Integrity Validation of User Space Code

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

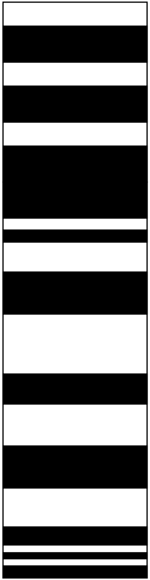
DFRWS

7th August 2013

Goal

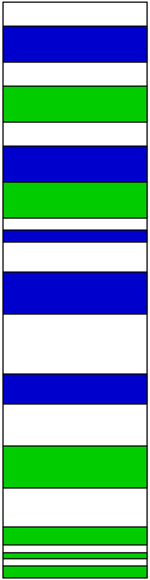
- Reduce amount of memory requiring manual analysis
- Highlight any memory that is potentially suspicious
  - e.g. malware
- Achieved by filtering out known code

Process Memory



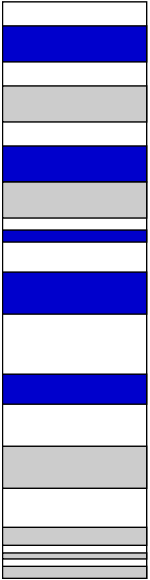
- Each process given its own view of memory
- User Space
  - Lower half of virtual memory
  - 0x00000000 - 0x80000000 (2GB) on 32 bit
  - Where process code and data is stored
- User space memory used by the process described by the VAD Tree

Code vs Data



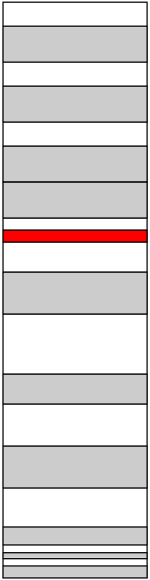
- Some memory is code, some memory is data
- Code must have executable permissions
  - Otherwise it will not run
- Memory permissions can be used to distinguish code and data
  - No Execute (NX) bit in Page Table Entry (PTE)
  - VAD permissions do not matter

Code on Windows



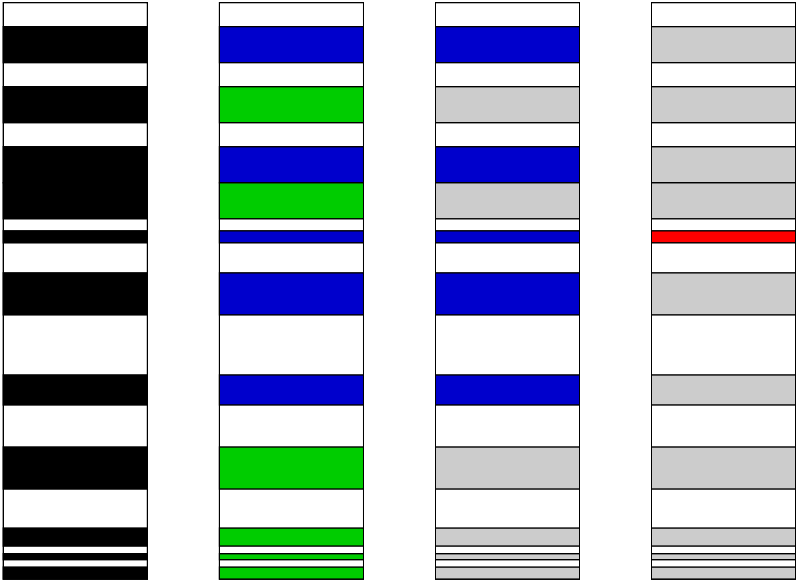
- Portable Executable (PE)
  - Format used by Windows for programs and code
  - .exe, .dll, .drv etc
- Format same in memory and on disk
  - Layout is different
- Content between memory and disk not quite the same
  - Code requires updating to reflect environment
  - Relocations and imports
  - Changes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

