Automatic Uncovering of Tap Points From Kernel Executions

Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin

University of Texas at Dallas

RAID 2016

Kernel Tap Point

- **An execution point**, e.g.,
  - an instruction
  - a function call
  - a function called in a particular context

  where active kernel execution monitoring, e.g., creation, traversal, or deletion of
  - processes
  - sockets
  - files
  - other kernel objects

  can be performed

Why Uncovering Them

```
sys_fork(){
    ...
    create_process();
    ...
}
```

Why Uncovering Them

```
sys_fork(){
    ...
    create_process();
    ...
}
```

- Increasingly, kernel malware is using the **internal functions** (e.g., `create_process`) to create kernel objects

Why Uncovering Them

```
sys_fork(){
     ...
     create_process();
     ...
}
```

- Increasingly, kernel malware is using the **internal functions** (e.g., `create_process`) to create kernel objects
- Identifying the **internal functions** or **instructions** will be useful in applications:
  - Virtual machine introspection
  - Kernel malware detection
  - Kernel malware profiling

Examples of Kernel Tap Points

Content		**Tap**	**Code**		
Read	**Write**				
			c14f30a0 <schedule>:		
			...		
		c14f33fd	c14f33fd:	mov	-0x58(%ebp),%edx
		c14f3400	c14f3400:	mov	-0x5c(%ebp),%eax
			...		
		c14f3405	c14f3405:	mov	%esp,0x318(%eax)
		c14f340b	c14f340b:	mov	0x318(%edx),%esp
			c14f3411:	movl	$0xc14f3433,0x320(%eax)
			c14f341b:	pushl	0x320(%edx)
			c14f3421:	mov	0x204(%edx),%ebx
			c14f3427:	mov	%ebx,%fs:0xc17f8694
			c14f342e:	jmp	c1001e80 <__switch_to>
			c14f3433:	pop	%ebp

Examples of Kernel Tap Points

Content		Tap	Code
Read	Write		
			`c14f30a0 <schedule>:`
			`...`
		c14f33fd	➡ `c14f33fd:    mov     -0x58(%ebp),%edx`
		c14f3400	`c14f3400:    mov     -0x5c(%ebp),%eax`
			`...`
		c14f3405	`c14f3405:    mov     %esp,0x318(%eax)`
		c14f340b	`c14f340b:    mov     0x318(%edx),%esp`
			`c14f3411:    movl    $0xc14f3433,0x320(%eax)`
			`c14f341b:    pushl   0x320(%edx)`
			`c14f3421:    mov     0x204(%edx),%ebx`
			`c14f3427:    mov     %ebx,%fs:0xc17f8694`
			`c14f342e:    jmp     c1001e80 <__switch_to>`
			`c14f3433:    pop     %ebp`

Examples of Kernel Tap Points

Content		Tap		Code
Read	**Write**			
				`c14f30a0 <schedule>:`
				`...`
`c035dc00`		`c14f33fd`	➡	`c14f33fd:    mov     -0x58(%ebp),%edx`
		`c14f3400`		`c14f3400:    mov     -0x5c(%ebp),%eax`
				`...`
		`c14f3405`		`c14f3405:    mov     %esp,0x318(%eax)`
		`c14f340b`		`c14f340b:    mov     0x318(%edx),%esp`
				`c14f3411:    movl    $0xc14f3433,0x320(%eax)`
				`c14f341b:    pushl   0x320(%edx)`
				`c14f3421:    mov     0x204(%edx),%ebx`
				`c14f3427:    mov     %ebx,%fs:0xc17f8694`
				`c14f342e:    jmp     c1001e80 <__switch_to>`
				`c14f3433:    pop     %ebp`

Examples of Kernel Tap Points

Content Read	 Write	Tap		Code
				`c14f30a0 <schedule>:`
				`...`
`c035dc00`		`c14f33fd`		`c14f33fd:   mov     -0x58(%ebp),%edx`
		`c14f3400`	➡	`c14f3400:   mov     -0x5c(%ebp),%eax`
				`...`
		`c14f3405`		`c14f3405:   mov     %esp,0x318(%eax)`
		`c14f340b`		`c14f340b:   mov     0x318(%edx),%esp`
				`c14f3411:   movl    $0xc14f3433,0x320(%eax)`
				`c14f341b:   pushl   0x320(%edx)`
				`c14f3421:   mov     0x204(%edx),%ebx`
				`c14f3427:   mov     %ebx,%fs:0xc17f8694`
				`c14f342e:   jmp     c1001e80 <__switch_to>`
				`c14f3433:   pop     %ebp`

Examples of Kernel Tap Points

Content		**Tap**		**Code**
Read	**Write**			
				`c14f30a0 <schedule>:`
				`...`
`c035dc00`		`c14f33fd`		`c14f33fd:    mov    -0x58(%ebp),%edx`
`cfe91690`		`c14f3400`	➡	`c14f3400:    mov    -0x5c(%ebp),%eax`
				`...`
		`c14f3405`		`c14f3405:    mov    %esp,0x318(%eax)`
		`c14f340b`		`c14f340b:    mov    0x318(%edx),%esp`
				`c14f3411:    movl   $0xc14f3433,0x320(%eax)`
				`c14f341b:    pushl  0x320(%edx)`
				`c14f3421:    mov    0x204(%edx),%ebx`
				`c14f3427:    mov    %ebx,%fs:0xc17f8694`
				`c14f342e:    jmp    c1001e80 <__switch_to>`
				`c14f3433:    pop    %ebp`

Examples of Kernel Tap Points

Content		**Tap**		**Code**
Read	**Write**			
				`c14f30a0 <schedule>:`
				`...`
`c035dc00`		`c14f33fd`		`c14f33fd:   mov    -0x58(%ebp),%edx`
`cfe91690`		`c14f3400`		`c14f3400:   mov    -0x5c(%ebp),%eax`
				`...`
		`c14f3405`	➡	`c14f3405:   mov    %esp,0x318(%eax)`
		`c14f340b`		`c14f340b:   mov    0x318(%edx),%esp`
				`c14f3411:   movl   $0xc14f3433,0x320(%eax)`
				`c14f341b:   pushl  0x320(%edx)`
				`c14f3421:   mov    0x204(%edx),%ebx`
				`c14f3427:   mov    %ebx,%fs:0xc17f8694`
				`c14f342e:   jmp    c1001e80 <__switch_to>`
				`c14f3433:   pop    %ebp`

