Detecting Rootkits With the RAI Runtime Application Inventory

Shabnam Aboughadareh
University of Texas at Arlington
(now @ Qualcomm)

Christoph Csallner
University of Texas at Arlington

SSPREW-6
Los Angeles, California
Dec 6, 2016

This material is based upon work supported by the National Science Foundation under Grants No. 1117369 and 1527398. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

System Integrity Checking Approaches

- Black- / White-listing programs or payloads
  - Behavior: System call sequences, memory accesses, ...
  - Signatures

- Runtime monitoring: VMs, instrumentation, ..

- Advanced platform integrity approaches
  - TPM for integrity measurement and remote attestation
  - Application signing
  - Secure boot
  - ...

Goal: Scale Integrity Checking to Legacy Systems

- Restarting applications: Costly → Avoid
  - Runtime memory attacks can be more dangerous on legacy systems
  - Security tools that may change process's address space (e.g., debuggers attaching to the process and add interrupts) may terminate program
  - **Goal: Deploy integrity checker on running applications**

- No modern security infrastructure
  - No TPM module/HW
  - No secure boot
  - **Everything on client machine in malware's reach (signatures, etc.) :-(**

Goal: Scale Integrity Checking to Legacy Systems

- Application’s valid signature/state at runtime not clear
  - **Address space of applications at “runtime” changes constantly:**
    - Loading libraries
    - Self modifying code, ...
  - Not specific to legacy systems / applications
  - Usage of TPM requires knowledge about trusted application state/signature
  - Secure boot (and usually usage of TPM) is considered “load-time” integrity

Goal: Scale Integrity Checking to Legacy Systems

- Most existing antivirus software (in charge of blacklisting/whitelisting) can be infected/disabled by a run-time attack without any sign for user/administrator to recognize the attack (not specific for legacy systems)
  - ➢ **Antivirus programs provide a huge attack surface for attackers**
    - High chance of 0-day vulnerabilities
    - High use of OS-level APIs (interesting targets for rootkits) for system monitoring

Assumptions & Threat Model

- Monitored application may run in user or kernel-mode or both
  - User system := OS + all applications running on OS
  - OS may run on hardware, VM, container
- User system may be under malware attack
  - Adversary has full access to the whole system: File system, all memory, …

- **Adversary may continuously hack OS + applications**
  - Inject code: Malicious payloads, hide its trace
  - Manipulate binaries on disk
  - Infect loaded images in memory
  - Obtain higher privilege level: Root, ..
  - Hook sys call table, overwrite code & read-only data sections

Design Keys

C1: Restarting the applications is often costly and should be avoided

K1: No interception in program execution:

→ No recompile, no restart

→ No modification within the application’s address space: Risks corruption / crash

C: Challenge

K: Design Key

Design Keys

C2: Lack of modern security infrastructure

C3: Lack of known valid signature/state for applications at runtime

K2: No root-of-trust

- **Idea:** Run multiple application instances (in different execution states) to create a dynamic whitelist of program states
  - Number of infected machines/applications is initially likely less than the number of non-infected machines
  - Many homogeneous instances already exist
    - Cloud
    - Local networks
    - End-users

Design Keys

C4: Attacks on Antivirus programs

K3: Tiny code with tiny trace on the systems (e.g., no/little usage of common OS-level APIs targeted by rootkits)

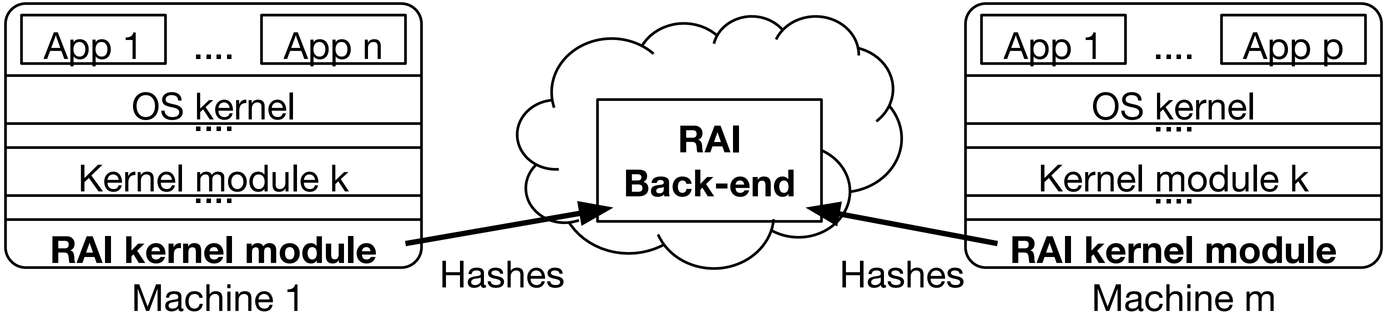
- Take the snapshot of memory frequently & calculate the hash of executable codes within address space of applications
  - Does not require usage of the APIs which are common targets of rootkits
- Process the hash outside the user’s machine (Client-Server scheme)
  - A kernel-mode driver as the client application
    - Full access to OS memory
    - Installed as a hidden driver, with random name and hidden network activity, small code and easy to apply obfuscation techniques