Malware

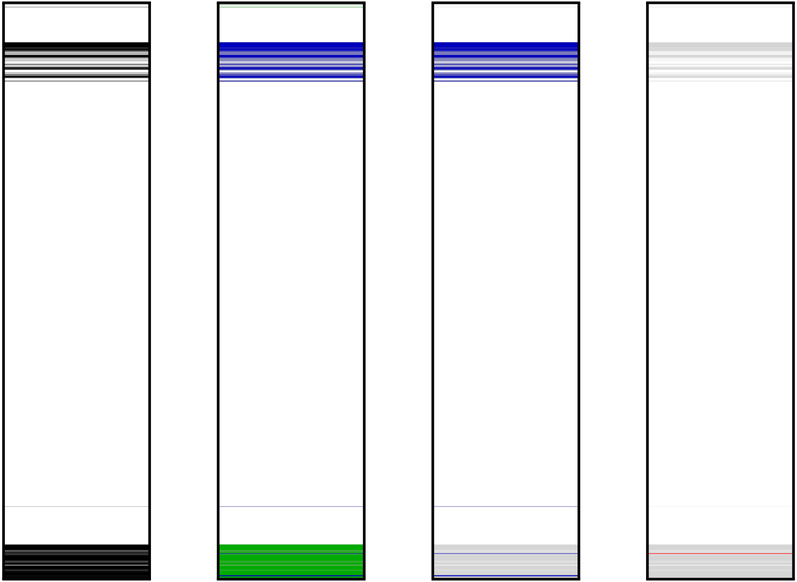


- Common need to determine if malware is running on the system
- Numerous ways in which that malware could have been loaded
- Locating that malware can be complicated

Reducing memory requiring analysis



Example for explorer.exe on Win7



Every process on a Windows 7 system

Every process on a Windows 7 system

Proposed solution

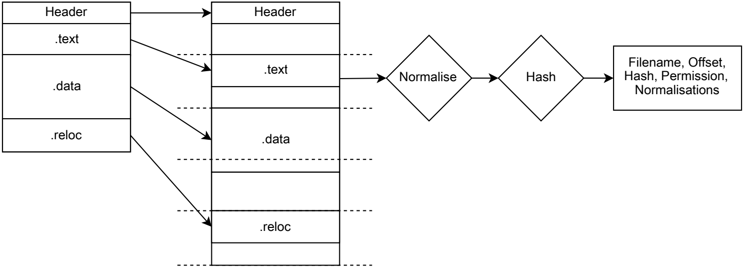
- Build hashes of trusted code from on disk
  - e.g. a default Windows install
- Apply hashes to code in user space memory
  - Apply in a manner that takes into account imports, relocations etc.
- Remove code that passes validation from further analysis
- Reduce memory requiring analysis from whole memory 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

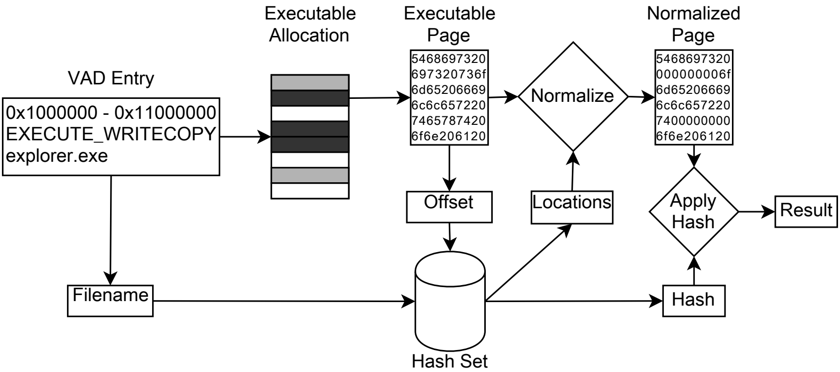
- Parse PE files on disk
- Convert PE to virtual layout
- Normalise variable locations
  - relocations, imports, etc.
- Hash normalised page
- Output a hash, list of normalised locations and metadata for each page
- Similar to Walters et al. [2008] approach



