Code Validation for Modern OS Kernels

Thomas Kittel
kittel@sec.in.tum.de
and
S. Vogl, T. Lengyel, J. Pfoh, C. Eckert

Lehrstuhl für Sicherheit in der Informatik / I20
Technische Universtität München

12/09/2014

Agenda

1. Motivation
2. Code patching mechanisms in the Linux Kernel
3. Architecture
4. Discussion
5. Conclusion

Motivation

- OS integrity is a fundamental requirement for malware detection
- Currently OS code is mainly thought to be static
- Malware research focuses on hook detection and DKOM
- Systems using UEFI and trusted boot for load time integrity are widely employed
- Run-time code integrity is still an open problem

Limitations of current approaches

- Currently code validation is mainly comparing hashes
- Each executable page is hashed and the hashes are stored in a (trusted) database
- The code identity is bound to the current hash
- If the hash changes, the code identity changes

Problem statement

- Kernel code in memory is not static
  - Code in memory is different to the code on disk
  - *Load-time* patching is applied while loading the code
  - *Run-time* patching is often applied by the kernel
- Malware only requires one control flow modification (4 Bytes)
- Every single change to the code has to be validated

Related Work

- Hash-based Approaches
  - **Copilot** Calculates hashes of all kernel code
  - **SBCFI** Moves the validation component out of the guest system
- Disable run-time modifications
  - **SecVisor** Forbids writing to kernel code pages
  - **MoRE** Uses the split TLB to direct write attempts to kernel code to another physical page
  - **Ianus** Forbids kernel modules to write to kernel code
- Load-time validation
  - **Patagonix** Partly considers load time modifications of kernel code

Load-time patching

- Relocation
  - resolving of external symbols
- Alternatve Instructions
  - Different instructions depending on current CPU features
- Paravirt Instructions
  - Using hypervisor functionality for (para-) virtualization
  - Also used on native and full-virtualized systems
  - XEN even inserts unconditional jumps
- Different instructions are used on different systems
- A lot of different possible hashes to maintain

Run-time patching

- SMP Locks
  - Locking mechanisms are only enabled if multiple CPUs are present
  - Could also be used to replace entire functions
- Jump Labels
  - *optimize "highly unlikely" code branches to the point that their normal overhead is close to zero*
  - patch kernel code by adding and removing branches
- Ftrace
  - An ftrace call is placed at the beginning of every function
  - Ftrace function calls are only enabled when requested by the user
- Future mechanisms:
  - Kpatch (based on Ftrace)
  - BPF (in-kernel VM)

- The integrity of each dynamic patch has to be validated separately

Example: Validation of Jump Labels

- A jump label may be disabled or enabled at any given time
- The jump target (location) of a jump label is stored in memory
- For validation the following has to be checked:
  1. The kernel control datastructure (the destination)
  2. The current state of the label (enabled / disabled)
  3. The kernel code itself
- A change is valid only if all three are consistent and reproducable

Run-time patching is complex

- Patches happen in multiple stages to avoid race conditions
- Therefore, patching is done as follows:
  - Replace the first byte with a breakpoint (CC)
  - Patch all further bytes of the patch
  - Patch the first byte
  - Wake up all threads that hit the breakpoint
- NOP sequences are used if no replacement code is required
  - The NOP sequences vary for each CPU architecture
- Keeping track of hashes is really hard

