0x1A Great Papers in Computer Security

Vitaly Shmatikov

http://www.cs.utexas.edu/~shmat/courses/cs380s/
Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems

(ASPLOS 2008)
Goal: Bypass an Insecure OS

- Secure software runs on commodity OS, thus even a 100% secure application can be compromised if the OS is compromised

- Goal of Overshadow: securely execute application even if the OS is not trusted
  - Guarantee confidentiality and integrity for application’s data in memory and on disk
  - Trust only VMM, not the OS

- Backward compatibility!
  - No modifications to OS or application binary
Virtual Machines

◆ Hardware-level abstraction
  • Virtual hardware: CPU, memory, chipset, I/O devices, etc.
  • Encapsulates all OS and application state

◆ Virtualization software
  • Extra level of indirection decouples hardware and OS
  • Multiplexes physical hardware across multiple “guest” VMs
  • Strong isolation between VMs
  • Manages physical resources, improves utilization
Key Idea: Cloaking

◆ VMM provides multiple views of application’s memory depending on who is looking
  - Application: unencrypted read-write access
  - Guest OS: “cloaked” view
    - Encrypted and integrity-protected
◆ Application/OS interaction mediated by shim
  - Public (unprotected) shim on guest OS
  - Private (protected) shim on application
Overshadow Architecture

- VMM switches between two views of memory
  - App sees normal view
  - OS sees encrypted view

- Shim manages application/OS interactions
  - Interposes on system calls, interrupts, faults, signals
  - Transparent to application

Two Virtualization Barriers
Memory Mapping: OS and VMM

virtual $\rightarrow$ “physical” $\rightarrow$ machine

guest OS

GVPN (guest virtual page number)

GPPN (guest physical page number)

MPN (machine page number)

shadow page tables

VMM
Multi-Shadowing

virtual $\rightarrow$ “physical”

guest OS

machine$_1$

view$_1$

machine$_2$

view$_2$

The view of memory is context-dependent!
Basic Cloaking Protocol

At any time, each page is mapped into only one of the two shadows
- App (A) sees plaintext via application shadow
- Kernel (K) sees ciphertext via system shadow

Protection metadata
- IV – random initialization vector
- H – secure hash of page contents
OS Accesses a Page

Virtual → Physical

Application’s view

Guest OS

OS’s view

Encrypted machine

Page is unmapped in current shadow ⇒ fault into VMM
VMM encrypts the page, computes integrity hash, remaps encrypted page into system shadow
Application Accesses a Page

Page is unmapped in current shadow $\Rightarrow$ fault into VMM

VMM verifies the integrity hash, decrypts the page, remaps plaintext page into application shadow
Cloaking Application Resources

◆ Protect memory-mapped objects
  • Stack, heap, mapped files, shared mmaps

◆ Make everything else look like a memory-mapped object
  • For example, emulate file read/write using mmap

◆ OS still manages application resources
  • Including demand-paged application memory
  • Moves cloaked data without seeing its true contents
  • Encryption/decryption typically infrequent
    – OS accesses application’s page ⇒ encrypt
    – Application accesses OS-touched page ⇒ decrypt
Shim

◆ Challenges

- Securely identify which application is running
- Securely transfer control between OS and application
- Adapt system calls

◆ Solution: shim

- OS-specific user-level program
- Linked into application address space
- Mostly cloaked, plus uncloaked trampolines and buffers
- Communicates with VMM via hypercalls
Hypercalls

- Used by shims to invoke VMM
- Uncloaked shim (untrusted, invoked by OS)
  - Can initialize a new cloaked context
    - When starting an application
  - Can enter and resume existing cloaked execution
    - When returning to a running application
- Cloaked shim (trusted, invoked by application)
  - Can cloak new memory regions (when is this needed?), unseal cloaked data, create new shadow contexts, access metadata cache
Secure Context Identification

◆ VMM must identify unique application contexts in order to switch shadow page tables
◆ Cloaked Thread Context (CTC)
  ● Sensitive data used for OS-application control transfers
    – Saved registers, entry points to shim functions, ASID (address space identifier – used to identify context), a special random value generated during initialization
  ● Uncloaked → cloaked (OS → application) transition: uncloaked shim makes a hypercall, passes ASID and the pointer to CTC to VMM, VMM verifies expected ASID and the random value
    – What prevents malicious OS from messing with CTC?
Handling Faults and Interrupts

1. App is executing
2. Fault traps into VMM
   • Saves and scrubs registers
   • Sets up trampoline back to shim so kernel can return
   • Transfers control to kernel
3. Kernel executes
   • Handles fault as usual
   • Returns to shim via trampoline
4. Shim hypercalls into VMM
   • Resume cloaked execution
5. VMM returns to app
   • Restores registers
   • Transfers control to app
Handling Systems Calls

- **Extra transitions**
  - Superset of fault handling
  - Handlers in cloaked shim interpose on system calls

