Host-Based Intrusion Detection

Vitaly Shmatikov
Reading Assignment

◆ Wagner and Dean. “Intrusion Detection via Static Analysis” (Oakland 2001).


After All Else Fails

◆ Intrusion prevention
  • Find buffer overflows and remove them
  • Use firewall to filter out malicious network traffic

◆ Intrusion detection is what you do after prevention has failed
  • Detect attack in progress
    – Network traffic patterns, suspicious system calls, etc.
  • Discover telltale system modifications
What Should Be Detected?

- Attempted and successful break-ins
- Attacks by legitimate users
  - For example, illegitimate use of root privileges
  - Unauthorized access to resources and data
- Trojan horses
- Viruses and worms
- Denial of service attacks
Where Are IDS Deployed?

- **Host-based**
  - Monitor activity on a single host
  - Advantage: better visibility into behavior of individual applications running on the host

- **Network-based (NIDS)**
  - Often placed on a router or firewall
  - Monitor traffic, examine packet headers and payloads
  - Advantage: single NIDS can protect many hosts and look for global patterns
Intrusion Detection Techniques

◆ Misuse detection
  • Use attack “signatures” (need a model of the attack)
    - Sequences of system calls, patterns of network traffic, etc.
  • Must know in advance what attacker will do (how?)
  • Can only detect known attacks

◆ Anomaly detection
  • Using a model of normal system behavior, try to detect deviations and abnormalities
    - E.g., raise an alarm when a statistically rare event(s) occurs
  • Can potentially detect unknown attacks

◆ Which is harder to do?
Misuse Detection (Signature-Based)

- Set of rules defining a behavioral signature likely to be associated with attack of a certain type
  - Example: buffer overflow
    - A setuid program spawns a shell with certain arguments
    - A network packet has lots of NOPs in it
    - Very long argument to a string function
  - Example: denial of service via SYN flooding
    - Large number of SYN packets without ACKs coming back
    - …or is this simply a poor network connection?

- Attack signatures are usually very specific and may miss variants of known attacks
  - Why not make signatures more general?
Extracting Misuse Signatures

- Use invariant characteristics of known attacks
  - Bodies of known viruses and worms, port numbers of applications with known buffer overflows, RET addresses of overflow exploits
  - Hard to handle mutations
    - Polymorphic viruses: each copy has a different body
- Big research challenge: fast, automatic extraction of signatures of new attacks
  - Honeypots are useful - try to attract malicious activity, be an early target
Anomaly Detection

Define a profile describing “normal” behavior
- Works best for “small”, well-defined systems (single program rather than huge multi-user OS)

Profile may be statistical
- Build it manually (this is hard)
- Use machine learning and data mining techniques
  - Log system activities for a while, then “train” IDS to recognize normal and abnormal patterns
- Risk: attacker trains IDS to accept his activity as normal
  - Daily low-volume port scan may train IDS to accept port scans

IDS flags deviations from the “normal” profile
Level of Monitoring

◆ Which types of events to monitor?
  • OS system calls
  • Command line
  • Network data (e.g., from routers and firewalls)
  • Processes
  • Keystrokes
  • File and device accesses
  • Memory accesses

◆ Auditing / monitoring should be scalable
STAT and USTAT

◆ Intrusion signature = sequence of system states
  • State machine describing the intrusion must be specified by an expert
    - Initial state: what system looks like before attack
    - Compromised state: what system looks like after attack
    - Intermediate states and transitions

◆ State transition analysis is then used to detect when system matches known intrusion pattern

[Ilgun, Porras]
**USTAT Example**

- User creates `File1`
  - `name(File1) == -*`
  - `typeof(File1) == link`
  - `owner(link_to(File1)) != user`
  - `name(link_to(File1)) exists_in Fileset #1`

- Access (user, euid) == root

- User executes `File1`

- Opens interactive subshell

- Creates symbolic link with same permissions and ownership as target

**State diagram of this attack**

- ***S1***: User creates `File1`
  - 1. Fileset #1 != empty
  - 2. Files are setuid

