Defeating Dynamic Data Kernel Rootkit Attacks via VMM-based Guest-Transparent Monitoring

Kernel Rootkits

- Target the OS kernels
- Hard to protect against and detect, because they target the core root of trust.
- Common techniques for hiding
  - Hijack control flow
    - Injection and execution of malicious code at kernel level
  - Manipulate kernel states

Dynamic data attack

- Modify dynamically allocated kernel data structures
- Without using injected code
- For example: /dev/kmem
- Techniques to prevent kernel code injection are ineffective
  - No kernel code may be modified!

Attack examples

- Shutting down SELinux
  - Change selinux_ops structure
- Resource status manipulation
  - Change data structures reporting network usage status to ifconfig
- Process Hiding
  - Remove process information from internal kernel tables.

Contribution

- System to prevent dynamic data attacks on kernel
- Actively intercept and prevent malicious modifications to critical kernel data
- Challenge:
  - Kernel data is dynamic in both location and content
  - How do you tell if modifications are valid or not?

Approach

- Monitor the execution of OS at the instruction-level from the VMM.
- Provide VMM with
  - a higher-level view of the kernel's memory
  - Policies to manage kernel memory
- At runtime,
  - Monitor key data structures
  - Enforce policies upon every write

Assumption

- Monitoring begins AFTER the OS has booted up and reached a stable state
- Otherwise too many changes to critical kernel data structures

What to monitor and protect?

- Location of data
  - static and dynamic
- Content of data
  - Invariant vs variant
- Number of instances
  - fixed and changing
- Most rootkits target static location, invariant content, and a fixed number of instances

Challenge: Semantic Gap

- VMM only has access to raw OS memory
- But to protect the OS, VMM needs to understand its internal structure
- This is the semantic gap.
- Means VMM needs extra help to bridge this gap
- Two primitives to bridge this gap:
  - Memoryguard
  - Watchpoint

Memoryguard

- Guard a data structure at a given range of memory address
- Any write to the guarded region is intercepted and validated by the VMM
- Similar to page-table protections, except that
- Memoryguard can apply more complex policies

Watchpoint

- Used to track pointer state changes that affect regions protected by memoryguard
- If the pointer moves, detect and update the corresponding memoryguard

Challenge in protecting dynamic data

- Which writes are valid and which aren't?
- Access invariant property:
  - For each data structure, explicitly enumerate all kernel functions that are allowed to/prohibited from modifying the data.
- Assumption: Allowed set of functions is fixed during the lifetime of the system

```
Input: A write address (addr), CPU instruction pointer (EIP)
1: if the update is scheduled then
2:    UpdateWatchPrimitives(GetScheduledTag())
3: end if
4: (prot, tag) ⇐ IsProtected (addr)
5: if prot = NotProtected then
6:    {allow the memory write on unprotected memory}
7: else
8:    if IsProhibitedCode (addr, EIP) then
9:       print "Code EIP attacks the target at addr."
10:      {prevent the memory write to addr by EIP}
11:   else
12:      if prot = WatchPoint then
13:         schedule the update of watch primitives with tag
14:      end if
15:      {allow the benign memory write}
16:   end if
17: end if
```

Fig. 1. An algorithm to verify a kernel memory write

Implementation

- QEMU/KVM platform
- Dynamic compiler to translate guest instructions to host instructions
- Cache of recently recompiled and used blocks
- Interception code implemented in cache

Preventing /dev/kmem attacks

- Invokes __generic_copy_from_user
- Policy: Disallow modifications from __generic_copy_from_user
- SELinux attack:
  - protect security_ops pointer, which is declared statically
- NIC data manipulation
  - Protect net_device_stats structure
- Process hiding attack:
  - Protect init_task_union which is the head of task struct list
  - Protect each task_struct with a memory guard

Optimizing Performance

- Allow user code to execute natively.
  - KQEMU acceleration module
- Only kernel code is intercepted and monitored
  - Every instruction in kernel code?

Runtime Overhead

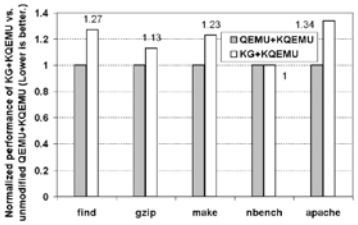

Fig. 3. Performance results of KG with KQEMU support compared to unmodified QEMU with KQEMU support. Results are normalized with respect to the unmodified VMM. In the apache benchmark, time per request is used. In the other benchmarks, the total execution time is used.

Key trick and limitations

- Main idea is the Access Invariant Concept
  - Set of functions allowed to modify a data structure is limited and does not change
  - Allow modifications from known good functions.
- Limitation:
  - Cannot detect malicious modifications by known good functions.
  - Mechanism needs to be very conservative.
    - Impact of false positives (denying valid write operations) can be incorrect kernel execution.
    - Better to suspend the VM and inform sysadmin with diagnostic information.