ICT Risk Assessment

Fabrizio Baiardi
f.baiardi@unipi.it

Syllabus

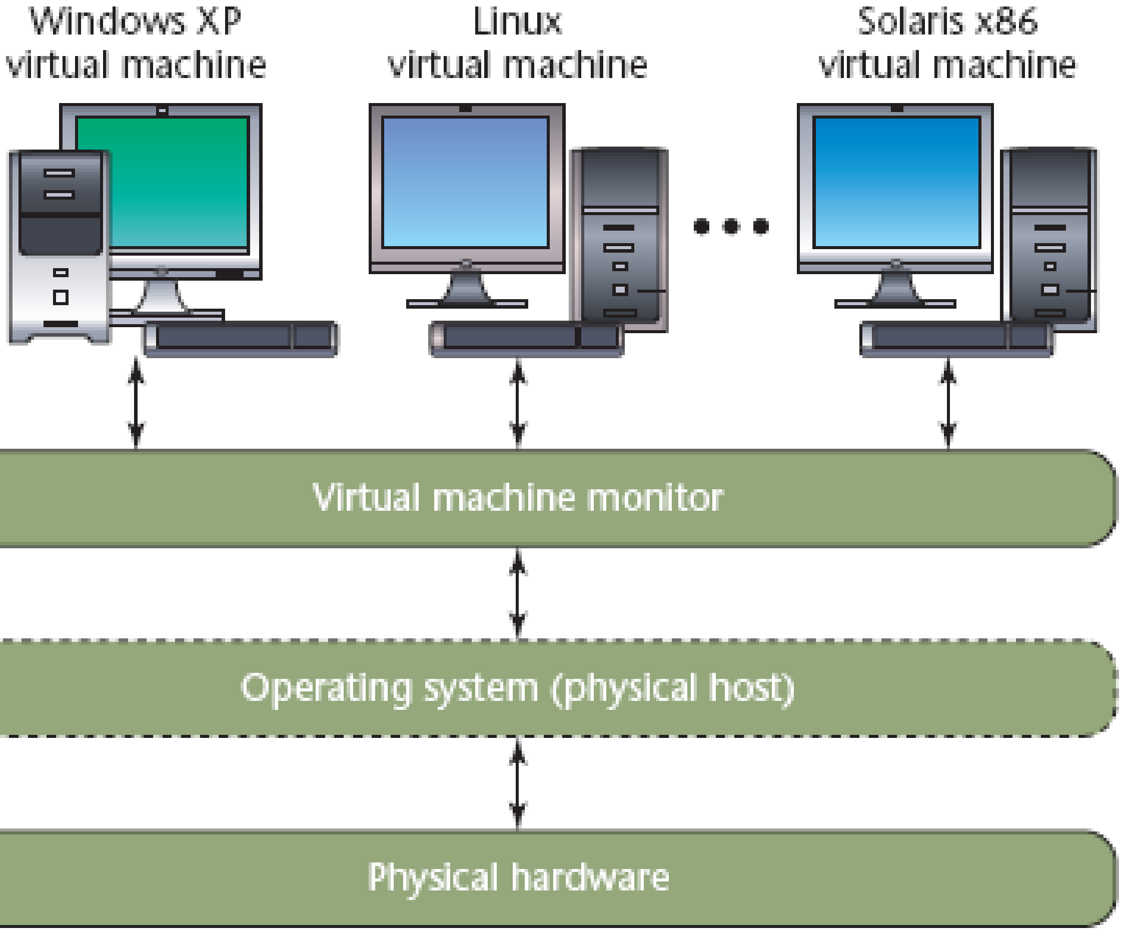
- Security
  - New Threat Model
  - New Attacks
  - Countermeasures

Introspection and a large number
of tools and security controls that
access the memory of a virtual machine

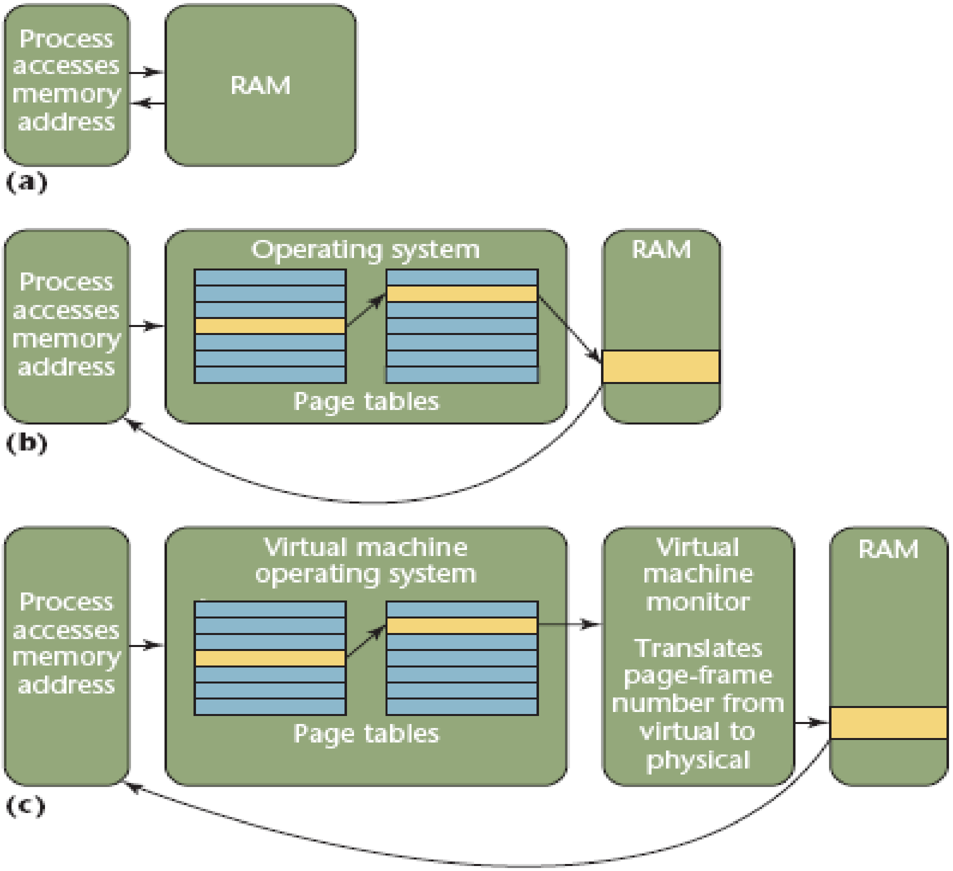
Virtual Machine Introspection

- VMI formally defines techniques and tools to monitor the VM run time behavior to protect the VM from internal and external attacks
- Inspect a VM from the outside to assess what's happening on the inside
- Possible for security tools
  - Virus scanners
  - Intrusion detection systems
- Observe and respond to VM events from a "safe" location outside the monitored machine
- This is another countermeasure that exploits virtualization. Another example of how virtualization changes the computing framework
- With respect to static attestation, virtual machine introspection implements a form of run time attestation that aims to discover not only which software a system runs but also its run time integrity

Virtualization Overview



Memory Mapping


Process accesses memory address
RAM
(a)
Process accesses memory address
Operating system
Page tables
RAM
(b)
Process accesses memory address
Virtual machine operating system
Page tables
Virtual machine monitor
Translates page-frame number from virtual to physical
RAM
(c)

Memory Mapping

- A process perspective
  - Request results in direct access to the memory address
- The OS layer has an active role in providing memory location access
  - Access the page table to map the logical memory address to a physical memory address
- VMM provides an abstraction layer between
  - Each VM OS's memory management
  - The underlying physical hardware
- VMM translates the VM-requested page frame number into a page frame number for the physical hardware
- Gives the VM access to that page

VMM Memory Accesses

- VMM accesses memory pages assigned to each VM directly by
  - VMM's active involvement in this process
  - Its elevated privileges
- Without the VM actually requesting the page
- Can also make those pages accessible to other Vms
- Another software module has a complete and transparent access to the memory pages of a virtual machine
- This requires an hardware module when working with physical machines

TPM vs. Introspection

TPM

- Root of trust rely on hardware
- Passive device
- Platform and software stack decide what to measure
- Need software update to change measurement coverage
- Can not detect compromise in software stack since verification

VM Introspection

- Root of trust rely on hypervisor
- Introspection agents = modules have the initiative
- Security vendor / policy dictate what to measure
- Coverage is content, and can change independently of VM
- Designed to continuously scan VMs and to detect compromise

Virtual Machine Introspection -1

- By implementing a physical machine through a virtual one, we can check the integrity of any component of the physical machine by evaluating a predicate on the state of the virtual one = on some memory subset of the physical one = from physical to information

- If the checks are delegated to the VMM the complexity of the VMM strongly increases together with the probability of a successful attack

- If the VMM has not been successfully attacked, then the same task can be delegated to another VM, the introspection one

- This is a dynamic, or semantic, attestation where the Introspection VM gives some assurance on the current status status of another VM

- Bootstrap = the Introspection VM assures the integrity of a component on another VM that, in turn, assures the integrity of another VM

Virtual Machine Introspection -2

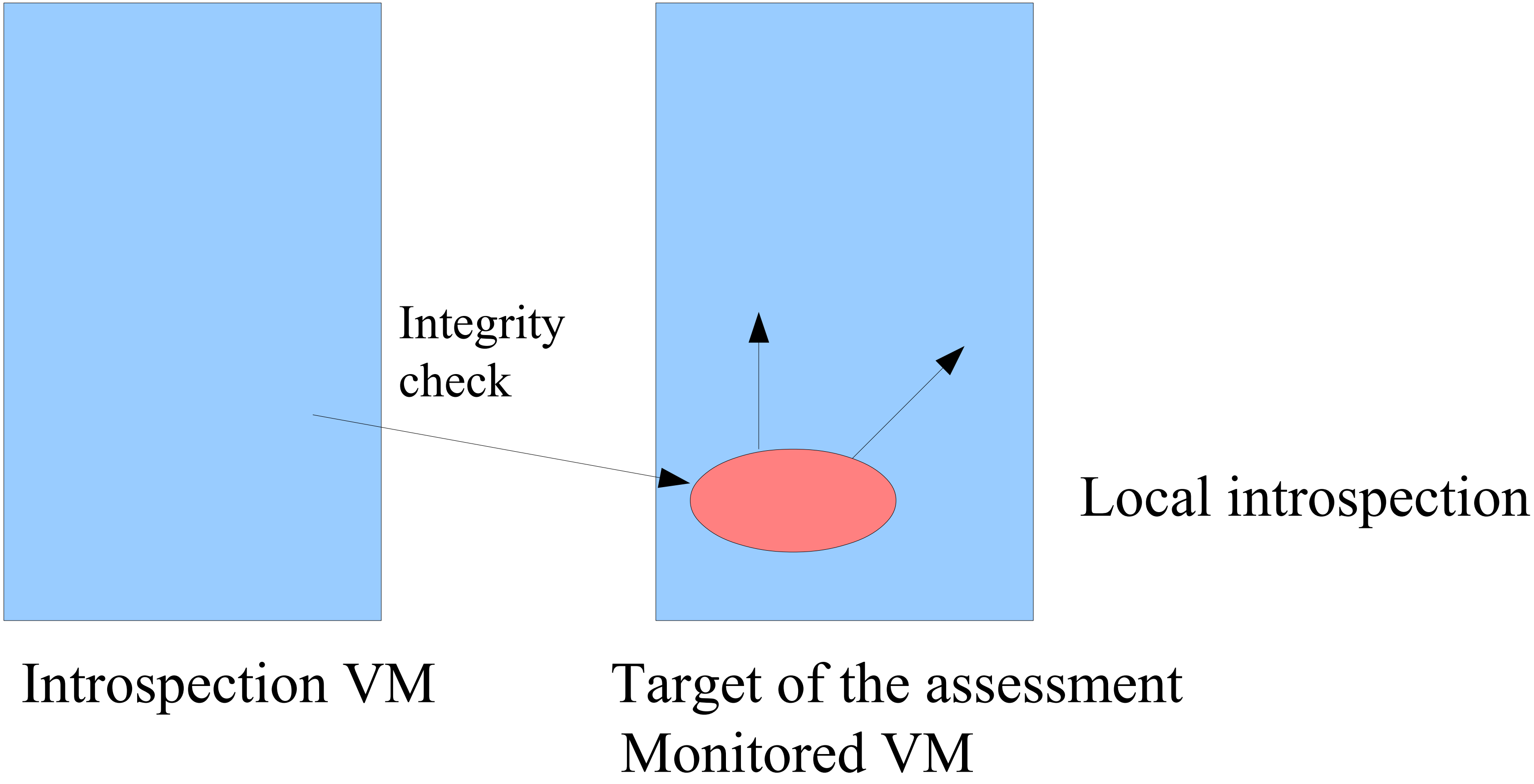
- There are alternative implementation of VMI
  - Asynchronous: the introspection VM evaluates some invariant independent of the current code the VM executes
  - Synchronous: the introspection VM monitors the execution of the other VM and, at some predefined moments,
    - freezes the VM execution
    - evaluates the invariant on the status of the frozen VM
    - resume the execution or kills the VM
- Synchronous is more complex because of the VMs synchronization
- Semantic gap problem:

  the Introspection VM access single memory positions but the invariant is defined at a higher abstraction level

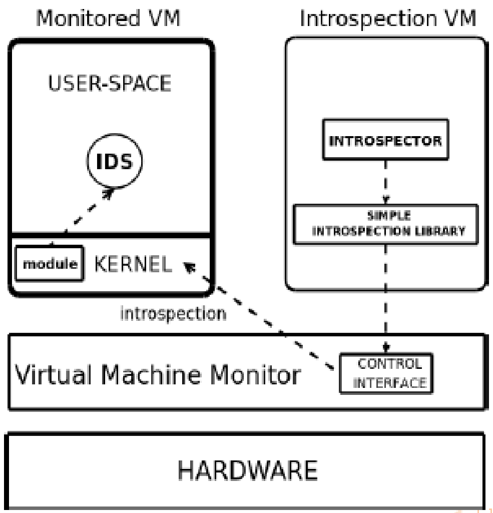
Virtual Machine Introspection - 3

- The VMM separates
  - The environment to be monitored, monitored VM
  - The monitoring environment, introspection VM
- The VMM separations results in controls more expensive but more robust than those implemented by two processes sharing memory
- To minimize the control overhead, a chain of trust is built where
  - some components in the monitored VM implement some control
  - the introspection VM checks the integrity of these components
- In any case, the controls requires the formal definition of a process self to be compared against the actual process behaviour

