End to End Defense against Rootkits in Cloud Environment

Design- Part 2

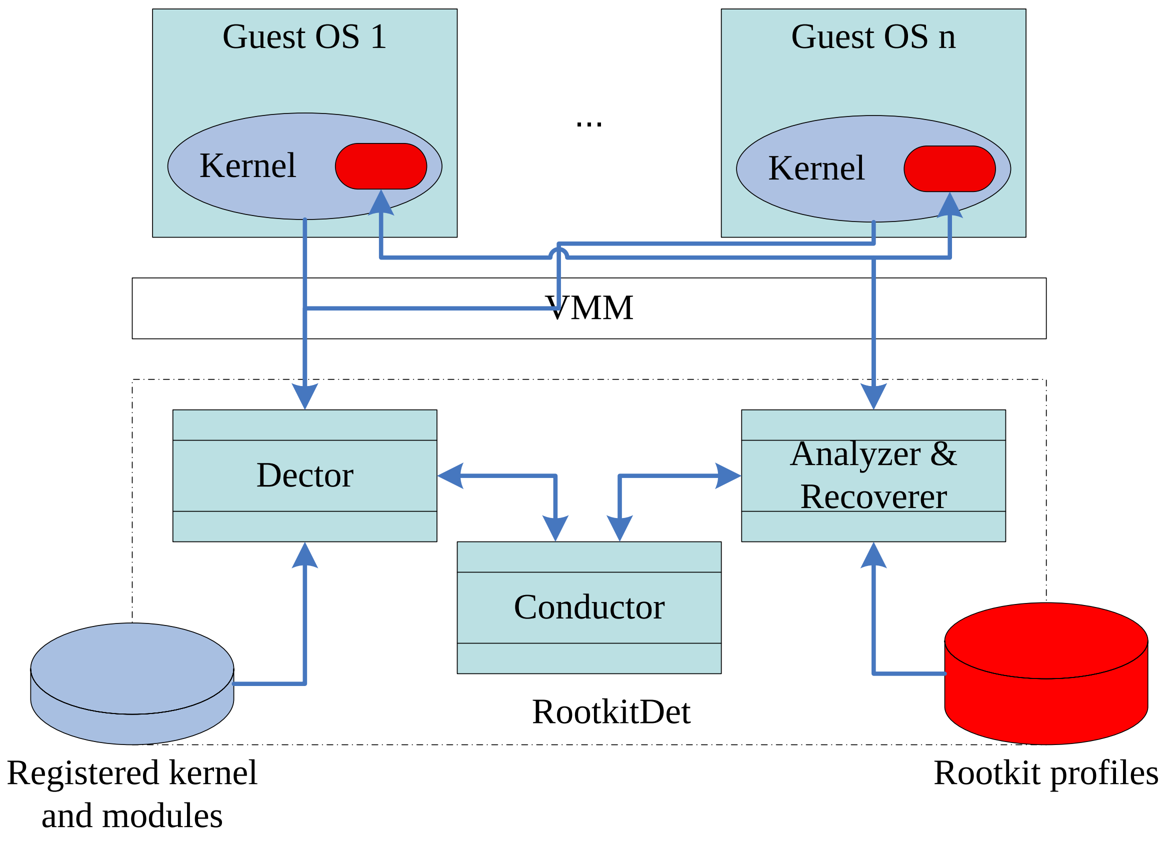
Sachin Shetty

Associate Professor
Electrical and Computer Engineering
Director, Cybersecurity Laboratory
Tennessee State University

RootKitDet- Scalability

- In order to monitor multiple guest OSes simultaneously, we expands our system into a scalable architecture
- For each guest OS, we generate related meta-data of the kernel in advance, which includes:
  - (1) system.map which contains names and locations of the kernel symbols,
  - (2) checksum of the kernel code which is used to detect modifications to the running kernel code,
  - (3) definitions of important data structures that might be referred to by the rootkits or during recovery,
  - (4) type information of important global variables and dynamically allocated objects and their relationship in the kernel.

RootkitDet: Scalable Architecture



RootKitDet: Scalability

- Our system takes advantages of the kernel’s meta-data to detect kernel-level rootkits and perform recovery.
- Besides, only one kernel’s meta-data is necessary if all of the guest OSes are using the same kernel
- Furthermore, several guest OSes can also runs on the same hypervisor if the hypervisor supports multiple guest OSes.

RootkitDet: Detection Procedure Challenges

- Under some particular conditions, inconsistency between the executable regions and loaded modules may occur in the kernel of guest OSes, which causes a false positive in detection procedure 1.
- **Case 1**: When a module is loading, the kernel allocates another executable region for its initialization code, which is released immediately after the initialization code is executed.
  - The temporary existence of initialization code of a module may cause a false positive.
  - We confirm the detection of rootkits only when the detector continuously reports rootkits 3 times.

