

Bridging the Semantic Gap Through Static Code Analysis

Christian Schneider, Jonas Pfoh, Claudia Eckert

`{schneidc,pfoh,eckertc}@in.tum.de`

Chair for IT Security
Technische Universität München
Munich, Germany

April 10, 2012

Outline

- 1 Motivation
 - Introducing InSight
 - Why debugging symbols are insufficient
- 2 Static Code Analysis
 - Step 1: Points-to Analysis
 - Step 2: Establishing Used-as Relations
- 3 Implementation
- 4 Conclusion

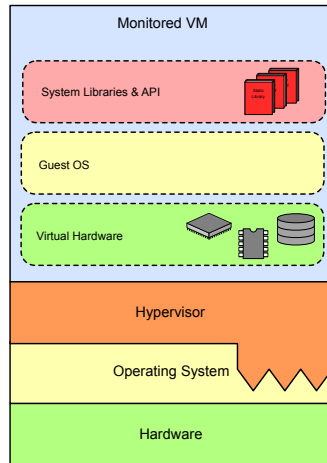
Outline

- 1 Motivation
 - Introducing InSight
 - Why debugging symbols are insufficient
- 2 Static Code Analysis
 - Step 1: Points-to Analysis
 - Step 2: Establishing Used-as Relations
- 3 Implementation
- 4 Conclusion

Virtual Machine Introspection (VMI)

[Garfinkel and Rosenblum(2003)]

VMI describes the act of **examining**, **monitoring** and **manipulating** a virtual machine from the **vantage point** of a **hypervisor**.



Semantic Gap

[Chen and Noble(2001)]

0800	0005	0800	0085	0800	0045	0900	00eb	0710	0008	0800	005d	0800	001d	0900	009b
0714	0053	0800	007d	0800	003d	0900	00db	0712	0017	0800	006d	0800	002d	0900	00bb
0800	000d	0800	008d	0800	004d	0900	00c3	0711	0003	0800	0053	0800	0013	0815	00c3
0713	0023	0800	0073	0800	0033	0900	00c7	0711	000b	0800	0063	0800	0023	0900	00a7
0800	0003	0800	0083	0800	0043	0900	00e7	0710	0007	0800	005b	0800	001b	0900	0097
0714	0043	0800	007b	0800	003b	0900	00d7	0712	0013	0800	006b	0800	002b	0900	00b7
0800	000b	0800	008b	0800	004b	0900	00c7	0710	0005	0800	0057	0800	0017	0840	0000
0713	0033	0800	0077	0800	0037	0900	00c1	0711	000f	0800	0067	0800	0027	0900	00af
0800	0007	0800	0087	0800	0047	0900	00ef	0710	0009	0800	005f	0800	001f	0900	009f
0714	0063	0800	007f	0800	003f	0900	00f7	0712	001b	0800	006f	0800	002f	0900	00bf
0800	000f	0800	008f	0800	004f	0900	00ff	0010	0010	0010	0010	0011	0011	0012	0012
0013	0013	0014	0014	0015	0015	0016	0016	0017	0017	0018	0018	0019	0019	001a	001a
001b	001b	001c	001c	001d	001d	0040	0040	0001	0002	0003	0004	0005	0007	0009	000d
0011	0019	0021	0031	0041	0061	0081	0081	0181	0201	0301	0401	0601	0801	0c01	
1001	1801	2001	3001	4001	6001	0000	0000	0010	0010	0010	0010	0010	0010	0010	0010
0011	0011	0011	0011	0012	0012	0012	0012	0013	0013	0013	0013	0014	0014	0014	0014
0015	0015	0015	0015	0010	00c9	00c4	0000	0003	0004	0005	0006	0007	0008	0009	000a
000b	000d	000f	0011	0013	0017	001b	001f	0023	002b	0033	003b	0043	0053	0063	0073
0083	00a3	00c3	00e3	0102	0000	0000	696b	696c	696c	2064	6964	7473	6e61	6563	7420
6f6f	6620	7261	6220	6361	006b	6e69	6176	696c	2064	6964	7473	6e61	6563	6320	646f
0065	6e69	6176	696c	2064	696c	6574	6176	2f6c	656c	676e	6874	6320	646f	0065	6e69
6f63	7272	6365	2074	6568	6461	7265	6320	6568	6b63	7500	6b6e	6f6e	6e77	6320	6d6f
7270	7365	6973	6e6f	6d20	7465	6f68	6301	6e69	6176	696c	2064	6977	646e	776f	7320
7a69	0065	6e69	6176	696c	2064	6c62	6301	206b	7974	6570	6900	766e	6c61	6469	7320
6f74	6572	2064	6c62	636f	206b	656c	676e	6874	0073	6e69	6176	696c	2064	6f63	6564
6c20	6e65	7467	7368	7320	7465	6900	766e	6c61	6469	6220	7469	6c20	6e65	7467	2064
6572	6570	7461	6900	766e	6c61	6469	6220	7465	6c61	6c2f	6e65	7467	7368	7320	