- ***S2***: User executes `File1`
  - 1. `name(File1) == -*`
  - 2. `typeof(File1) == link`
  - 3. `owner(link_to(File1)) != user`
  - 4. `name(link_to(File1)) exists_in Fileset #1`

- ***S3***
Statistical Anomaly Detection

- Compute statistics of certain system activities
- Report an alert if statistics outside range
- Example: IDES (Denning, mid-1980s)
  - For each user, store daily count of certain activities
    - For example, fraction of hours spent reading email
  - Maintain list of counts for several days
  - Report anomaly if count is outside weighted norm

Problem: the most unpredictable user is the most important
“Self-Immunology” Approach

Normal profile: short sequences of system calls

- Use strace on UNIX

... open, read, write, mmap, mmap, getrlimit, open, close ...

Remember last K events

Compute % of traces that have been seen before. Is it above the threshold?

Y → normal
N → abnormal

Raise alarm if a high fraction of system call sequences haven’t been observed before

[Forrest]
System Call Interposition

Observation: all sensitive system resources are accessed via OS system call interface
- Files, sockets, etc.

Idea: monitor all system calls and block those that violate security policy
- Inline reference monitors
- Language-level: Java runtime environment inspects stack of the function attempting to access a sensitive resource to check whether it is permitted to do so
- Common OS-level approach: system call wrapper
  - Want to do this without modifying OS kernel (why?)
Janus  [Berkeley project, 1996]
Policy Design

- Designing a good system call policy is not easy
- When should a system call be permitted and when should it be denied?
- Example: ghostscript
  - Needs to open X windows
  - Needs to make X windows calls
  - But what if ghostscript reads characters you type in another X window?
Trapping System Calls: ptrace()

- ptrace() – can register a callback that will be called whenever process makes a system call
  - Coarse: trace all calls or none
  - If traced process forks, must fork the monitor, too

Note: Janus used ptrace initially, later discarded…
Problems and Pitfalls

- Incorrectly mirroring OS state
- Overlooking indirect paths to resources
  - Inter-process sockets, core dumps
- Race conditions (TOCTTOU)
  - Symbolic links, relative paths, shared thread meta-data
- Unintended consequences of denying OS calls
  - Process dropped privileges using setuid but didn’t check value returned by setuid… and monitor denied the call
- Bugs in reference monitors and safety checks
  - What if runtime environment has a buffer overflow?
Incorrectly Mirroring OS State

Policy: “process can bind TCP sockets on port 80, but cannot bind UDP sockets”

6 = socket(UDP, …)  Monitor: “6 is UDP socket”
7 = socket(TCP, …)  Monitor: “7 is TCP socket”
close(7)
dup2(6,7)  Monitor’s state now inconsistent with OS
bind(7, …)  Monitor: “7 is TCP socket, Ok to bind”
Oops!
TOCTTOU in Syscall Interposition

◆ User-level program makes a system call
  • Direct arguments in stack variables or registers
  • Indirect arguments are passed as pointers

◆ Wrapper enforces some security condition
  • Arguments are copied into kernel memory (why?) and analyzed and/or substituted by the syscall wrapper

◆ What if arguments change right here?

◆ If permitted by the wrapper, the call proceeds
  • Arguments are copied into kernel memory
  • Kernel executes the call
Exploiting TOCTTOU Conditions

- Page fault on an indirect syscall argument
  - Force wrapper to wait on disk I/O and use a concurrent process to replace already checked arguments
  - Example: `rename()` – see Watson’s paper
    - Page out the target path of `rename()` to disk
    - Wrapper checks the source path, then waits for target path
    - Concurrent attack process replaces the source path

- Voluntary thread sleeps
  - Example: `TCP connect()` – see Watson’s paper
    - Kernel copies in the arguments
    - Thread calling `connect()` waits for a TCP ACK
    - Concurrent attack process replaces the arguments
TOCTTOU via a Page Fault

[Diagram showing the timeline of events involving kernel and user processes, with annotations for attack vectors and specific system calls.]
TOCTTOU on Sysjail

Exploitable race window between two copyin() calls

Process 1

kernel

user

P1 sets original address

Sysjail copies in and checks original IP address

bind() system call copies in replaced address for use in operation

Shared Memory

0.0.0.0

192.168.100.1

Process 2

user

P2 replaces address in shared memory from second processor

slide 24
Mitigating TOCTTOU

- Make pages with syscall arguments read-only
  - Tricky implementation issues
  - Prevents concurrent access to data on the same page

- Avoid shared memory between user process, syscall wrapper and the kernel
  - Argument caches used by both wrapper and kernel
  - Message passing instead of argument copying
    - Why does this help?

