0x1A Great Papers in Computer Security

Vitaly Shmatikov

http://www.cs.utexas.edu/~shmat/courses/cs380s/
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
  • Discover telltale system modifications
What Should Be Detected?

- Attempted and successful break-ins
- Attacks by legitimate users
  - Illegitimate use of root privileges, unauthorized access to resources and data ...
- Malware
  - Trojan horses, rootkits, viruses, worms ...
- Denial of service attacks
Intrusion Detection Systems (IDS)

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

- **Network-based (NIDS)**
  - Often placed on a router, firewall, or network gateway
  - 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
  - Can potentially detect unknown (zero-day) 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, RET addresses of memory exploits, port numbers of applications with known vulnerabilities
  - Hard to handle mutations
    - Polymorphic viruses: each copy has a different body

- Big research challenge: fast, automatic extraction of signatures of new attacks
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
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

  ... `open, read, write, mmap`
  `read, write, mmap, mmap`
  `write, mmap, mmap, getrlimit`
  `mmap, mmap, getrlimit, open`
  ...

- 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]
Level of Monitoring

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

◆ Auditing / monitoring should be scalable
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?
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 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
R. Watson

Exploiting Concurrency Vulnerabilities in System Call Wrappers

(WOOT 2007)
Exploiting TOCTTOU Conditions

 Forced wait on disk I/O

- **Example: rename()**
  - Page out the target path of rename() to disk
  - Kernel copies in the source path, then waits for target path
  - Concurrent attack process replaces the source path
  - Postcondition checker sees the replaced source path

Voluntary thread sleeps

- **Example: TCP connect()**
  - Kernel copies in the arguments
  - Thread calling connect() waits for a TCP ACK
  - Concurrent attack process replaces the arguments
TOCTTOU via a Page Fault

[Watson]
TOCTTOU on Sysjail

Exploitable race window between two copyin() calls

- Process 1 (kernel): P1 sets original address
- Shared Memory: 0.0.0.0 to 192.168.100.1
- Process 2 (user): P2 replaces address in shared memory from second processor

[Watson]
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?

- Atomicity using system transactions

- Integrate security checks into the kernel?
D. Wagner, D. Dean

Intrusion Detection via Static Analysis

(Oakland 2001)
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
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!
NFA Example

- Monitoring is $O(|V|)$ per system call
- Problem: attacker can exploit impossible paths
  - The model has no information about stack state!
Another NFA Example

```c
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
- **Fast**: runtime verification is cheap

<table>
<thead>
<tr>
<th></th>
<th>Inaccurate</th>
<th>Accurate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fast</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 a 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
PDA Example

mysetuid

setuid

log

d push A

log

d push B

write

log

d pop B

exec

setuid

log

d pop A

log

Another PDA Example

Wagner and Dean

```c
f(int x) {
    x ? getuid() : geteuid();
    x++;
}
g() {
    fd = open("foo", O_RDONLY);
    f(0); close(fd); f(1);      // Entry(f) ::= getuid() Exit(f) | geteuid() Exit(f)
    exit(0);
}
```

```plaintext
while (true)
    case pop() of
        Entry(f) ⇒ push(Exit(f)); push(getuid());
        Entry(f) ⇒ push(Exit(f)); push(geteuid());
        Exit(f)  ⇒ no-op
        Entry(g) ⇒ push(v); push(open());
        v        ⇒ push(v'); push(Entry(f))
        v'       ⇒ push(w); push(close());
        w        ⇒ push(w'); push(Entry(f))
        w'       ⇒ push(Exit(g)); push(exit())
        Exit(g)  ⇒ no-op
        a ∈ Σ    ⇒ read and consume a from the input
        otherwise ⇒ enter the error state, Wrong
```
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 and returns) explicit symbols in the 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 the 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

<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>

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

[Giffin]
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.]