Virtual Machine Introspection -1 - bis

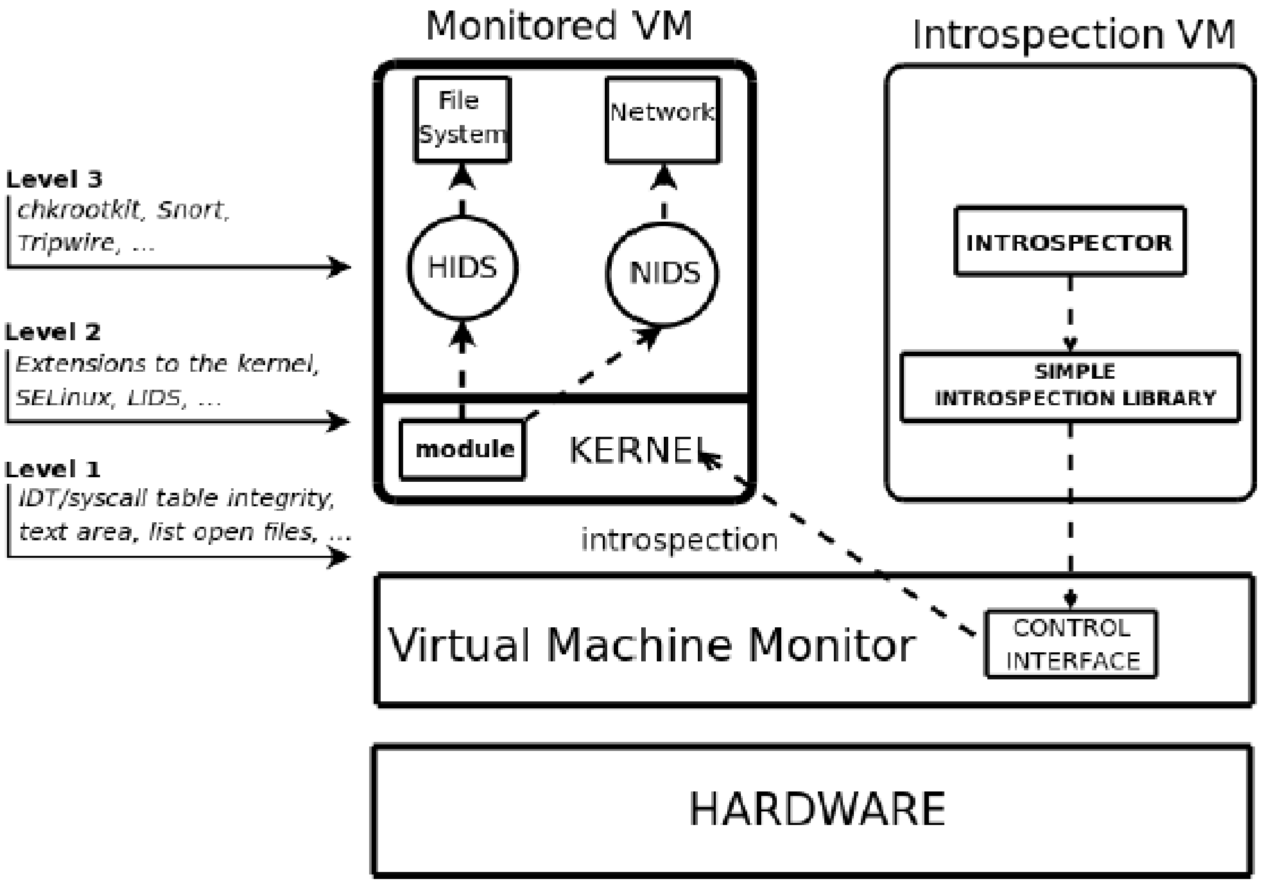


VM Introspection: the modular solution



- A simple introspection library to access the memory of the Monitored VM
- A kernel module that checks the integrity of the IDS on the Monitored VM
- The integrity of the kernel of the Monitored VM is protected by the Introspector in the Introspection VM
- Definition of the Introspector depends upon that of the module in the kernel
- Checks can be implemented anytime a given number of kernel invocation has occurred

Chain of Trust


Monitored VM
Introspection VM
File System
Network
HIDS
NIDS
module
KERNEL
INTROSPECTOR
SIMPLE INTROSPECTION LIBRARY
Level 3
chkrootkit, Snort, Tripwire, ...
Level 2
Extensions to the kernel, SELinux, LIDS, ...
Level 1
IDT/syscall table integrity, text area, list open files, ...
introspection
Virtual Machine Monitor
CONTROL INTERFACE
HARDWARE

Further advantages of VMI

- Full visibility of the system running inside the Monitored VM: the Introspection VM can access every Monitored VM component, such as the main memory or the processor's registers.

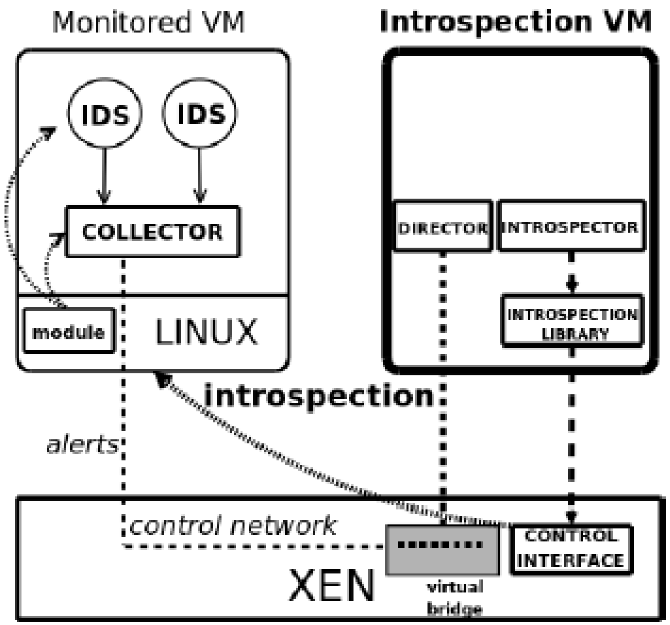
- Transparency: the security checks can be implemented without modifying the software on the Monitored-VM and they are almost invisible even to time based control

  - If the underlying architecture fully support virtualization, no software on the Monitored VM has to be updated
  - Otherwise we may have to modify the kernel but not the application running on the Monitored VM

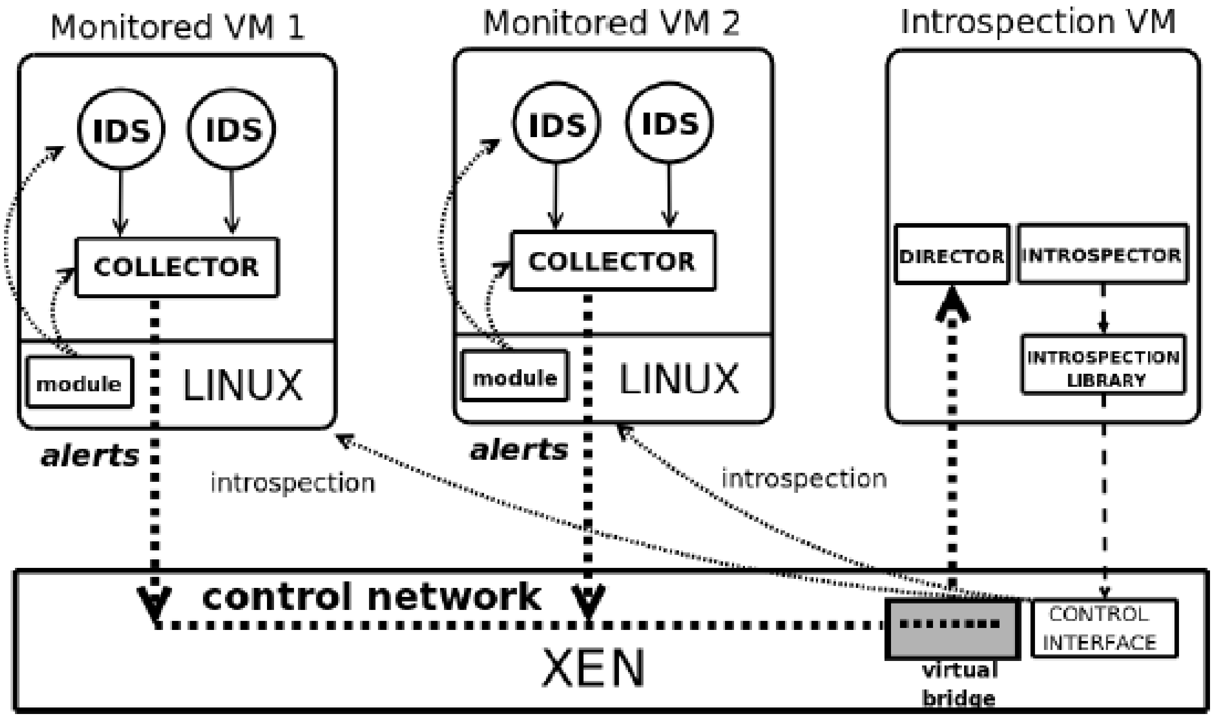
A full HIDS: Introspection and Alerts

Introspection VM: monitors all the VMs.

- The introspector protects kernel integrity.
- The director:
  1. collects the alerts;
  2. executes actions in response to an alert: stops a VM.



A more general case



Semantic Integrity and Introspection

A trivial attack classification

- Attacks against user-level processes:
  - the attacker injects some code into a process
  - the attacker diverges the original control-flow to execute the injected code.
- Attacks against the kernel modify
  - some kernel functionalities
  - the kernel behavior to hide any sign of the attack.
- User level attacks are the first step of a complex attack = a privilege escalation that, after increasing the attacker privileges results in the execution of an attack against
  - the kernel or
  - the kernel and then the VMM

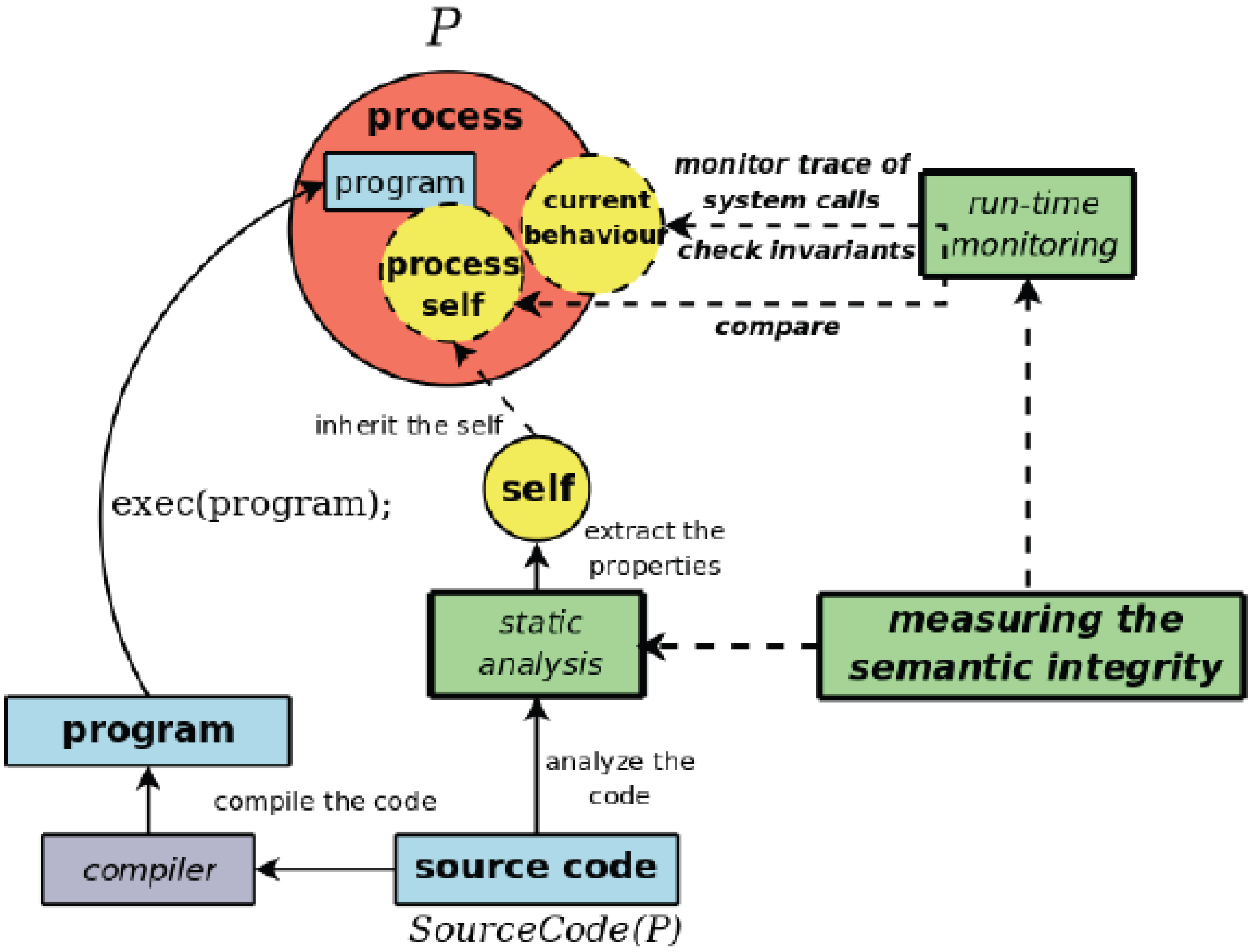
Process Self

- Process Self
  - The process properties that determine its run-time behavior
  - It can be approximated through static analysis.
- Axiom: Any difference between the process current behavior and the process self is due to an attack.

- Measuring the semantic integrity:
  - the approximation of the process self
  - the monitoring of the actual process behavior to assure its coherence with the process self.