Bridging the Gap: Out-of-Band Delivery

Common Approach: utilize kernel debugging symbols

- Use symbols for:
 - Layout and size of kernel data structures
 - Virtual address of global variables and functions
 - Emulate virtual-to-physical address translation in software
- ⇒ Complex engineering task

Introducing InSight

[Schneider et al.(2011)]

Features:

- Stand-alone VMI tool to bridge the semantic gap
- Uses **debugging symbols** as foundation
- **Shell-like** interface for interactive inspection
- **JavaScript engine** for automated analysis
- Works for x86 **32 bit** (w/ PAE) and **64 bit Linux** guests
- Supports **any hypervisor** providing guest memory access

```

}
0x10 pti: 0x2bf7 spinlock_t
0x10 slab: 0x5436 struct kmem_cache *
0x10 first_page: 0x880 struct page *
}
0x20 0x543c union (anon.)
{
0x20 index: 0x89 long unsigned int
0x20 freelist: 0x30e void *
}
0x28 lru: 0x2b51 struct list_head
>>> mem q task_struct_cachep
task_struct_cachep: struct kmem_cache (ID 0x511d) @ 0xfffff88001f81100
0. 0x0000 flags : long unsigned int = 262144
1. 0x0008 size : int = 1808
2. 0x000c objsize : int = 1800
3. 0x0010 offset : int = 0
4. 0x0018 oo : struct kmem_cache_order_objects = ...
5. 0x0020 local_node : struct kmem_cache_node = ...
6. 0x0060 max : struct kmem_cache_order_objects = ...
7. 0x0068 min : struct kmem_cache_order_objects = ...
8. 0x0070 allocflags : gfp_t = 16384
9. 0x0074 refcount : int = 1
10. 0x0078 ctor : void (*)(void *) * = NULL
11. 0x0080 imuse : int = 1800
12. 0x0084 align : int = 16
13. 0x0088 min_partial : long unsigned int = 10
14. 0x0090 name : const char * = "task_struct"
15. 0x0098 list : struct list_head = ...
16. 0x00a8 kobj : struct kobject = ...
17. 0x00e8 rewrite_node_defrag_ratio : int = 1000
18. 0x00f0 node : struct kmem_cache_node *[54] = ... @ 0xfffff88001f811020
19. 0x02f0 cpu_slab : struct kmem_cache_cpu *[512] = ... @ 0xfffff880001811f00
>>>

```

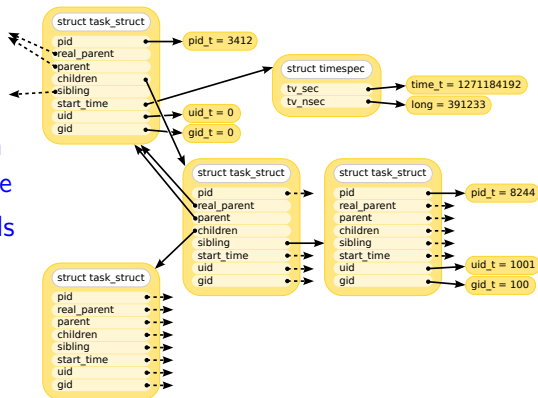
Introducing InSight (cont.)

[Schneider et al.(2011)]

Functionality so far

- Read objects from **known locations** with **known type**
- Follow **typed pointer fields** to further objects

But...