Examples of Kernel Tap Points

Content		Tap	Code
Read	Write		
			`c14f30a0 <schedule>:`
			`...`
c035dc00		c14f33fd	`c14f33fd:  mov    -0x58(%ebp),%edx`
cfe91690		c14f3400	`c14f3400:  mov    -0x5c(%ebp),%eax`
			`...`
	c20f0120	c14f3405	`c14f3405:  mov    %esp,0x318(%eax)`
		c14f340b	`c14f340b:  mov    0x318(%edx),%esp`
			`c14f3411:  movl   $0xc14f3433,0x320(%eax)`
			`c14f341b:  pushl  0x320(%edx)`
			`c14f3421:  mov    0x204(%edx),%ebx`
			`c14f3427:  mov    %ebx,%fs:0xc17f8694`
			`c14f342e:  jmp    c1001e80 <__switch_to>`
			`c14f3433:  pop    %ebp`

Examples of Kernel Tap Points

Content		**Tap**	**Code**
Read	**Write**		
			`c14f30a0 <schedule>:`
			`...`
`c035dc00`		`c14f33fd`	`c14f33fd:   mov     -0x58(%ebp),%edx`
`cfe91690`		`c14f3400`	`c14f3400:   mov     -0x5c(%ebp),%eax`
			`...`
	`c20f0120`	`c14f3405`	`c14f3405:   mov     %esp,0x318(%eax)`
		`c14f340b`	➡ `c14f340b:   mov     0x318(%edx),%esp`
			`c14f3411:   movl    $0xc14f3433,0x320(%eax)`
			`c14f341b:   pushl   0x320(%edx)`
			`c14f3421:   mov     0x204(%edx),%ebx`
			`c14f3427:   mov     %ebx,%fs:0xc17f8694`
			`c14f342e:   jmp     c1001e80 <__switch_to>`
			`c14f3433:   pop     %ebp`

Examples of Kernel Tap Points

Content		**Tap**		**Code**
Read	**Write**			
				`c14f30a0 <schedule>:`
				`...`
`c035dc00`		`c14f33fd`		`c14f33fd:    mov     -0x58(%ebp),%edx`
`cfe91690`		`c14f3400`		`c14f3400:    mov     -0x5c(%ebp),%eax`
				`...`
	`c20f0120`	`c14f3405`		`c14f3405:    mov     %esp,0x318(%eax)`
`c24e0fe4`		`c14f340b`	➡	`c14f340b:    mov     0x318(%edx),%esp`
				`c14f3411:    movl    $0xc14f3433,0x320(%eax)`
				`c14f341b:    pushl   0x320(%edx)`
				`c14f3421:    mov     0x204(%edx),%ebx`
				`c14f3427:    mov     %ebx,%fs:0xc17f8694`
				`c14f342e:    jmp     c1001e80 <__switch_to>`
				`c14f3433:    pop     %ebp`

Examples of Kernel Tap Points

Content		Tap
Read	**Write**	
c035dc00		c14f33fd
cfe91690		c14f3400
	c20f0120	c14f3405
c24e0fe4		c14f340b

Switched-to task

Code

```
c14f30a0 <schedule>:
...
c14f33fd:   mov     -0x58(%ebp),%edx
c14f3400:   mov     -0x5c(%ebp),%eax
...
c14f3405:   mov     %esp,0x318(%eax)
c14f340b:   mov     0x318(%edx),%esp
c14f3411:   movl    $0xc14f3433,0x320(%eax)
c14f341b:   pushl   0x320(%edx)
c14f3421:   mov     0x204(%edx),%ebx
c14f3427:   mov     %ebx,%fs:0xc17f8694
c14f342e:   jmp     c1001e80 <__switch_to>
c14f3433:   pop     %ebp
```

Examples of Kernel Tap Points

Content		Tap		Code
Read	Write			
				`c14f30a0 <schedule>:`
				`...`
c035dc00	Switched-to task	c14f33fd		`c14f33fd:    mov    -0x58(%ebp),%edx`
cfe91690	Switched-from task	c14f3400		`c14f3400:    mov    -0x5c(%ebp),%eax`
				`...`
	c20f0120	c14f3405		`c14f3405:    mov    %esp,0x318(%eax)`
c24e0fe4		c14f340b	➡	`c14f340b:    mov    0x318(%edx),%esp`
				`c14f3411:    movl   $0xc14f3433,0x320(%eax)`
				`c14f341b:    pushl  0x320(%edx)`
				`c14f3421:    mov    0x204(%edx),%ebx`
				`c14f3427:    mov    %ebx,%fs:0xc17f8694`
				`c14f342e:    jmp    c1001e80 <__switch_to>`
				`c14f3433:    pop    %ebp`

Why Uncovering the Tap Points is Challenging

1. Large code base of an OS kernel
    - Millions of instructions
    - Hundrends of thousands of functions
    - Tens of thousands of kernel objects
2. Complicated control flow
    - Asynchronized events
        - Interrupts (e.g., timer, keystrokes)
    - Non standard control flow
        - Exceptions (e.g., page fault)

Introducing AUTOTAP

AUTOTAP: a system for AUTOmatic uncovering of TAP points directly from kernel executions.

Introducing AUTOTAP

AUTOTAP: a system for AUTOmatic uncovering of TAP points directly from kernel executions.