- If P is a generic process that we want to protect.
  - Self (P) refers to the process self of P
  - SourceCode(P) is the source code of P program= syntactic integrity

All the relations



Self and OS calls

- It is widely accepted that an abstract description of a process self should consider the OS calls issued by the process
- Any attempt to violate the security policy, to hide the trace of an attack, to avoid intrusion detection mechanisms involves some interaction with the OS
- Hence the process self should be defined in terms of the OS calls
- Two alternative approaches to define the self
    - Monitor and learn = anomaly detection
    - Deduce the self from some representation of the process code
- The first is more general as it does not require the availabilty of the source code but less accurate

Self: Alternative Descriptions

Default Allow = black list (no politically correct)

- Forbidden Calls: the set of system calls that P cannot issue
- Forbidden Parameters: the set of system calls that P cannot issue and assertions on the parameters it cannot transmit to a call

Default Deny = whitelist (even more no politically correct)

- Hashing or Memory Invariants; memory invariants to be evaluated anytime P issues a given system call
- Allowed Calls: the set of system calls that P can issue and assertions on their parameters
- Enriched Traces: the sequence(s) of system calls that P may issues in one execution; each call may be coupled with an assertion on the process memory

Enriched Traces

- A set of enriched traces fully describes alternative legal behaviors of P

- Proper static tools may be designed to map SourceCode(P) into Self (P) described through enriched traces = <CFG(P), IT(P) >

- CFG(P) =
  - context-free grammar that defines the system call traces that P may issue during its execution
  - a set of strings on an alphabet with a symbol for each system call

- IT(P)= a set of invariants {I(P, 1), …, I(P, n) }, each associated with a program point i, 1≤i≤n, where P invokes a system call.

Grammar Generation Algorithm - 1

- A static tool can generate CFG(P) while traversing AST(P), the abstrax syntax tree of the program of P

- CFG(P) = < T, F, S, R > where
  - T is a set of terminal symbols with one symbol for each distinct system call in SourceCode(P)
  - F is a set of non-terminal symbols, one for each function defined in SourceCode(P); each symbol corresponds to a subset of T.
  - S is the starting symbol, which corresponds to main;
  - R is the set of production rules X →B where
    - X is a non-terminal symbol
    - B a sequence of terminal and non-terminal symbols.

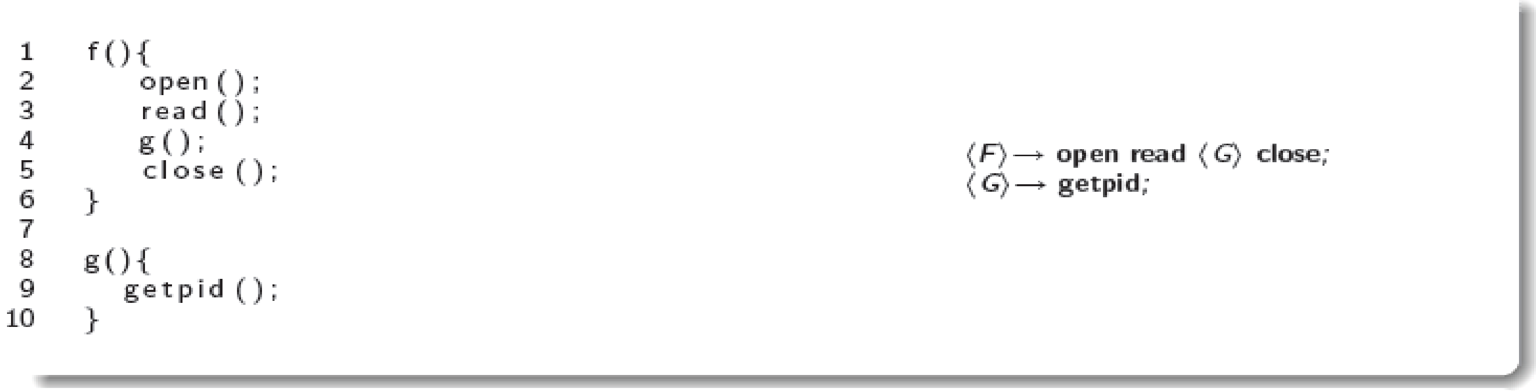
Grammar Generation Algorithm - 2

- GGA analyzes AST(P) and for each function *fun* defined inSourceCode(P) it inserts into F a new non-terminal symbol S_{fun} and a new rule Rnew into R with S_{fun} as its left-hand-side
- To generate the right-hand side of the rule, GGA linearly scans the definition of fun in SourceCode(P)
- Distinct production rules may be generated, according to the type of statements in the body of *fun*.
- For each statement, GGA generates a new rule and adds a new symbol to the right-hand side of Rnew .
- In this way, CFG(P) represents the system calls that *fun* can invoke and the ordering among the invocations in the body of *fun*.

Grammar Generation Algorithm - 2

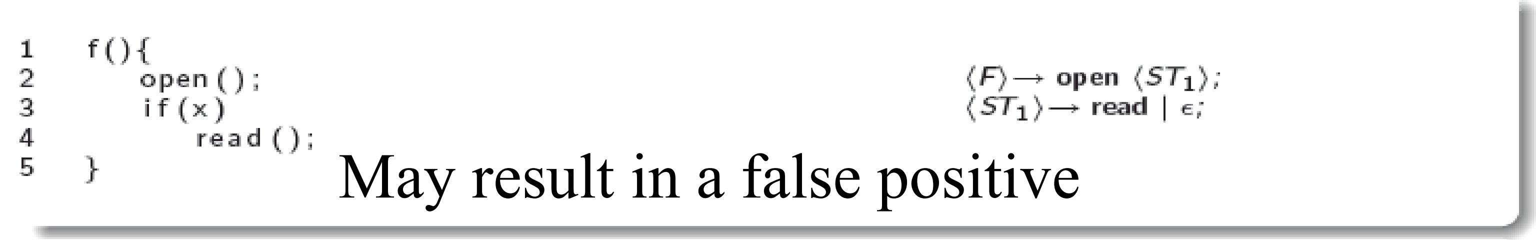
- GGA analyzes AST(P) and for each function *fun* defined inSourceCode(P) it inserts into F a new non-terminal symbol S_{fun} and a new rule Rnew into R with S_{fun} as its left-hand-side
- To generate the right-hand side of the rule, GGA linearly scans the definition of fun in SourceCode(P)
- Distinct production rules may be generated, according to the type of statements in the body of *fun*.
- For each statement, GGA generates a new rule and adds a new symbol to the right-hand side of Rnew .
- In this way, CFG(P) represents the system calls that *fun* can invoke and the ordering among the invocations in the body of *fun*.

Grammar Generation Algorithm - 3



```
f(){
    open();
    read();
    g();
    close();
}

g(){
   getpid();
}
```

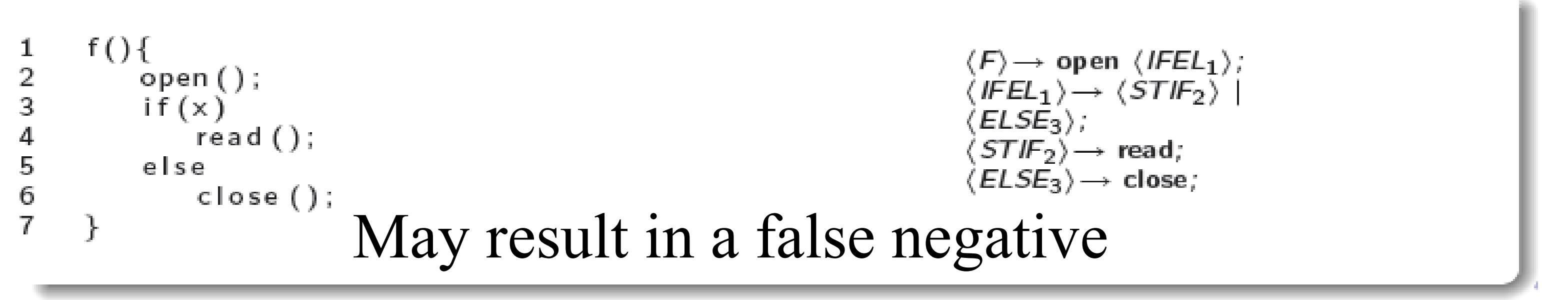
$\langle F\rangle \longrightarrow$ open read $\langle G\rangle$ close;
$\langle G\rangle \longrightarrow$ getpid;



```
f(){
    open();
    if(x)
        read();
}
```

$\langle F\rangle \longrightarrow$ open $\langle ST_1\rangle$;
$\langle ST_1\rangle \longrightarrow$ read | ϵ;

May result in a false positive



```
f(){
    open();
    if(x)
        read();
    else
        close();
}
```

$\langle F\rangle \longrightarrow$ open $\langle IFEL_1\rangle$;
$\langle IFEL_1\rangle \longrightarrow \langle STIF_2\rangle$ |
$\langle ELSE_3\rangle$;
$\langle STIF_2\rangle \longrightarrow$ read;
$\langle ELSE_3\rangle \longrightarrow$ close;

May result in a false negative

Assertion Generator -1

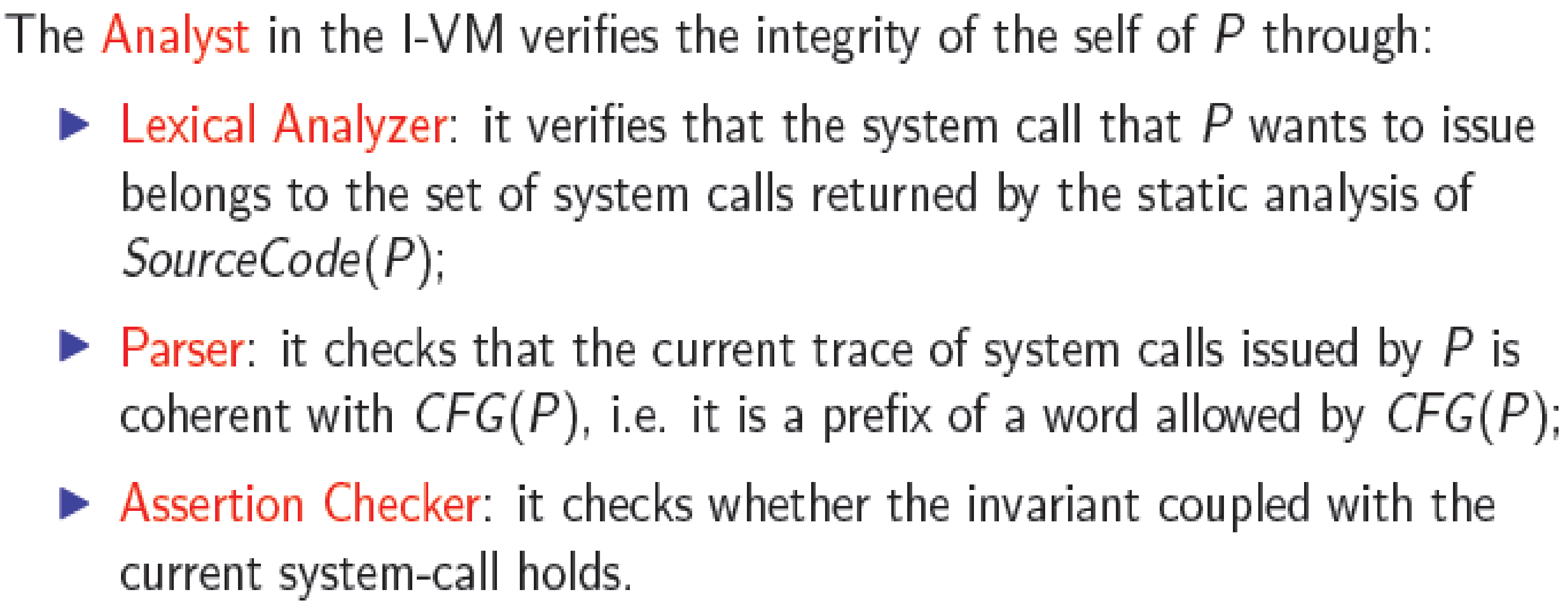
- The Assertion Generator traverses AST(P) and analyzes the variables, functions and language statements to build the invariant table (IT(P)).
- To simplify the analysis, we restrict to:
  - integer variables: only files and socket descriptors to express relations among these variables and the system calls;
  - string variables: in case of arrays of char statically declared, functions to manipulate strings are treated like assignments;
  - struct members: only integer or string type field.

Assertion Generator - 2

Any assertion is the composition of any of the followings:

- Parameters assertions. They express data-ow relations among parameters of distinct calls, e.g. the file descriptor in a read call is the result of a previous open call.
- File Assertions. To prevent symlink and race condition attacks, they check, as an example, that the real file-name corresponding to the le descriptor belongs to a known directory.
- Buffer length assertions. They check that the length of the string passed to a vulnerable function is not larger than the local buffer to hold it.
- Conditional statements assertions. They prevent problems due to impossible paths by relating a system call and the expression in the guard of a conditional statement (important difference wrt self described as CFG only)