Design Keys

C5: Monitored binaries may change frequently in main memory at runtime
→ e.g., by an ongoing malware attack

K4: Verifying binaries at startup is not sufficient

RAI: Runtime Application Inventory



- Minimal client can be deployed at application runtime & tries to hide itself
- At each client: Periodically monitor (hash) physical memory
- Server requests hashes in random intervals, compares hashes: Infers dynamic white-list
- Detect common rootkit attacks in user- & kernel-mode (current implementation on Linux)
  - Code injection & binary patching (on memory and disk) → monitor executable sections
  - Sensitive data manipulated → monitoring binaries' .readonly section in memory: sys call table, ..

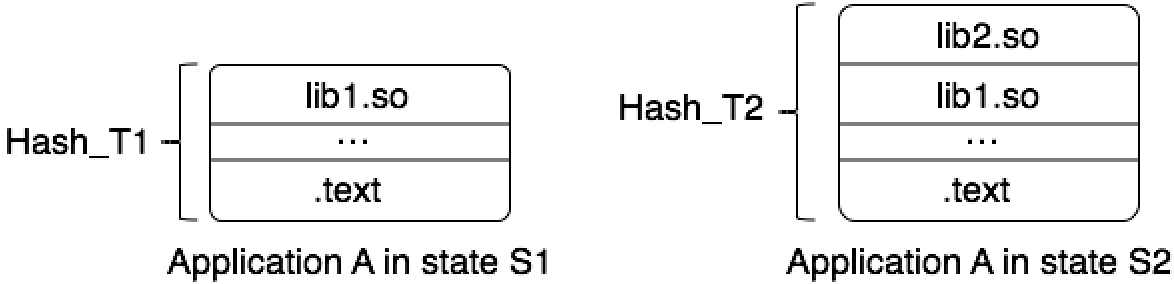
Server Component

- Keep track of participating machines
  - Maintain current / previous hashes
- Manage client-server communication: Initiate, retries, ..

Monitoring Different Program States

Dynamic monitoring challenge: Different hashes for different program states

- Approach :
  - Send hash of each executable segment separately (e.g., VMA in Linux)
  - Backend application(s) is/are in charge of matching the states (identifying/comparing similar segments)

