CS 6V81-05
Kernel Rootkit Defense I: Graph-based Scanning Approach

Zhiqiang Lin
Department of Computer Science
The University of Texas at Dallas

September 2\textsuperscript{nd}, 2011
Outline

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

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

Credit: [Petroni et al, USENIX Security 2006]
How to detect the hidden object

**Traversing** memory

1. From global memory, to all reachable data [KOP]
2. Deriving signature, and using it to scan memory [SigGraph]
Memory graph approach used in software debugging

Thomas Zimmermann and Andreas Zeller, Visualizing Memory Graphs, 2001
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;
}
struct thread (R) {
    [0] struct task *task;
}
struct memory (M) {
    [0] struct vma *mmap;
    [4] void (*map_area)(struct memory* mmap);
}
struct signal (S) {
    [0] struct task_status *status;
}
```

Static Derivation

\[S_T(x) \rightarrow S_R(*(x + 0)) \land S_M(*x + 4)) \land \]
\[S_S(*x + 8)) \land S_T(*x + 12))\] (1)
\[S_R(x) \rightarrow S_T(*x + 0)) \] (2)
\[S_M(x) \rightarrow S_{VA}(*x + 0)) \land S_{FP}(*x + 4))\] (3)
\[S_S(x) \rightarrow S_{TS}(*x + 0)) \] (4)
Basic Idea

Sample Data Structure

```
struct task (T) {
    [0] struct thread *thread;
    [4] struct memory *mm;
    [8] struct signal *signal;
    [12] struct task *parent;
}
struct thread (R) {
    [0] struct task *task;
}
struct memory (M) {
    [0] struct vma *mmap;
    [4] void (*map_area)(struct memory* mmmap);
}
struct signal (S) {
    [0] struct task_status *status;
}
```

Static Derivation

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

\[
S_T(x) \rightarrow S_T(*(*(x + 0) + 0)) \land
S_{VA}(*/(*(x + 4) + 0)) \land
S_{FP}(*/(*(x + 4) + 4)) \land
S_{TS}(*/(*(x + 8) + 0))) \land
S_T(*(x + 12))
\]

(5)
**Basic Idea**

**Sample Data Structure**

```c
tstruct task (T) {
    [0] struct thread *thread;
    [4] struct memory *mm;
    [8] struct signal *signal;
    [12] struct task *parent;
}
tstruct thread (R) {
    [0] struct task *task;
}
tstruct memory (M) {
    [0] struct vma *mmap;
    [4] void (*map_area)
        (struct memory* mmap);
}
tstruct signal (S) {
    [0] struct task_status *status;
}
```

**Inverse Inference**

\[
S_T(x) \leftarrow S_T(*((x + 0) + 0)) \land \\
S_{VA}(*((x + 4) + 0)) \land \\
S_{FP}(*((x + 4) + 4)) \land \\
S_{TS}(*((x + 8) + 0)) \land \\
S_T(*(x + 12))
\] (6)
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;
}
struct thread (R) {
    [0] struct task *task;
}
struct memory (M) {
    [0] struct vma *mmap;
    [4] void (*map_area)(struct memory* mmap);
}
struct signal (S) {
    [0] struct task_status *status;
}
```

**Graph Representation**

The graph shows the relationships between different data structures:

- **T** (Task): 0
- **R** (Thread): 4
- **M** (Memory): 8
- **S** (Signal): 12

Each node represents a data structure, and the edges indicate dependencies or connections between them.
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;
}

struct thread (R) {
    [0] struct task *task;
}

struct memory (M) {
    [0] struct vma *mmap;
    [4] void (*map_area)(struct memory* mmap);
}

struct signal (S) {
    [0] struct task_status *status;
}
```

Graph Representation

```
T
  /|
 / |
/  \
R  M
    /|
    / |
    /  \
   T  VM
        /|
        / |
        /  \
       FP  T*
```

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;
}
struct thread (R) {
    [0] struct task *task;
}
struct memory (M) {
    [0] struct vma *mmap;
    [4] void (*map_area)(struct memory* mmap);
}
struct signal (S) {
    [0] struct task_status *status;
}
```

Graph Representation

```
T
  /  \
/    |
R 0   M
  /  \
/    |
T  4   S
    /  \
    /    |
   0     8
  /  \
/    |
T  12 T*
  /  \
/    |
   0  T*
```