Key Approaches

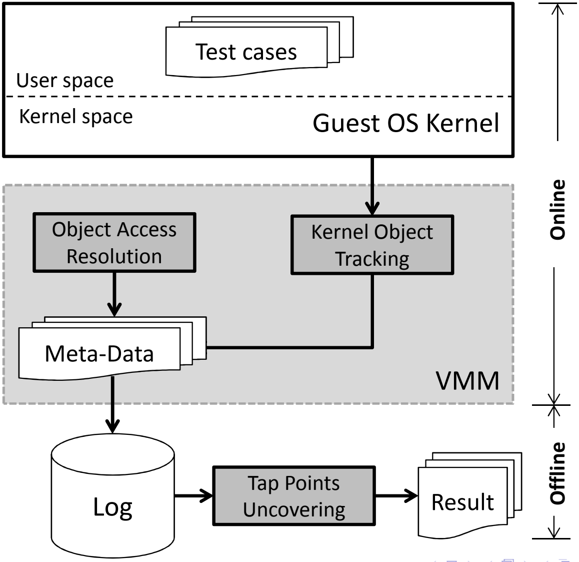
1. **Classifying** the complicated execution contexts into hierarchical structures
2. **Associating** kernel objects with the identified execution context
3. **Deriving** the TAP points based on the execution contexts and the identified kernel objects
    - From object access (read, write, allocation, deallocation, initialize, traversal)
    - From hardware level events (e.g., interrupts)
    - From system call level events

to infer the meaning of instructions and functions

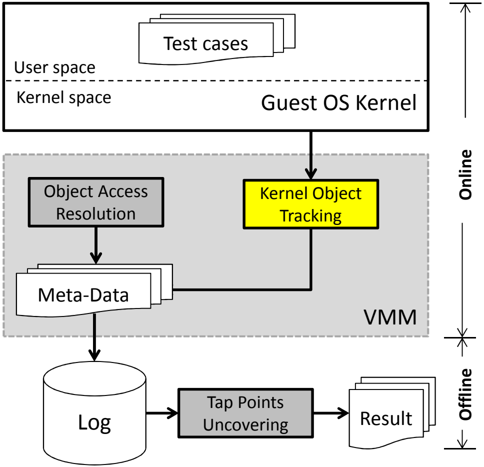
Scope and Assumptions

1. Linux kernel and x86 architecture
2. Assume the knowledge of kernel APIs and its argument types
   - `kmalloc`, `kfree`
   - `kmem_cache_alloc`, `kmem_cache_free`
   - `vmalloc`, `vfree`.
3. Access of (some) header files for kernel driver development (they are open and needed when developing kernel modules)

How AutoTap Works

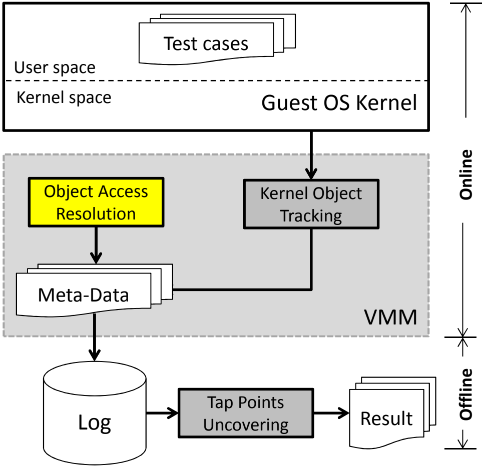


Kernel Object Tracking (ARGOS [ZL15])



1. Tracking the object life time (kmalloc/kfree etc)
2. Assigning a static type to the dynamic object (callsite-chain of kmalloc)
3. Tracking the object size (well-known APIs, header files)
4. Tracking object relations (flow propagation, REWARDS [LZX10])

Object Access Resolution



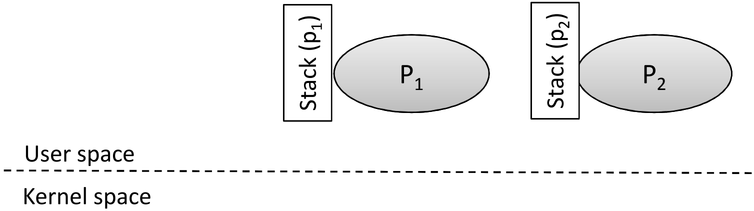
Goal

Identify the specific kernel execution context, when an instruction accessing a monitored object.

