On Correctness, Robustness and Coverage of Memory Analysis

Heng Yin
Assistant Professor

Department of Electrical Engineering and Computer Science, Syracuse University
What is memory analysis?

• Input:
  – A memory snapshot (or dump) of a running physical machine or virtual machine

• Analysis:
  – Scan data structure signatures
  – Traverse data structures

• Output:
  – Semantic knowledge extracted from the snapshot
  – Examples: running processes, loaded modules, network connections, …
Existing Work

• Signature Scanning
  – Signature tools in Volatility
  – Robust signature [Dolan-Gavitt et al. CCS 2009]
  – Graph signature [Lin et al. NDSS 2011]

• Data Structure Traversal
  – Traversal tools in Volatility
  – KOP/MAS from Microsoft Research
Why is it useful?

• Digital Forensics
  – Collect crime evidence

• Virtual Machine Introspection
  – Monitor VM activities from outside, desired for IaaS cloud providers

• Malware Detection
  – Find malware (especially kernel rootkit) footprint in memory

*Unique Advantage: Code and data must be loaded into memory to be executed and accessed*
What are the challenges?

• Semantic Gap
  – Data structure definitions (Limited documentation for closed-source OSes)
  – Global variables (The roots for traversal)
  – Generic pointers (void *, struct list_head, LIST_ENTRY)
  – Consequence: low coverage and reduced accuracy

• Memory Manipulation Attacks
  – Pointer manipulation (add/remove/change a pointer, unlink an object)
  – Value manipulation (obtain fraudulent info)
  – Consequence: low robustness and low trustworthiness
Questions to Investigate

• Q1: Correctness (in non-deceptive settings)
  – How can we know if a memory analysis tool produces correct results, especially for closed-source OSes?

• Q2: Robustness (in deceptive settings)
  – To what extent a memory analysis tool can still produce correct results, if the memory snapshot has been compromised?

• Q3: Improving the state-of-the-art
  – Can we develop a better memory analysis tool (Higher coverage, more robust)?

We focus on OS kernel space. User-level memory analysis is beyond the scope for now.
Q1: Correctness Study

• How to obtain the ground truth: Kernel Data Structure Graph
• We construct it on the fly
  – Whole-system dynamic binary analysis
  – Use DECAFE, an open-source platform
  – Hook malloc/free functions

https://code.google.com/p/decaf-platform/
An example graph

• How do we map these objects into their types?

Caller1=804e4020  Caller1=804e4020  Caller1=804e4020
Caller2=805210a0  Caller2=805210a0  Caller2=805210a0
Caller3=804d30ab  Caller3=804d30ab  Caller3=804d30ab
Size=128          Size=128          Size=128
Mapping Caller List to Type

- **Rationale:** Objects allocated at the same calling context have the same type

## Caller List to Type Mapping in Windows 7

<table>
<thead>
<tr>
<th>Object</th>
<th>Caller 3</th>
<th>Caller 2</th>
<th>Caller 1</th>
<th>Size</th>
<th>In Volatility</th>
<th>In DSG</th>
</tr>
</thead>
<tbody>
<tr>
<td>_EPROCESS</td>
<td>PspCreateProcess</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>728</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>_EPROCESS</td>
<td>PspCreateProcess</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>744</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>_ETHREAD</td>
<td>PspAllocateThread</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>736</td>
<td>413</td>
<td>413</td>
</tr>
<tr>
<td>_DRIVER_OBJECT</td>
<td>IoCreateDriver</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>252</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>_DRIVER_OBJECT</td>
<td>IoCreateDriver</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>236</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>_DRIVER_OBJECT</td>
<td>IoCreateDriver</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>236</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>_DRIVER_OBJECT</td>
<td>EtwpStartAutoLogger</td>
<td>EtwpAllocateTraceBufferPool</td>
<td>EtwpAllocateFreeBuffers</td>
<td>65536</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>_DRIVER_OBJECT</td>
<td>IoCreateStreamFileObjectLIte</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>236</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>_DRIVER_OBJECT</td>
<td>IoCreateStreamFileObjectLIte</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>88</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>_KTHREAD</td>
<td>NtCreateMutant</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>72</td>
<td>234</td>
<td>234</td>
</tr>
<tr>
<td>_FILEOBJECT</td>
<td>IoCreateStreamFileObjectLIte</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>160</td>
<td>43</td>
<td>67</td>
</tr>
<tr>
<td>_FILEOBJECT</td>
<td>IoCreateStreamFileObjectLIte</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>176</td>
<td>136</td>
<td>179</td>
</tr>
<tr>
<td>_FILEOBJECT</td>
<td>IoCreateStreamFileObjectLIte</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>176</td>
<td>2025</td>
<td>2135</td>
</tr>
<tr>
<td>_FILEOBJECT</td>
<td>IoAllocateRealFileObject</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>160</td>
<td>182</td>
<td>321</td>
</tr>
<tr>
<td>_FILEOBJECT</td>
<td>IoAllocateRealFileObject</td>
<td>ObpCreateObject</td>
<td>ObpAllocateObject</td>
<td>160</td>
<td>182</td>
<td>321</td>
</tr>
</tbody>
</table>

