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

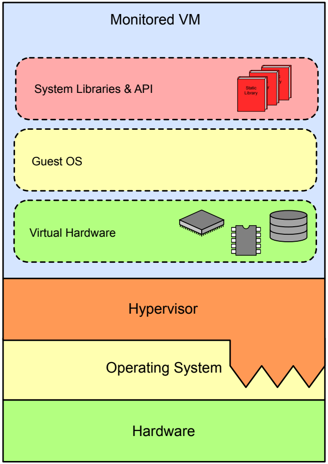
1. Motivation
   - Introducing InSight
   - Why debugging symbols are insufficient
2. Static Code Analysis
   - Step 1: Points-to Analysis
   - Step 2: Establishing Used-as Relations
3. Implementation
4. Conclusion

Outline

Virtual Machine Introspection (VMI)

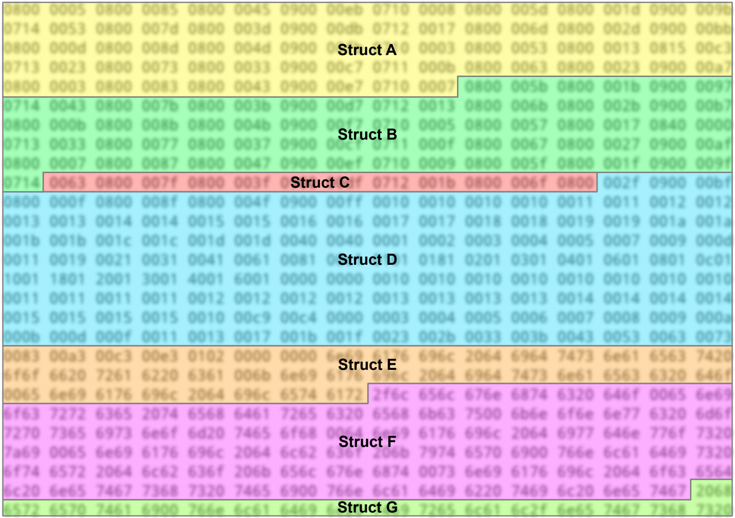
[Garfinkel and Rosenblum(2003)]

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)]



Bridging the Gap: Out-of-Band Delivery

Common Approach: utilize kernel debugging symbols

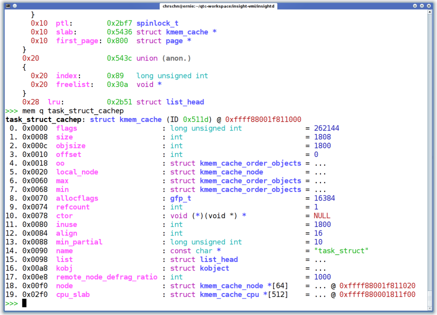
- Use symbols for:
  - Layout and size of kernel data structures
  - Virtual address of global variables and functions
- Emulate virtual-to-physical address translation in software

⇒ 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



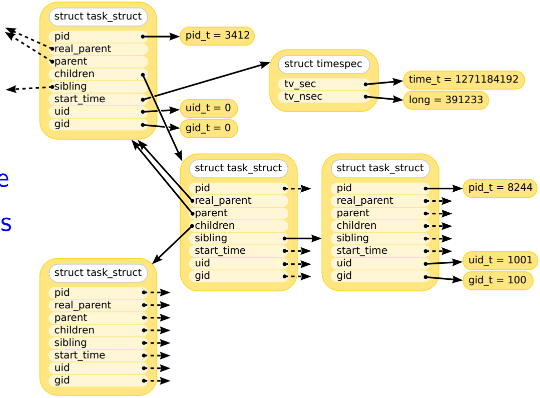
- Works for x86 32 bit (w/ PAE) and 64 bit Linux guests
- Supports any hypervisor providing guest memory access

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...



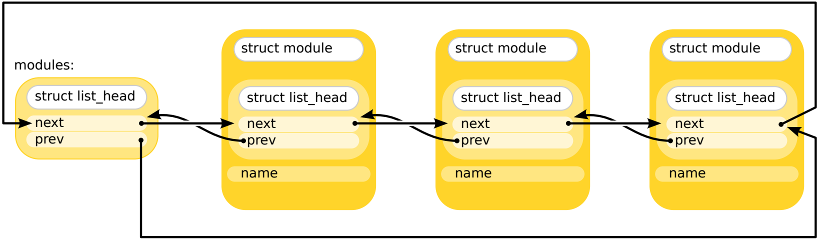
Why debugging symbols are insufficient

```
struct list_head {
    struct list_head *next, *prev;
};

struct module {
    struct list_head list;
    char name[60];
    /* ... */
};

struct list_head modules;

struct module* find_module(const char *name)
{
    struct module *mod;
    list_for_each_entry(mod, &modules, list)
    {
        if (strcmp(mod->name, name) == 0)
        return mod;
    }
    return NULL;
}
```

