PsycoTrace: Virtual and Transparent Monitoring of a Process Self

Fabrizio Baiardi, Dario Maggiari
Polo G. Marconi, La Spezia
Università di Pisa, Italy
{*baiardi, maggiari*}*@di.unipi.it*

Daniele Sgandurra, Francesco Tamberi
Dipartimento di Informatica
Università di Pisa, Italy
{*daniele, tamberi*}*@di.unipi.it*

Abstract

PsycoTrace is a set of tools to protect a process P from attacks that alter P self as specified by its source code. P self is specified in terms of legal traces of system calls and of assertions on P status paired with each call. In turn, legal traces are specified through a context-free grammar returned by a static analysis of P program that may also compute assertions. At run-time, each time P invokes a system call, PsycoTrace checks that the trace is coherent with the grammar and assertions are satisfied. To increase overall robustness, PsycoTrace's run-time tool relies on two virtual machines that run, respectively, P and the monitoring system. This strongly separates the monitored machine that runs P from the monitoring one. The current implementation is fully transparent to P but not to the OS because a kernel module in the monitored machine intercepts system calls.

We describe PsycoTrace overall architecture and focus on the run-time and introspection tools that enable the monitoring machine to check that a trace is legal and to transparently access the memory of the other machine to evaluate assertions. Lastly, a preliminary evaluation of the run-time overhead is discussed.

1. Introduction

Several attacks, such as buffer overflows, modify a process intended behavior [16], i.e. the process self. To protect a process P from these attacks, PsycoTrace integrates static and dynamic tools that, respectively, compute a description of P expected behavior and compare the specification against the actual behavior of P. Since PsycoTrace assumes that security critical operations are implemented through system calls only, it specifies P behavior through both $CFG(P)$, i.e. a context-free grammar that defines the traces of system calls that P may issue, and a set of assertions on the values of the call parameters to simplify the detection of *mimicry attacks* [7].

To check P's run-time behavior in a robust and unobtrusive manner, PsycoTrace fully exploits virtualization by concurrently executing two virtual machines (VMs), the *monitored VM* (Mon-VM) and the *introspection VM* (I-VM). The Mon-VM executes the system that runs P and it intercepts system calls and alerts the I-VM. In this way, any time P executes an invocation i of a system call, the I-VM checks that the actual trace is a prefix of at least one generated by $CFG(P)$ and it evaluates the assertion $A(P, i)$ paired with i. $A(P, i)$ is a function of P variables that holds any time the control reaches i and it may either be deduced through a static analysis of P source code or generated by observing properties that hold during the execution of P. The I-VM retrieves variable values through the PsycoTrace *Introspection Library* that exploits virtual machine introspection [11] to directly and transparently access the physical memory and the processor's registers of the Mon-VM. Besides implementing a transparent monitoring, the adoption of two virtual machines strongly increases the complexity of an attack against PsycoTrace, because an attacker should at first subvert the Mon-VM in an undetected way and then attack the I-VM through a vulnerability of the virtual machine monitor.

This paper is focused on PsycoTrace's run-time tools that control the traces of P and on the Introspection Library to evaluate assertions. Section 2 recalls the main concepts underlying PsycoTrace and describes its run-time architecture and the current implementation. Section 3 discusses a preliminary evaluation of effectiveness and of the execution overhead. Section 4 discusses the current limitations of PsycoTrace and outlines future developments. Section 5 reviews some related works. Finally, Sect. 6 draws a set of conclusions.

2. PsycoTrace Architecture

After briefly discussing the main concepts underlying PsycoTrace, namely virtualization, introspection and context-free grammar, this section describes the current implementation of the run-time tools.

2.1. Background

Virtualization and Introspection. Besides robustness, a main goal of PsycoTrace is *fully transparency*, i.e. the monitoring of an application should not require any updates to the applications or the underlying operating system (OS).