We obtain debug symbols to map these callers to kernel function names.
Correctness for Basic Volatility Tools

- False positives are quite common in several tools, because signature patterns are not strong enough.
- 6 false negatives in filescan are due to an implementation error. We reported a patch to Volatility.
Correctness and Efficiency for Robust Signatures

<table>
<thead>
<tr>
<th>Tool</th>
<th>512 MB</th>
<th></th>
<th>1 GB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Obj</td>
<td>FN/FP</td>
<td>Time</td>
</tr>
<tr>
<td>SigField</td>
<td>641s</td>
<td>25</td>
<td>0/0</td>
<td>1283s</td>
</tr>
<tr>
<td>SigField_opt</td>
<td>341s</td>
<td>25</td>
<td>0/0</td>
<td>715s</td>
</tr>
<tr>
<td>SigGraph</td>
<td>494s</td>
<td>25</td>
<td>0/0</td>
<td>1006s</td>
</tr>
</tbody>
</table>

• We did not observe any errors
• Efficiency is quite low: spend several minutes to scan one memory dump, over 10 minutes for big dumps (1GB)
  – Cannot use these schemes for realtime VM scanning
• Basic signature tools are much faster
  – Only a few seconds
  – Skip large memory regions based on unreliable heuristics
Q1: Summary

- We obtained the ground truth by
  - dynamically monitoring kernel memory allocation and de-allocation
  - performing type inference to map these objects to their types

- Signature tools often raise false positives
  - Devise a good signature is hard
  - How to determine if an object is dead?

- Even one tool has an implementation error
  - Due to incorrect understanding of kernel data structure definition

- Robust signature schemes are slow!
  - Has to examine every byte at least once
Q2: Robustness Study

• Deceptive scenario
  – The OS kernel has been compromised
  – The attacker can write to arbitrary memory locations to deceive security analysis
  – The attacker does not want to crash the system

• Questions
  – Which data structure fields can be modified (mutable)?
  – How can these mutations affect memory analysis tools?
Q2: Our Approach

• Automatic Mutation Testing
  1. Start a test program inside VM
  2. Save and pause the VM
  3. Locate a kernel data structure belonging to that test program
  4. Select a field to mutate
  5. Resume the VM to finish the test program
     (system may crash or the program may terminate prematurely)
  6. Go to Step 2
Q2: Test Cases and Mutation Rules

<table>
<thead>
<tr>
<th>No</th>
<th>Test Case</th>
</tr>
</thead>
</table>
| 1  | Start test program  
    Test Point 1: mutate process & thread related values  
    Run other test cases |
| 2  | Load a user DLL  
    Test Point 2: mutate DLL related values  
    Call a function in the DLL repeatedly  
    Unload the DLL |
| 3  | Load a kernel module  
    Test Point 3: mutate kernel module values  
    Send IO requests to the kernel module  
    Unload the kernel module |
| 4  | Open two files, one each for read and write  
    Test Point 4: mutate file values  
    Read and write the two files repeatedly  
    close the files |
| 5  | Open a TCP connection  
    Test Point 5: mutate values related to this connection  
    Send and receive data through this connection  
    Close the connection |
| 6  | Open a registry key (Windows only)  
    Test Point 6: Mutate registry key related values  
    Read and write this registry key repeatedly  
    Close the key |

<table>
<thead>
<tr>
<th>Type</th>
<th>Mutation Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>0, copy from another ID, increment or decrement by a small constant</td>
</tr>
<tr>
<td>Size/Offset</td>
<td>0, increment or decrement by a small constant</td>
</tr>
<tr>
<td>String</td>
<td>&quot;&quot;, copy from another string, mutate one character</td>
</tr>
</tbody>
</table>
### Q2: Our Findings