Why debugging symbols are insufficient

```
1  struct list_head {
2      struct list_head *next, *prev;
3  };
4
5  struct module {
6      struct list_head list;
7      char name[60];
8      /* ... */
9  };
10
11 struct list_head modules;
12
13 struct module* find_module(const char *name)
14 {
15     struct module *mod;
16     list_for_each_entry(mod, &modules, list)
17     {
18         if (strcmp(mod->name, name) == 0)
19             return mod;
20     }
21     return NULL;
22 }
```

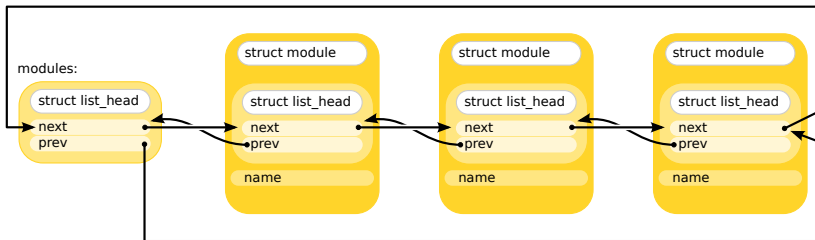
Why debugging symbols are insufficient (cont.)

```

1  struct module* find_module(const char * name)
2  {
3      struct module * mod;
4      /* Original code: list_for_each_entry(mod, &modules, list) */
5      for (mod = ({
6          const typeof(((typeof(*mod) *) 0)->list) * __mptr = (&modules->next);
7          (typeof(*mod) *) ((char *) __mptr - __builtin_offsetof(typeof(*mod), list));
8      });
9      __builtin_prefetch(mod->list.next), &mod->list != (&modules);
10     mod = ({
11         const typeof(((typeof(*mod) *) 0)->list) * __mptr = (mod->list.next);
12         (typeof(*mod) *) ((char *) __mptr - __builtin_offsetof(typeof(*mod), list));
13     })))
14     {
15         if (strcmp(mod->name, name) == 0)
16             return mod;
17     }
18     return ((void *) 0);
19 }

```

Why debugging symbols are insufficient (cont.)



Example: lsmod in JavaScript

Manually apply expert knowledge

```
1  function lsmod()
2  {
3      // type of variable "modules" is list_head
4      var head = new Instance("modules");
5      var m = head.next;
6      m.ChangeType("module");
7      // offset for address correction
8      var offset = m.MemberOffset("list");
9      m.AddToAddress(-offset);
10     // correct head as well for loop terminaten
11     head.AddToAddress(-offset);
12     // iterate over all modules
13     do {
14         print(m.name + " " + m.args);
15         m = m.list.next;
16         m.ChangeType("module");
17         m.AddToAddress(-offset);
18     } while (m && m.Address() != head.Address());
19 }
```

Summary

Problems

Runtime **pointer** and **type manipulations** are not reflected in the debugging symbols:

- type casts from `void*` pointers
- type casts from integer types
- pointer arithmetic
- variable length arrays

Possible solution

Static **analysis** of the kernel's **source code** to detect such runtime operations and augment the debugging symbols

Outline

- 1 Motivation
 - Introducing InSight
 - Why debugging symbols are insufficient
- 2 Static Code Analysis
 - Step 1: Points-to Analysis
 - Step 2: Establishing Used-as Relations
- 3 Implementation
- 4 Conclusion

Static Code Analysis

Questions our code analysis can answer:

- 1 Is a global variable or structure field **used as** a type that differs from its declaration?
- 2 How to **transform** a source value (field/variable) to derive the next object's address?

Our approach:

- **Type centric** analysis
- Captures arbitrary **pointer arithmetic**
- **Over-approximation** of possible pointer types
→ Increase object coverage at cost of type uncertainty

We call this the **used-as analysis**.