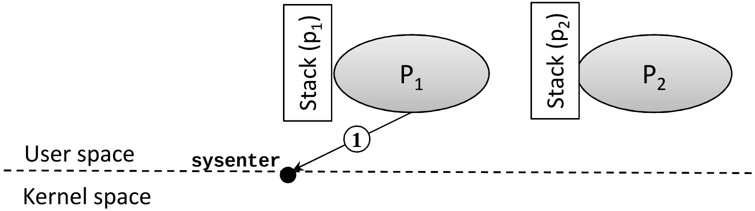
Challenges

1. Context switches
2. Interrupts (bottom half, top half)
3. kernel thread

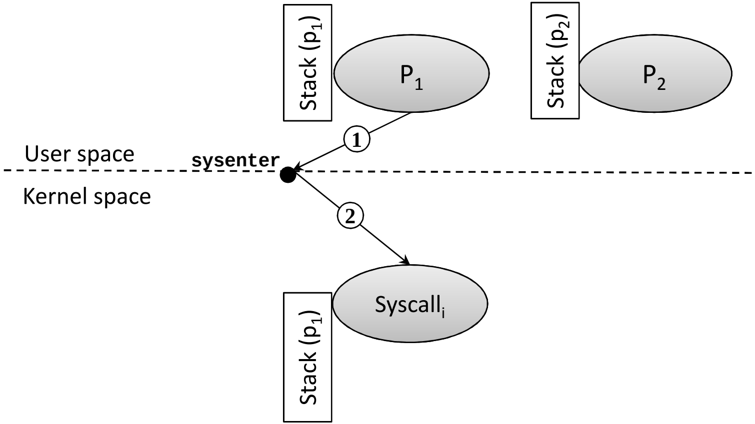
Object Access Resolution



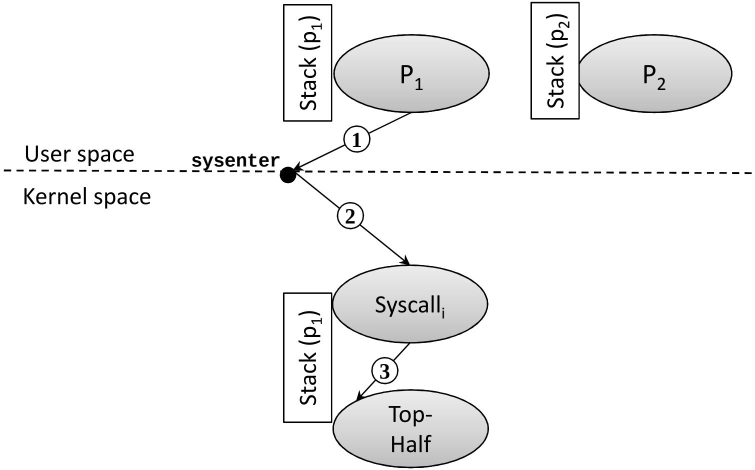
Object Access Resolution



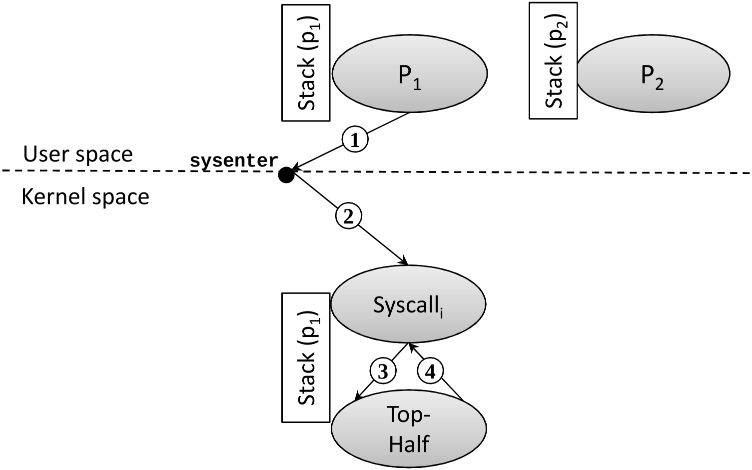
Object Access Resolution



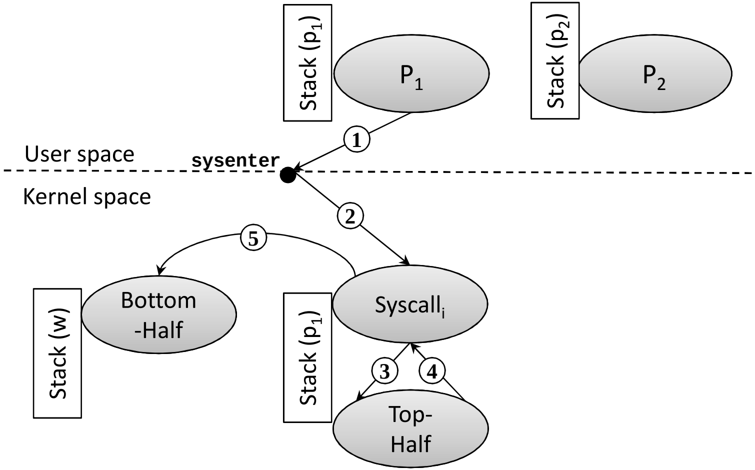
Object Access Resolution



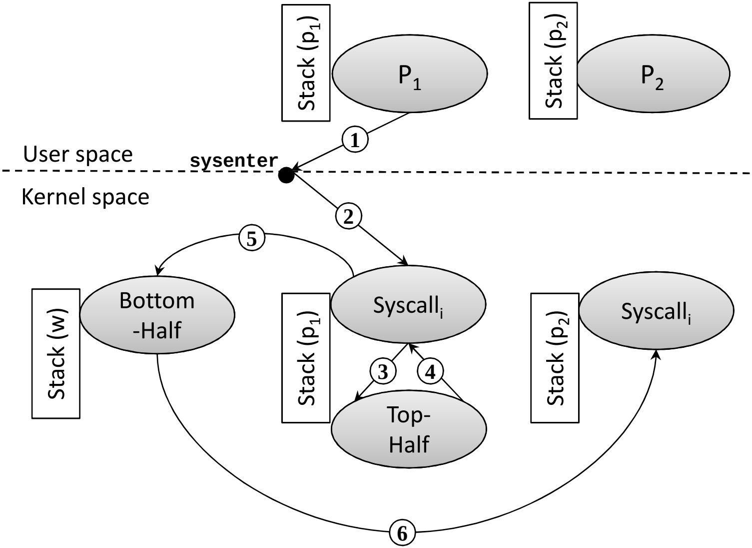
Object Access Resolution



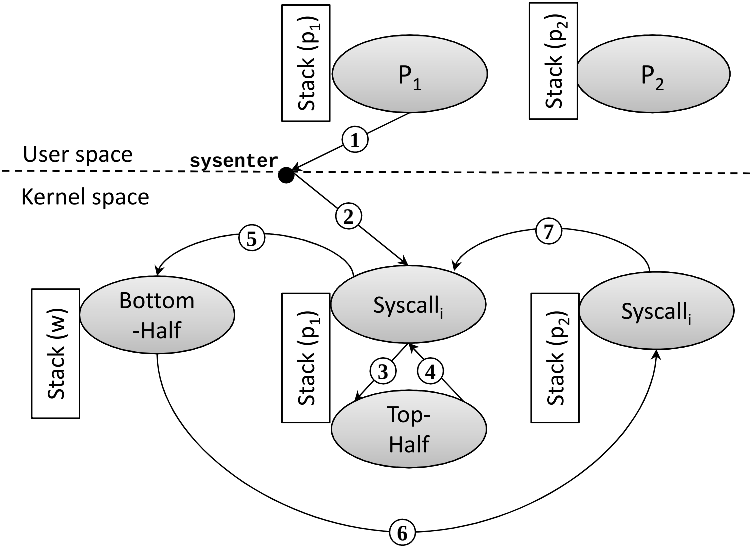
Object Access Resolution



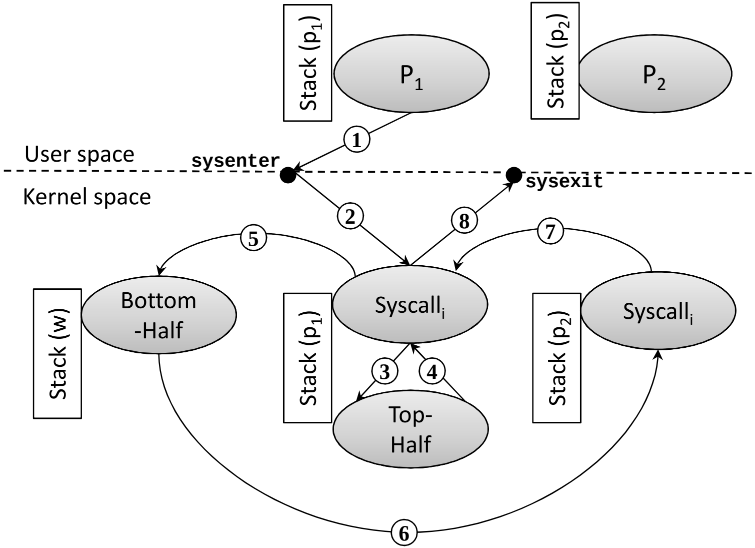
Object Access Resolution



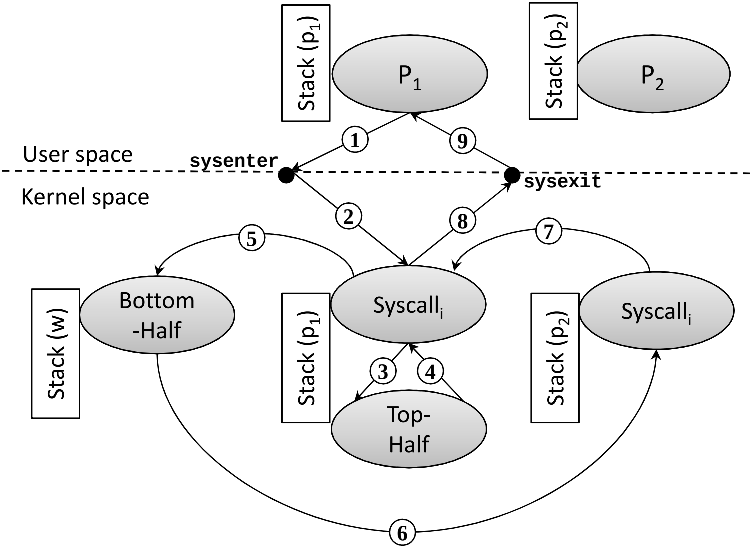
Object Access Resolution



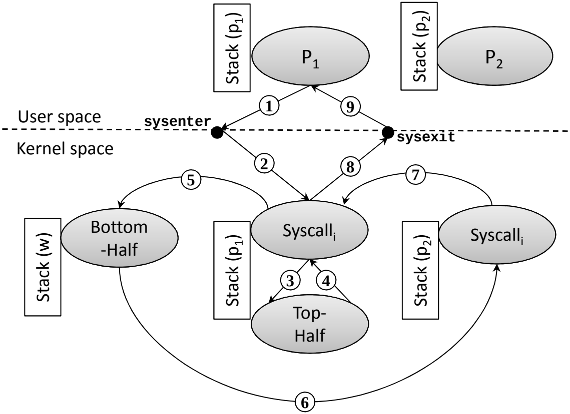
Object Access Resolution



Object Access Resolution



Object Access Resolution



Hierarchy

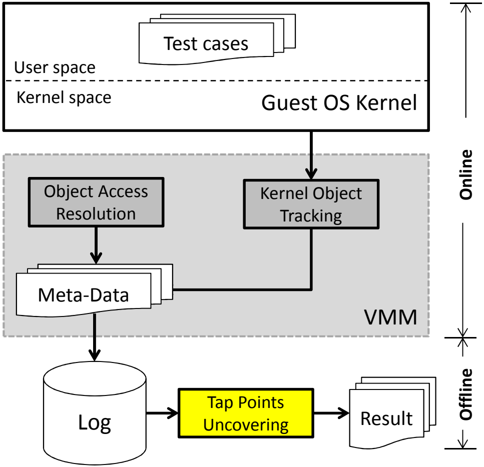
1. Top level
   1. system call
   2. top-half
   3. bottom-half
2. Middle level (function call chain)
3. Lowest level (instructions)

Object Access Resolution

Key Observations

1. Tracking `sysenter/sysexit`, and the `eax` ⇒ system call context
2. Tracking the `esp` changes—context switches need to exchange kernel stack (`esp`) ⇒ context switches
3. Interrupt handler
   - The begining of an interupt handler and the ending `iret` ⇒ top half
   - Kernel stack (`esp`) exchange, no `sysenter` ⇒ bottom half