<table>
<thead>
<tr>
<th>Category: Structures</th>
<th>Semantic Field</th>
<th>Mutability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process: struct EPROCESS</td>
<td>UniqueProcessId, ExitStatus, ImageFileName, CreateTime, GrantedAccess, InheritedFromUniqueProcessId, ObjectTable, HANDLE_COUNT, ObjectHeader, ObjectTyp</td>
<td>✓</td>
</tr>
<tr>
<td>Thread: struct ETHREAD</td>
<td>StartAddress, Cid, UniqueThreadId, ObjectHeader, ObjectType</td>
<td>✓</td>
</tr>
<tr>
<td>DLL &amp; Kernel Module: struct LDR_DATA_TABLE_ENTRY</td>
<td>DllBase, EntryPoint, FullDllName, BaseDllName, Flags, LoadCount, PatchInfo</td>
<td>✓</td>
</tr>
<tr>
<td>Registry Key, CM_KEY_NODE</td>
<td>Name, NameLength, LastWriteTime, SubkeyCounts, Flags, Signature, Parent, Security</td>
<td>✓</td>
</tr>
<tr>
<td>Network</td>
<td>TCPT_OBJECT, RemoteIpAddress, TCPT_OBJECT, RemotePort, TCPT_OBJECT, LocalAddress, TCPT_OBJECT, LocalPort, TCPT_OBJECT, Pid</td>
<td>✓</td>
</tr>
<tr>
<td>Memory Pool: struct POOL_HEADER</td>
<td>PoolTag, BlockSize</td>
<td>✓</td>
</tr>
</tbody>
</table>

For Windows XP SP3:

- Many fields are mutable, and thus untrustworthy. Bad news!

For Linux 2.6:

- Many fields are mutable, and thus untrustworthy. Bad news!
Q2: Insights

- **Network related values are not mutable**
  - Local and remote IP addresses and ports

- **Process ID is mutable**
  - In Windows, PID in EThread is not mutable

- **Strings are mutable**
  - Process name, file name, module name, pool tag, etc.

- **Timestamps are mutable**
  - E.g. process creation and termination time
Q3: Improving the state-of-the-art

- Q3.1: Can we leverage information redundancy to defeat value manipulation attacks?
  - E.g., PID appears in multiple data structures. Attackers need to modify all these copies for complete deception

- Q3.2: Can we improve the coverage and accuracy of data structure identification from a global view?
  - Prior approaches evaluate each data structure individually
Q3.1: How to find duplicate values in kernel space?

- Trace kernel execution
- For each insn, perform bidirectional data flow analysis to track duplicate values

---

**Algorithm 1 Dynamic Duplicate Value Analysis**

```
procedure DYNVALUEANALYSIS(Trace t)
    for all instruction i ∈ t do
        if i.type is assignment operation then
            for each src & dst byte pair (u, v) do
                DoAssign(u, v)
            end for
        else
            for each byte v in the dst operand do
                DoRemove(v)
            end for
        end if
    end for
end procedure

procedure DoAssign(u, v)
    for all variable r ∈ S_v do
        S_r ← S_r - {v}
    end for
    for all variable r ∈ S_u do
        S_r ← S_r + {v}
    end for
    S_v ← S_u
end procedure

procedure DoRemove(v)
    for all variable r ∈ S_v do
        S_r ← S_r - {v}
    end for
end procedure
```
Q3.1: Testing Procedure

- Start test program
- Trace kernel execution
- Pause VM
- Save VM state
- Compute duplicate values
- Locate semantic values
- Observe VM execution
- Revert to the saved state
- Mutate a value or a value set
- Resume VM
Q3.1: Mutating duplicate values in Windows XP

