Transparent Monitoring of a Process Self in a Virtual Environment
PhD Lunchtime Seminar

Daniele Sgandurra

Università di Pisa

24 Giugno 2008
Outline

Background
  Process Self
  Attacks Against the Self
  Dynamic and Static Analysis

PsycoTrace
  Virtualization
  Static Analysis
  Run-Time Support
  Conclusions
Considered Problem

- Protecting a process from attacks that alter the intended behavior of the process program.
- We want to preserve the original semantics of the program.
- We are not interested in logic errors, such as:
  - authentication errors (weak passwords);
  - malicious behavior.
A Sense of Self for Processes

- Notion of **process self**: the program the process executes.
- Based on traces of **system calls**:
  - dynamic analysis;
  - static analysis.
- Assumption: a process can implement **security critical operations** only through system calls.
- We are not interested in Denial of Service Attacks (DOS).
System Calls

User space

Kernel space

Transparent Monitoring of a Process Self in a Virtual Environment

Daniele Sgandurra
Definitions
Definitions

- **Software bug**: an error in a computer program that prevents it from behaving as intended.
- **Vulnerability**: a bug that has security implications for the computer system executing the program.
- **Exploit**: a program that takes advantage of a vulnerability to violate the system security policy.
- **Attack**: the effective exploitation of a vulnerability through the execution of an exploit.
Buffer Overflow

- A vulnerability: the program stores more data into a buffer than the memory space reserved for it.
- The attacker may overwrite data that controls the program executions:
  - control-hijacking attack: the attacker is able to diverge the control flow of the process to execute its own instructions;
  - malicious code is executed instead of the original one.
- If the program has the rights of invoking any system call, the attacker gains control of the system.
Process Virtual Memory Layout

The problem is related to the run-time support:

1. process virtual memory layout;
2. function calling conventions.
Function Calling Conventions

- High mem addr
- Stack growth
- Low mem addr

- Arguments
- Return address
- Saved frame pointer
- Local variables

- Frame pointer
- Stack pointer

- High mem addr
- Char *str
- Return address
- Saved frame pointer
- Char buff[5]

- Overwrite return address
- Strcpy(buff, str)
Forrest (1996) described a model for the process self through sequences of system calls.

- The algorithm has two stages:
  - scan traces of normal behavior and build up a database of normal patterns, i.e. observed sequences of system calls;
  - scan new traces that might contain abnormal behavior, looking for patterns not present in the normal database.

- High number of false positives.
Dynamic Analysis

Slide a window of size $k+1$ across the trace and record which calls follow which within the sliding window.

- Suppose $k = 3$ and are given the following sequence of system calls to define normal behavior: `open, read, mmap, mmap, open, getrlimit, mmap, close`

- If we were given a new trace of calls: `open, read, mmap, open, open, getrlimit, mmap, close`

- This trace would generate 4 mismatches:
  - if the miss rate exceeds the threshold $\rightarrow$ abnormal activity.
Static Analysis

Proposal by Wagner and Dean (2001).

▸ Use static analysis to derive a specification from the program source code and to reduce false alarms.

▸ Approach:
  1. derive the specification of expected system calls by statically analyzing the source code;
  2. model the application as a transition system;
  3. monitor system call trace at run-time;
  4. check for compliance to the model.
Dynamic and Static Analysis

Static Analysis: Trivial Model

- Build the model $S$ as the set of systems calls that the application can ever make:
  $$S = \{\text{open, read, close}\}.$$
- Enforcement: if any system call not in $S$ is observed, raise an alarm.
- Properties:
  - simple, easy, efficient;
  - fail to detect many attacks.
Static Analysis: Callgraph Model

- Idea: retain some ordering information.
- Model: represent the system call trace as an NDFA:
  - derived from the control-flow graph;
  - transitions: system calls + transfer of control;
  - non-deterministic: cannot predict which branch to take.
- Enforcement: simulate the NDFA on the observed trace:
  - non-accepting state raises an alarm.
Static Analysis: Callgraph Model

Properties:

- more precise that trivial model;
- no false positives;
- impossible paths are allowed: \( w \rightarrow \text{Entry}(f) \rightarrow \text{Exit}(f) \rightarrow v'. \)
Static Analysis: Further Models

- **Abstract stack model** builds and simulates a NDPDA and compares the application call stack with a set of call stacks.
- **Dick Model** includes null system calls to eliminate impossible paths.
- **VPStatic** extracts context information about stack activity of the monitored program to define a deterministic model.
Static Analysis?

```c
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
    // guaranteed to be random.
}
```
PsycoTrace

- PsycoTrace is a virtualization-based monitoring system aimed at protecting a process self.
  - Virtualization Technology: Xen.
- Based on static analysis of the code:
  - legal traces of system calls that the process can invoke.
- It can can pair an invariant with each point of the program where a system call is issued:
  - virtual machine introspection to evaluate invariants.
Virtualization

- **Virtual Machine Monitor** (VMM): manages Virtual Machines (VMs), i.e. execution environments that emulate, at software, the behavior of the underlying physical machine.
- A standard machine can **support several VMs**, each running a distinct OS.
Grammar of System Call Sequences

- We deduce the possible **valid sequences of system calls** that the process can issue from the source code:
  - we encode them using a **context-free grammar**.
- **Meta-compilation-compilation** approach.
- At runtime, a sequence of system calls is valid only if it is a **prefix** of at least one string generated by the grammar.
Grammar of System Call Sequences: An Example

```c
int n = 5;

foo()
{
    n--;  // Decrease n
    open();  // Open file
    write();  // Write to file
    if(n) foo();  // Recurse if n is not zero
    close();  // Close file
}

int main(int argc)
{
    if(argc)
        foo();  // If there are arguments, call foo
    else
    {
        open();  // Open file
        read();  // Read from file
        close();  // Close file
    }
}
```

\[
\langle \text{main} \rangle \rightarrow \langle \text{foo} \rangle \mid \text{open read close}; \\
\langle \text{foo} \rangle \rightarrow \text{open write } (\langle \text{foo} \rangle)? \text{close};
\]
Grammar of System Call Sequences: ghttpd

<MAIN> → <COUNT_VHOSTS> <MALLOC> <READINCONFIG> <A> <OPENLOG> socket <B> setsockopt
bind <B> listen <B> <CLOSELOG> <GETPWNAM> ...

<A> → <INETD_SERVER> | ε

<B> → <PERROR> <SYSLOG> <CLOSELOG> exit | ε

... 

<INETD_SERVER> → (exit)? ...

<COUNT_VHOSTS> → ...

...
Architecture

PsycoTrace architecture is based on two virtual machines:

- the Monitored VM (Mon-VM), i.e. the VM executing the process to be monitored;
- the Introspection VM (I-VM), i.e. the VM monitoring the process through virtual machine introspection:
  - parser;
  - assertion checker;
  - introspection library.
Monitored VM

It runs the monitored process.

- **HiMod**: Linux Kernel Module to hijack system calls.
- Only a subset of system calls is traced: most critical ones.
Introspection VM

The I-VM runs the monitoring systems.

- The **Analyst** applies the consistency checks.
- It exploits the **introspection library** to access the Mon-VM.
Introspection VM

- A Bison-generated parser to check the process CFG.
- Evaluate invariants. These are generated dynamically (e.g. Daikon).
- The Introspection Library allows the Analyst to access any region of memory of the process and VCPU registers:
  - visibility: access Mon-VM’s state from a lower level;
  - robustness: introspect Mon-VM from a distinct VM.
Here Comes Trouble: Mimicry Attacks

