**CS 6V81-05**

Kernel Rootkit Defense I: Graph-based Scanning Approach

Zhiqiang Lin

Department of Computer Science
The University of Texas at Dallas

September 2nd, 2011

---

**Outline**

- Background
- SigGraph
- KOP
- Summary

---

**Kernel rootkit**

**How to detect the hidden object**

**Traversing memory**

- From global memory, to all reachable data [KOP]
- Deriving signature, and using it to scan memory [SigGraph]

---

**Memory graph approach used in software debugging**

---

*Credit: [Petroni et al, USENIX Security 2006]*

---

*Fig. 3. A simple memory graph*
Outline

1 Background
2 SigGraph
3 KOP
4 Summary

Basic Idea

Sample Data Structure

```c
struct task (T) {
    [0] struct thread *thread;
    [4] struct memory *mm;
    [8] struct signal *signal;
    [12] struct task *parent;
}
```

Static Derivation

Substituting symbols in rule (1) using rules (2), (3) and (4), we further have

\[
S_r(x) \rightarrow S_r(x + 0) \land S_r(x + 4) \land S_r(x + 8) \land S_r(x + 12)
\]

(6)

Inverse Inference

\[
S_r(x) \leftarrow S_r(x + 0) \land S_r(x + 4) \land S_r(x + 8) \land S_r(x + 12)
\]

(6)

How to use it

Signature scanning example (Suppose 2 layers)
Challenges

Key Challenge
A signature should be unique, non-isomorphic with other signatures

Sample Data Structure

struct B {
    [0] E * b1;
    [4] B * b2;
}

IPP(B) = 0 · E · 4 · B

Solution

Definition
Given a data structure T, let its pointer field offsets be f1, f2, ..., and fn, pointing to types t1, t2, ..., and tn, resp. Its immediate pointer pattern, denoted as IPP(T), is defined as follows.

\[ IPP(T) = f_1 \cdot f_2 \cdot (f_3 - f_2) \cdot (f_4 - f_3) \cdot ... \cdot (f_n - f_{n-1}) \cdot f_n \]

Substring problem
Determine IPP(A) is a substring of IPP(X).

Problem Formulation

IPP(A) = 0 · B · 12 · C · 6 · D

IPP(X) = 0 · Y · 28 · BB · 12 · CC · 6 · DD

Example
**Practical Issues**

- Null pointers
- Void pointers (KOP[CCS09]'s contribution)
- User-level pointers
- Special pointers
  - LIST_POISON1 (0x00100100)
  - LIST_POISON2 (0x00200200)
  - SPINLOCK_MAGIC (0xdeadbeef)
- Pointer-like values
- Undecided pointers (union)

**Solution**

Pruning a few noisy pointer fields (identified by profiler) does not degenerate the uniqueness of the graph-based signatures.

**Evaluation**

### Table: Experimental results of signature uniqueness test

<table>
<thead>
<tr>
<th>Kernel version</th>
<th>#Total structs</th>
<th>#Pointer structs</th>
<th>#Unique sig</th>
<th>Percent</th>
<th>$\delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.15-1</td>
<td>8890</td>
<td>3597</td>
<td>3229</td>
<td>89.76%</td>
<td>2.31</td>
</tr>
<tr>
<td>2.6.15-1</td>
<td>17800</td>
<td>4852</td>
<td>4355</td>
<td>88.76%</td>
<td>2.46</td>
</tr>
<tr>
<td>2.6.20-1</td>
<td>14992</td>
<td>6098</td>
<td>5395</td>
<td>88.50%</td>
<td>2.54</td>
</tr>
<tr>
<td>2.6.24-26</td>
<td>13981</td>
<td>6427</td>
<td>5945</td>
<td>87.83%</td>
<td>2.67</td>
</tr>
</tbody>
</table>

**Related work**

- Kernel memory mapping and analysis
  - Copilot [Petroni et al., Security 04], [Petroni et al., CCS 07]
  - Gibralter [Baliga et al., ACSAC 06]
  - KOP [Carbone et al., CCS 09]

- Memory forensics
  - Memory graph-based: Redhat crash utility, KOP
  - Value-invariant Signature: Klist [Plutkowska, 2003], GREPEXEC [bugcheck, 2006], Volatility [Walters, 2006], [Schuster, 2006], [Dolan-Gavitt et al., CCS 08]
  - Dynamic heap type inference [Polishchuk et al., 2007]