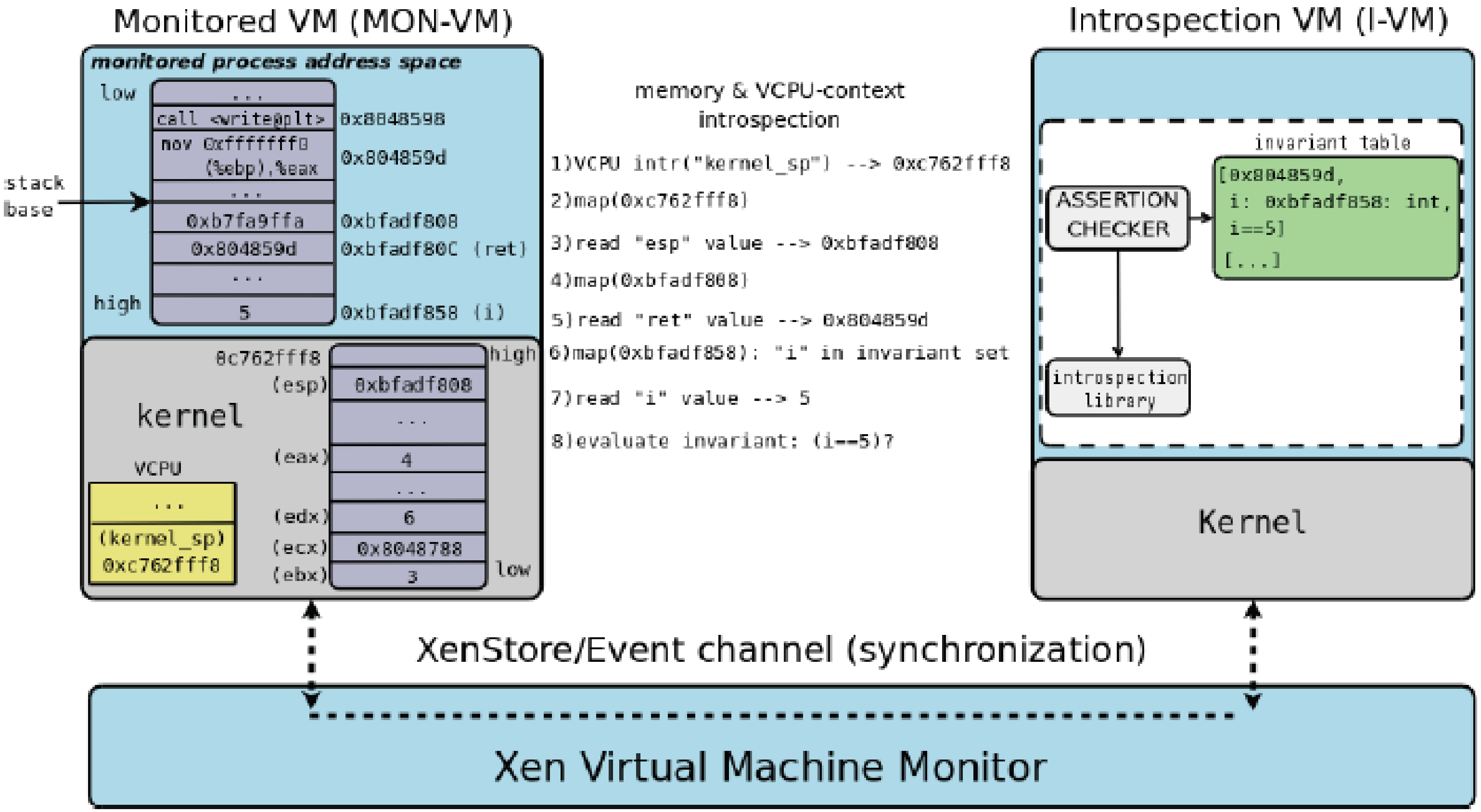
The Analyst - 2



The Analyst in the I-VM verifies the integrity of the self of P through:

- Lexical Analyzer: it verifies that the system call that P wants to issue belongs to the set of system calls returned by the static analysis of $SourceCode(P)$;
- Parser: it checks that the current trace of system calls issued by P is coherent with $CFG(P)$, i.e. it is a prefix of a word allowed by $CFG(P)$;
- Assertion Checker: it checks whether the invariant coupled with the current system-call holds.

Invariant Evaluation - 1



Monitored VM (MON-VM)

Introspection VM (I-VM)

memory & VCPU-context introspection

```
1)VCPU intr("kernel_sp") --> 0xc762fff8
2)map(0xc762fff8)
3)read "esp" value --> 0xbfadf808
4)map(0xbfadf808)
5)read "ret" value --> 0x804859d
6)map(0xbfadf858): "i" in invariant set
7)read "i" value --> 5
8)evaluate invariant: (i==5)?
```

XenStore/Event channel (synchronization)

Xen Virtual Machine Monitor

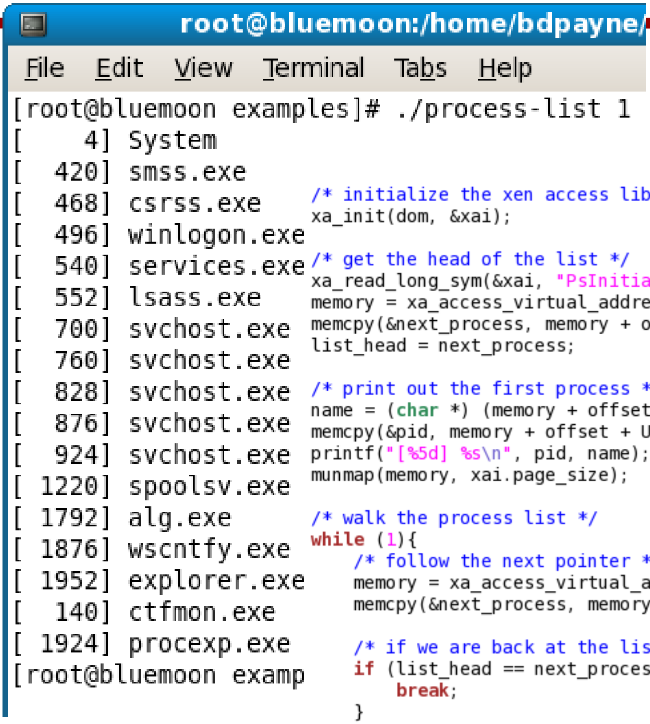
Memory Monitoring Implementation

Xen overview

- Runs directly on the physical hardware
- Special management domain is called Dom0 to provide a management interface
- The VMM gives Dom0 system access to a control library
  - create, destroy, start, pause, stop, and allocate resources to VMs from Dom0
- Provides drivers for the host’s physical hardware
- Can also request that memory pages allocated to unprivileged VMs

The XenAccess Library

- An open source VM introspection library
- Access to virtual addresses, kernel symbols, and more
- Works with Xen and dd-style memory image files
- Released in Spring 2006
- Maintained by Georgia Tech Inf. Sec. Center to encourage more research
- http://www.xenaccess.org


root@bluemoon:/home/bdpayne/
File Edit View Terminal Tabs Help

```
[root@bluemoon examples]# ./process-list 1
[    4] System
[  420] smss.exe
[  468] csrss.exe
[  496] winlogon.exe
[  540] services.exe
[  552] lsass.exe
[  700] svchost.exe
[  760] svchost.exe
[  828] svchost.exe
[  876] svchost.exe
[  924] svchost.exe
[ 1220] spoolsv.exe
[ 1792] alg.exe
[ 1876] wscntfy.exe
[ 1952] explorer.exe
[  140] ctfmon.exe
[ 1924] procexp.exe
[root@bluemoon examp
```

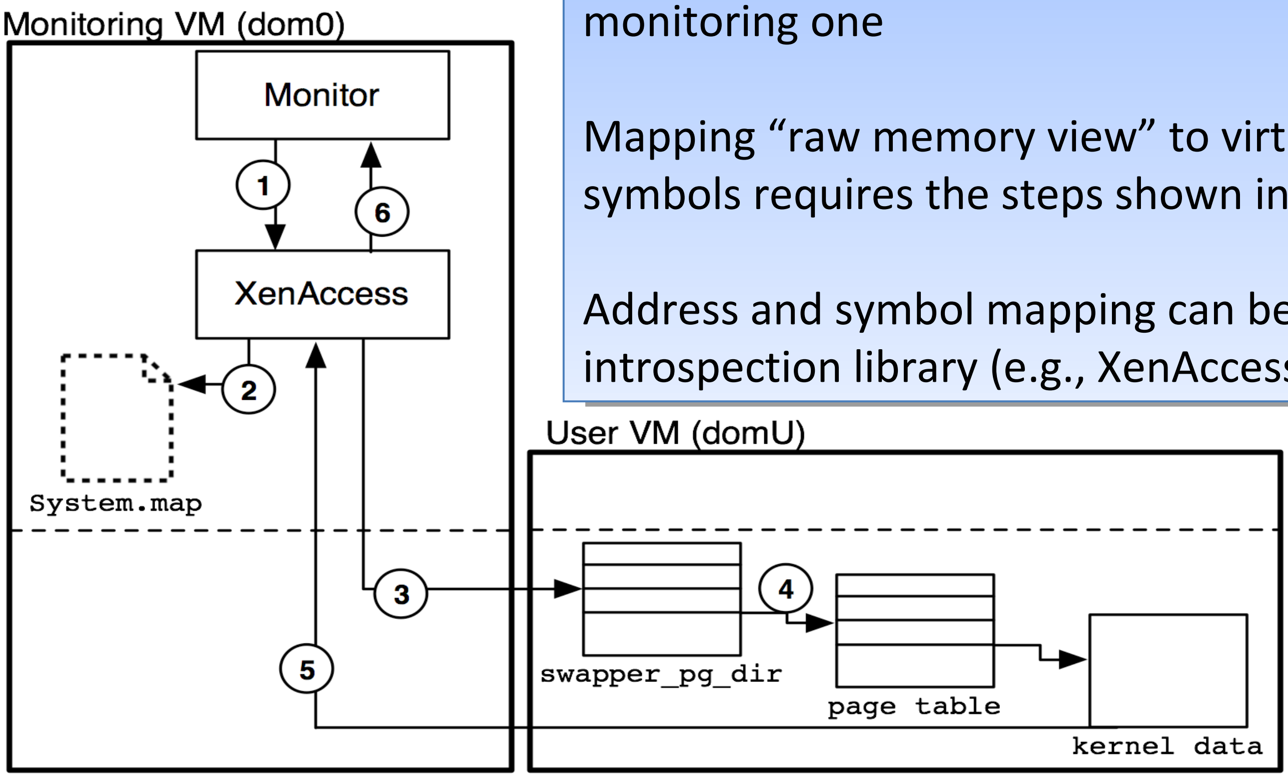
```
/* initialize the xen access library */
xa_init(dom, &xai);

/* get the head of the list */
xa_read_long_sym(&xai, "PsInitialSystemProcess", &list_head);
memory = xa_access_virtual_address(&xai, list_head, &offset);
memcpy(&next_process, memory + offset + ActiveProcessLinks_OFFSET, 4);
list_head = next_process;

/* print out the first process */
name = (char *) (memory + offset + ImageFileName_OFFSET);
memcpy(&pid, memory + offset + UniqueProcessId_OFFSET, 4);
printf("[%5d] %s\n", pid, name);
munmap(memory, xai.page_size);

/* walk the process list */
while (1){
    /* follow the next pointer */
    memory = xa_access_virtual_address(&xai, next_process, &offset);
    memcpy(&next_process, memory + offset, 4);

    /* if we are back at the list head, we are done */
    if (list_head == next_process){
        break;
    }

    /* print out the next process */
    name = (char *) (memory + offset + ImageFileName_OFFSET -
           ActiveProcessLinks_OFFSET);
    memcpy(&pid, memory + offset + UniqueProcessId_OFFSET -
           ActiveProcessLinks_OFFSET, 4);
    printf("[%5d] %s\n", pid, name);
    munmap(memory, xai.page_size);
}

/* cleanup */
xa_destroy(&xai);
```

Passive Monitoring

To monitor application memory of another virtual machine we have to map the memory into an address of the monitoring one

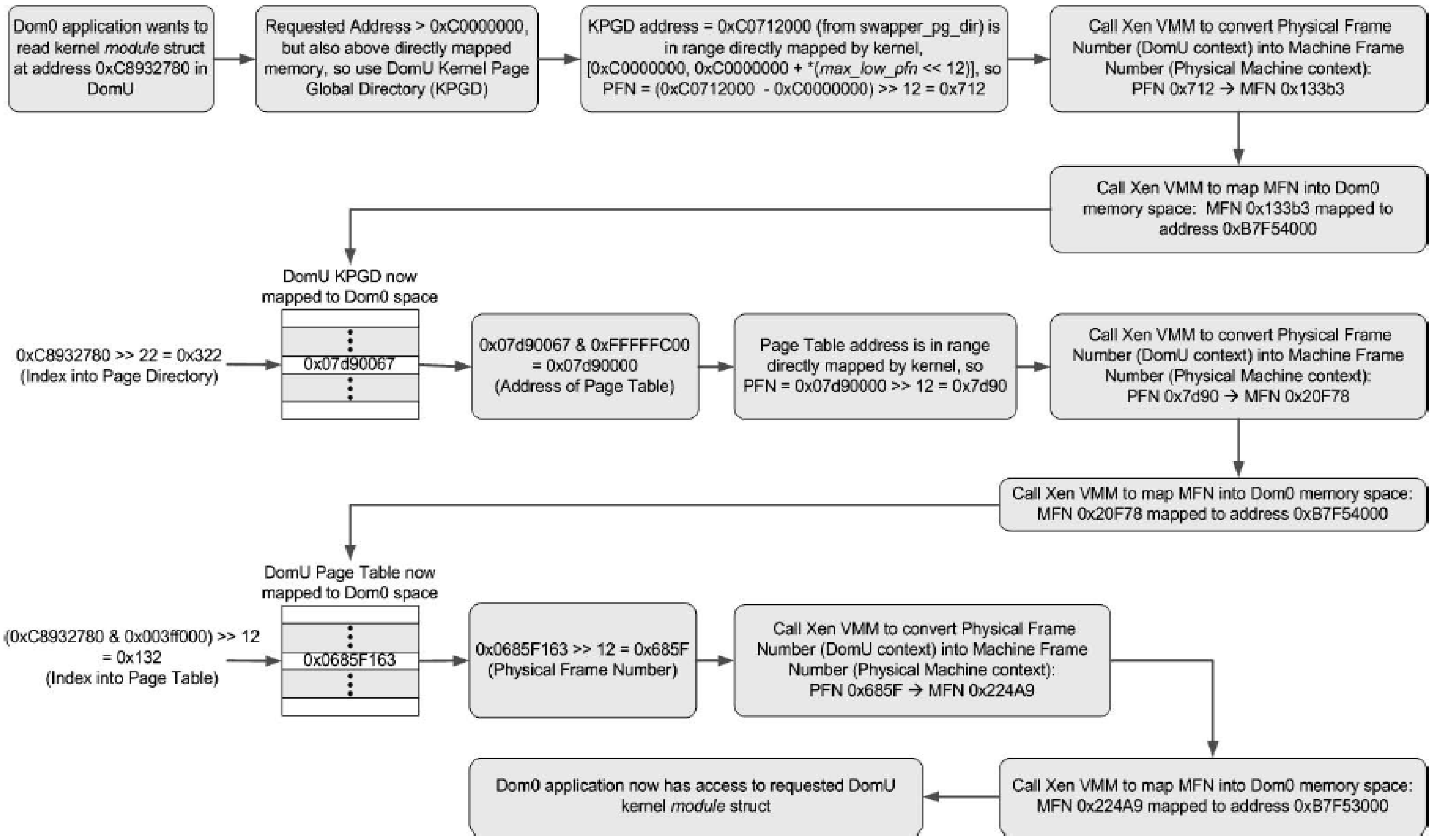
Mapping "raw memory view" to virtual addresses and symbols requires the steps shown in figure below.