Transparency guarantees that any application can be monitored and it also increases robustness because the monitoring cannot be avoided even by attacking the underlying OS. The basic technologies to achieve transparency and robustness are virtualization and introspection. Virtualization introduces the virtual machine monitor (VMM), which is a thin software layer that runs on top of a physical machine and that creates, manages and monitors virtual machines (VMs). Each VM is an execution environments that emulate, at software, the behavior of the underlying physical machine. Virtual machine introspection (VMI) extends the VMM so that a privileged VM can analyze any data structure of any other VM in full detail. VMI does not require any hardware support and it offers full system visibility because the VMM can access every VM component, such as the main memory or the processor's registers. According to the previous considerations, PsycoTrace's run-time architecture is built around two VMs, the monitored VM (Mon-VM), which executes P, and the introspection VM (I-VM), which monitors P through VMI. The Mon-VM transfers control to the I-VM each time P reaches an invocation i of a system call. The I-VM can check that the trace satisfies $CFG(P)$ and it evaluates $A(P, i)$ through VMI. Assertions can involve each component of the Mon-VM, for example any program variable. A further benefit of this solution is that the tools that implement the checks run at the user-level on the I-VM and this strongly simplifies their implementation with respect to a kernel-level solution.

The current version of PsycoTrace is not fully transparent because a module in the Mon-VM kernel traps the system calls and notifies them to the I-VM. The module integrity is periodically checked by the I-VM through VMI so that we do not need to assume the integrity of the Mon-VM kernel. Xen [6] is the virtualization technology that has been chosen, mainly because of its high performance and complete integration with the Linux kernel.

Legal System Call Traces. To achieve an optimal compromise between accuracy and efficiency, a context-free grammar $CFG(P)$ describes the traces generated by any execution of P. At run-time, the sequence of system calls that P generates up to a given instant, i.e. P current trace, is legal if and only if it is *coherent with* $CFG(P)$, i.e. it is a prefix of at least one string of the language $L(P)$ generated by $CFG(P)$.

Formally, $CFG(P)$ is a tuple $< T, F, S, R >$, where:

- T is the set of terminal symbols, one for each call P invokes;
- F is the set of non-terminal symbols, one for each function defined in the source code;
- S is the starting symbol, which corresponds to the `main` function;
- R is the set of production rules.

A PsycoTrace tool automatically generates $CFG(P)$ by applying *GGA* (Grammar Generating Algorithm) to linearly scan each function defined in P source code. For each definition of a function `fun`, GGA inserts a new symbol `FUN` into F and a new rule $\texttt{R}_{new}$ into R with `FUN` as its left-hand-side. GGA generates the right-hand-side by sequentially scanning the definition of `fun` and by applying the following rules to each instruction block `B` in `fun`:

- if `B` does not include conditional or loop statements, GGA appends to the right-hand-side of $\texttt{R}_{new}$ the calls and functions invoked inside `B` in the same order they appear in `B`;
- if `B` is a `if(cond)` $\{\texttt{block}_1\}$ `else` $\{\texttt{block}_2\}$ statement, GGA generates a new rule:

  `<A>` $\rightarrow$ `<B>` | `<C>`

  where `B` and `C` are the left-hand-sides of the productions generated by the recursive application of GGA to $\texttt{block}_1$ and $\texttt{block}_2$ respectively. Furthermore, it appends the symbol `A` to the right-hand of $\texttt{R}_{new}$; if the `else` branch is missing, then `<C>` is equal to ϵ, i.e. the empty string;
- if `B` is a `if(cond)` $\{\texttt{block}_1\}$ `else if(cond)` $\{\texttt{block}_2\}$ `...` `else if(cond)` $\{\texttt{block}_n\}$ statement or, equivalently, a `switch` statement, GGA generates a new rule:

  `<A>` $\rightarrow$ `<B`$_1$`>` | `<B`$_2$`>` | ... | `<B`$_n$`>`

  where $\texttt{B}_1$, $\texttt{B}_2$ ... $\texttt{B}_n$ are the left-hand-sides of the productions returned by the recursive application of GGA to $\texttt{block}_1$, $\texttt{block}_2$, ... , $\texttt{block}_n$ respectively. Furthermore, it appends the symbol `A` to the right-hand of $\texttt{R}_{new}$;
- finally, if `B` is a `while(cond)` $\{\texttt{block}_1\}$ or, equivalently, a `for` cycle, GGA appends $\texttt{(A)}^*$ to the right-hand side of `R`, and GGA generates a new non-terminal symbol `A` that is the left-hand-side of the productions generated by the recursive application of GGA to $\texttt{block}_1$.

Anytime P issues a system call, PsycoTrace parses the current trace to check if it is coherent with $CFG(P)$. To reduce parsing complexity due to non-determinism that arises because of conditional statements, we represent $CFG(P)$ as a Non-Deterministic Push-Down Automaton (NPDA) and apply Bison [5] to generate a parser that implements a bottom-up syntactic analysis.

Assertions. By pairing an assertion with each call, PsycoTrace can detect non-control-data attacks [3], [4]. Assertions can either be deduced by monitoring the program execution [8] or computed by applying a static tool to P source code [1]. Any non-empty assertion is the conjunction of assertions in the following classes:

- Parameters assertions. They express data-flow 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 filename corresponding to the file 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 *impossible paths* [16] by relating a system call and the expression in the guard of a conditional statement. As an example, in `if(uid == 0) then` $\texttt{syscall}_1$ `else` $\texttt{syscall}_2$, we pair the assertion `uid == 0` with $\texttt{syscall}_1$, to prevent a normal user from executing the same call of the root user. They may also check that the current return address matches the call issued by P.

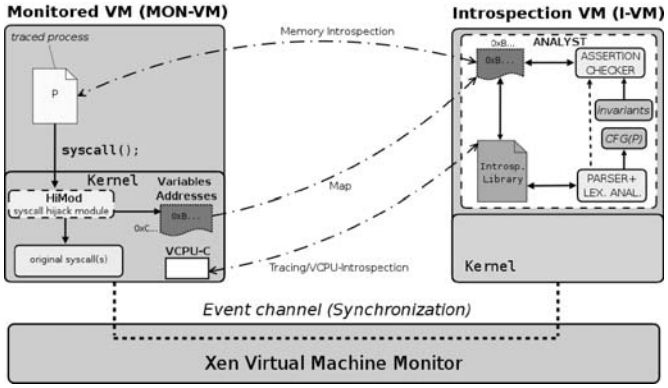

Figure 1. PsycoTrace Architecture.

2.2. Current Prototype

The implementation of PsycoTrace's run-time includes a set of tools to trace P system calls and to evaluate assertions. The source code of the current prototype is available at http://www.di.unipi.it/~daniele/projects/psycotrace/psycotrace.php.

System Call Tracing and Run-Time Checks. The current run-time architecture exploits two tools: the *HiMod* and the *Analyst* (see Fig. 1). The HiMod runs in the kernel of the Mon-VM to hijack system calls issued by P. Each time P invokes a system call, the HiMod traps the call and, before servicing it, it informs the Analyst. The Analyst in the I-VM runs a Bison-generated parser and an Assertion Checker that, respectively, check P trace and evaluate any assertion paired with the current call. The Assertion Checker accesses the Mon-VM memory and its VCPU register through the Introspection Library. Since the interactions between the Analyst and the HiMod have to be synchronous because the state of P cannot be updated during assertion evaluation, the Analyst freezes the execution of the Mon-VM and resumes it only if the security checks are satisfied.

The HiMod and the Analyst are synchronized through an *event channel*, a Xen data structure that emulates the interrupt mechanism. The Analyst allocates a new event channel and waits for notifications from the HiMod. Each notification corresponds to a call that P wants to issue. Since just a few calls in the Linux kernel can be exploited to attack a process [2], PsycoTrace monitors only critical system calls.