RootkitDet: Detection Procedure Challenges

- **Case 2**: When a module is unloaded, the kernel doesn’t release related regions until the total size reaches a threshold.
- The lazy clean-up may also cause false positive.
- We require a subtle modification to the kernel source code to release all free regions once a module is unloaded.
- This modification doesn’t affect the efficiency of the kernel because unloading modules happens rarely in general.

RootkitDet: Detection Procedure Challenges

- **Case 3**: Unused space usually exists below the code of a module because of the pagealigned allocation of memory.
- As far as we know, the kernel doesn’t clear the memory regions allocated for modules before loading modules into them.
- As a result, the unused space may contain nonzero data, which cause a false positive in detection procedure 2.
- To eliminate this kind of false positive, we require a subtle modification to the kernel source code to clear the last page of memory regions allocated for modules.

RootkitDet: Detection Procedure Challenges

- The code of a module varies with the relocation address of the module when it is loaded into the kernel.
- We can’t compute checksums of modules in detection procedure 3 because the original object files of modules are not required when the detector computes checksums in our system.
- To reduce the work of the detector, the original checksums of modules are provided by the conductor
- The detector computes current checksums of legitimate code respectively, and compares them with original checksums. Any mismatch means modifications to the legitimate code.

RootkitDet: Detection Procedure

- Detection procedures are performed periodically instead of being triggered like Patagonix
- Rootkits that are erased immediately after execution are out the scope of the system,
- RootkitDet focuses only on the kernel space instead of the space of all processes.
- Overhead of periodical detection is small.
- Unused space of modules should be checked lthough the pages are already legitimate to be executed.
- RootkitDet more flexible to adjust periods of detection procedures.

RootkitDet: Diagnosis

- To categorize the detected rootkit, we investigate well known typical rootkits according to the intentions that rootkits achieve and the tactics that rootkits adopt.
- For each typical rootkit, we generate a profile to describe its tactic to achieve its intention as well as modified data structures and objects that we should recover.

RootkitDet: Diagnosis

- Generate profiles of typical rootkits manually due to the following reasons.
- First, rootkits may achieve different intentions together, and understanding the intentions and related tactics of rootkits requires manual effort.
- Second, data structures and objects that are accessed in the same tactic might subtly vary with the kernel version.
- Third, rootkits may implement the same tactic in different ways.

RootkitDet: Diagnosis

- To apply the profiles of known rootkits during diagnosis, we translate the profiles into ones that coordinate with the kernel running in the guest OS monitored by our system.
- Then the profiles of known rootkits are ready to categorize the detected rootkit.
- Categorization is done by matching certain characteristic information (collected from the detected rootkit) against the set of pre-generated profiles.

RootkitDet: Diagnosis

- RootkitDet system performs static analysis on the code of rootkit to collect characteristic information.
- The characteristic information is divided into two groups.
  - One group is the control flow information. Usually, a rootkit calls to some kernel functions to achieve its intentions, which we name external function call s.
  - The other group is the global variables and dynamically allocated data structures accessed by the rootkit. In general, to access special data structure maintained by the kernel, the rootkit has to find it starting from a global variable and tracking down according to the relationship among different data structures.

RootkitDet: Diagnosis

- A global variable is actually a kernel symbol and usually accessed by its address which is constant.
- The characteristic information collected through static analysis is binary.
- RootkitDet system translates the characteristi information according to the meta-data of the kernel.
- Translated information is then used to categorize the detected rootkit.

RootkitDet: Diagnosis

- We extract the characteristic information of the rootkit through static analysis.
- We focus on external function calls and memory access during static analysis instead of the control flow of the code.
- Determine the values of CPU registers during static analysis.
- We create a static machine with a special CPU and stack to execute the code of rootkit statically.

RootkitDet: Diagnosis

- First, we use a pair < val, flags> to represent the value of a register, in which val represents the value while flags indicates validation of each byte of val .
- We update the pair instead of the value of registers when we execute instructions.
- Second, when an instruction involves read of memory other than the stack, we update val by the value of the memory and set flags by a value indicating val totally valid.
- Finally, some instructions load hard-coded immediate values into registers.

RootkitDet: Diagnosis

- In that case, we also update the flags of the target register according to the size of immediate value and the instruction type.
- In consequence, the values of registers that we can determine during static analysis are independent of execution environments.
- In most cases, we can determine the external function calls and accesses to global variables of the kernel, which we can use to infer the behavior of the suspicious code

RootkitDet: Diagnosis

- Example binary code snippet(middle), with its associated C snippet(left) and associated output of static analysis

```
static struct dentry* adore_proc_lookup(
           struct inode *i,
           struct dentry* d,
           struct namedata* nd)
{
        ...
        task_unlock(current);
        return orig_proc_lookup(i, d, nd);
}
```

```
...
mov  %fs:0xc1416454, %eax
incb 0x330(%eax)
mov 0x8(%esp), %eax
mov %ebp, %ecx
mov %edi, %edx
call *0xd0c0d4d4
...
```

```
...
c1416454   r--   current
d0c0d4d4   r--   c10e6a08
c10e6a08   --x   proc_root_lookup
...
```

RootkitDet: Recovery

- RootkitDet attempts to recover the infected kernel according to the profile of the rootkit.
- Data structures and objects that are modified by the rootkit are described in the profile of the rootkit.
- Combined with the meta-data of the kernel, recovery driven profile is derived from the profile of the rootkit.
- Recovery-driven profile describes how to locate the modified data structures and objects and how to recover them.