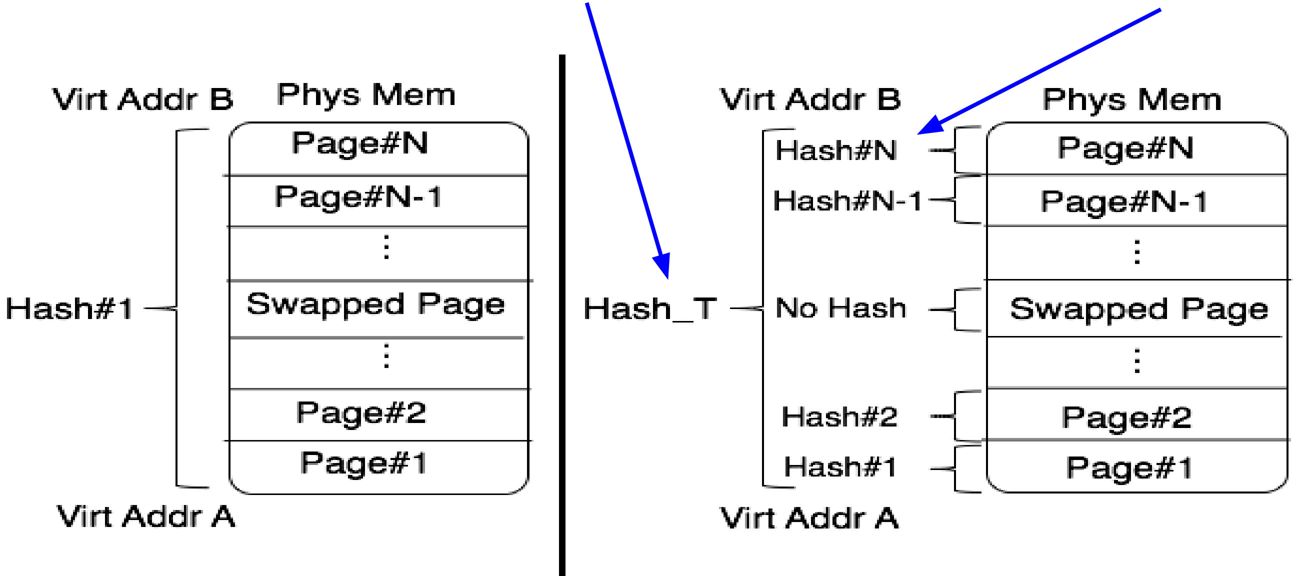


Client: De-Relocate Addresses

- Position Dependent Code: Relative addresses to the “load” address of application cause different hashes
  - ➢ Identifying and de-relocating requires access to the file (e.g., .reloc section in ELF files)
    - ○ Trusting the files on disk is in contrast with “No root of trust” design key
    - ○ After loading into the memory, there is no trace for identifying which file belongs to which kernel module (Linux)
  - ➢ Heuristic Approach: Disassemble the memory contents, find position dependent addresses and de-relocate them
    - ○ Use Distorm disassembler: Provides convenient access to opcodes and operands via its “decomposer” feature

Two-level Hash of Physical Memory

- Observation 1: Unassigned virtual address for unused memory pages (Linux)
  - ➢ Get hash of physical memory pages
- Observation 2: Swapped-out pages can result in inaccurate hash
  - ➢ Obtain two-level hash. When needed: Compare level-1 hashes



Prototype Implementation

- Implementation took several shortcuts
    - Fixing these: Future work

- Client component: Kernel module
    - Listens directly on the network for server commands → Obvious security problems
    - Advanced features missing
    - Failure recovery, security / authentication, availability, scalability, performance management
- DoS attacks on server component possible
- ...

Research Questions (RQ) & Hypotheses (H)

- **Overall: Promising for online rootkit detection?**
  - True / false negatives less interesting: Combine with other detection approaches anyway
- RQ1: Does runtime overhead preclude RAI from detecting rootkits online?
  - [-] Compensating for code load order / address space layout randomization expensive
  - H1: RAI can be useful if client machines have significant resources available
- RQ2: Do false positives preclude RAI from being used in production?
  - [-] x86 disassembly undecidable → Zero false positives impossible in general
  - H2: RAI's average false positive rate can remain below 10%
- RQ3: Can RAI detect common types of kernel / user level rootkit attacks?
  - H3: RAI can detect common rootkit attack types online.
- RQ4: Does RAI scale to geographically widely distributed deployments?
  - H4: RAI can detect rootkit attacks within a few minutes, even if the RAI-monitored applications are running on geographically widely distributed machines

Evaluation: Own & Third-party Subjects

	Subject	Mode	Location
1.	Exchanging libraries via LD_PRELOAD	User	Memory
2.	Exchanging libraries via ld.so.preload (Jynxkit)	User	Memory
3.	Patching the user-mode program loader	User	Disk
4.	Diverting process execution (InjectSO)	User	Memory
5.	Hooking the system call table	Kernel	Memory
6.	In-line function patching (Suterusu)	Kernel	Memory

Evaluation: Two Setups

1. RQ1 to RQ3: “Local” setup
   1. All clients on same physical machine
      1. 40 VMware Ubuntu Linux
      2. Two groups with the same kernel versions (2.6 and 3), 512 MB RAM, 32 and 64 bits processors
      3. 100 user-mode applications and 41 kernel modules
   2. One VM is dedicated to the server: Ubuntu 12.04 LTS, 64 bits

2. RQ4: AWS
   1. Sets from 6 to 60 clients equally distributed over 10 AWS regions
      1. Similar setups to local experiment
   2. 90 user-mode applications
   3. Ubuntu 12.04 LTS, 30 GB RAM as the server running in Oregon

RQ1: Moderate Runtime Overhead

Activity	Target	Location	Slowdown(%)
Hash	User-mode app	Client	8
Hash	Kernel code/data	Client	3
Hash + De-relocation	Kernel module	Client	20
Compare hashes	20 VMs	Server	15
Logging received data	20 VMs	Server	20