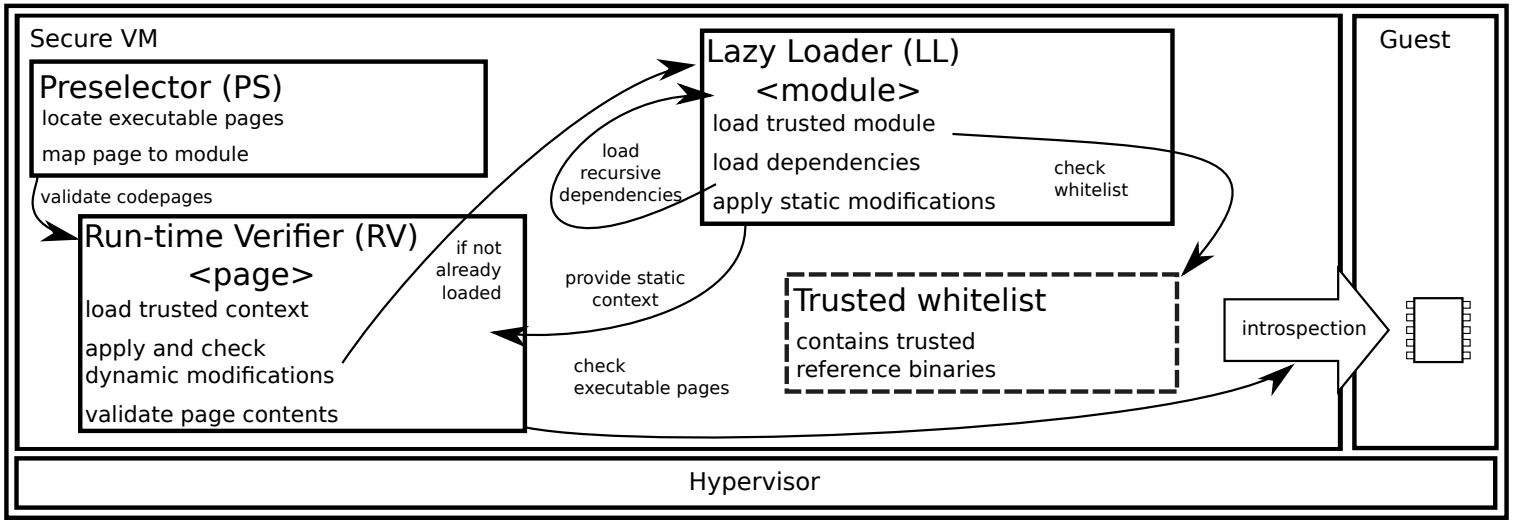
Basic concept

- Resemble the loading and patching process
- Validate legitimate changes

- Employ VMI for increased trust
- Do not trust the introspected guest
- Reference of kernel code in a trusted environment

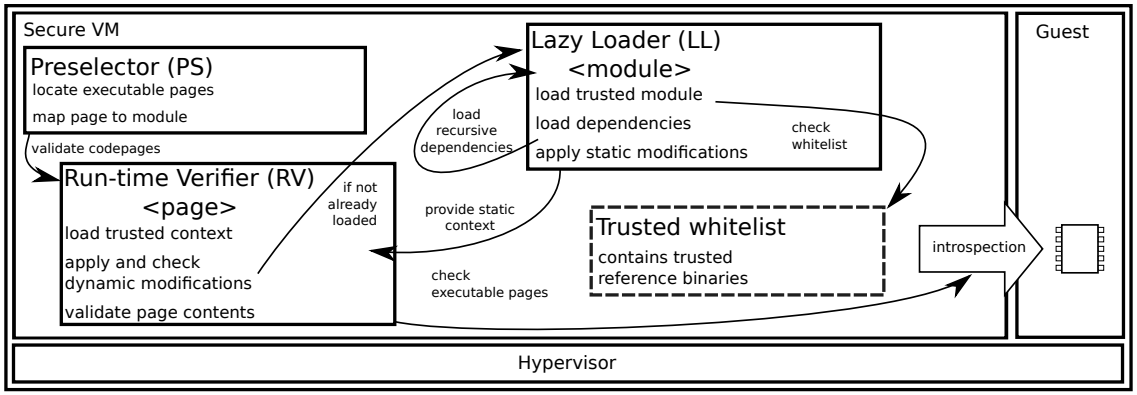
- Validation component is executed in the host or a dedicated VM

Proposed architecture



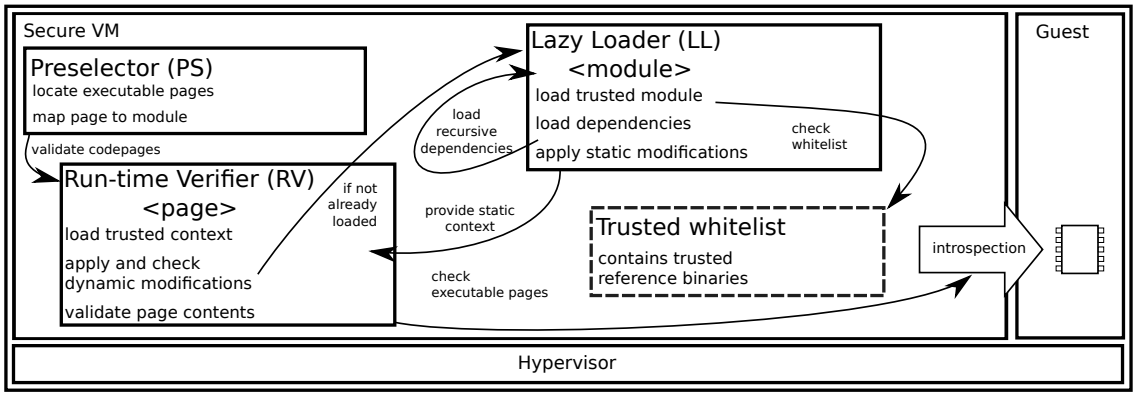
- Main components
  - Preselector
  - Lazy Loader
  - Runtime Verifier