Address and symbol mapping can be performed by a VM introspection library (e.g., XenAccess)



BD Payne, M Carbone, and W Lee. *Secure and Flexible Monitoring of Virtual Machines*. In ACSAC 2007.

Steps for Passive Monitoring



Dom0 application wants to read kernel *module* struct at address 0xC8932780 in DomU

Requested Address > 0xC0000000, but also above directly mapped memory, so use DomU Kernel Page Global Directory (KPGD)

KPGD address = 0xC0712000 (from swapper_pg_dir) is in range directly mapped by kernel, [0xC0000000, 0xC0000000 + *(*max_low_pfn* << 12)], so PFN = (0xC0712000 - 0xC0000000) >> 12 = 0x712

Call Xen VMM to convert Physical Frame Number (DomU context) into Machine Frame Number (Physical Machine context): PFN 0x712 → MFN 0x133b3

Call Xen VMM to map MFN into Dom0 memory space: MFN 0x133b3 mapped to address 0xB7F54000

DomU KPGD now mapped to Dom0 space

0xC8932780 >> 22 = 0x322 (Index into Page Directory)

0x07d90067

0x07d90067 & 0xFFFFFC00 = 0x07d90000 (Address of Page Table)

Page Table address is in range directly mapped by kernel, so PFN = 0x07d90000 >> 12 = 0x7d90

Call Xen VMM to convert Physical Frame Number (DomU context) into Machine Frame Number (Physical Machine context): PFN 0x7d90 → MFN 0x20F78

Call Xen VMM to map MFN into Dom0 memory space: MFN 0x20F78 mapped to address 0xB7F54000

DomU Page Table now mapped to Dom0 space

(0xC8932780 & 0x003ff000) >> 12 = 0x132 (Index into Page Table)

0x0685F163

0x0685F163 >> 12 = 0x685F (Physical Frame Number)

Call Xen VMM to convert Physical Frame Number (DomU context) into Machine Frame Number (Physical Machine context): PFN 0x685F → MFN 0x224A9

Call Xen VMM to map MFN into Dom0 memory space: MFN 0x224A9 mapped to address 0xB7F53000

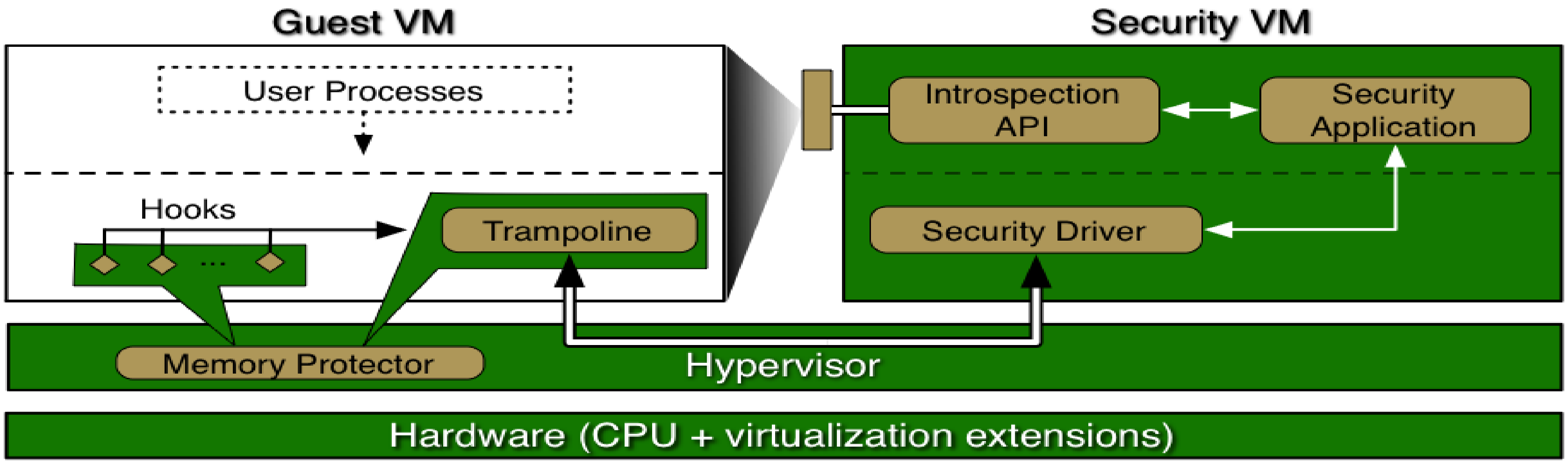
Dom0 application now has access to requested DomU kernel *module* struct

Active Monitoring

Monitoring application receives event notification from Guest VM when code execution reaches one of the hooks installed in the Guest VM kernel.

Hooks and all associated code are protected from tampering using hypervisor-enforced memory protections (i.e., User VM can not modify these security-critical components).

Hooks invoke trampoline, which transfers control to the security application.



BD Payne, M Carbone, M Sharif, and W Lee. *An Arch for Secure Active Monitoring Using Virtualization*. In Oakland 2008.

Ether: Experiments

- Two tools to test the Ether framework:
  - EtherUnpack: extracts hidden code from obfuscated malware
  - EtherTrace: Records system calls executed by obfuscated malware
- Evaluation
  - EtherUnpack: how well current tools extract hidden code by obfuscating a test binary and looking for a known string in the extracted code
  - EtherTrace: a test binary which executes a set of known operations obfuscated and then observes if these operations were logged by the tool

Ether: EtherUnpack Results

Packing Tool	PolyUnpack	Renovo	EtherUnpack
Armadillo	no	no	yes
Aspack	no	yes	yes
Asprotect	yes	yes	yes
FSG	yes	yes	yes
MEW	yes	yes	yes
MoleBox	no	yes	yes
Morphine	yes	yes	yes
Obsidium	no	no	yes
PECompact	no	yes	yes
Themida	no	yes	yes
Themida VM	no	no	yes
UPX	yes	yes	yes
UPX Scrambled	yes	yes	yes
WinUPack	no	yes	yes
Yoda’s Protector	no	yes	yes

Ether: EtherUnpack Results

PolyUnpack = Approach is based on the observation that sequences of packed or hidden code in a malware instance can be made self-identifying when its runtime execution is checked against its static code model

Renovo = An approach based on the observation that sequences of packed or hidden code in a malware instance can be made self-identifying when its runtime execution is checked against its static code model. Any new code is considered as an attack

Both use virtual machine emulators

Ether: EtherTrace Results

Packing Tool	Norman Sandbox	Anubis	EtherTrace
None	yes	yes	yes
Armadillo	no	no	yes
UPX	yes	yes	yes
Upack	yes	yes	yes
Themida	yes	yes	yes
PECompact	yes	yes	yes
ASPack	yes	yes	yes
FSG	yes	yes	yes
ASProtect	yes	no	yes
WinUpack	yes	yes	yes
tElock	yes	no	yes
PKLITE32	yes	yes	yes
Yoda’s Protector	no	yes	yes
NsPack	yes	yes	yes
MEW	yes	yes	yes
nPack	yes	yes	yes
RLPack	yes	yes	yes
RCryptor	yes	yes	yes

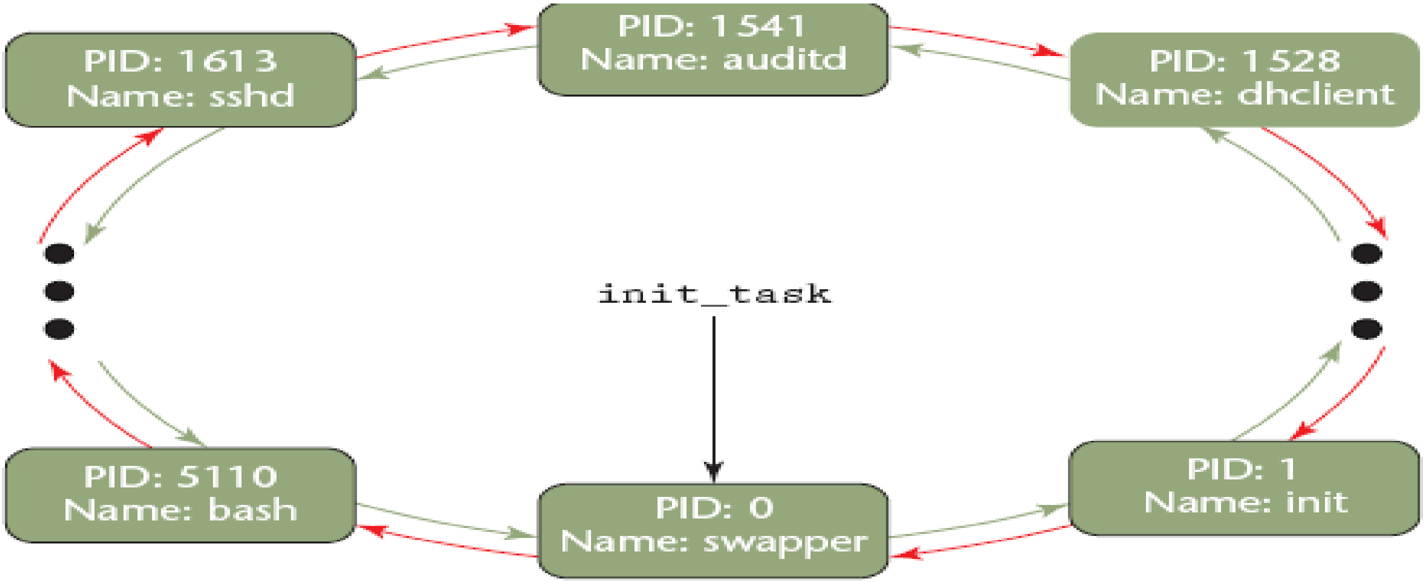
VIX

- Virtual Introspection for Xen
- Place in the privileged Dom0 VM
- Interact through a stable API
- Reduce the application's ability to perform inline processing (requests in real time)

How VIX works

- Pauses operation of the target VM
- Maps some of its memory into the Dom0
- Acquires and decodes the memory pages
- Resumes operation of the target VM
- Reference task_struct data structures
  - process ID, process name, memory map, and execution time
- Traverses the list of task_structs

List of task_structs



Linux stores this list as a circular double-linked list

Each kernel version has an associated memory address for the first process

VMI Functionality

- Not depend on any VM OS functionality for information
- VIX application
  - vix-ps,
  - vix-netstat,
  - vix-lsof,
  - vix-pstrings,
  - vix-lsmod,
  - vix-pmap, and
  - vix-top
- vix-ps
  - Traverse the entire task list
  - Output as the ps command

VM Introspection - VMware Initiatives

Security API’s

- Designed for security productization
- Agent runs within a VM
- Capabilities
  - Memory access events
  - Selected CPU events
  - VM lifecycle events
  - Access to VM memory & CPU state
  - Page Table walker

Security APIs (VMsafe)

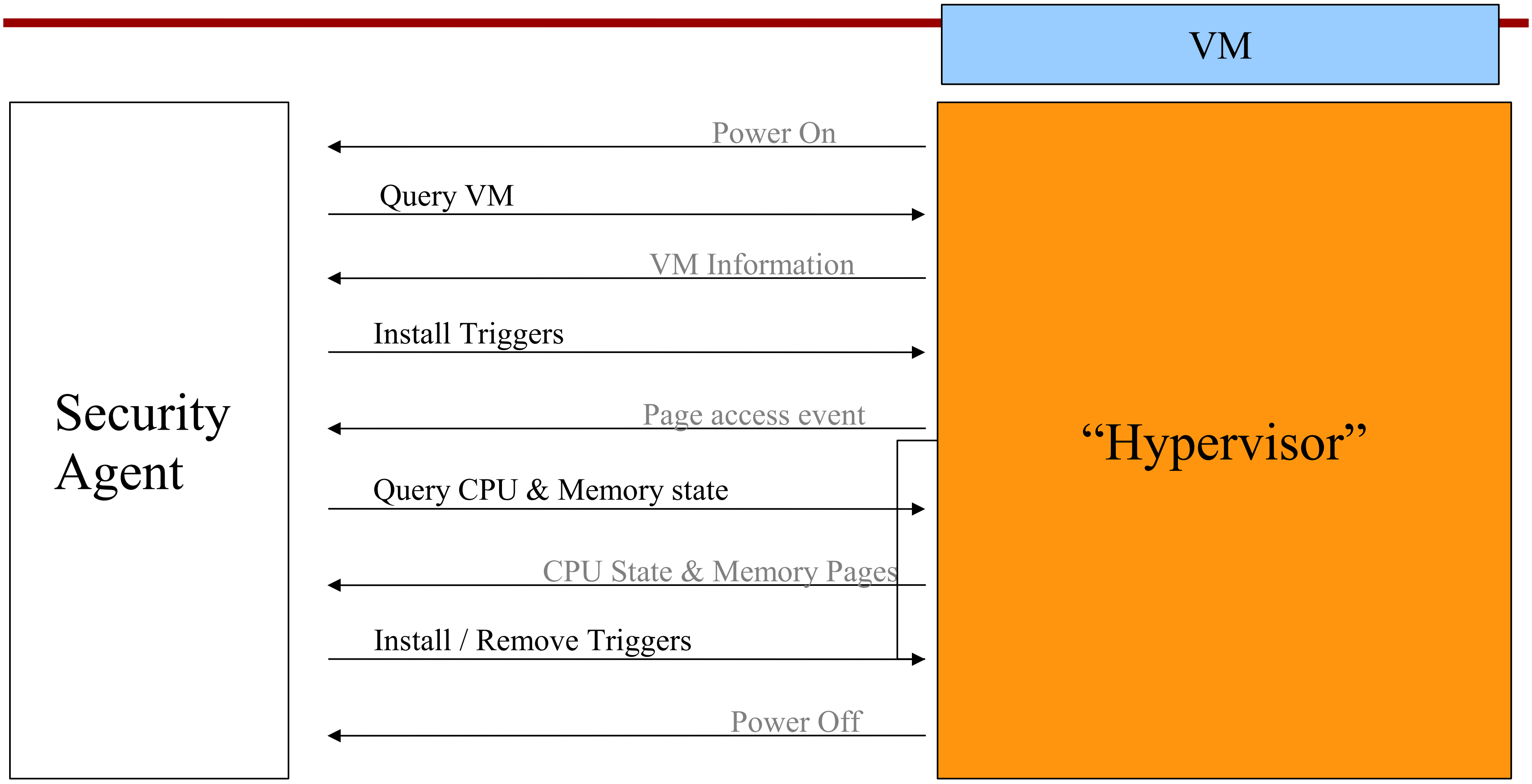
- A new security technology for virtualized environments that can help to protect your virtual infrastructure in ways previously not possible with physical machines.
- VMsafe provides a unique capability for virtualized environments through an API-sharing program to develop security products
- VMsafe enables third-party security products to gain the same visibility as the hypervisor into the operation of a virtual machine to identify and eliminate malware. Security vendors can leverage VMsafe to detect and eliminate malware that is undetectable on physical machines.
- This advanced protection is achieved through fine-grained visibility into the virtual hardware resources of memory, CPU, disk and I/O systems of the virtual machine that can be used to monitor every aspect of the execution of the system and stop malware