<table>
<thead>
<tr>
<th>Primary Field</th>
<th># of Dups</th>
<th>Type of Duplicates</th>
<th>Immutable Duplicates</th>
<th>Set Mutability</th>
</tr>
</thead>
<tbody>
<tr>
<td>_EPROCESS.UniqueProcessId</td>
<td>36</td>
<td>_ETHREAD.Cid.UniqueProcess, _HANDLE_TABLE.UniqueProcessId, _CM_KEY_BODY.ProcessId, _EPROCESS.InheritedFromUniqueProcessId, _ETIMER.Lock, _TEB.ClientId, _TEB.RealClientId, 0x9b57b6d0, 0x9cdaef0, 0x9ce697c...</td>
<td>_ETHREAD.Cid.UniqueProcess</td>
<td>✗</td>
</tr>
<tr>
<td>_EPROCESS.ImageFileName</td>
<td>4</td>
<td>_OBJECT_NAME_INFORMATION.Name, _RTL_USER_PROC_PARAMS.ImagePathName, _SE_AUDIT_PROCESS_INFO.ImageFileName</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>_EPROCESS.CreateTime</td>
<td>2</td>
<td>_ETHREAD.CreateTime</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>_EPROCESS.ActiveThreads</td>
<td>2</td>
<td>_EPROCESS.ActiveThreadsHighWatermark</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>_HANDLE_TABLE.HandleCount</td>
<td>2</td>
<td>_HANDLE_TABLE.HandleCountHighWatermark</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>_FILE_OBJECT.FileName (Data file)</td>
<td>7</td>
<td>0x003a948c, 0x822df33a, 0x822df35c, ...</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>_LDR_DATA_TABLE_ENTRY.FullDllName</td>
<td>3</td>
<td>_LDR_DATA_TABLE_ENTRY.FullDllName</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>_LDR_DATA_TABLE_ENTRY.BaseDllName</td>
<td>3</td>
<td>_LDR_DATA_TABLE_ENTRY.BaseDllName</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>_CM_KEY_NODE.LastWriteTime</td>
<td>2</td>
<td>0x9b43ea60</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>_CM_KEY_NODE.Parent</td>
<td>4</td>
<td>0x94d20a20, 0x9adc7940, 0x9adc7948</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>_CM_KEY_NODE.Security</td>
<td>2</td>
<td>0x822c7880</td>
<td>_CM_KEY_NODE.Security</td>
<td>✗</td>
</tr>
<tr>
<td>_ETHREAD.StartAddress</td>
<td>2</td>
<td>_SECTION_OBJECT.StartingVa</td>
<td>_SECTION_OBJECT.StartingVa</td>
<td>✗</td>
</tr>
</tbody>
</table>
Q3.1: Mutating duplicate values in Linux

<table>
<thead>
<tr>
<th>Primary Field</th>
<th># of Dups</th>
<th>Type of Duplicates</th>
<th>Immutable Duplicates</th>
<th>Set Mutability</th>
</tr>
</thead>
<tbody>
<tr>
<td>task_struct.pid</td>
<td>4</td>
<td>task_struct.t_gid, task_struct.t_gid(lwp), 0xf63916dc</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>task_struct.comm</td>
<td>2</td>
<td>task_struct.comm(lwp)</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>task_struct.static_prio</td>
<td>3</td>
<td>task.parent.static_prio, task.static_prio (lwp)</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>task_struct.exit_code</td>
<td>3</td>
<td>task.parent.exit_code, task.exit_code (lwp)</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>task_struct.fds</td>
<td>3</td>
<td>0xf7179080, 0xf61bae84</td>
<td>0xf7179080, task fds</td>
<td>x</td>
</tr>
<tr>
<td>module.name</td>
<td>2</td>
<td>0xdf93c524c</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>module.num_syms</td>
<td>12</td>
<td>module.num_kp, 0xe086c15c, 0xe086c170...</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>vma.vm_start</td>
<td>2</td>
<td>vma.vm_end</td>
<td>vma.vm_start</td>
<td>x</td>
</tr>
<tr>
<td>vma.vm_end</td>
<td>2</td>
<td>vma.vm_end</td>
<td>vma.vm_end</td>
<td>x</td>
</tr>
<tr>
<td>dentry.d_name</td>
<td>2</td>
<td>0x5f83f0d8</td>
<td>None</td>
<td>✓</td>
</tr>
<tr>
<td>inet_sock.saddr</td>
<td>24</td>
<td>inet_sock.rcv_saddr 0xde49147c ...</td>
<td>inet_sock.rcv_saddr 0xde49147c ...</td>
<td>x</td>
</tr>
<tr>
<td>inet_sock.daddr</td>
<td>24</td>
<td>inet_sock.rcv_saddr 0xde49147c ...</td>
<td>inet_sock.daddr 0xde49147c ...</td>
<td>x</td>
</tr>
</tbody>
</table>
Q3.1: Observations