Preselector



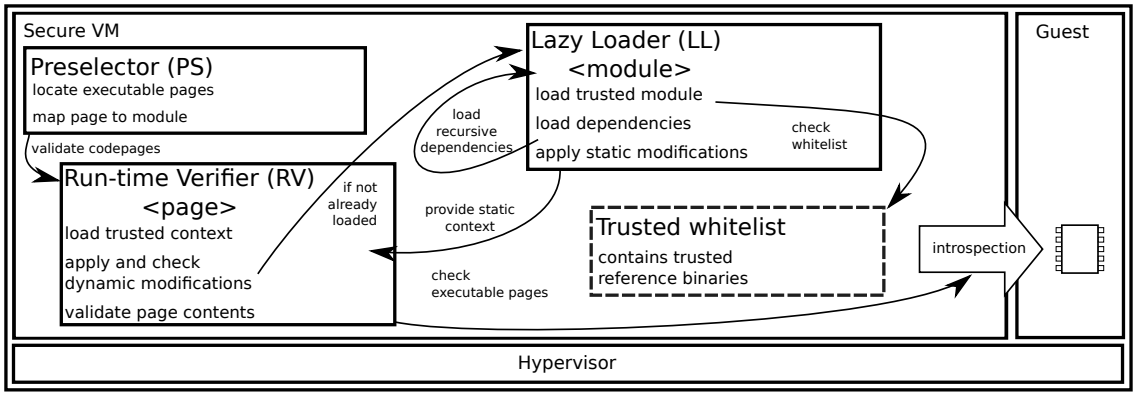
- Walks the page-tables of the monitored guest
- Identifies executable supervisor pages
  - Maps pages to corresponding kernel module
  - Separates code pages from executable data pages
- Calls Run-time Verifier to check code pages

Run-time Verifier



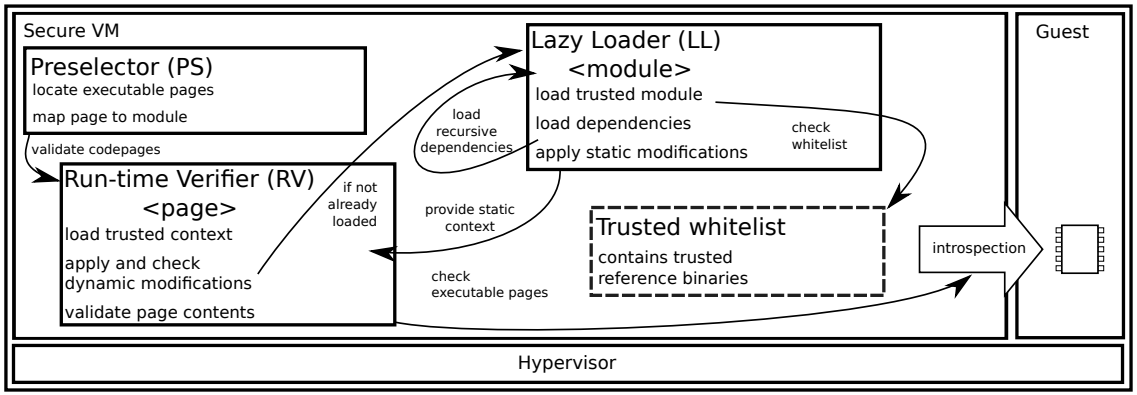
- Requests trusted module context from Lazy Loader

Lazy Loader



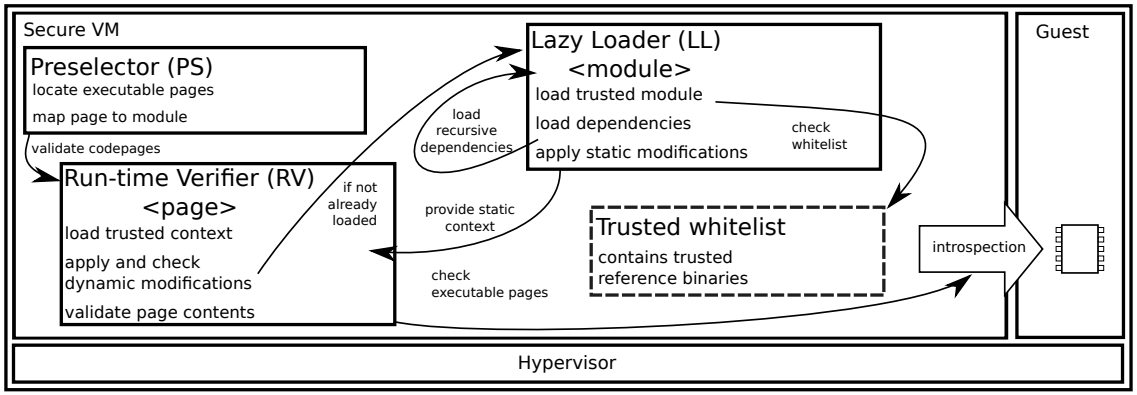
- Resembles linux loading process
  - Loads reference binary from trusted storage (referenced by name)
  - Resolves and loads dependencies
  - Applies load-time modifications
- Provides static context for Run-time Verifier
  - List of patchable locations together with meta information