- If an exploit contains only **allowed system call subtraces**, the intrusion will remain undetected.
- Attacks based on **parameters of system calls**.

```c
read() write() close() munmap() sigprocmask() wait4()
sigprocmask() sigaction() alarm() time() stat() read()
alarm() sigprocmask() setreuid() fstat() getpid()
time() write() time() getpid() sigaction() socketcall()
sigaction() close() flock() getpid() lseek() read()
kill() lseek() flock() sigaction() alarm() time()
stat() write() open() fstat() mmap() read() open()
fstat() munmap() read() close() munmap() brk() fcntl()
setregid() open() fcntl() chroot() chdir() setreuid()
stat() stat() stat() lstat() open() fcntl() fstat()
lseek() getdents() fcntl() fstat() lseek() getdents()
close() write() time() open() fstat() mmap() read()
close() munmap() brk() fcntl() setregid() open() fcntl()
chroot() chdir() setreuid() lstat() lstat() lstat()
lstat() open() fcntl() brk() fstat() lseek() getdents()
lseek() getdents() time() stat() write() time() open()
getpid() sigaction() socketcall() sigaction() umask()
sigaction() alarm() time() stat() read() alarm()
getrlimit() pipe() for() fcntl() fstat() mmap() lseek()
close() brk() time() getpid() sigaction() socketcall()
sigaction() chdir() sigaction() sigaction() write()
munmap() munmap() munmap() exit()
```
Invariant Examples

Invariants help in limiting mimicry attacks.

- **Syscalls parameters assertions.** They exploit data-flow properties: relations among values of syscall parameters:
  - check that the file descriptor of a `read` system call is the same returned by a previous `open`.

- **Assertions on files.** They help avoiding symlink attacks:
  - check that the path of a requested file is under a known dir.
Invariant Examples

- **Buffer length assertions:**
  - e.g.: check that the length of the string passed to a vulnerable function is not larger than the local buffer.

- **Invariants on conditional statements.** They reduce the non-determinism of the grammar:
  - `syscall_1` if `(uid==0)` `syscall_2` else `syscall_3`. (If the trace is `syscall_1`, `syscall_2` check that `uid==0`).
**Trusted Computing Base (TCB)**

The set of all computer components that are critical to security.

- **Xen** small size: \( \sim 100K \) vs \( \sim 100M \) LoC of a modern OS.
- **Xen** narrow interface: \( \sim 30 \) hypercalls vs \( \sim 300 \) syscalls.
- **I-VM**: small kernel, no users, no open ports.
- **Mon-VM**: HiMod needs to be protected.
Process Monitoring

Monitored VM (MON-VM)

- traced process
- Kernel
- HiMod syscalls hijack module
- syscall Addresses: 0x8...
- original syscall(s)
- VCPU-C

Introspection VM (I-VM)

- ANALYST
- ASSERTION CHECKER
- Invariants
- CFG(P)
- INTROSP. LIBRARY
- PARSER + LEX. ANAL.

Xen Virtual Machine Monitor

Transparent Monitoring of a Process Self in a Virtual Environment

Daniele Sgandurra
Process Monitoring

Monitored VM (MON-VM)

- traced process
- syscall()
- HiMod
  - syscall hijack module
  - 0xB

Kernel

- Variables
  - Addresses
- VCPU-C

Introspection VM (I-VM)

- ANALYST
  - ASSERTION CHECKER
  - invariants
  - CFG(P)

- Intros. Library
  - PARSER + LEX. ANAL.

Kernel

Event channel (Synchronization)

Xen Virtual Machine Monitor

Transparent Monitoring of a Process Self in a Virtual Environment

Daniele Sgandurra
Process Monitoring

Monitored VM (MON-VM)

- traced process
- syscall();
- Kernel
- HiMod
  - syscall hijack module
  - original syscall(s)
- VCPU-C

Introspection VM (I-VM)

- Memory Introspection
- Map
- Tracing/VCPU-Introspection

Event channel (Synchronization)

Xen Virtual Machine Monitor

Daniele Sgandurra

Transparent Monitoring of a Process Self in a Virtual Environment
Summing-up

- **Virtualization-based** monitoring system:
  - robustness;
  - transparency.
- **Notion of process self:**
  - system call context-free grammar;
  - invariants.
- Acceptable overhead (<15%).
Future Works

- Static deduction of invariants: e.g. abstract interpretation.
- Complete transparency on Mon-VM:
  - IDT-hijacking;
  - NX-BIT;
  - VMexit on int instructions.
- Monitoring child processes.
- Signal handling: stack unwinding.
Acknowledgments

This work has been done in collaboration with:

Fabrizio Baiardi
Dario Maggiari
Francesco Tamberi
Questions?