How to use it

Sample Data Structure

0xc001c0a8: 0xc002c0a8 0xc002bee0 0xc002caa0 0xc00ddbb0
0xc002c0a8: 0xc12a0e7c 0xc727faa8 0xbfbb9195 0x00000009
0xc002bee0: 0xc001c114 0xc001c16c 0xffb29122 0x00201001
0xc002caa0: 0xb002ca20 0xb021d00a 0xc05b9f5c 0x00000000
0xc00ddbb0: 0xc12a0e7c 0xc727faa8 0xc001c114 0xc001c16c

Signature scanning example (Suppose 2 layers)
Challenges

Key Challenge

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

Sample Data Structure

```c
struct B {
    [0] E * b1;
    [4] B * b2;
} struct G {
    ... [10] int * g;
} struct BB {
    [0] EE * b1;
    [4] BE * b2;
} struct E {
    ...
    [12] G * e1;
    ...
    [24] H * e3;
}
struct GG {
    ...
    [4] char * gg1;
    ...
    [8] char * gg2;
    ...
    [24] HH * e3;
}
```

Signature Graph
Isomorphism

Sample Data Structure

```c
struct B {
    [0] E * b1;
    [4] B * b2;
} struct G {
    ... [10] int * g;
} struct BB {
    [0] EE * b1;
    [4] BE * b2;
} struct E {
    ... [12] G * e1;
    ... [24] H * e3;
} struct GG {
    ... [4] char * gg1;
    ... [8] char * gg2;
} struct EE {
    ... [12] GG * e1;
    ... [24] HH * e3;
}
```

Signature Graph

```
  B
 /|
/  |
E  B
 /|
G  H
 /|  
|  B*
```

```
  BB
 /|
/  |
EE  BB
 /|
GG  HH
 /|  
|  BB*
```
Isomorphism

Sample Data Structure

```
struct B {
    [0] E * b1;
    [4] B * b2;}
struct E {
    ...
    [12] G * e1;
    ...
    [24] H * e3;}
struct G {
    ...
    [10] int * g;}
struct GG {
    ...
    [4] char * gg1;
    [8] char * gg2;}
struct BB {
    [0] EE * b1;
    [4] BE * b2;}
struct EE {
    ...
    [12] GG * e1;
    ...
    [24] HH * e3;}
```

Signature Graph

```
G  B  E  B
H  B  E  B
10 12 24 0 4
   12 24
       0
```

```
GG  BB  EE  BB
HH  BB  BB
0  0  4  0
   12 12
     8
```
Subgraph Isomorphism

**Sample Data Structure**

```c
struct A {
    [0] struct B * a1;
    ...
    [12] struct C * a2;
    ...
    [18] struct D * a3;
};

struct X {
    [8] struct Y * x1;
    ...
    [36] struct BB * x2;
    ...
    [48] struct CC * x3;
    ...
    [54] struct DD * x4;
};
```

**Signature Graph**

- **Signature Graph**
  - A: 0, 12, 18
  - B
  - C
  - D

- **Signature Graph**
  - X: 8, 36, 48, 54
  - Y
  - BB
  - CC
  - DD
Definition

Given a data structure $T$, let its pointer field offsets be $f_1$, $f_2$, ..., and $f_n$, pointing to types $t_1$, $t_2$, ..., and $t_n$, resp. Its immediate pointer pattern, denoted as $IPP(T)$, is defined as follows.

$$IPP(T) = f_1 \cdot t_1 \cdot (f_2 - f_1) \cdot t_2 \cdot (f_3 - f_2) \cdot t_3 \cdot \ldots \cdot (f_n - f_{n-1}) \cdot t_n.$$ 

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

Given a data structure $T$, let its pointer field offsets be $f_1$, $f_2$, ..., and $f_n$, pointing to types $t_1$, $t_2$, ..., and $t_n$, resp. Its immediate pointer pattern, denoted as $IPP(T)$, is defined as follows.

$$IPP(T) = f_1 \cdot t_1 \cdot (f_2 - f_1) \cdot t_2 \cdot (f_3 - f_2) \cdot t_3 \cdot ... \cdot (f_n - f_{n-1}) \cdot t_n.$$ 

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

Definition

Given a data structure $T$, let its pointer field offsets be $f_1$, $f_2$, ..., and $f_n$, pointing to types $t_1$, $t_2$, ..., and $t_n$, resp. Its *immediate pointer pattern*, denoted as $IPP(T)$, is defined as follows.

$$IPP(T) = f_1 \cdot t_1 \cdot (f_2 - f_1) \cdot t_2 \cdot (f_3 - f_2) \cdot t_3 \cdot ... \cdot (f_n - f_{n-1}) \cdot t_n.$$