Run-time Verifier



- Receives trusted module context from Lazy Loader
- Apply run-time patches to the trusted reference
- Validate differences between code and trusted reference
  - integrity of the kernel state (relevant data)
  - reconstruct and verify dynamic modifications

Continuous Validation



- While loading our system an initial validation is conducted
- Then, only write accesses to the kernel code need to be validated

Detection of malicious code

- Preselector
  - Pages not related to any kernel component
  - Pages containing data
- Lazy Loader
  - No trusted reference binary available
- Run-time Verifier
  - Detection of inconsistent kernel state
  - Detection of unknown / unverifiable changes in code segments

Evaluation - Effectiveness

- This investigation is mainly based on the Linux kernel
  - The implementation was tested with Linux 3.8
- We tested our system to detect 4 different linux rootkits
  - All 4 rootkits were detected
  - All other changes to kernel code were also reported
- No false positives during our tests
  - Legitimate kernel module loading is supported

Evaluation - Performance

- Initialization required
  - ca. 4 seconds in our test system
- Fast validation after the system is initialized
  - 141 executable code pages in our test system
  - $0.279s$ to validate all pages
  - ca. $2ms$ overhead for each single page
- The test results are based on continuous code validation
- As patching is an infrequent event, event based validation would futher improve performance

Evaluation - Security Considerations

- Our system uses untrusted (non-binding) information about the guest
- This is not an issue for our system, as:
  - The page tables are derived from the virtual hardware
  - Hidden kernel modules can not hide their code pages[1]
  - The list of symbols is derived from the trusted binary representation
  - The current system hardware state is derived from the hypervisor
  - Paravirtualization is handled by a whitelist of referenced symbols
  - For Ftrace and Jump Labels the current state of the corresponding data structures is also checked for integrity

[1] Well ...

Evaluation - Security Considerations

We ensure that:

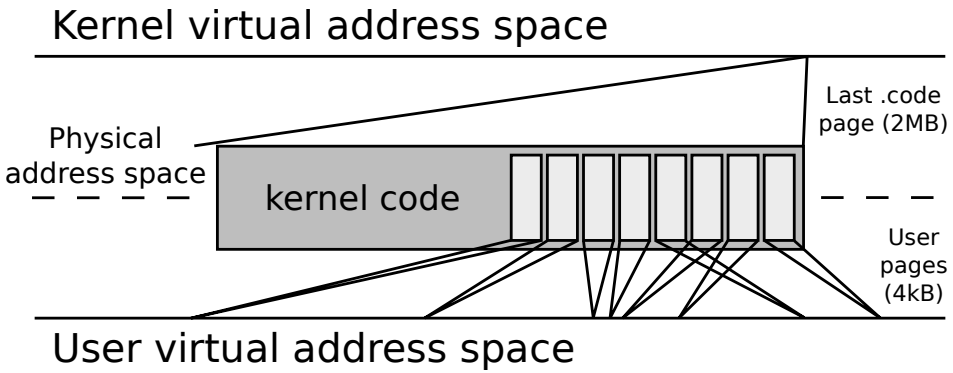
- all detected code pages are related to known modules
- the state information in kernel data structures is consistent
- dynamic calls only use known verified functions

Sidenote — Executable data pages

- The kernels data pages are executable by default (unused NX)
  - This is also the default for memory allocated with *kmalloc*
  - Data pages can not be validated
- Lowers the requirements for code execution vulnerabilities
  - The attacker can directly use an exploited buffer to execute his payload

Sidenote — Double page mapping

- Parts of the kernels code segment are also mapped to userspace
  - An argument for this design decision is not to waste memory
- This violates the barrier between userspace and kernel space
  - The attacker is able to extract the mapped pages from userspace
  - He can calculate the virtual address of that page in the kernel
  - To exploit that he only has to jump to that location



- Similar issue with kernel identity mapping found by Kemerlis et al.

Future Work

- Event-based implementation is work-in-progress
- Identify more data structures that are relevant for control flow
  - Stack frame validation using the information generated by this system
  - Tracepoints can call arbitrary functions when requested
  - Support Live Kernel function patching
  - Support for BPF (a in-kernel virtual machine)
- Extend framework for userspace code validation (White et al.)
- Extend framework for Windows guests

Conclusion

- Kernel code integrity is an open issue
- Existing systems are limited and / or incomplete
- We presented a prototype to validate dynamic code changes
- Our system is effective while having a low overhead
  - Still room for improvements (event-based validation)
- We found two architectural problems in the Linux kernel