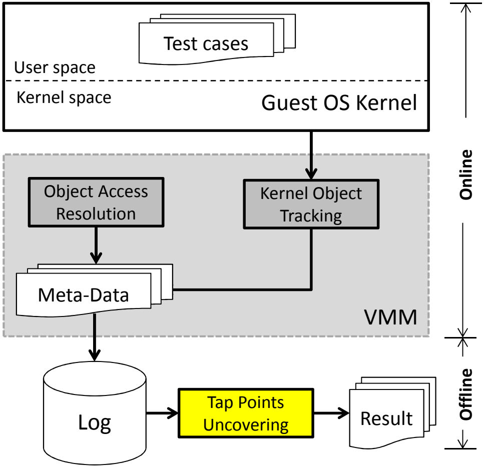
Tap Points Uncovering



Goal

Perform an offline analysis to further derive the tap points for each type of kernel object

Tap Points Uncovering



Goal

Perform an offline analysis to further derive the tap points for each type of kernel object

Tap Points of Interest

1. Object Creation
2. Object Deletion
3. Object Traversal
4. Object Field Read
5. Object Write
6. Object Initialization

Tap Points Uncovering

Category	Behavior
Creation (O_i)	O_i is created by calling `kmalloc`
Deletion (O_i)	O_i is freed by calling `kfree`
Read (O_i, F_j)	A memory read field F_j of O_i
Traversal (O_i, F_j)	Read (O_i, F_j) $\wedge$ $F_j \in$ pointer field
Write (O_i, F_j)	A memory write to field j of O_i
Initialization (O_i, F_j)	Write (O_i, F_j) $\wedge$ first time write to F_j
Others	Other contexts, e.g., periodical access

Table: Resolved access types based on the behavior.

Tap Points Uncovering

Category	Behavior
Creation (O_i)	O_i is created by calling `kmalloc`
Deletion (O_i)	O_i is freed by calling `kfree`
Read (O_i, F_j)	A memory read field F_j of O_i
Traversal (O_i, F_j)	Read (O_i, F_j) $\wedge$ $F_j \in$ pointer field
Write (O_i, F_j)	A memory write to field j of O_i
Initialization (O_i, F_j)	Write (O_i, F_j) $\wedge$ first time write to F_j
Others	Other contexts, e.g., periodical access

Table: Resolved access types based on the behavior.

