methods
  - Memory based or virtual machine based approaches
Questions

- Code
  - https://github.com/a-white/
- Questions?