RootkitDet: Recovery: Control Data

- Expected values of the control data are locations of kernel functions.
- Data structures and objects maintained by the kernel can always be found tracking down from some global variable.
- Address of global variables are constant and can be found in the meta-data of the kernel.
- Recovery-driven profile for control data describes the tracking path from the global variable to the object containing the control data.

RootkitDet: Recovery: Control Data

- For example, a rootkit may overwrite the pointers of functions registered with the virtual file system layer by the pseudo random number generator (PRNG) to disable the PRNG [17].
- The pointers of functions registered by the PRNG are stored in structures random fops and urandom fops , which are located in the object devlist , a list of memory devices that is a global variable.
- Recovery-driven profile for the functions registered by PRNG contains the address of devlist , offsets of random fops and urandom fops in devlist as well as the real addresses of the functions registered by the PRNG.

RootkitDet: Recovery: Non Control Data

- Non-control data is different because the original values are either lost forever or not easy to calculate.
- Non-control data is different in the way to locate the related data structures or objects.
- For example, a rootkit hides a process by removing related item from the pid hash table.
- Then we can’t find the process tracking down from the pid hash table.
- The only way to find the process is tracking down from init task and checking each process whether it is not linked into the pid hash table.

RootkitDet: Recovery: Non Control Data

- As a result, the recovery-driven profile for non-control data describes how to restore the broken links or resolve violations of invariants as well as the tracking path from the global variable to the object containing the non-control data.
- If the original value of a non-control data are lost forever, we can not recover it.
- For example, we can not recover the entropy pool of PRNG if it is zeroed by a rootkit

RootkitDet: Design Requirements

- Scalable.
  - RootkitDet should support detection of kernel-level rootkits on multi-VMs,
- Low overhead.
  - Performance is critical in cloud environment because the cost a cloud user should pay depends on the resources that he consumes.
- Easy to adopt.
  - Xen[23] and KVM[12] are both used to create VMs in the cloud, It should be easy to deploy RootkitDet system in the cloud base on both Xen and KVM.

RootkitDet: Detection Method

- Procedure 1 - Detect whether extra executable regions exist in the kernel space.
- Procedure 2 - Detect whether some code resides in unused spaces of modules.
- Procedure 3 - Detect whether the code of kernel or modules are modified.
- We can conclude that kernel-level rootkits exist if and only if any of the procedures above comes true.
- The only precondition is that the NX feature is enabled.

RootkitDet: Detection Method

- As we know, the kernel runs in a VM, so this precondition only depends on the settings of the physical machine and the kernel in the VM can not change it.
- In addition, we only read some registers and memory of the VM in all of the three procedures.
- We neither monitor the execution of the VM nor keep watch on some registers or memory of the VM.
- As a consequence, the overhead of this detection method is pretty low

RootKitDet Design Summary

- RootkitDet system consists of one conductor and multiple detectors.
- Conductor runs on the host OS as a user space process.
  - Communicates with all of the detectors through IPC.
  - Sends detection commands to the detectors, and receives responses back.
  - If rootkits are detected, it raises alert.
- Detector detects kernel-level rootkits in a VM by reading its registers and memory.
  - In order to conveniently access the VM's registers and memory, the detector is integrated into the qemu-kvm hypervisor

RootKitDet Design Summary

- Registration procedures
  - Registration of kernel
  - Registration of loadable kernel modules (LKMs)
- Detection procedures
  - Detect extra executable regions in kernel space
  - Detect code residing in unused space of LKMs
  - Detect malicious modifications to legitimate code
- Challenges
  - Inconsistency of executable regions when LKM is unloaded
    - Kernel frees unused virtual memory area in a lazy manner
  - Module’s code is variable due to the relocation
    - Relocation address and symbols of itself
    - Symbols of main kernel, even other modules

RootKitDet Design Summary

- Static analysis
  - Characteristic information of detected rootkits
    - External function calls
    - Global variables and dynamically allocated objects accessed by the code
- Categorization
  - Profiles of known rootkits
    - The tactic adopted by the rootkit to achieve its intention
    - Data structures that are modified according to its tactic
  - Detail the profiles
    - Translate symbols into addresses according to the running kernel

RootKitDet Design Summary

- Recovery-driven profile
  - Derived from the profile of the rootkit and meta-data of the running kernel
- Recovery of control data
  - Expected values are constant and known
  - Tracking down from some global variable
- Recovery of non-control data
  - Expected values can be inferred if logical relations among non-control data and other objects in the kernel