Tap Points Uncovering

Category	Behavior
Creation (O_i)	O_i is created by calling `kmalloc`
Deletion (O_i)	O_i is freed by calling `kfree`
Read (O_i, F_j)	A memory read field F_j of O_i
Traversal (O_i, F_j)	Read (O_i, F_j) $\wedge$ $F_j \in$ pointer field
Write (O_i, F_j)	A memory write to field j of O_i
Initialization (O_i, F_j)	Write (O_i, F_j) $\wedge$ first time write to F_j
Others	Other contexts, e.g., periodical access

Table: Resolved access types based on the behavior.

Experiment Setup

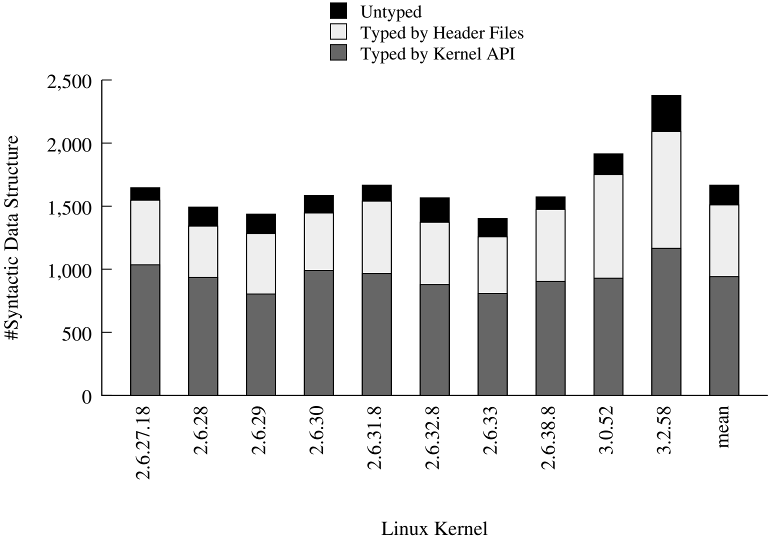
Experiment Environment

- QEMU-1.6.2
- 64-bit Intel Core i-7 CPU with 8GB physical memory
- Host OS: `ubuntu-12.04` with `3.5.0-51-generic`.

Input to AUTOTAP

1. System call specification
2. Kernel API specification
3. Kernel header files
4. Test suites:
    - Linux Kernel Test Suite: `ltp-20140115`
    - User Level: `spec2006`, `lmbench-2alpha8`

Type Resolution Result for Each Kernel



Tap Points for Important Kernel Data Structures

Category	**Semantic Type**	**#Syntactic Type**	*Creation*		*Deletion*		$R_{Traversal}$		$N_{Traversal}$		F_{Read}	
			$\lvert PC \rvert$	$\lvert FC \rvert$	$\lvert PC \rvert$	$\lvert FC \rvert$	$\lvert PC \rvert$	$\lvert FC \rvert$	$\lvert PC \rvert$	$\lvert FC \rvert$	$\lvert PC \rvert$	$\lvert FC \rvert$
Process	task_struct	6	1	0	1	0	98	93	725	6	1024	24
	pid	6	1	0	1	0	2	1	15	3	50	1
	task_delay_info	6	1	0	1	0	0	0	0	0	24	4
	task_xstate	7	2	0	1	0	0	0	0	0	38	1
	taskstats	2	1	0	1	0	0	0	0	0	27	0
Memory	anon_vma	7	1	0	1	0	0	0	5	1	8	1
	mm_struct	4	2	0	1	0	0	0	21	8	235	32
	vm_area_struct	44	7	0	2	0	84	94	113	1	395	1
Network	TCP	3	0	1	0	1	7	0	74	8	1023	137
	UDP	2	0	1	0	1	0	0	0	0	0	84
	UNIX	4	0	1	0	1	8	0	29	4	118	36
	neighbour	7	1	0	1	0	2	0	4	0	113	15
	inet_peer	1	1	0	1	0	0	0	0	0	23	1
	rtable	7	1	0	1	0	0	0	11	0	155	3
	nsproxy	1	1	0	1	0	0	0	1	0	6	0
	request_sock_TCP	2	1	0	1	0	0	0	1	0	70	8
	skbuff_fclone	7	0	1	0	1	0	0	76	78	89	161
	skbuff_head	53	1	1	0	1	1	0	152	78	148	161
	sock_alloc	4	1	0	1	0	0	4	64	2	59	34

Table: The statistics for the uncovered tap points for the observed semantic types of `linux-2.6.32.8` in slab/slub allocators

Tap Points for Important Kernel Data Structures

Category	Semantic Type	#Syntactic Type	*Creation*		*Deletion*		$R_{Traversal}$		$N_{Traversal}$		F_{Read}	
			$\lvert PC\rvert$	$\lvert FC\rvert$	$\lvert PC\rvert$	$\lvert FC\rvert$	$\lvert PC\rvert$	$\lvert FC\rvert$	$\lvert PC\rvert$	$\lvert FC\rvert$	$\lvert PC\rvert$	$\lvert FC\rvert$
Process	task_struct	6	1	0	1	0	98	93	725	6	1024	24
	pid	6	1	0	1	0	2	1	15	3	50	1
	task_delay_info	6	1	0	1	0	0	0	0	0	24	4
	task_xstate	7	2	0	1	0	0	0	0	0	38	1
	taskstats	2	1	0	1	0	0	0	0	0	27	0
Memory	anon_vma	7	1	0	1	0	0	0	5	1	8	1
	mm_struct	4	2	0	1	0	0	0	21	8	235	32
	vm_area_struct	44	7	0	2	0	84	94	113	1	395	1
Network	TCP	3	0	1	0	1	7	0	74	8	1023	137
	UDP	2	0	1	0	1	0	0	0	0	0	84
	UNIX	4	0	1	0	1	8	0	29	4	118	36
	neighbour	7	1	0	1	0	2	0	4	0	113	15
	inet_peer	1	1	0	1	0	0	0	0	0	23	1
	rtable	7	1	0	1	0	0	0	11	0	155	3
	nsproxy	1	1	0	1	0	0	0	1	0	6	0
	request_sock_TCP	2	1	0	1	0	0	0	1	0	70	8
	skbuff_fclone	7	0	1	0	1	0	0	76	78	89	161
	skbuff_head	53	1	1	0	1	1	0	152	78	148	161
	sock_alloc	4	1	0	1	0	0	4	64	2	59	34