Security APIs (VMsafe)

Goals

- Better than physical
  - Exploit hypervisor interposition to place new security agent
  - Provide security coverage for the VM kernel (and apps)
- Hypervisor as a Base of Trust
  - Divide responsibilities between the hypervisor and in-VM agent
  - Hypervisor covers the VM kernel, the rest from within the VM
  - Insure in-VM security agent execution and correctness
- Security as an infrastructure service
  - “Agent less” security services for Vms
  - Flexible OS independent solutions

Verify-Before-Execute Flow



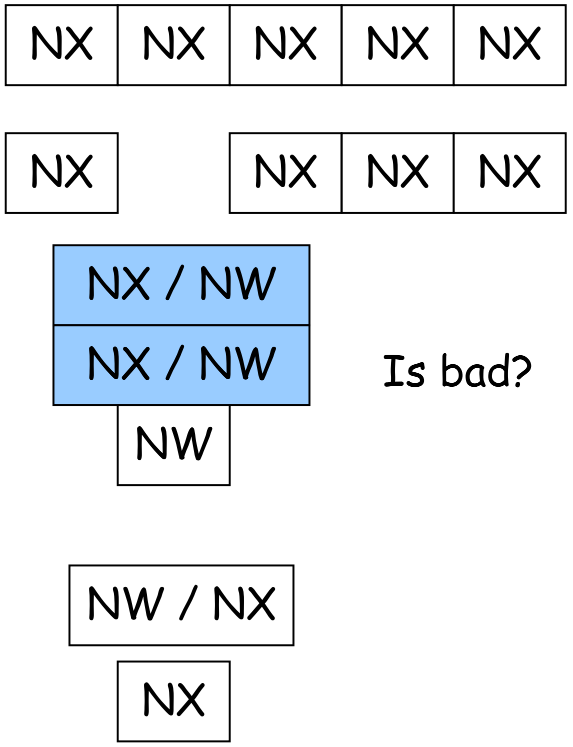
Sample Introspection Agents

Verify-Before-Execute

- Utilize memory introspection to validate all executing pages

Flow

- Trace all pages for execution access
- On execution detection
  - Trace for page modification
  - Verify if page contain malware
  - Remove execution trace
- On modification detection
  - Trace for execution
  - Remove modification trace



Security APIs – Use cases

VM Kernel coverage

- Detect infection in early boot process
  - Device opt ROM attacks
  - Boot loader
  - Boot records
  - OS image
- Detect code injection due to kernel vulnerabilities
- Detect self modifying code in kernel
- Lock kernel after initialization

Security APIs – Use cases cont’

- Watch dog services
  - Liveness check for in-VM security agent
    - Detect agent presence
    - Verify agent periodic execution
    - Protect agent code and static data

VMsafe – Network Introspection

- Capabilities
  - Place an inline network agent on any VM virtual nic
  - Allow reading, injecting, modifying, and dropping packets.

- Benefits
  - Efficiently monitor inter-VM network communication
  - Integrated support for live migration

- Virtualization only applications
  - Correlate VM internals with network policy. (using CPU/ Memory inspections one can learn OS version, patch level, configuration etc)
  - Build a trusted distributed firewall.

Retrospective Security

- Motivation
  - Detect whether you have been attacked in the past
  - Detect if you might be still compromised by a past attack
- Method
  - VMware Record & Replay allow for a deterministic replay of VM using recorded logs
  - Potentially the recordings have captured an attack
  - The security API's are detached from the recorded VM (unlike in-VM agent) and can attach to a replay session

Retrospective Security

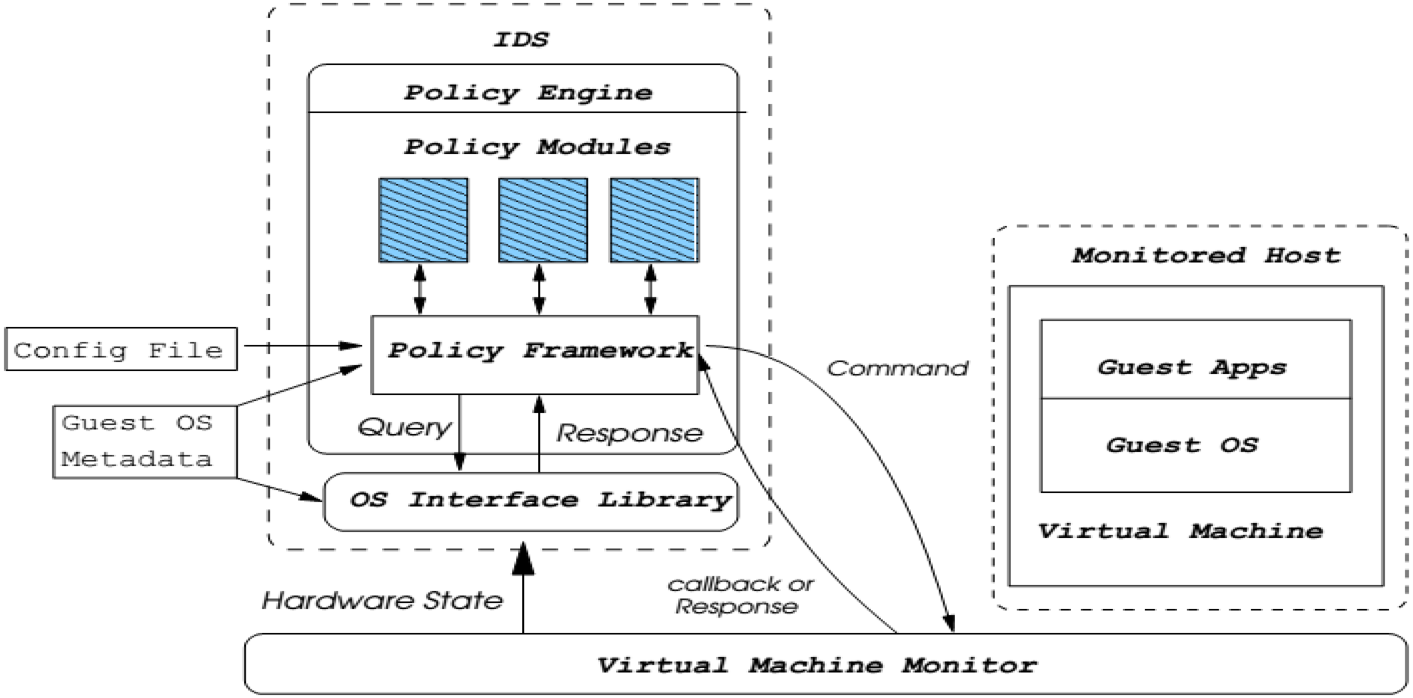
- What is it good for?
  - Run more aggressive policies that will not be acceptable in production environments
  - Discover 0days used to exploit your system
  - Learn how the malware / attacker have navigated your system
  - Use data tainting technique to detect any side effects that still exist on your system
  - Possibly clean the finding from last step on your production VM.
  - Learn about the scope of the damage done to your system, i.e. what is the extent of data leakage

Threat Monitoring/Interfering

- Other approaches are possible
- An important classification is
  - Monitor subject
  - Interfere with subject
- Only monitor subject behavior
  - Livewire
  - Monitor a system can only detect and report problems
- Interfere with subject behavior
  - LycosID, μDenali
  - Can actually respond to a detected threat
  - Might terminate the relevant processes or VM
  - Might reduce the resources available to the VM (starve the attacker)

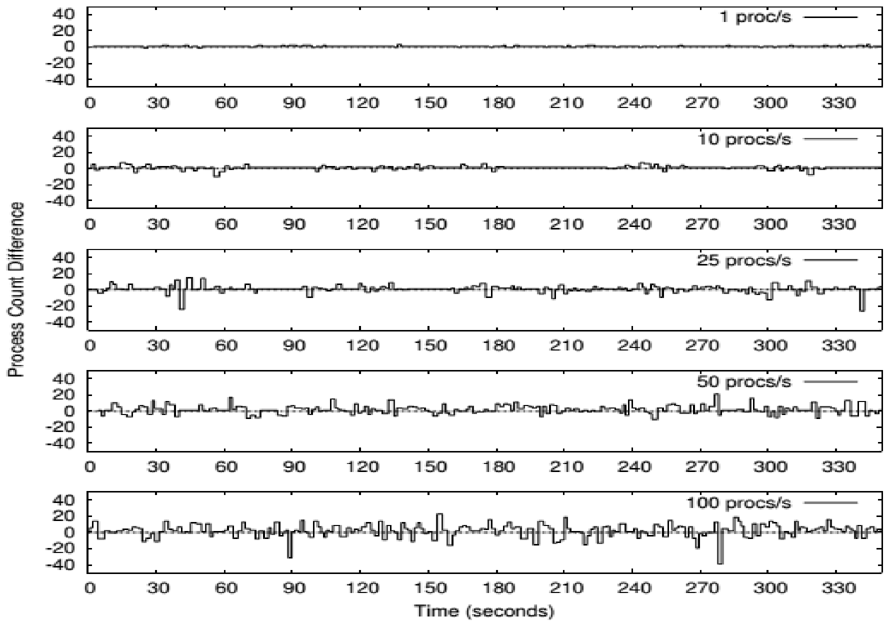
Livewire

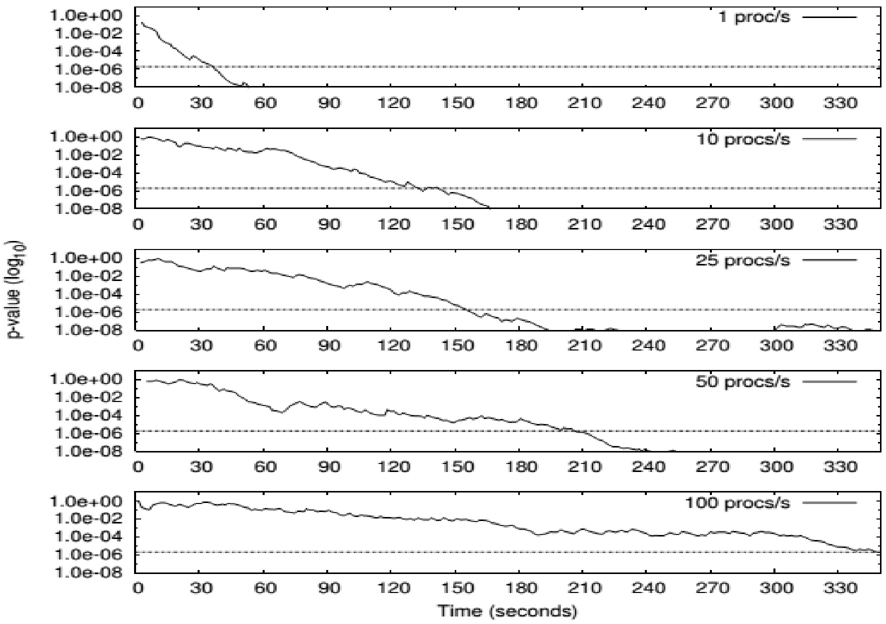
- An early host-based intrusion detection system
- Monitors VMs to gather information and detect attacks
- Merely reports it rather than interfering



LycosID

- Uses crossview validation techniques to compare running processes = compares high level and low level view of an object
- Patches running code to enable reliable identification of hidden processes





Manitou

- A VMI designed to detect malware
- Compares known instruction-page hashes with memory-page hashes at runtime before starting a program
- The instruction-page is corrupted and nonexecutable if no match
- A self attestation model

Manitou

- A VMI designed to detect malware
- Compares known instruction-page hashes with memory-page hashes at runtime before starting a program
- The instruction-page is corrupted and nonexecutable if no match
- A self attestation model

Semantic Awareness

- Account for different guest OS
- provide information that is more detailed
- parse kernel memory to build a process table map
- Unaware VMI simply see memory as bits