**Conclusion**

- Points-to relations can be leveraged to generate graph-based signatures for brute force scanning
- SigGraph, a framework that generates non-isomorphic structural-invariant signatures Complements value-invariant signatures
- Applications:
  - Kernel memory forensics
  - Kernel rootkit detection
**KOP**

- **Goal**: Identifying hidden object.
- **Basic approach**: start from global variables and follow pointers in objects recursively, mapping each object along the way
- **Challenges**:
  - Locate objects
  - Identify their types

  Credit: part of the slides in KOP section are compiled from Martim Carbone's CCS'09 presentation

**State of the art**

- **Previous work**
  - Linked lists – Manual Annotation
  - *(void *) pointers* – Not Solved
  - Unions – Not Solved
  - Dynamic arrays – Not Solved

  Poor coverage of existing techniques
  “Only 27% coverage of the dynamic kernel data in Windows Vista SP1!”

**Architecture**

Input
- OS kernel + drivers source code

Goal
- Generating extended type graph
  - Nodes are type definitions
  - Edges
    - Typed pointers
    - Candidates for generic pointers (linked lists and void*)

Techniques
- **points-to-analysis**
  - They prioritize precision over performance
  - Only needs to be run once for each OS

Credit: this figure is from Martim Carbone’s CCS’09 presentation
# Memory Analysis

- Start from global variables and traverse the memory snapshot using the extended type graph
- Final output: object graph

## Issues

- Typed pointer traversal
- Generic pointer traversal
  - Memory allocation boundaries (hard constraint)
  - Valid pointer values (soft constraint)
- Dynamic array identification
- Robustness

# Dynamic Array Identification

- Done after the object traversal
- Assumes knowledge of active memory blocks
- Candidates: memory blocks with a single object found at their beginning

```c
1. d_array = alloc(256*sizeof(Obj))
2. struct Obj {
   ...
   struct Obj2 d_array[0/1]
}
```

- Test candidates using type selection constraints

# Robustness

- Traversal is very sensitive to errors in type identification
- How to minimize their occurrence and impact?
  - Traverse in multiple rounds, gradually decreasing the confidence level and expanding the traversal
  - Apply a safeguard mechanism at every step

# Integrity Checking Applications

- Function pointer checking
  - Subverted Function Pointer Detector (SFPD)
  - Higher function pointer coverage due to KOP
- Hidden object discovery
  - General Hidden Object Scanning Tool (GHOST)

# General Hidden Object Scanning Tool (GHOST)

- Function pointers
  - Commonly targeted by kernel malware
- Explicit Pointer
  - `void (*handler)(int, int, char);`
  - Obtained from type definitions
- Implicit Pointer
  - `unsigned int handler;
   - Type definitions are not enough in this case!
   - They rely on their points-to analysis to identify implicit function pointer candidates`

# KOP’s Implementation

- 16,000 LOC in C#
- Phoenix compiler + Windows Debugger API
- Target: Windows Vista SP1 kernel
Evaluation

### Coverage

<table>
<thead>
<tr>
<th></th>
<th>Previous (%)</th>
<th>KOP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clean-boot</td>
<td>29.3</td>
<td>98.9</td>
</tr>
<tr>
<td>Stress-test</td>
<td>28.0</td>
<td>98.8</td>
</tr>
</tbody>
</table>

### Performance Overhead

<table>
<thead>
<tr>
<th>Component</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static analyzer</td>
<td>48 hours</td>
</tr>
<tr>
<td>Memory analyzer</td>
<td>8 minutes</td>
</tr>
</tbody>
</table>

Evaluation - SFPD

Zero false positives and false negatives for all the 8 rootkits tested.

Limitations

- Third-party drivers
- Memory analysis
  - In some cases pointers are not enough
  - Intentional memory pollution
  - Assumes integrity of addressing structures
- Offline analysis
  - There is plenty of room for optimizations

Conclusion

- Dynamic kernel data integrity is hard
- Kernel Object Pinpointer (KOP)
  - Addresses challenges with mapping kernel objects
  - Results: very high coverage on Vista SP1
- Security applications
  - Find malicious function pointers and hidden objects
  - Directly benefits from KOP’s high coverage

Outline

1. Background
2. SigGraph
3. KOP
4. Summary

Summary

- Both SigGraph and KOP traverse pointers.
- KOP relies on root, SigGraph does not
- KOP requires access to source code, SigGraph requires access to data structure definition (which can be acquired from debugging information)
- KOP is a top-down approach, SigGraph is a bottom-up approach

Next Lecture

Mitchell Adair and Kevin Weaver will be presenting exciting vulnerability analysis papers