Table: The statistics for the uncovered tap points for the observed semantic types of `linux-2.6.32.8` in slab/slub allocators

Tap Points for Important Kernel Data Structures

Category	**Semantic Type**	**#Syntactic Type**	*Creation*		*Deletion*		$R_{Traversal}$		$N_{Traversal}$		F_{Read}	
			PC	*FC*	*PC*	*FC*	*PC*	*FC*	*PC*	*FC*	*PC*	*FC*
File	bio-0	94	0	1	0	1	3	0	18	0	123	30
	biovec-16	5	0	1	0	1	0	0	0	0	0	26
	biovec-64	4	0	1	0	1	0	0	0	0	1	30
	io_context	17	1	0	1	0	0	0	7	2	15	7
	request	60	0	1	0	1	13	99	22	0	164	2
	dentry	85	1	0	1	0	80	4	321	4	197	10
	ext2_inode_info	4	1	0	1	0	6	17	74	12	136	262
	ext3_inode_info	21	1	0	1	0	6	19	38	35	580	348
	fasync_struct	1	1	0	1	0	0	0	1	0	1	1
	file_lock	10	1	0	1	0	11	6	17	0	113	3
	files_struct	4	1	0	1	0	0	3	25	10	41	41
	file	33	1	0	1	0	4	5	227	7	352	4
	fs_struct	4	1	0	1	0	0	0	9	2	44	3
	inode	5	1	0	1	0	2	5	5	8	15	113
	journal_handle	124	1	0	1	0	0	0	28	0	25	0
	journal_head	82	1	0	1	0	19	0	66	0	50	0
	proc_inode	9	1	0	1	0	0	0	6	3	33	95
	sysfs_dirent	36	1	0	1	0	12	0	7	0	31	0
	vfsmount	4	1	0	1	0	31	0	21	8	63	3

Table: The statistics for the uncovered tap points for the observed semantic types of `linux-2.6.32.8` in slab/slub allocators

Tap Points for Important Kernel Data Structures

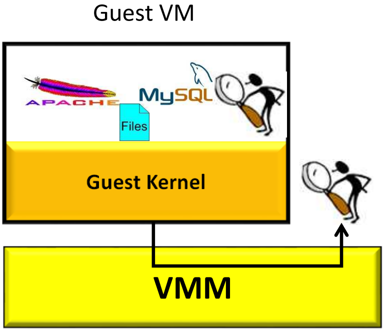
Category	Semantic Type	#Syntactic Type	*Creation*		*Deletion*		$R_{Traversal}$		$N_{Traversal}$		F_{Read}	
			$\|PC\|$	$\|FC\|$	$\|PC\|$	$\|FC\|$	$\|PC\|$	$\|FC\|$	$\|PC\|$	$\|FC\|$	$\|PC\|$	$\|FC\|$
IPC	mqueue_inode_info	1	1	0	1	0	0	0	15	2	37	49
	shmem_inode_info	8	1	0	1	0	0	4	0	16	107	194
Signal	fsnotify_event	19	1	0	1	0	1	0	8	2	24	2
	inotify_event_private_data	19	2	0	1	0	0	0	3	0	2	0
	inotify_inode_mark_entry	1	1	0	1	0	1	0	7	1	25	1
	sighand_struct	6	1	0	1	0	0	0	0	0	66	4
	signal_struct	6	1	0	1	0	0	12	11	4	265	36
	sigqueue	17	1	0	1	0	4	2	8	2	8	0
Security	cred	41	2	0	1	0	0	3	28	3	352	1
	key	4	1	0	1	0	0	10	4	0	53	3
Other	buffer_head	61	1	0	1	0	20	0	21	0	423	0
	cfq_io_context	17	1	0	1	0	2	0	15	3	39	1
	cfq_queue	15	1	0	1	0	0	0	17	5	106	1
	idr_layer	12	1	0	3	0	5	5	1	3	19	3
	names_cache	58	2	0	3	0	0	0	0	0	16	10
	k_itimers	1	1	0	1	0	1	0	12	0	24	24
	radix_tree_node	56	1	0	1	0	10	3	2	3	22	9
	jbd_revoke_record_s	14	1	0	1	0	1	0	0	0	7	0

Table: The statistics for the uncovered tap points for the observed semantic types of `linux-2.6.32.8` in slab/slub allocators

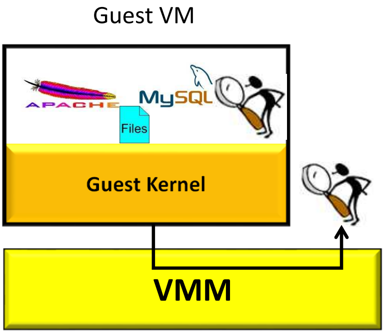
Applications—Hidden Process Identification

- Providing invisible service to attackers
- Typical approaches to hide a process:
  1. Modifying `ps/pslist` binary
  2. Modifying the system libraries (e.g., `glibc`), dynamic linker structures (`plt/got` table), system call tables, or corresponding operating system functions that report system status
  3. Direct kernel object manipulation (DKOM).

How to detect the hidden process?



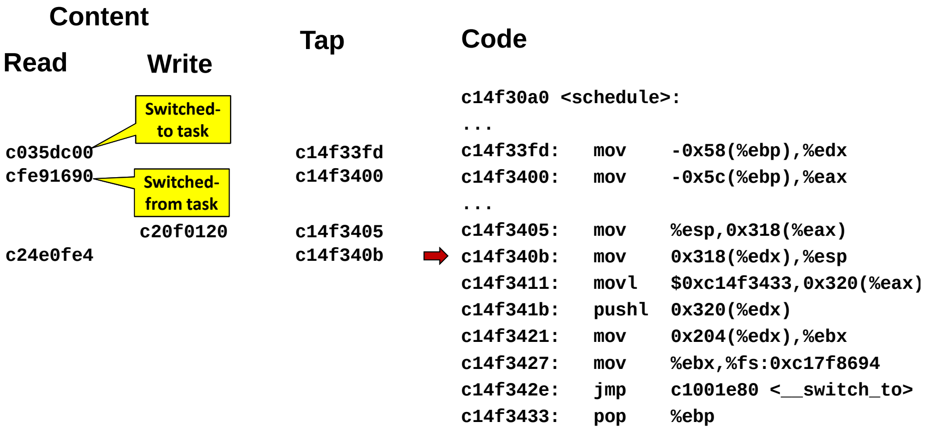
How to detect the hidden process?