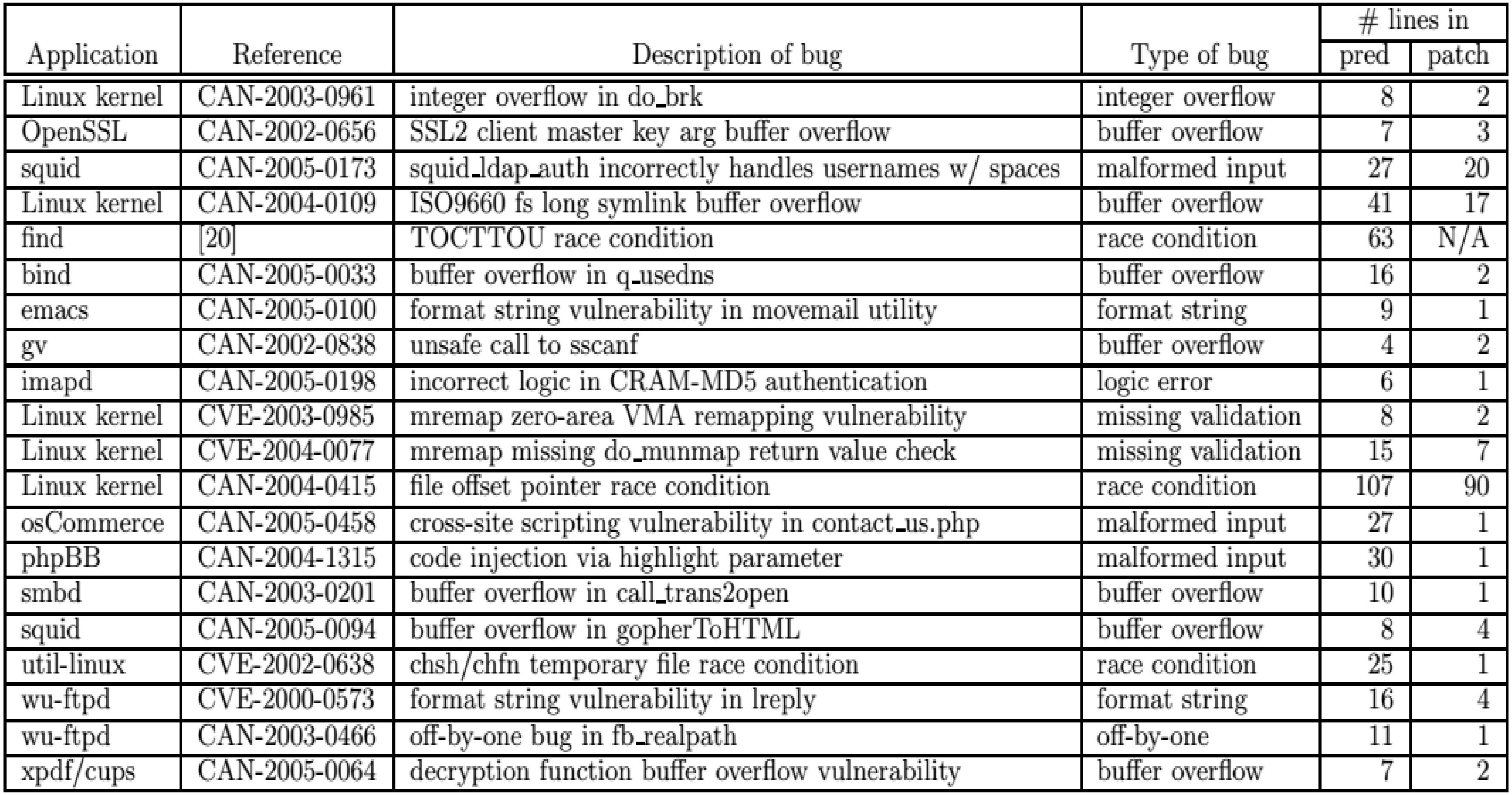
LARES

- Gives each VM an internal "hook"
  - Activate an external monitoring control upon execution
- Monitor can interrupt execution and pass control to a security mechanism
  - The hook is injected into the VM OS
  - Hypervisor write-protects both the hook and the transfers control
  - Triggers at a meaningful system execution point

IntroVirt

- It supports the construction of vulnerability specific predicates
- Attempt to bridge the "semantic gap" between
  - The VMI application
  - The target VM
- Using functionality on the target VM itself to lend context to the acquired data
- Basic mechanism insert assertion + replay VM

IntroVirt: the patch complexity



Application	Reference	Description of bug	Type of bug	# lines in	
				pred	patch
Linux kernel	CAN-2003-0961	integer overflow in do_brk	integer overflow	8	2
OpenSSL	CAN-2002-0656	SSL2 client master key arg buffer overflow	buffer overflow	7	3
squid	CAN-2005-0173	squid_ldap_auth incorrectly handles usernames w/ spaces	malformed input	27	20
Linux kernel	CAN-2004-0109	ISO9660 fs long symlink buffer overflow	buffer overflow	41	17
find	[20]	TOCTTOU race condition	race condition	63	N/A
bind	CAN-2005-0033	buffer overflow in q_usedns	buffer overflow	16	2
emacs	CAN-2005-0100	format string vulnerability in movemail utility	format string	9	1
gv	CAN-2002-0838	unsafe call to sscanf	buffer overflow	4	2
imapd	CAN-2005-0198	incorrect logic in CRAM-MD5 authentication	logic error	6	1
Linux kernel	CVE-2003-0985	mremap zero-area VMA remapping vulnerability	missing validation	8	2
Linux kernel	CVE-2004-0077	mremap missing do_munmap return value check	missing validation	15	7
Linux kernel	CAN-2004-0415	file offset pointer race condition	race condition	107	90
osCommerce	CAN-2005-0458	cross-site scripting vulnerability in contact_us.php	malformed input	27	1
phpBB	CAN-2004-1315	code injection via highlight parameter	malformed input	30	1
smbd	CAN-2003-0201	buffer overflow in call_trans2open	buffer overflow	10	1
squid	CAN-2005-0094	buffer overflow in gopherToHTML	buffer overflow	8	4
util-linux	CVE-2002-0638	chsh/chfn temporary file race condition	race condition	25	1
wu-ftpd	CVE-2000-0573	format string vulnerability in lreply	format string	16	4
wu-ftpd	CAN-2003-0466	off-by-one bug in fb_realpath	off-by-one	11	1
xpdf/cups	CAN-2005-0064	decryption function buffer overflow vulnerability	buffer overflow	7	2

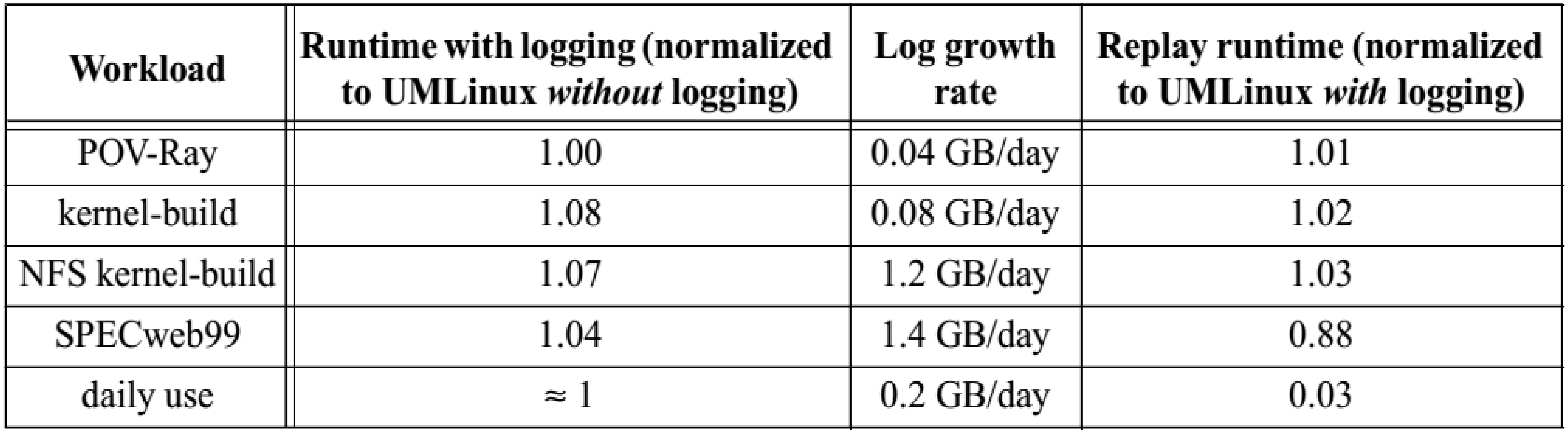
Event Replay

- Ability to replay, or log events on a VM is useful
  - Debugging OSs
  - Replaying compromises
- VM must record (in a log file) enough information to reconstruct interesting portions
- The penalty is to record extra information

Revirt

- An example of a logging VMI
- Serves as the basis for time-traveling VMs that allow replay from any previous VM state

ReVirt

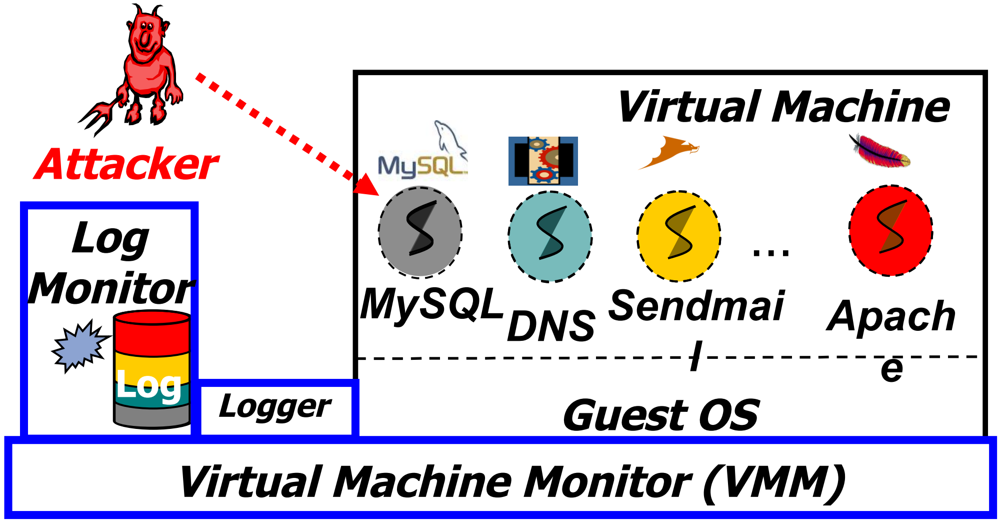


Workload	Runtime with logging (normalized to UMLinux *without* logging)	Log growth rate	Replay runtime (normalized to UMLinux *with* logging)
POV-Ray	1.00	0.04 GB/day	1.01
kernel-build	1.08	0.08 GB/day	1.02
NFS kernel-build	1.07	1.2 GB/day	1.03
SPECweb99	1.04	1.4 GB/day	0.88
daily use	≈ 1	0.2 GB/day	0.03

Tainting

APPROACH

- Track OS-level information flow provenance by assigning a unique identifier (color) to each potential malware entry point
- Color individual processes/data based on their interaction with potential entry points or other previously colored processes/data
- Color-based identification of malware contaminations
- Color-based reduction of log data to be analyzed
- Highlight event anomalies via abnormal color interactions present in logs
- Leverage virtual machine technology for tamper resistance of log coloring