Used-As Analysis

Prerequisites:

- Kernel debugging symbols
- Pre-processed source code

```

1  struct module* find_module(const char * name)
2  {
3      struct module * mod;
4      /* Original code: list_for_each_entry(mod, &modules, list) */
5      for (mod = (
6          const typeof(((typeof(*mod) *) 0)->list) + __nptr = ((&modules)->next);
7          typeof(*mod) *) (char *) __nptr - __builtin_offsetof(typeof(*mod), list));
8      );
9          __builtin_prefetch(mod->list.next, &mod->list != (&modules));
10     mod = (
11         const typeof(((typeof(*mod) *) 0)->list) + __nptr = (mod->list.next);
12         typeof(*mod) *) (char *) __nptr - __builtin_offsetof(typeof(*mod), list));
13     );
14     {
15         if (strcmp(mod->name, name) == 0)
16             return mod;
17     }
18     return ((void *) 0);
19 }

```

Involves two steps:

- 1 Points-To Analysis
 - Detects memory aliasing between symbols (variables/pointers)
 - Reveals **indirect type usages** through local (pointer) variables
- 2 Establishing used-as relations
 - Find type usages contradicting their declaration
 - **type casts**
 - Record how value is transformed to target address
 - **pointer arithmetic**

Step 1: Points-to Analysis

Characteristics:

- structure/union field sensitive
- intra-procedural
- control-flow insensitive
- works on complete C expressions:

```
x = y + 8 * sizeof(int);
```

```
z = x & ~0xFF;
```

```
z ↦ {(y + 8 · sizeof(int)) & ~0xFF}
```

Result: transitive closure of points-to map

Step 2: Establishing Used-as Relations

Find used-as relations for

- **global variables** of pointer or integer type
- **structure/union fields** of pointer or integer type

Analysis overview:

- 1 Examine type usages under consideration of points-to map in
 - assignment statements
 - initializers
 - pointer dereferences after type casts
 - function parameters
 - return statements
- 2 Find mismatching source and target type
- 3 Identify corresponding context structure/union
- 4 Link alternative type along with arithmetic expression
 - to global variable or
 - to field of context structure/union

Step 2: Establishing Used-as Relations

Type usages

```

1  struct A { int value; struct A *next; };
2  struct B { void *data; }
3
4  struct A* func1(struct A *a) { return a; }
5
6  struct A* func2()
7  {
8      struct B b;
9      struct A a = { 0, b.data };      // initializer (struct)
10     struct A *pa = b.data;          // initializer (variable)
11     pa = b.data;                    // assignment
12     a = *((struct A*)b.data);       // dereference (*)
13     ((struct A*)b.data)->value++;   // dereference (->)
14     pa = func1(b.data);             // function parameter
15     return b.data;                 // return statement
16 }

```

Result: Field 'data' of `struct B` having type `void*` is used as `struct A*` with expression `(struct B).data`.

Step 2: Establishing Used-as Relations

Achieving type context sensitivity

Problem

- Used-as relations are often **unique** to their **context** (embedding) type
- Propagating such relations to other contexts would increase ambiguity

Solution

- **Copy embedded structures/unions** uniquely for embedding type
- Record used-as relations for this copy's members

Outline

- 1 Motivation
 - Introducing InSight
 - Why debugging symbols are insufficient
- 2 Static Code Analysis
 - Step 1: Points-to Analysis
 - Step 2: Establishing Used-as Relations
- 3 Implementation
- 4 Conclusion

Extension of InSight for Used-as Analysis

Required extensions:

- Consolidation of types from debugging symbols
- Parser for C with GCC extensions
- Semantic analyzer for “type flow” within statements and expressions
- Evaluator for C expressions, including many GCC builtins

Results of Used-As Analysis

Experiments with Debian 6.0, AMD64, Kernel 2.6.32

- Analysis required < 20 min. for 20 mio. LoC (584 MB)
- 11,382 unique types in total

Used-as relations in...

- 233 of 23,949 global variables
- 225 of 3,012 unique struct/union types
- 812 struct/union unique types with 908 members
 - 541 struct list_head
 - 18 struct hlist_head
 - 15 struct rb_root
 - 7 struct device

Example: lsmod in JavaScript

Manually apply expert knowledge

```
1  function lsmod()
2  {
3      // type of variable "modules" is list_head
4      var head = new Instance("modules");
5      var m = head.next;
6      m.ChangeType("module");
7      // offset for address correction
8      var offset = m.MemberOffset("list");
9      m.AddToAddress(-offset);
10     // correct head as well for loop terminaten
11     head.AddToAddress(-offset);
12     // iterate over all modules
13     do {
14         print(m.name + " " + m.args);
15         m = m.list.next;
16         m.ChangeType("module");
17         m.AddToAddress(-offset);
18     } while (m && m.Address() != head.Address());
19 }
```