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

$$IPP(B) = 0 \cdot E \cdot 4 \cdot B$$
Example

Sample Data Structure

```c
struct A {
    [0] struct B * a1;
    ...
    [12] struct C * a2;
    ...
    [18] struct D * a3;
}
```

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

Pointer Expansion

A

0 12 18

B C D
**Sample Data Structure**

```c
struct A {
    struct B * a1;
    ...
    struct C * a2;
    ...
    struct D * a3;
}
```

```c
struct B {
    E * b1;
    B * b2;
}
```

**Pointer Expansion**

\[
IPP(A) = 0 \cdot B \cdot 12 \cdot C \cdot 6 \cdot D
\]

\[
\overset{B}{\Rightarrow} 0 \cdot (0 \cdot E \cdot 4 \cdot B) \cdot 12 \cdot C \cdot 6 \cdot D
\]
Problem Formulation

Signature Graph

\[
\text{IPP}(A) = 0 \cdot B \cdot 12 \cdot C \cdot 6 \cdot D
\]

Signature Graph

\[
\text{IPP}(X) = 0 \cdot Y \cdot 28 \cdot BB \cdot 12 \cdot CC \cdot 6 \cdot DD
\]

Substring problem

Determine IPP(A) is a substring of IPP(X).
Practical Issues

1. Null pointers
2. Void pointers (KOP[CCS09]’s contribution)
3. User-level pointers
4. Special pointers
   - LIST_POISON1 (0x00100100)
   - LIST_POISON2 (0x00200200)
   - SPINLOCK_MAGIC (0xdead4ead)
5. Pointer-like values
6. 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>
<thead>
<tr>
<th>Kernel version</th>
<th>#Total structs</th>
<th>Signature Statistics</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#Pointer structs</td>
<td>#Unique Sig.</td>
</tr>
<tr>
<td>2.6.15-1</td>
<td>8850</td>
<td>3597</td>
<td>3229</td>
</tr>
<tr>
<td>2.6.18-1</td>
<td>11800</td>
<td>4882</td>
<td>4305</td>
</tr>
<tr>
<td>2.6.20-15</td>
<td>14992</td>
<td>6096</td>
<td>5395</td>
</tr>
<tr>
<td>2.6.24-26</td>
<td>15901</td>
<td>6427</td>
<td>5645</td>
</tr>
<tr>
<td>2.6.31-1</td>
<td>26799</td>
<td>9957</td>
<td>8683</td>
</tr>
</tbody>
</table>

**Table:** Experimental results of signature uniqueness test
## Evaluation

<table>
<thead>
<tr>
<th>Kernel version</th>
<th>Number of signatures in different steps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2.6.15-1</td>
<td>1355</td>
</tr>
<tr>
<td>2.6.18-1</td>
<td>1820</td>
</tr>
<tr>
<td>2.6.20-15</td>
<td>2137</td>
</tr>
<tr>
<td>2.6.24-26</td>
<td>2172</td>
</tr>
<tr>
<td>2.6.31-1</td>
<td>3364</td>
</tr>
</tbody>
</table>

