CS 6V81-05

Kruiser: Semi-synchronized Non-blocking Concurrent Kernel
Heap Buffer Overflow Monitoring

Donghai Tian, Qiang Zeng, Dinghao Wu, Peng Liu, Changzhen Hu

Presenter: Shishir K Prasad
Department of Computer Science
University of Texas at Dallas

March 28th, 2012
Outline

1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
Outline

1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
The Problem

**Problem**
- Kernel heap buffer overflow vulnerabilities
- Few practical counter-measures present

**Previous Solutions**
- Security enforcement inlined into kernel execution
- Performance concerns
- Compatibility issues with mainstream kernels
Outline

1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync'd Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
Kruiser

Overview

- Concurrent kernel heap buffer overflow monitor
- Enforcement not inlined with kernel execution but performed in a concurrent monitor process
- Taking advantage of the now popular multi-core architectures
- Virtualization for security and performance isolation of the monitor process
High-level Idea

Key Concepts

- Place a secure canary word at the end of every heap buffer and check its integrity
- If a canary is found to be tampered, buffer overflow is detected
- *Cruising*: Infinite loop to check kernel heap space.
- Semi-sync’d non-blocking monitoring algorithm using a lock-free data structure
- Meta-data maintained is page-level, rather than for each individual buffer
- Out-of-VM monitor and In-VM meta-data
Key Concepts
1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
Challenges & Solutions

Synchronization

- Lock based approach
- Lock free approach
## Challenges & Solutions

### Synchronization
- Lock based approach
- Lock free approach

### Self-protection & Canary counterfeit
- Virtualization
- DMM for efficiency
- SIM (Secure In VM) framework
- Cryptographic canaries
### Challenges & Solutions

#### Synchronization
- Lock based approach
- Lock free approach

#### Self-protection & Canary counterfeit
- Virtualization
- DMM for efficiency
- SIM (Secure In VM) framework
- Cryptographic canaries

#### Compatibility
- Hook only the alloc/dealloc routines of the kernel
- Custom data structure for holding meta-data
Outline

1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
Page Identity Array

- What makes up a page’s identity?
  - Signature, Access Control, Accounting...
  - Page frame used for heap or not
  - If yes, then meta-data used to locate canaries
Page Identity Array

- What makes up a page’s identity?
  - Signature, Access Control, Accounting...
  - Page frame used for heap or not
  - If yes, then meta-data used to locate canaries

- Size of PIA is fixed because kernel memory address space is fixed

- Insight into kernel heap management
  - A kernel page used for heap is divided into buffer objects of equal size
  - All buffers in this page are arranged as an array
  - Given a heap page and the address of the initial buffer object, all other buffers can be located
  - Thus meta-data to be stored in quite small
Page Identity Array

- What makes up a page’s identity?
  - Signature, Access Control, Accounting...
  - Page frame used for heap or not
  - If yes, then meta-data used to locate canaries
- Size of PIA is fixed because kernel memory address space is fixed
- Insight into kernel heap management
  - A kernel page used for heap is divided into buffer objects of equal size
  - All buffers in this page are arranged as an array
  - Given a heap page and the address of the initial buffer object, all other buffers can be located
  - Thus meta-data to be stored in quite small
- Adding a page to the heap pool
  - Canaries within the page are initialized
  - PIA entry is updated
- Flat PIA structure may not be scalable for 64bit systems
Outline

1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
Race Conditions

- Two processes access PIA: Monitor and process adding/removing pages from the heap pool
- Non-atomic operations on the PIA entries
- W-W and R-W hazards
Race Conditions

- Two processes access PIA: Monitor and process adding/removing pages from the heap pool
- Non-atomic operations on the PIA entries
- W-W and R-W hazards
- TOCTTOU: Time Of Check To Time Of Use
  - Step1: If page is in heap pool then goto step 2; else go to next page
  - Step2: check canaries within the page
  - Answer: Double-check mechanism

ABA hazard
Read value A from some location, perform tests
In order to double check, read that location again
Some other process might have changed the value from A to B and back to A within this time
Race Conditions

- Two processes access PIA: Monitor and process adding/removing pages from the heap pool
- Non-atomic operations on the PIA entries
- W-W and R-W hazards
- TOCTTOU: Time Of Check To Time Of Use
  - Step1: If page is in heap pool then goto step 2; else go to next page
  - Step2: check canaries within the page
  - Answer: Double-check mechanism
- ABA hazard
  - Read value A from some location, perform tests
  - In order to double check, read that location again
  - Some other process might have changed the value from A to B and back to A within this time
Outline

1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
Semi-sync'd Non-blocking Cruising

- A word-sized "Version Number"
  - If page is not in heap pool, even number
  - If page is in heap pool, odd number
Semi-sync’d Non-blocking Cruising

- A word-sized "Version Number"
  - If page is not in heap pool, even number
  - If page is in heap pool, odd number

- Avoiding concurrent PIA entry updates
  - Hop on the kernel for a *free-ride*
  - PIA entry update operations interposed into the kernel’s add/remove page routines
  - PIA meta-data entry update is fast
Semi-sync’d Non-blocking Cruising

- A word-sized "Version Number"
  - If page is not in heap pool, even number
  - If page is in heap pool, odd number