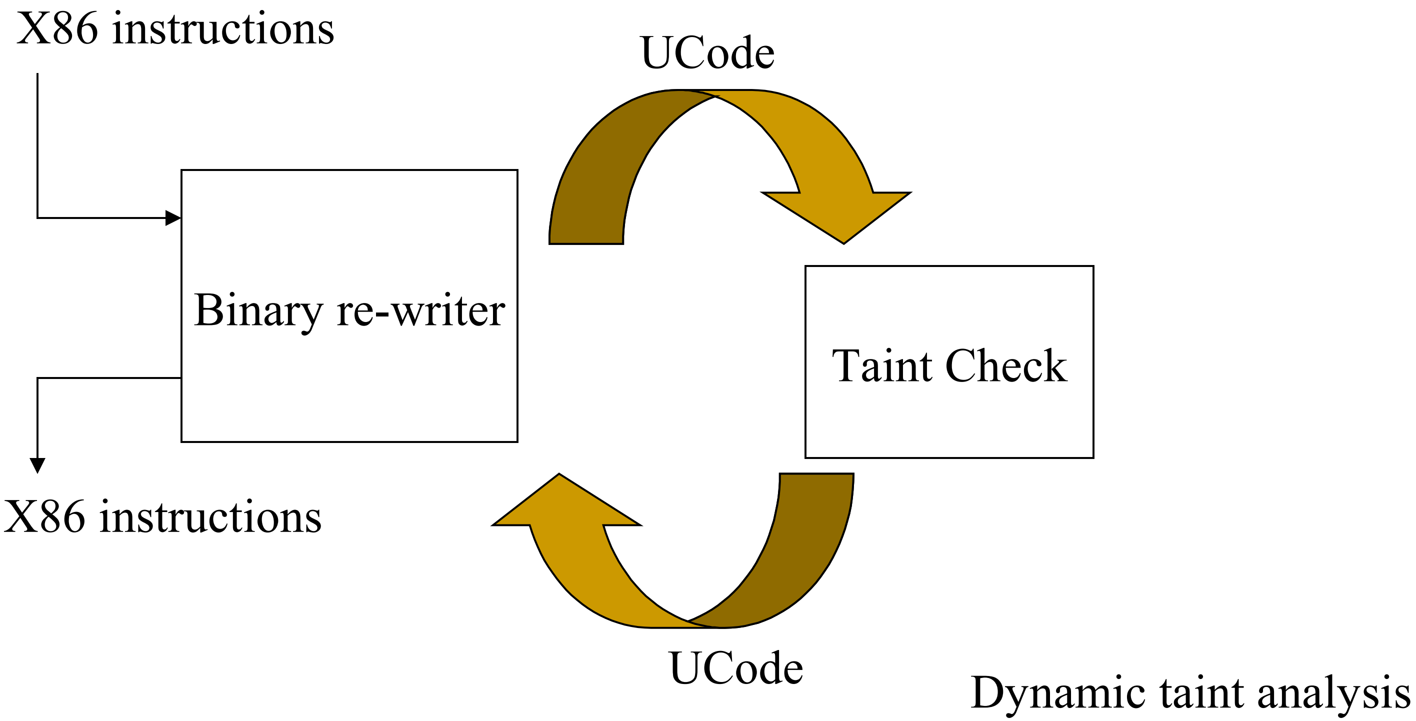


Dynamic taint analysis

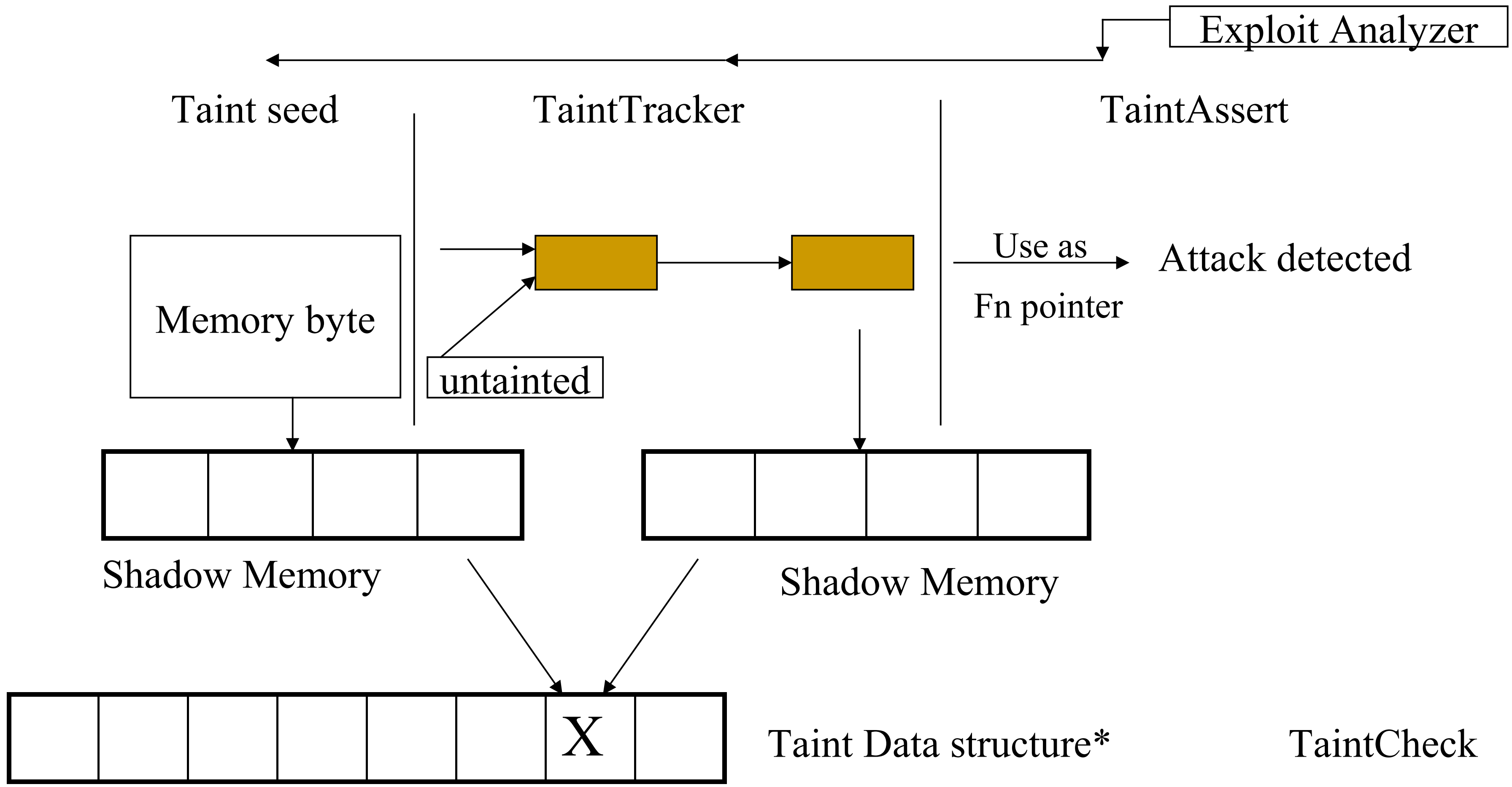
1. Taint analysis should be applied anytime a malicious user input can be the vector of an attack. Very important even in the case of web applications.
2. Mark input data as “tainted”
3. Monitor program execution to track how tainted attributes propagate
4. Check when tainted data is used in dangerous ways

Dynamic taint analysis

TaintCheck performs dynamic taint analysis on a program by running the program in its own emulation environment.



Dynamic taint analysis



*TDS holds the system call number, a snapshot of the current stack, and a copy of the data that was written

Dynamic taint analysis

TaintSeed

- It marks any data from untrusted sources as “tainted”
  - Each byte of memory has a four-byte shadow memory that stores a pointer to a Taint data structure if that location is tainted, or a NULL pointer if it is not.

Memory is mapped to TDS

Dynamic taint analysis

TaintTracker

- It tracks each instruction that manipulates data in order to determine whether the result is tainted.
    - When the result of an instruction is tainted by one of the operands, TaintTracker sets the shadow memory of the result to point to the same Taint data structure as the tainted operand.

Memory is mapped to TDS

Result is mapped to TDS

Dynamic taint analysis

TaintTracker

- It tracks each instruction that manipulates data in order to determine whether the result is tainted.
  - When the result of an instruction is tainted by one of the operands, TaintTracker sets the shadow memory of the result to point to the same Taint data structure as the tainted operand.

Memory is mapped to TDS

Result is mapped to TDS

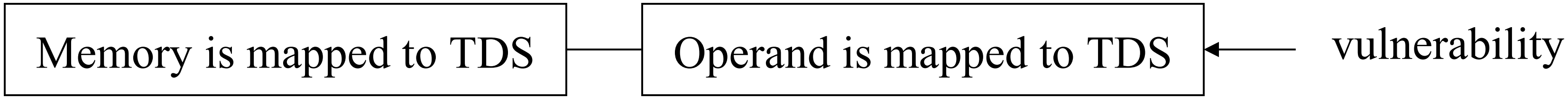
Dynamic taint analysis

TaintAssert

- It checks whether tainted data is used in ways that its policy defines as illegitimate

Exploit Analyzer

- The Exploit Analyzer can provide useful information about how the exploit happened, and what the exploit attempts to do.



Memory is mapped to TDS

Operand is mapped to TDS

vulnerability

Dynamic taint analysis

Types of attacks detected by TaintCheck are

- Overwrite attack
    - jump targets (such as return addresses, function pointers, and function pointer offsets), whether altered to point to existing code (existing code attack) or injected code (code injection attack).
- Format string attacks
    - an attacker provides a malicious format string to trick the program into leaking data or into writing an attacker-chosen value to an attacker-chosen memory address.
        - E.g.. use of %s and %x format tokens to print data from the stack or possibly other locations in memory.

Dynamic taint analysis

Why to use TaintCheck ?

- **Does not require source code or specially compiled binaries.**
- **Reliably detects most overwrite attacks.**
- **Has no known false positives.**
- **Enables automatic semantic analysis based signature generation.**

Evaluation

False Negatives

- A false negative occurs if an attacker can cause sensitive data to take on a value without that data becoming tainted.
  - E.g. if (x == 0) y = 0; else if (x == 1) y = 1; ...
- If values are copied from hard-coded literals, rather than arithmetically derived from the input.
  - IIS translates ASCII input into Unicode via a table
- If TaintCheck is configured to trust inputs that should not be trusted.
  - data from the network could be first written to a file on disk, and then read back into memory.

Evaluation

False Positives

- TaintCheck detects that tainted data is being used in an illegitimate way even when there is no attack taking place.
  - It indicates, there are vulnerabilities in the program
    - E.g. A program uses tainted data as a format string, but makes sure it does not use it in a malicious way.

Evaluation

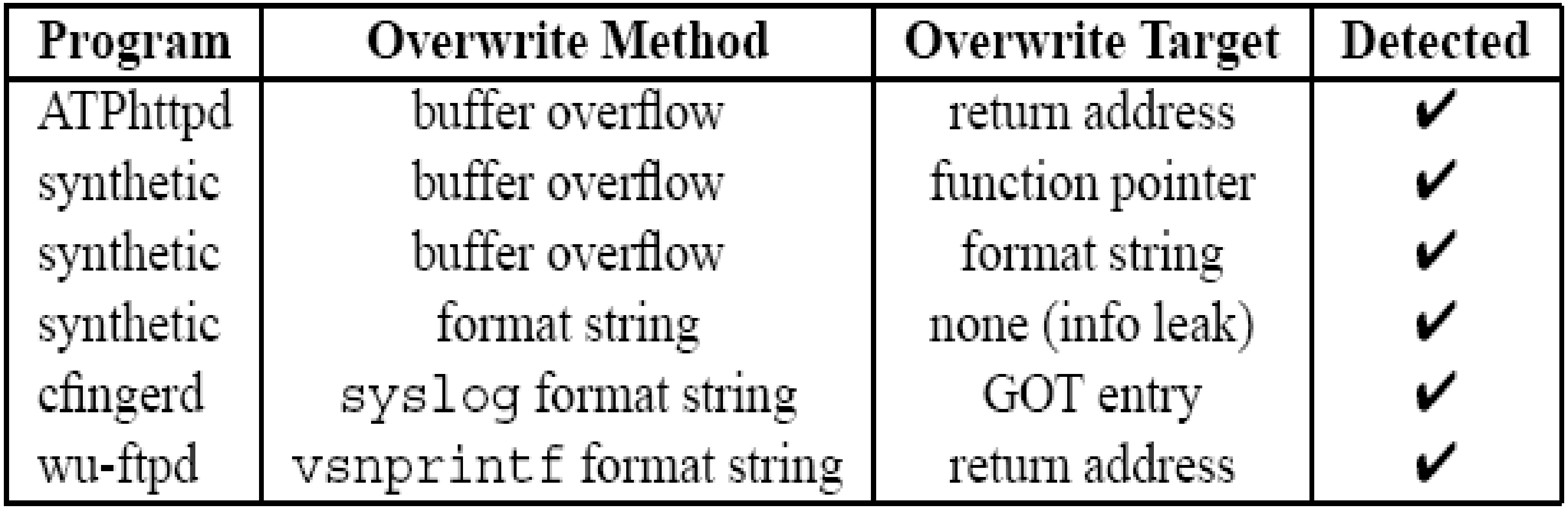
Synthetic

- To detect
  - Overwritten return addresses
  - Overwritten function pointer
  - Format string vulnerability

Actual exploits

- ATPhttpd exploit (buffer overflow)
- Cfingerd exploit (format string vulnerability)
- Wu-ftpd exploit (format string vulnerability)

Evaluation



Program	Overwrite Method	Overwrite Target	Detected
ATPhttpd	buffer overflow	return address	✓
synthetic	buffer overflow	function pointer	✓
synthetic	buffer overflow	format string	✓
synthetic	format string	none (info leak)	✓
cfingerd	`syslog` format string	GOT entry	✓
wu-ftpd	`vsnprintf` format string	return address	✓

Evaluation

Performance

- CPU bound
  - a 2.00 GHz Pentium 4, and 512 MB of RAM, running RedHat 8.0. was used to compress bzip2(15mb)
    - Normal runtime 8.2s
    - Valgrind nullgrind skin runtime25.6s (3.1 times longer)
    - Memcheck runtime 109s (13.3 times longer)
    - TaintCheck runtime 305s (37.2 times longer)
- Short-lived
- Common case

Evaluation

Performance

- CPU bound
- Short-lived
  - Basic blocks are cached and hence the penalty is acceptable over long lived programs. For short lived programs it is still significantly large
    - Normal runtime for Cfingerd was0.0222s
    - Valgrind nullgrind skin runtime took 13 times longer
    - Memcheck runtime took 32 times longer
    - TaintCheck runtime took 13 times longer
- Common case

Evaluation

Performance

- CPU bound
- Short-lived
- Common case
  - For network services the latency experienced is due to network and/or disk I/O and the TaintCheck performance penalty should not be noticeable

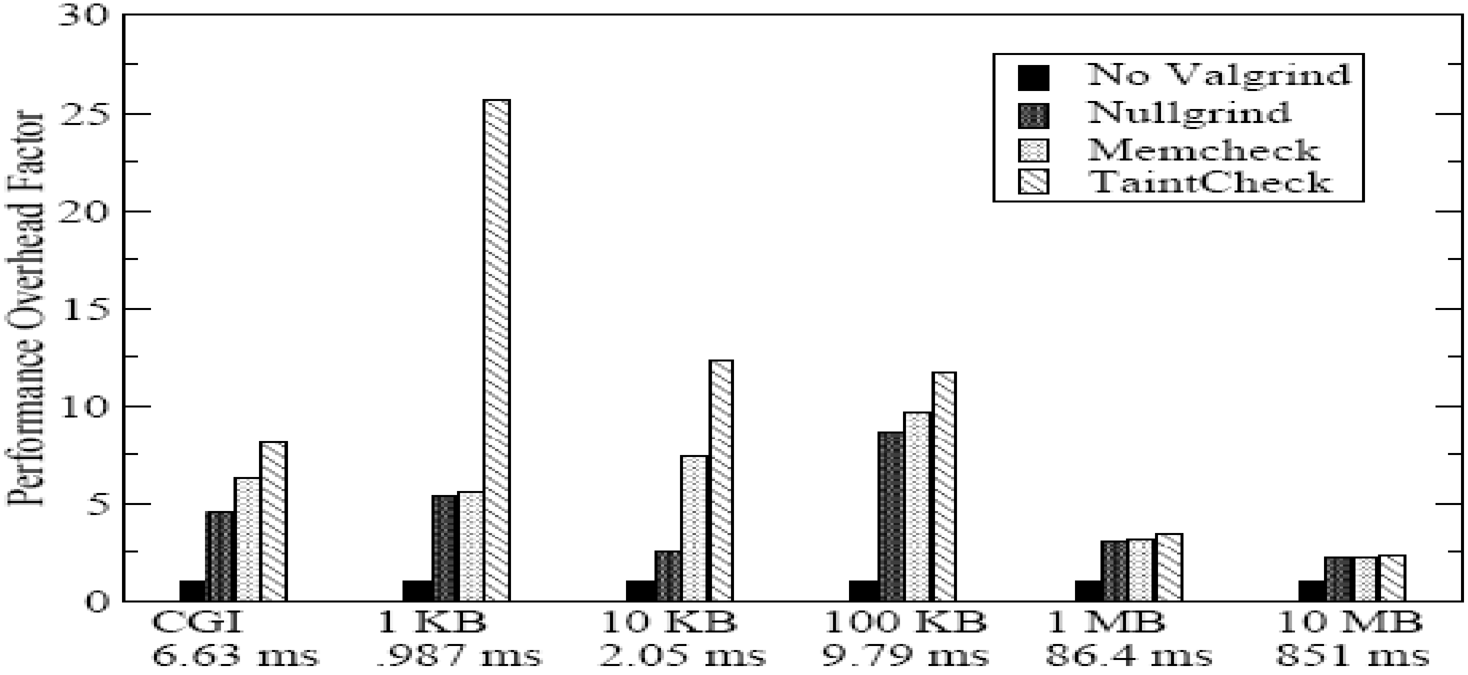
Application

It is not practical to implement TaintCheck as a standalone due to the performance overhead

- TaintCheck enabled honeypots could use TaintCheck to monitor all of its network services
  - TaintCheck will verify the exploit and provide additional information about the detected attack
- TaintCheck with OS randomization
  - identify which request contained an attack and generate signature for the attack or blocking future requests from the user.
- TaintCheck in a distributed environment

Evaluation

Performance


Performance Overhead Factor
No Valgrind
Nullgrind
Memcheck
TaintCheck
CGI
6.63 ms
1 KB
.987 ms
10 KB
2.05 ms
100 KB
9.79 ms
1 MB
86.4 ms
10 MB
851 ms