- System transactions

- Integrate security checks into the kernel
  - Requires OS modification!
Interposition + Static Analysis

Assumption: attack requires making system calls
1. Analyze the program to determine its expected behavior
2. Monitor actual behavior
3. Flag an intrusion if there is a deviation from the expected behavior
   • System call trace of the application is constrained to be consistent with the source or binary code
   • Main advantage: a conservative model of expected behavior will have zero false positives
Runtime Monitoring

◆ One approach: run slave copy of application
  • Replication is hard; lots of non-determinism in code
    – Random number generation, process scheduling, interaction with outside environment
  • Slave is exposed to the same risks as master
    – Any security flaw in the master is also present in the slave
  • Virtual machines make problem easier!

◆ Another approach: simulate control flow of the monitored application
Trivial “Bag-O’Calls” Model

- Determine the set $S$ of all system calls that an application can potentially make
  - Lose all information about relative call order
- At runtime, check for each call whether it belongs to this set
- Problem: large number of false negatives
  - Attacker can use any system call from $S$
- Problem: $|S|$ very big for large applications
Callgraph Model

- Build a control-flow graph of the application by static analysis of its source or binary code
- Result: non-deterministic finite-state automaton (NFA) over the set of system calls
  - Each vertex executes at most one system call
  - Edges are system calls or empty transitions
  - Implicit transition to special “Wrong” state for all system calls other than the ones in original code
  - All other states are accepting
- System call automaton is conservative
  - No false positives!

[Wagner and Dean]
NFA Example

- No false positives
- Monitoring is $O(|V|)$ per system call
- Problem: attacker can exploit impossible paths
  - The model has no information about stack state!
void mysetuid (uid_t uid) {
    setuid(uid);
    log("Set UID", 7);
}

void log (char *msg, int len) {
    write(fd, msg, len);
}

void myexec (char *src) {
    log("Execing", 7);
    exec("/bin/ls");
}
NFA Permits Impossible Paths

Impossible execution path is permitted by NFA!
## NFA: Modeling Tradeoffs

A good model should be...

- **Accurate**: closely models expected execution
  - Need context sensitivity!
- **Fast**: runtime verification is cheap

<table>
<thead>
<tr>
<th></th>
<th>Inaccurate</th>
<th>Accurate</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Slow</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fast</strong></td>
<td>NFA</td>
<td></td>
</tr>
</tbody>
</table>
Abstract Stack Model

- NFA is not precise, loses stack information
- Alternative: model application as a context-free language over the set of system calls
  - Build non-deterministic pushdown automaton (PDA)
  - Each symbol on the PDA stack corresponds to single stack frame in the actual call stack
  - All valid call sequences accepted by PDA; enter “Wrong” state when an impossible call is made
Another PDA Example

[Wagner and Dean]

\[
f\text{(int x) \{}
  x ? \text{getuid() : geteuid();}
  x++;
\}

\text{g() \{}
  \text{fd = open("foo", O_RDONLY);}
  f(0); close(fd); f(1);
  \text{exit(0);}
\}
\]

\[
\begin{align*}
\text{Entry}(f) & ::= \text{getuid()} \ \text{Exit}(f) \\
& \quad | \text{geteuid()} \ \text{Exit}(f) \\
\text{Exit}(f) & ::= \epsilon \\
\text{Entry}(g) & ::= \text{open()} \ v \\
  v & ::= \text{Entry}(f) \ v' \\
  v' & ::= \text{close()} \ w \\
  w & ::= \text{Entry}(f) \ w' \\
  w' & ::= \text{exit()} \ \text{Exit}(g) \\
\text{Exit}(g) & ::= \epsilon \\
\end{align*}
\]

while (true)