When the HiMod informs the Analyst that P has invoked a system call, the Analyst suspends the Mon-VM and retrieves the system call number from the `EAX` register in the Mon-VM through VCPU-Introspection. Then, the Analyst passes this number to the lexical analyzer that, in turn, transmits the system call token to the parser as an input. If the current call does not belong to the terminal alphabet symbols, the parser returns an error. Otherwise, it resumes the parsing from the point reached when analyzing the previous call. In this way, PsycoTrace implements a *stream-oriented parsing* that strongly simplifies the parsing with respect to the case where each invocation starts a new derivation from the $CFG(P)$ starting symbol.

Introspection Library. If the parsing is successful, the I-VM invokes the Assertion Checker module to evaluate the assertion paired with the call. The Assertion Checker accesses P memory through an *Introspection Library* that supports x86/32 architectures, either with regular paging or PAE. This library enables the Assertion Checker to traverse the page tables of P to translate a virtual address of any variable into a machine address, i.e. a physical address in Xen terminology. In this way, the Assertion Checker can map any page of P into its address space and access the variables of P. The library implements two introspection functions, *Memory Introspection* and *VCPU-Context Introspection*. The former accesses the memory of a VM both at the user and at the kernel level while the latter retrieves the state of a virtual processor of a VM.

To implement user-space memory introspection, the *Introspection Library* needs to access any physical memory location allocated to the Mon-VM that corresponds to a virtual address of P. To translate this virtual address, the Introspection Library directly accesses the page tables (PTs) of P and then follows the pointer to walk the paging levels to discover the mapping between a virtual and a physical address. In the case of para-virtualization, the addresses in all the page levels and in the registers of a virtual context of a VM are machine addresses. This implies that the Introspection Library has to map three pages to translate a virtual address into a machine address. Pages are mapped through the `xc_map_foreign_range()` function.

Taking into account Xen memory management, the library translates a pseudo-physical address PPA paired with a kernel symbol in four steps: (i) translate PPA into a

machine address MA using the physical-to-machine (P2M) table. MA does not reference the kernel symbol because it is relocated, i.e. Xen adds a further level of indirection to the kernel pseudo-physical addresses; (ii) ask Xen to map the page at the base address of MA and retrieve from the resulting offset the relocated pseudo-physical address PPA_2 of the kernel symbol; (iii) access the P2M table to translate PPA_2 into the corresponding machine address MA_2; (iv) ask Xen to map the page at the base address of MA_2 into the address space of the Assertion Checker process. This page stores the kernel data structure pointed to by the kernel symbol. When the Introspection Library has mapped the page that stores the pointer to the kernel page directory and referenced to by the `swapper_pg_dir` symbol, it can translate pseudo-physical addresses by accessing the kernel PTs in the same way of a process virtual address. In this case, the Introspection Library sequentially maps three pages instead of four.

Finally, the VCPU-Context introspection allows the Analyst to monitor, and modify, the content of any Mon-VM's register that Xen saves in a *Virtual CPU-Context* paired with each VM. The Analyst exploits VCPU-Context introspection to retrieve the current system call number by reading the `EAX` register and the value of any system call parameter stored or pointed to by a VCPU register.

3. Tests

This section shows a preliminary evaluation of PsycoTrace effectiveness and of the current prototype's run-time overhead. The HiMod consist of 2.5K lines of C code generated by a 450 lines Perl script. To generate the HiMod, the script parses the definition of the Linux system calls. The Analyst, including the Introspection Library, consists of about 3K lines of C code. The implementation of the extended C grammar that includes the semantic actions to generate $CFG(P)$ is about 1K lines of C++ code. Finally, a typical parser generated by Bison requires about 3K lines of C code.