CPU time metric

- The most reliable source (tamper-proof) for rootkit detection.
- How to get the CPU execution time for a process using the tap points?

The Tap Points Catching the CPU Execution



Content		Tap
Read	Write	
c035dc00		c14f33fd
cfe91690		c14f3400
	c20f0120	c14f3405
c24e0fe4		c14f340b

Switched-to task

Switched-from task

Code

```
  c14f30a0 <schedule>:
  ...
  c14f33fd:    mov     -0x58(%ebp),%edx
  c14f3400:    mov     -0x5c(%ebp),%eax
  ...
  c14f3405:    mov     %esp,0x318(%eax)
➡ c14f340b:    mov     0x318(%edx),%esp
  c14f3411:    movl    $0xc14f3433,0x320(%eax)
  c14f341b:    pushl   0x320(%edx)
  c14f3421:    mov     0x204(%edx),%ebx
  c14f3427:    mov     %ebx,%fs:0xc17f8694
  c14f342e:    jmp     c1001e80 <__switch_to>
  c14f3433:    pop     %ebp
```

Tested Rootkit

Rootkits	Process Hiding Mechanism	Detected?
ps_hide	Fake `ps` binary with process hiding function	✓
libprocesshider	Override `glibc`'s `readdir` to hide process	✓
LinuxFu	Hide the process by deleting its `task_struct` from task list	✓

Table: Process Hiding Rootkits

Limitation and Future Work

1. The effectiveness relies on coverage of the dynamic analysis
2. Only a few types of TAP points (e.g., creation, deletion, read, write, and traversal) are supported
3. Only demonstrated our techniques with Linux Kernel and need to test with other kernels (FreeBSD, Windows, etc.)

Related Works

Tap Points Uncovering

1. TZB [DGLHL13]: Mining (`memgrep`) the memory access points for user level applications, to identify the places for active monitoring

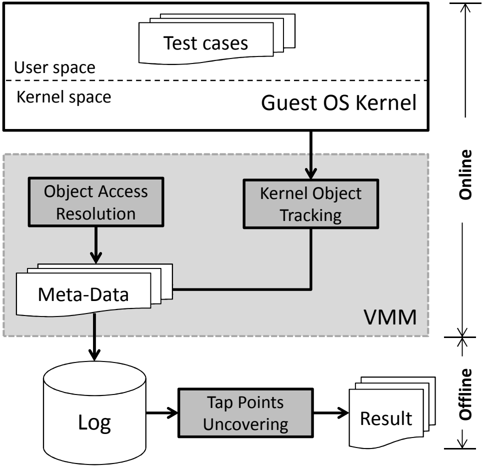
Data Structure Reverse Engineering

1. Aggregate structure identification (ASI) [RFT99], value set analysis (VSA) [BR04, RB08]
2. Laika [CSXK08], REWARDS [LZX10], TIE [LAB11], Howard [SSB11], ARGOS [ZL15], and PointerScope [ZPL+12]

Virtual Machine Introspection

1. VMI [GR03]
2. Hidden process detection (e.g, [JWX07, JADAD08, DGLZ+11])

Summary: AUTOTAP



1. The first system to infer kernel tap points from execution
2. Starting from syscall, exported kernel APIs, data structure definitions
3. Tracking kernel objects, resolving kernel execution context and associating them
4. Deriving TAP points based on how kernel objects get accessed

Thank you

References I

- Gogul Balakrishnan and Thomas Reps, Analyzing memory accesses in x86 executables, CC, Mar. 2004.
- Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King, Digging for data structures, Proceeding of 8th Symposium on Operating System Design and Implementation (OSDI'08) (San Diego, CA), December, 2008, pp. 231–244.
- Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee, Tappan zee (north) bridge: Mining memory accesses for introspection, Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2013.
- Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee, Virtuoso: Narrowing the semantic gap in virtual machine introspection, Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland, CA, USA), 2011, pp. 297–312.
- Yangchun Fu and Zhiqiang Lin, Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection, Proceedings of 33rd IEEE Symposium on Security and Privacy, May 2012.
- ______, Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery, Proceedings of the Ninth Annual International Conference on Virtual Execution Environments (Houston, TX), March 2013.
- Tal Garfinkel and Mendel Rosenblum, A virtual machine introspection based architecture for intrusion detection, Proceedings Network and Distributed Systems Security Symposium (NDSS'03), February 2003, pp. 38–53.
- Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, Vmm-based hidden process detection and identification using lycosid, Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (Seattle, WA, USA), VEE '08, ACM, 2008, pp. 91–100.

References II

Xuxian Jiang, Xinyuan Wang, and Dongyan Xu, Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction, Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07) (Alexandria, Virginia, USA), ACM, 2007, pp. 128–138.

JongHyup Lee, Thanassis Avgerinos, and David Brumley, Tie: Principled reverse engineering of types in binary programs, Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11) (San Diego, CA), February 2011.

Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu, Automatic reverse engineering of data structures from binary execution, Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10) (San Diego, CA), February 2010.

Thomas W. Reps and Gogul Balakrishnan, Improved memory-access analysis for x86 executables, Proceedings of International Conference on Compiler Construction (CC'08), 2008, pp. 16–35.

G. Ramalingam, John Field, and Frank Tip, Aggregate structure identification and its application to program analysis, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL'99) (San Antonio, Texas), ACM, 1999, pp. 119–132.

Asia Slowinska, Traian Stancescu, and Herbert Bos, Howard: A dynamic excavator for reverse engineering data structures, Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11) (San Diego, CA), February 2011.

Junyuan Zeng and Zhiqiang Lin, Towards automatic inference of kernel object semantics from binary code, Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15) (Kyoto, Japan), November 2015.

References III

Mingwei Zhang, Aravind Prakash, Xiaolei Li, Zhenkai Liang, and Heng Yin, Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis, Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12) (San Diego, CA), February 2012.