\text{case pop() of}

\begin{align*}
\text{Entry}(f) & \Rightarrow \text{push(Exit}(f)); \text{push(getuid())} \\
\text{Entry}(f) & \Rightarrow \text{push(Exit}(f)); \text{push(geteuid())} \\
\text{Exit}(f) & \Rightarrow \text{no-op} \\
\text{Entry}(g) & \Rightarrow \text{push}(v); \text{push(open())} \\
  v & \Rightarrow \text{push}(v'); \text{push(Entry}(f)) \\
  v' & \Rightarrow \text{push}(w); \text{push(close())} \\
  w & \Rightarrow \text{push}(w'); \text{push(Entry}(f)) \\
  w' & \Rightarrow \text{push(Exit}(g)); \text{push(exit())} \\
\text{Exit}(g) & \Rightarrow \text{no-op} \\
  a \in \Sigma & \Rightarrow \text{read and consume } a \text{ from the input} \\
\text{otherwise} & \Rightarrow \text{enter the error state, Wrong}
\end{align*}

slide 36
PDA: Modeling Tradeoffs

Non-deterministic PDA has high cost

- Forward reachability algorithm is cubic in automaton size
- Unusable for online checking

<table>
<thead>
<tr>
<th></th>
<th>Inaccurate</th>
<th>Accurate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow</td>
<td></td>
<td>PDA</td>
</tr>
<tr>
<td>Fast</td>
<td>NFA</td>
<td></td>
</tr>
</tbody>
</table>
Dyck Model

- Idea: make stack updates (i.e., function calls) explicit symbols in the automaton alphabet
  - Result: stack-deterministic PDA

- At each moment, the monitor knows where the monitored application is in its call stack
  - Only one valid stack configuration at any given time

- How does monitor learn about function calls?
  - Use binary rewriting to instrument the code to issue special “null” system calls to notify the monitor
    - Potential high cost of introducing many new system calls
  - Can’t rely on instrumentation if application is corrupted
Example of Dyck Model

Runtime monitor now “sees” these transitions
CFG Extraction Issues

◆ Function pointers
  • Every pointer could refer to any function whose address is taken

◆ Signals
  • Pre- and post-guard extra paths due to signal handlers

◆ setjmp() and longjmp()
  • At runtime, maintain list of all call stacks possible at a setjmp()
  • At longjmp() append this list to current state
## System Call Processing Complexity

$n$ is state count
$m$ is transition count

<table>
<thead>
<tr>
<th>Model</th>
<th>Time &amp; Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFA</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>PDA</td>
<td>$O(nm^2)$</td>
</tr>
<tr>
<td>Dyck</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Dyck: Runtime Overheads

Execution times in seconds

<table>
<thead>
<tr>
<th>Program</th>
<th>Unverified execution</th>
<th>Verified against Dyck</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>procmail</td>
<td>0.5</td>
<td>0.8</td>
<td>56%</td>
</tr>
<tr>
<td>gzip</td>
<td>4.4</td>
<td>4.4</td>
<td>1%</td>
</tr>
<tr>
<td>eject</td>
<td>5.1</td>
<td>5.2</td>
<td>2%</td>
</tr>
<tr>
<td>fdformat</td>
<td>112.4</td>
<td>112.4</td>
<td>0%</td>
</tr>
<tr>
<td>cat</td>
<td>18.4</td>
<td>19.9</td>
<td>8%</td>
</tr>
</tbody>
</table>

◆ Many tricks to improve performance
  • Use static analysis to eliminate unnecessary null system calls
  • Dynamic “squelching” of null calls
Persistent Interposition Attacks

Observation: malicious behavior need not involve system call anomalies

Hide malicious code inside a server
- Inject via a memory corruption attack
- Hook into a normal execution path (how?)