Sample Hashes

Filename	Offset	Normalised Hash	Executable	To Normalise
ntdll.dll	0	721652da644c8b8be9c27909f76319ca1e2c6648	0	
ntdll.dll	32	0e04ac081fdd61f63a9efbf46154578da56d15cc	1	35d 4df d3a
ntdll.dll	45	d1d6e5357344dbb74957c0eec9c98cd703ab4222	1	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
ntdll.dll	5b	e6cc914ef3095a5a7e5f967a92a57c1c5779a806	1	fb5

Applying hashes



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

```
PID           Verified  Failed  Unverifiable  Unknown   Name
------------  --------  ------  ------------  -------   ---------------------------------------------------
00004                1       0             0        0   System
00268                3       0             0        0   smss.exe
00372               17       0             0        0   csrss.exe
                                ...
00764               85       0             1        0   svchost.exe
    01110000         0       0             2        0   ole32.dll executable alloc (Unverifiable)
                                ...
02376              100       0             6        0   wmpnetwk.exe
    003a0000         0       0             2        0   ole32.dll executable alloc (Unverifiable)
    6cd00000        47       0            11        0   msmpge2enc.dll (Executable Data)
    6ced0000       103       0            26        0   blackbox.dll (Unverifiable / Executable Data)
    6de80000       165       0            11        0   drmv2clt.dll (Executable Data)
    6dfa0000        57       0            11        0   wmdrmdev.dll (Executable Data)
                                ...
Totals
    Allocations   2076       0             7        0
    Pages        38788       0            73        0

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

Implementation

- Implemented 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

Malware	Executable Pages	Pages Failed	Pages Verified	Executable PE Data	Unverifiable Allocations	Unknown Allocations
No Sample	18701	0	100.00%	0	25	0
Cridex.B	18808	38	99.80%	0	25	4
Cridex.E	16964	28	99.83%	0	25	3
Dexter	37506	0	100.00%	0	25	2
NGRBot	19700	332	98.31%	0	25	44
Shylock	19583	30	99.85%	0	25	7
Spyeye	18564	107	99.42%	0	25	23
TDL3	19719	14	99.93%	0	25	49
TDL4	19911	14	99.93%	0	25	52
Vobfus	18322	0	100.00%	0	25	3
ZeroAccess	19644	0	100.00%	0	25	10

Application Results - Win 7

Program	Executable Pages	Pages Failed	Pages Verified	Executable PE Data	Unverifiable Allocations	Unknown Allocations
7zip	583	0	100.00%	0	0	0
Adobe Reader	3478	42	98.79%	0	0	17
Chrome	10867	9	99.92%	32	0	25
Excel	2419	6	99.75%	0	0	2
Firefox	4480	5	99.89%	0	0	5
Google Talk	2951	0	100.00%	0	0	0
Internet Explorer	3794	27	99.29%	0	1	1
iTunes	5991	0	100.00%	11	0	0
Notepad++	1651	0	100.00%	0	0	0
Outlook	6981	11	99.84%	1	0	4
Pidgin	2720	0	100.00%	0	0	0
Powerpoint	3558	2023	43.14%	972	0	10
Skype	7320	4216	42.40%	262	0	2
Thunderbird	4247	5	99.88%	0	0	5
VLC	2073	0	100.00%	0	0	0
Winamp	3810	0	100.00%	0	0	18
Windows Media Player	3160	1	99.97%	0	0	1
Winrar	1457	0	100.00%	11	0	11
Wordpad	1545	0	100.00%	0	0	1
Word	3403	9	99.74%	0	0	2

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?

References I

Keong, T. C. (2004). Dynamic Forking of Win32 EXE. Available `http://www.security.org.sg/code/loadexe.html`. Last Accessed 14/06/12.

Ligh, M. H. (2009). Malfind Volatility Plugin. Available `http://mnin.blogspot.com.au/2009/01/malfind-volatility-plug-in.html`. Last Accessed 19/04/11.

Rutkowska, J. (2005). System virginity verifier. In *Proceedings of the Hack In The Box Security Conference*.

Walters, A., Matheny, B., and White, D. (2008). Using Hashing to Improve Volatile Memory Forensic Analysis. In *Proceedings of the American Acadaemy of Forensic Sciences Annual Meeting*.