**Table:** Detail statistics on our static signatures
| Data Structure    | || | SigGraph Signature | | Value Invariant Signature |
|-------------------|---|-----------------|---|-----------------|---|
|                   | | | $R$ | $FP'$ | $FP$ | $FN$ | $R$ | $FP'$ | $FP$ | $FN$ |
| task_struct       | 88 | 88 | 0.00% | 0.00% | 0.00% | 88 | 0.00% | 0.00% | 0.00% |
| thread_info       | 88 | 88 | 0.00% | 0.00% | 0.00% | 93 | 6.45% | 6.45% | 1.08% |
| key               | 22 | 22 | 0.00% | 0.00% | 0.00% | 19 | 0.00% | 0.00% | 15.79% |
| mm_struct         | 52 | 54 | 3.70% | 0.00% | 0.00% | 55 | 5.45% | 0.00% | 0.00% |
| vm_area           | 2174 | 2233 | 2.64% | 0.40% | 0.00% | 2405 | 9.61% | 7.52% | 0.00% |
| kmem_cache        | 127 | 127 | 0.00% | 0.00% | 0.00% | 5124 | 97.52% | 97.52% | 0.00% |
| files_struct      | 53 | 53 | 0.00% | 0.00% | 0.00% | 50 | 0.00% | 0.00% | 6.00% |
| fs_struct         | 52 | 60 | 13.33% | 0.00% | 0.00% | 60 | 13.33% | 0.00% | 0.00% |
| file              | 791 | 791 | 0.00% | 0.00% | 0.00% | 791 | 0.00% | 0.00% | 0.00% |
| dentry            | 31816 | 38611 | 17.60% | 0.01% | 0.00% | 31816 | 0.00% | 0.00% | 0.00% |
| proc_inode        | 885 | 885 | 0.00% | 0.00% | 0.00% | 470 | 0.00% | 0.00% | 88.30% |
| ext3_inode        | 38153 | 38153 | 0.00% | 0.00% | 0.00% | 38153 | 0.00% | 0.00% | 0.00% |
| vsmount           | 28 | 28 | 0.00% | 0.00% | 0.00% | 28 | 0.00% | 0.00% | 0.00% |
| sysfs_dirent      | 2105 | 2116 | 0.52% | 0.52% | 0.00% | 88823 | 97.63% | 97.63% | 0.00% |
| socket Alloc      | 75 | 75 | 0.00% | 0.00% | 0.00% | 75 | 0.00% | 0.00% | 0.00% |
| socket            | 55 | 55 | 0.00% | 0.00% | 0.00% | 49 | 0.00% | 0.00% | 12.24% |
| sock              | 55 | 55 | 0.00% | 0.00% | 0.00% | 43 | 0.00% | 0.00% | 27.90% |
| bdev_inode        | 25 | 25 | 0.00% | 0.00% | 0.00% | 24 | 0.00% | 0.00% | 4.17% |
| signal_struct     | 73 | 73 | 0.00% | 0.00% | 0.00% | 72 | 0.00% | 0.00% | 1.39% |
| user_struct       | 10 | 10 | 0.00% | 0.00% | 0.00% | 10591 | 99.91% | 99.91% | 0.00% |

**Table:** Experimental results of SigGraph signatures and value invariant-based signatures.
Related work

- **Kernel memory mapping and analysis**
  - Copilot [Petroni et al., Security 04], [Petroni et al., CCS 07]
  - Gibraltar [Baliga et al., ACSAC 08]
  - KOP [Carbone et al., CCS 09]

- **Memory forensics**
  - Memory graph-based: Redhat crash utility, KOP
  - Value-invariant Signature: Klist [Rutkowska, 2003], GREPEXEC [bugcheck, 2006], Volatility [Walters, 2006], [Schuster, 2006], [Dolan-Gavitt et al., CCS 09]

- **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
1. Background
2. SigGraph
3. KOP
4. Summary
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
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!"
Goal

Kernel Object Pinpointer (KOP)

100% coverage and accuracy by addressing previous challenges
**Goal**

**Kernel Object Pinpointer (KOP)**
- 100% coverage and accuracy by addressing previous challenges

**Assumptions**
- Access to the OS source code
- Ability to capture a memory snapshot
Architecture

Credit: this figure is from Martim Carbone's CCS'09 presentation
Static Analysis

Input

- OS kernel + drivers source code
Static Analysis

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*)
## Static Analysis

### 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
Memory Analysis

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

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

Issues

1. Typed pointer traversal
2. Generic pointer traversal
   - 1. Memory allocation boundaries (hard constraint)
   - 2. Valid pointer values (soft constraint)
3. Dynamic array identification
4. 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
  
  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?
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)
Function pointers

- Commonly targeted by kernel malware
Function pointers
- Commonly targeted by kernel malware

Explicit Pointer
- `void (*handler)(int, int, char);`
- Obtained from type definitions
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>Snapshot</th>
<th>Previous (%)</th>
<th>KOP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clean-boot</td>
<td>26.8</td>
<td>98.9</td>
</tr>
<tr>
<td>Stress-test</td>
<td>28.0</td>
<td>98.8</td>
</tr>
</tbody>
</table>
### Coverage

<table>
<thead>
<tr>
<th>Snapshot</th>
<th>Previous (%)</th>
<th>KOP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clean-boot</td>
<td>26.8</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
Limitations

- Third-party drivers
- Memory analysis
  - In some cases pointers are not enough
  - Intentional memory pollution
  - Assumes integrity of addressing structures
- Offline analysis
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)
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
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
Both **SigGraph** and **KOP** traverse pointers.
Summary

- Both **SigGraph** and **KOP** traverse pointers.
- **KOP** relies on root, **SigGraph** does not.
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).
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
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