Why debugging symbols are insufficient (cont.)

```
struct module* find_module(const char * name)
{
    struct module * mod;
    /* Original code: list_for_each_entry(mod, &modules, list) */
    for (mod = ({
            const typeof(((typeof(*mod) *) 0)->list) * __mptr = ((&modules)->next);
            (typeof(*mod) *) ((char *) __mptr - __builtin_offsetof(typeof(*mod), list));
        });
        __builtin_prefetch(mod->list.next), &mod->list != (&modules);
        mod = ({
            const typeof(((typeof(*mod) *) 0)->list) * __mptr = (mod->list.next);
            (typeof(*mod) *) ((char *) __mptr - __builtin_offsetof(typeof(*mod), list));
        }))
    {
        if (strcmp(mod->name, name) == 0)
            return mod;
    }
    return ((void *) 0);
}
```

Why debugging symbols are insufficient (cont.)



Example: lsmod in JavaScript

Manually apply expert knowledge

```
function lsmod()
{
    // type of variable "modules" is list_head
    var head = new Instance("modules");
    var m = head.next;
    m.ChangeType("module");
    // offset for address correction
    var offset = m.MemberOffset("list");
    m.AddToAddress(-offset);
    // correct head as well for loop terminaten
    head.AddToAddress(-offset);
    // iterate over all modules
    do {
        print(m.name + " " + m.args);
        m = m.list.next;
        m.ChangeType("module");
        m.AddToAddress(-offset);
    } while (m && m.Address() != head.Address());
}
```

Summary

Problems

Runtime pointer and type manipluations 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

```c
struct module* find_module(const char * name)
{
    struct module * mod;
    /* Original code: list_for_each_entry(mod, &modules, list) */
    for (mod = ({
            const typeof(((typeof(*mod) *) 0)->list) * __mptr = ((&modules)->next);
            (typeof(*mod) *) ((char *) __mptr - __builtin_offsetof(typeof(*mod), list));
        });
        __builtin_prefetch(mod->list.next), &mod->list != (&modules);
        mod = ({
            const typeof(((typeof(*mod) *) 0)->list) * __mptr = (mod->list.next);
            (typeof(*mod) *) ((char *) __mptr - __builtin_offsetof(typeof(*mod), list));
        }))
    {
        if (strcmp(mod->name, name) == 0)
            return mod;
    }
    return ((void *) 0);
}
```

Prerequisites:

- Kernel debugging symbols
- Pre-processed source code

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;
```

$z \mapsto \{(y + 8 \cdot \mathrm{sizeof}(int))\ \&\ \tilde{\ }\mathrm{0xFF}\}$

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

```
struct A { int value; struct A *next; };
struct B { void *data; }

struct A* func1(struct A *a) { return a; }

struct A* func2()
{
    struct B b;
    struct A a = { 0, b.data };     // initializer (struct)
    struct A *pa = b.data;          // initializer (variable)
    pa = b.data;                    // assignment
    a = *((struct A*)b.data);       // dereference (*)
    ((struct A*)b.data)->value++;   // dereference (->)
    pa = func1(b.data);             // function parameter
    return b.data;                  // return statement
}
```

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

```
function lsmod()
{
    // type of variable "modules" is list_head
    var head = new Instance("modules");
    var m = head.next;
    m.ChangeType("module");
    // offset for address correction
    var offset = m.MemberOffset("list");
    m.AddToAddress(-offset);
    // correct head as well for loop terminaten
    head.AddToAddress(-offset);
    // iterate over all modules
    do {
        print(m.name + " " + m.args);
        m = m.list.next;
        m.ChangeType("module");
        m.AddToAddress(-offset);
    } while (m && m.Address() != head.Address());
}
```

Example: lsmod in JavaScript

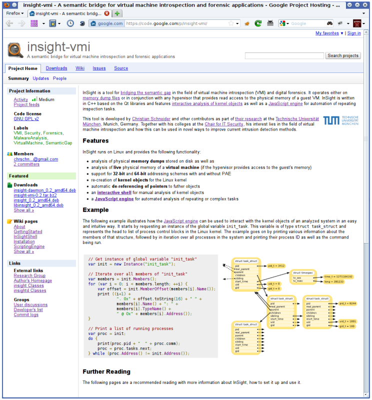
Automatic application of used-as relations

```
function lsmod()
{
    // type of variable "modules" is list_head
    var head = new Instance("modules");
    // iterate over all modules
    var m = head.next;
    while (m.MemberAddress("list") != head.Address()) {
        print(m.name + " " + m.args);
        m = m.list.next;
    }
}
```

Outline

Conclusion

- Used-as analysis captures type usages contradicting declared type
- Extracts arithmetic expression to retrieve target object address
- Extension of InSight mimics dynamic pointer maniplulations 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/`

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.