- **System call adaptation**
  - Arguments may be pointers to cloaked memory
  - Marshal and unmarshal via buffer in uncloaked shim
  - More complex: pipes, signals, fork, file I/O
Marshalling Syscall Arguments

- For some system calls, OS needs to read or modify arguments in caller’s address space
  - Path names, socket structures, etc.
  - This does not work with cloaked applications (why?)
- Instead, arguments are marshalled into a buffer in the uncloaked shim and registers are modified so that the call uses this buffer as the new source or destination
- Results are copied back into the cloaked application’s memory
Resuming Cloaked Execution

- OS can ask to resume cloaked execution from a “wrong” point, but integrity checking will fail unless the CTC is mapped in the proper location
  - What’s the “right” point to resume execution?
- VMM will always enter cloaked execution with proper saved registers, including the IP, and all application pages unaltered (why?)
- Thus, OS can only cause a cloaked execution to be resumed at the proper point in the proper application code
Signal Handling

- Parts of the shim cannot be preempted
- Application registers a signal handler $\Rightarrow$ the shim emulates the OS and records it in a table
- Signal is received $\Rightarrow$ shim passes to VMM the signal, parameters, context in which it occurred
  - If during a cloaked execution, VMM passes control to a proper signal entry point in the shim
  - If during a shim execution, VMM either rolls back the execution to the last application system call entry, or defers signal delivery until shim returns to application
Cloaked File I/O

- Interpose on I/O system calls
  - Read, write, lseek, fstat, etc.

- Uncloaked files use simple marshalling

- Cloaked files emulated using memory
  - Emulate read and write using mmap
    - Copy data to/from memory-mapped buffers
  - Decrypted automatically when read by application, encrypted automatically when flushed to disk by OS
  - Shim caches mapped file regions (1MB chunks)
  - Prepend file header containing size, offset, etc.
Protection Metadata

- VMM enforces integrity, ordering, freshness for application’s memory pages

- **Metadata** for each memory page tracks what’s supposed to be in it
  - IV – random initialization vector
  - H – secure integrity hash of page content
  - VMM keeps the mapping \((\text{ASID}, \text{GVPN}) \rightarrow (\text{IV}, \text{H})\)
    - ASID = “application” (address space) identifier
    - GVPN = guest virtual page number
Managing Protection Metadata

Application (SID)

Daemon

Guest

VMM

miss
load
evict
find

(start,end) → (RID,RPN)

(RID,RPN) → (IV,H)

VA RANGE → RESOURCE

RESOURCE → METADATA

MLB (per-ASID)

MDC (per-VM)
Details of Metadata Protection

- **Protected resources**: files and memory regions
  - \((\text{RID}, \text{RPN})\) – unique resource id, app page number

- **Metadata lookup in VMM**:
  \((\text{ASID}, \text{VPN}) \rightarrow \text{(RID, RPN)} \rightarrow \text{(IV, H)}\)
  - Shim tracks mappings \((\text{start, end}) \rightarrow \text{(RID, RPN)}\)
    - VMM caches these mappings in “metadata lookaside buffer” \((\text{MLB})\), upcalls into shim on MLB miss
  - **Indirection needed to support sharing and persistence**
    - Two processes of the same app may access same resource
    - Application may want to keep a resource between executions
    - Persistent metadata is stored securely in the guest filesystem
Cloning a Cloaked Process

- Allocate local storage for new thread
- Copy parent’s CTC and fix pointers to the new thread’s local storage
- Change instruction pointer and stack pointer in the child’s CTC
- Set up the uncloaked stack so that the child starts execution in a special `child_start` function within the child’s shim, it finishes initialization
Cloning Metadata

- **Problem:** copy-on-write private memory regions shared between a process and its clone
- **If parent encrypts shared memory after the fork, how does the child find metadata for decrypting?**
- **Solution:** data structure with metadata information, mirroring the process trees
  - Whenever a page is encrypted, new metadata (random IV, hash) is propagated to all children with pages whose contents existed prior to the fork
Security Guarantees (1)

◆ OS cannot modify or inject application code
  - Application code resides in cloaked memory, where it is encrypted and integrity-protected
  - Any modifications detected because page contents won’t match the hash in VMM’s metadata cache

◆ OS cannot modify application’s instruction pointer
  - All application registers are saved in the cloaked thread context (CTC) after all faults/interrupts/syscalls and restored when cloaked execution resumes
  - CTC resides in cloaked memory and is encrypted and integrity-protected, so the OS can’t read or modify it
Security Guarantees (2)

- OS cannot tamper with the loader
  - Before entering cloaked execution, VMM verifies that the shim was loaded properly by comparing hashes of the appropriate memory pages with expected values
    - If check fails, the application can access resources only in encrypted form

- OS can execute an arbitrary program instead, but it cannot access any protected data
Overshadow: Key Ideas

- VM-based protection of application data – even if the OS is compromised!
- No modifications to OS or applications
  - Shim extends the “reach” of VMM
- Multi-shadowing and cloaking
  - Use the shim and faults into VMM to switch between encrypted and unencrypted views on all transitions between the application and the OS