• Duplicate values do exist for many important semantic values

• Unfortunately, most of these duplicate values are mutable both collectively and individually

• In very limited cases, checking duplicate values can be helpful to defeat value manipulation attacks
Q3.2: A Network Perspective

- Kernel data structure graph is a **small-world network**

### 5 dumps for Windows XP

<table>
<thead>
<tr>
<th></th>
<th>Nodes</th>
<th>Links</th>
<th>Diameter</th>
<th>Center</th>
<th>Clustering Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18249</td>
<td>173153</td>
<td>2.81</td>
<td>ntoskml.exe</td>
<td>0.070</td>
</tr>
<tr>
<td>2</td>
<td>22713</td>
<td>205683</td>
<td>3.77</td>
<td>ntoskml.exe</td>
<td>0.067</td>
</tr>
<tr>
<td>3</td>
<td>20078</td>
<td>221224</td>
<td>2.81</td>
<td>ntoskml.exe</td>
<td>0.074</td>
</tr>
<tr>
<td>4</td>
<td>29811</td>
<td>222231</td>
<td>3.11</td>
<td>CMCa</td>
<td>0.061</td>
</tr>
<tr>
<td>5</td>
<td>33277</td>
<td>200816</td>
<td>5.01</td>
<td>CMCa</td>
<td>0.053</td>
</tr>
</tbody>
</table>
• Points-to relation reveals the objects’ types
  – Deterministically (for typed pointers)
  – Probabilistically (for generic pointers)

• We perform supervised learning
  – Use DECAF to collect labeled memory dumps
  – Construct a pointer-constraint model

• We perform network-based inference
  – Adapt random surfer model (a page rank algorithm)
Q3.2: Scan, Traverse and Infer
More details…

Will be presented in Thursday 11am

<table>
<thead>
<tr>
<th>Time</th>
<th>Orleans A</th>
<th>Orleans B</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:00-10:30</td>
<td><strong>Break (Foyer)</strong></td>
<td></td>
</tr>
</tbody>
</table>
| 10:30-12:00 | **Panel: The Attacker Among Us: Insider Threats Within the Energy Sector**
Moderator: Dr. William (Bill) Claycomb, CERT Insider Threat Center - Carnegie Mellon University
Panelists (listed alphabetically):
Dr. Nader Mehravari, Cyber Security Solutions, Software Engineering Institute
Dr. Shawn Taylor, Sandia National Laboratories
Mr. Randy Trzeciak, CERT Insider Threat Center - Carnegie Mellon University

**Securing Memory and Storage**
Chair: Cristina Serban
SEER: Practical Memory Virus Scanning as a Service
Jason Gionta, North Carolina State University; Ahmed Azab, Samsung Electronics Co., Ltd.; William Enck, North Carolina State University; Peng Ning, North Carolina State University; Xiaolan Zhang, Google Inc.

**MACE: High-Coverage and Robust Memory Analysis For Commodity Operating Systems**
Qian Feng, Syracuse University; Aravind Prakash, Syracuse University; Heng Yin, Syracuse University; Zhiqiang Lin, University of Texas at Dallas

**Assisted Deletion of Related Content**
Hubert Ritzdorf, ETH Zurich; Nikolaos Karapanos, ETH Zurich; Srdjan Capkun, ETH Zurich
Q3.2: Result Highlights

• Achieve > 95% coverage for WinXP/Win7
  – Include both documented and undocumented objects

• With 80% typed pointers removed, our coverage degradation is negligible
  – Thanks to the small-world network

• Outperform Volatility
  – Detect rootkit footprint in undocumented objects
  – Resilient against pool tag manipulations
  – In contrast, volatility heavily relies on pool tags and thus is completely defeated
Summary

• We conducted a systematic study on memory analysis
  – With respect to correctness and robustness
  – Explore ways to improve it

• We made the following conclusions
  – The existing tools may produce erroneous results, and may even have implementation errors.
  – The robustness of these tools are questionable, given that attackers can freely manipulate many semantic values
  – Exploiting duplicate values can improve the robustness, but the improvement is marginal
  – Exploiting pointer relations is a promising direction. It can improve coverage and defeat pointer manipulation attacks.
For more details, please read

- “On the Trustworthiness of Memory Analysis --- An Empirical Study from the Perspective of Binary Execution”, IEEE Transactions on Dependable and Secure Computing
Questions?