RQ2: False Positives in De-Relocation

Kernel Module	Number of Pages	False Hash	False Positive (%)
E1000	22	3	13.6
Vmwgfx	18	2	11.1
Ttm	11	1	9.0
Drm	33	1	3.0
Bluetooth	55	3	5.4
Rfcomm	9	1	11.1
Psmouse	16	2	12.5
All 41 modules	**314**	**13**	**4.1**

RQ3: Rootkit Detection Rate

- 100% rootkit detection
- Identified the injected code
- Identified the manipulated pages

RQ4: Scaling to Larger Deployments

Total Clients	6	12	18	24	30	36	42	48	54	60
Delay (s)	0.4	1.0	1.1	1.6	1.9	2.1	2.9	3.3	3.9	4.4

Limitations / Future Work

- Return-oriented programming attacks
- Quantitative comparison (comparison with similar approaches)
- Experiment on legacy systems

Related Work (1/2)

- Cloud-based antivirus: CloudAV, ..
  - Reduce attack surface on client
  - But rely on manually curated blacklists (slow)
- Rootkit detector built on Pioneer
  - But needs prior knowledge of applications & makes strong machine / connection assumptions
- File integrity checking: Tripwire, SVV, ..
  - Compare in-memory with on-disk
  - But cannot deploy during malware attack: Malware may have changed files on disk
- Traditional malware detection: Nickle, Poker, ..
  - E.g.: Static symbolic execution
  - But kernel-only, large attack surface, cannot apply in an ongoing malware attack

Related Work (2/2)

- Static instrumentation
    - But requires recompile & restart

Acknowledgments

- Matthew Elder
- Nathan Evans
- Azzedine Benameur
- Anonymous SSPREW reviewers

Questions

Attacks Started by Zero-Day Exploits

- May inject malicious behavior into a trusted application
    - White-listing the app does not help
- May be persistent, manipulate infected system to hide itself
- Recent code-injection rootkit examples: Stuxnet, Duqu, Flame, ..

- **Traditional anti-virus not effective**
    - May take weeks for tool to receive signature to catch malware
    - Since most antivirus tools rely on blacklist of known malware signatures
    - Antivirus vendor needs time to distill malware into signatures & deploy the signatures

Many Homogeneous Instances

- Assume: Many application instances running at the same time
- Common in large distributed applications
    - Data centers, cloud
    - Popular stand-alone client-side end-user applications

- **Our scheme works best if relatively few app variants in use**
    - App version, how they have been compiled
    - Common: Developers use same compiler for long time, few app versions in wide use

Pinpointing Possible Attacks

- Server receives hashes from all clients
- Cluster the clients: Within each cluster:
    - Each member has same architecture & OS version
    - **Detect outliers**

- **[+] Does not need prior knowledge**
    - About original binaries / signatures / blacklists / whitelist
- **[+] Detects potential malware attacks immediately**
    - Don’t have to wait until third party releases corresponding malware signatures

Obtain Virtual Address Ranges

- Consult Linux kernel's symbol table (system.map)?
  - Maps between name and address
  - But does not work if the kernel uses run-time address randomization
- Traverse the I/O memory resources & kernel code's physical address range?
  - RAM's child resources: Kernel code, data, and uninitialized data sections (bss)
  - But does not work for the kernel's read-only data
- **Call Linux kallsyms function**
  - Extracts all symbols (e.g., functions and variables) from the kernel
  - Commonly used by Linux debuggers
  - Extract address of symbols that mark start / end of kernel code & read-only data segments
    - _stext, _etext, __start_rodata, __end_rodata
- Outside kernel: Traverse kernel heap structures: task_struct, module_struct

Hashing Physical Memory

- Linux kernel code & read-only data: Straightforward
  - In contiguous physical addresses
  - Never swapped out
- Special case: Legacy machines w/ more physical than virtual memory
  - Access main memory > 4GB on 32-bit x86 (4GB virtual address space)
  - Linux kernel dynamically maps a set of virtual addresses to a larger set of physical addresses
  - kmap HIGH MEMORY
- Linux may recycle virtual address
  - Even if the pointed-to page is still valid
- Reads pages directly from Linux page cache
  - Page may still be valid but only be reachable via page cache

Conclusions

- First approach
    - To determine which code is running on which machines
    - That is designed to work even when deployed on legacy systems under malware attacks
- Designed to be more effective at detecting rootkits in legacy applications
    - Than state of the art