Example: lsmod in JavaScript

Automatic application of used-as relations

```
1  function lsmod()
2  {
3      // type of variable "modules" is list_head
4      var head = new Instance("modules");
5      // iterate over all modules
6      var m = head.next;
7      while (m.MemberAddress("list") != head.Address()) {
8          print(m.name + " " + m.args);
9          m = m.list.next;
10     }
11 }
```

Outline

- 1 Motivation
 - Introducing InSight
 - Why debugging symbols are insufficient
- 2 Static Code Analysis
 - Step 1: Points-to Analysis
 - Step 2: Establishing Used-as Relations
- 3 Implementation
- 4 Conclusion

Conclusion

- Used-as analysis captures **type usages** contradicting declared type
- Extracts **arithmetic expression** to retrieve target object address
- Extension of **InSight** mimics dynamic pointer manipulations through kernel
- Highly advanced approach for **recreation of kernel state** from hypervisor's perspective

Released under **GPLv2 license:**

<https://code.google.com/p/insight-vmi/>

insight-vmi
A semantic bridge for virtual machine introspection and forensic applications

Project Information

Summary

insight is a tool for [debugging the kernel](#) in the field of virtual machine introspection (VMI) and digital forensics. It operates either on [topical kernel](#) or in conjunction with any hypervisor that provides read access to the physical memory of a guest VM. insight is written in C++ based on the Qt libraries and features [interactive analysis of kernel objects](#), as well as a [JSON-CAT engine](#) for automation of repeating inspection tasks.

This tool is developed by [Christian Schuster](#) and other contributors as part of [Open research at the Technische Universität München](#), Munich, Germany. Together with his colleagues at the [Chair for IT Security](#), his interest lies in the field of [static machine introspection](#) and how this can be used to [investigate](#) to improve current intrusion detection methods.

Features

Insight runs on Linux and provides the following functionality:

- analysis of physical memory dumps stored on disk as well as
- analysis of live physical memory of a virtual machine (this hypervisor provides access to the guest's memory)
- support for 32-bit and 64-bit addressing schemes with and without PAE
- no restriction of kernel objects to the Linux kernel
- automatic de-referencing of pointers to kernel objects
- an [API interface](#) for manual analysis of kernel objects
- a [JSON-CAT engine](#) for automated analysis of repeating or complex tasks

Example

The following example illustrates how the [JSON-CAT engine](#) can be used to interact with the kernel objects of an analyzed system in an easy and intuitive way. It starts by requesting an instance of the global variable `task_list`. This variable is of type `struct task_struct` and represents the heads to the list of process control blocks in the Linux kernel. The example goes on by printing unique information about the members of that structure, followed by iterating over all processes in the system and printing their process ID as well as the command being run.

```
// Get instance of global variable "task_list"
var task = new JSONCat("task_list");

// Iterate over all members of "task_list"
var members = task.Members();
for (var i = 0; i < members.length; ++i)
  var object = task.MemberOfTask(members[i].Name());
  print ("[" + i + "] = " + object.ToString();

// Print a list of running processes
var proc = task;
do {
  print(proc.pid + " = " + proc.cmd);
  proc = proc.task.next;
} while (proc.Address() != 0);
```

Further Reading

The following pages are a recommended reading with more information about insight, how to set it up and use it.

References



P. M. Chen and B. D. Noble.

When virtual is better than real.

In *Proc. of the 8th Workshop on Hot Topics in Operating Systems*, page 133. IEEE, 2001.



T. Garfinkel and M. Rosenblum.

A virtual machine introspection based architecture for intrusion detection.

In *Proc. of NDSS*, pages 191–206, 2003.



J. Pfoh, C. Schneider, and C. Eckert.

Nitro: Hardware-based system call tracing for virtual machines.

In *Advances in Information and Computer Security*, LNCS. Springer, Nov. 2011.



C. Schneider, J. Pfoh, and C. Eckert.

A universal semantic bridge for virtual machine introspection.

In *Information Systems Security*, volume 7093 of LNCS, pages 370–373. Springer, 2011.