- Avoiding concurrent PIA entry updates
  - Hop on the kernel for a free-ride
  - PIA entry update operations interposed into the kernel’s add/remove page routines
  - PIA meta-data entry update is fast

- Avoiding use of inconsistent entry value
  - Use a double-check algorithm
  - Step1: Read the version number, then read other entry fields
  - Step2: Read version number again
  - Scan page only if two reads return the same odd number
Semi-sync’d Non-blocking Cruising

- Wait-free monitoring
  - Simple reads, writes and memory barriers
  - Monitor process is lightly sync’d via version numbers
  - Other processes are not blocked by the monitor
  - Implies one-way synchronization

- W-W is sync’d using the kernel’s critical sections

- R-W is not sync’d **but double-checking is used**
Outline

1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
Outline

1 The Problem
2 Kruiser
3 High-level Idea
4 Challenges & Solutions
5 Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6 System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
Background

- Slab allocator in Linux
- 2 types of caches: general & specific
- A cache is made up of $\geq 1$ slabs
- Each slab occupies $\geq 1$ contiguous pages in memory
- A slab contains buffer objects of the same type
Outline

1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
Architecture

- VMM: Memory Mapping
- Dom0 VM: Monitor process runs here
- DomU VM
  - The **monitored** kernel runs here
  - The PIA & interposition also reside here, protected using SIM
Outline

1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
Direct Memory Mapping

- VMI leads to high overhead due to frequent calls to the VMM
- DMM enables only one-time involvement of the VMM
- Basic idea: Modify monitor’s page tables
- Stage 1
  - Monitor allocates a chunk of memory
  - Thus it obtains a contiguous range of virtual addresses
  - Max size = 896MB on 32bit systems
  - Access the allocated memory to force creation of PTEs
Direct Memory Mapping

- VMI leads to high overhead due to frequent calls to the VMM
- DMM enables only one-time involvement of the VMM
- Basic idea: Modify monitor’s page tables

Stage 1
- Monitor allocates a chunk of memory
- Thus it obtains a contiguous range of virtual addresses
- Max size = 896MB on 32bit systems
- Access the allocated memory to force creation of PTEs

Stage 2
- Custom driver reclaims allocated pages but retains the PTEs
- Saves a lot of memory
Stage 3

- Custom driver informs Memory Mapper to perform DMM
- Lookup DomU’s P2M table and collect MFNs
- Modify monitor’s PTEs by changing the PFNs to the obtained MFNs
DMM continued

- **Stage 3**
  - Custom driver informs Memory Mapper to perform DMM
  - Lookup DomU’s P2M table and collect MFNs
  - Modify monitor’s PTEs by changing the PFNs to the obtained MFNs

- **Stage 4**
  - Once PIA is initialized in DomU, VMM is notified via hypercall
  - VMM then notifies monitor process to start cruising
DMM Conceptualization

Monitor Process

Target OS Kernel

Machine Physical Memory

Page directory

Page table
Outline

1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
SIM Framework

- Placing data and code in another VM/hypervisor => performance overhead
- Partition the DomU address space: Kernel & Monitor address space
SIM Framework

- Placing data and code in another VM/hypervisor => performance overhead
- Partition the DomU address space: Kernel & Monitor address space
- Interposition code and meta-data placed in *monitor address space*
- Access monitor address space *only via protected gates*
- Process in kernel address space cannot access monitor address space and vice versa
SIM Framework

- Placing data and code in another VM/hypervisor => performance overhead
- Partition the DomU address space: Kernel & Monitor address space
- Interposition code and meta-data placed in monitor address space
- Access monitor address space only via protected gates
- Process in kernel address space cannot access monitor address space and vice versa
- Necessitates creation of SPTs with different access permissions for kernel and In-VM monitor
- Transition code is executable in both regions and it modifies the CR3 register => change root of target SPT
SIM Framework

Kernel Address Space

- **R** Kernel Code
- **X**

- **R** Kernel Data
- **W**

- Monitor Code

- Monitor Data

- **R** Transition Code
- **X**

Monitor Address Space

- **R** Kernel Code

- **R** Kernel Data
- **W**

- Monitor Code
- **W**

- Monitor Data
- **X**

- **R** Transition Code
- **X**
Outline

1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
Placement of Canaries

- Placing 2 canaries surrounding each buffer results in mis-alignment in L1 cache
- Only one canary used. Underflow can still be detected (except for 1st object in the slab)
- Secure Canary generation

![Diagram showing placement of canaries in specific and general cache contexts.](image)
Outline

1. The Problem
2. Kruiser
3. High-level Idea
4. Challenges & Solutions
5. Kernel Cruising
   - Page Identity Array
   - Race Conditions
   - Semi-sync’d Non-blocking Cruising
6. System Design & Implementation
   - Background
   - Architecture
   - Direct Memory Mapping
   - In-VM Protection
   - Placing Canaries
   - The PIA structure
struct PIA_entry{
    unsigned int version;
    short mem; // the starting address of the first object
    short slab_size; // the size of the slab descriptor
    int obj_size; // the actual size used by each object
    int buffer_size; // the whole size for each object
    int number; // the number of objects in this slab
    long key; // the key for canary verification
};
References

2. P2M, PFN, MFN: http://lxr.free-electrons.com/source/arch/x86/xen/p2m.c