Malicious code communicates with its master by “piggybacking” on normal network I/O
- No anomalous system calls
- No anomalous arguments to any calls except those that read and write

[Parampalli et al.]
Virtual Machine Monitors

Software layer between hardware and OS virtualizes and manages hardware resources
History of Virtual Machines

- **IBM VM/370 – A VMM for IBM mainframe**
  - Multiple OS environments on expensive hardware
  - Desirable when few machines around

- **Popular research idea in 1960s and 1970s**
  - Entire conferences on virtual machine monitors
  - Hardware/VMM/OS designed together

- **Interest died out in the 1980s and 1990s**
  - Hardware got cheap
  - OS became more more powerful (e.g., multi-user)
VMware virtual machine is an application execution environment with its own operating system.
VM Terminology

- **VMM** (Virtual Machine Monitor) - software that creates VMs
- **Host** - system running the VMM
- **Guest** - “monitored host” running inside a VM
Isolation at Multiple Levels

◆ **Data security**
  - Each VM is managed independently
    - Different OS, disks (files, registry), MAC address (IP address)
    - Data sharing is not possible; mandatory I/O interposition

◆ **Fault isolation**
  - Crashes are contained within a VM

◆ **Performance**
  - Can guarantee performance levels for individual VMs on VMWare ESX server

◆ No assumptions required for software inside a VM (important for security!)
Observation by Host System

◆ “See without being seen” advantage
  • Very difficult within a computer, possible on host

◆ Observation points
  • Networking (through vmnet), physical memory, disk I/O and any other I/O
Virtual Machine-Based IDS

- Run the monitored host within a virtual machine (VM) sandbox on a different host
- Run the intrusion detection system (IDS) outside the VM
- Allow the IDS to pause the VM and inspect the hardware state of host
- Policy modules determine if the state is good or bad, and how to respond
VMIDDS Architecture

[Diagram showing the architecture of VMI IDS, including Policy Engine, Policy Framework, OS Interface Library, Config File, Guest OS Metadata, Command, Query, Response, Hardware State, callback or Response, Guest Apps, Guest OS, and Virtual Machine Monitor.]

slide 51
Components of VMI IDS

- **OS interface library**
  - Interprets hardware state into OS-level events
    - For example, list of all processes
  - Hard to implement and tied to a particular guest OS

- **Policy modules**
  - Determine if the OS has been compromised and what action to take
  - Many detection techniques can be implemented as policy modules and thus used to prevent intrusions
Livewire

- VMware + interposition/inspection hooks
- Six sample policy modules
  - Most related to existing host-based intrusion detection techniques
- Uses crash as OS interface library
  - Linux crash dump examination tool
  - Same goal: interpret the host’s raw memory in terms of OS-level events

[Garfinkel and Rosenblum]
Livewire Policy Modules

◆ User program integrity detector
  • Periodically hashes unchanging sections of running programs, compares to those of known good originals

◆ Signature detector
  • Periodically scans guest memory for substrings belonging to known malware
    - Finds malware in unexpected places, like filesystem cache

◆ Lie detector
  • Detects inconsistencies between hardware state and what is reported by user-level programs (ls, netstat, …)

◆ Raw socket detector
Enforcing Confinement Policies

- **Event-driven modules** run in response to a change in hardware state
- **Memory access enforcer**
  - Prevents sensitive portions of the kernel from being modified
- **NIC access enforcer**
  - Prevents the guest’s network interface card (NIC) from entering promiscuous mode or having a non-authorized MAC address
Other Advantages

♦ Resistant to “swap-out” malware
  • Sophisticated malware might detect an IDS running on the infected host and remove itself from memory when a scan is performed
  • Difficult to detect a scanner when the scanner is running outside your VM

♦ Can save entire system state for forensics
♦ Fail closed
VM Issues

◆ Guest can often tell that he is running inside a VM
  • Timing: measure time required for disk access
    - VM may try to run clock slower to prevent this attack…
    - ...but slow clock may break an application like music player

◆ Better visibility into guest ⇒ worse performance

◆ OS interface library is complex, can be fooled

◆ Policy problem
  • Even with perfect visibility into the monitored system, how to differentiate good and bad behavior?
  • Hard to express and enforce fine-grained policies
    - “Do not write any SSNs from payroll into Web database”