To evaluate PsycoTrace effectiveness, we considered `ghttpd`, a HTTP server, that contains a known vulnerability [15]. Table 1 shows a subset of the grammar generated by the PsycoTrace static tools. Upper-case letters denote non-terminal symbols that are paired either with functions declared in the source code of `ghttpd`, such as `inetd_server`, or with `libc` functions, such as `malloc`, that may issue system calls. In the first case, system calls were directly extracted from the source code, whereas a binary analysis deduced invocations of `libc` functions. The execution of the `ghttpd` exploit results in a remote shell with the privileges of the remote server process. After a successful injection, the trace of P included a `setuid` call followed by a `brk`. The parser immediately signals these calls because they are forbidden by $CFG(P)$.

```
<MAIN> → <COUNT_VHOSTS> <MALLOC>
 <READINCONFIG> <A> <OPENLOG> socket <B> ...
<A> → <INETD_SERVER> | ϵ
<B> → <PERROR> <SYSLOG> <CLOSELOG> exit | ϵ
<INETD_SERVER> → (exit)? ...
<COUNT_VHOSTS> → ...
...
```

Table 1. Fragment of `ghttpd` Grammar

All the tests to evaluate PsycoTrace overhead refer to a system including a Pentium Centrino Duo T2250 1.7GHz. 128MB of physical RAM were allocated to the Mon-VM running a Linux Debian distribution, and 874 MB RAM to the I-VM. The Xen version was 3.1.0, while the Mon-VM Linux kernel version was 2.6.18-xen. We considered two executions of `bunzip2` on the Mon-VM to uncompress the Linux kernel, with and without trace parsing. The execution time increases from 19.896 sec to 24.268 sec. The rather large overhead, 21.97%, is due to the high rate of call invocations of `bunzip`. To optimize introspection of variables mapped into the same page, PsycoTrace exploits a software TLB that records the pairing among virtual and machine addresses. Before translating a virtual address va, the Introspection Library searches the TLB for the virtual address of the page including va. If the address is found, then the page is already mapped in the Assertion Checker memory. The average time to map a page of P into the Assertion Checker address space is about 50μsec. The software TLB reduces the access time to 20μsec anytime the same page stores several variables. In this way, the Assertion Checker can access the variables without mapping further pages of the virtual address space of P. Finally, taking into account the rate of system call invocations of `ghttpd`, the average execution time overhead is at most 10% with respect to the execution of `ghttpd` on the Mon-VM when no check is applied.

4. Future Developments

Since PsycoTrace is focused on malicious code injection, it does not cover security problems due to flawed application logic. A further problem is due to non-standard control-flows that a static analysis cannot handle very easily such as function pointers, dynamic linking and OS signals. For the time being, we neglect function pointers, so we may miss some system calls. Dynamic linking enables a process to load at run-time any library and it can be handled through a run-time update of $CFG(P)$ each time a new library is loaded. The handling of OS signals is rather complex for a static tool because signal handler could issue any call and signals may be delivered at any moment. While PsycoTrace prototype neglects signals, we are developing a solution

where the I-VM applies introspection to access P stack and check if a signal handler has invoked the call. Hence, the Analyst can check the handler's grammar or stop the parsing until the handler exits. The final prototype will achieve fully transparency through a trapping mechanism based upon the mapping of the pages storing the system call handler or by exploiting the `int` trapping mechanism supported by recent processors.

5. Related Work

Wagner and Dean [16] define a static analysis of the application source code that returns a specification of the expected application behavior. They discuss the *callgraph model*, built by analyzing the control-flow of the program, which is then extended to the *abstract stack model* to take into account impossible paths. Finally, they introduce the *digraph model* to simplify the framework implementation. Instead, PsycoTrace derives a context-free grammar directly from the source code rather than from the control-flow and the abstract stack and exploits virtualization to take advantages of its isolation properties and apply introspection. [13] discusses a model to detect malicious system calls through a static analysis of the binary program that builds a model representing any remote call streams the process may generate. The *Dick Model* [12] includes a stack to record function call return locations, by using precalls and postcalls, and null system calls to eliminate impossible paths and to simulate stack operations. *VtPath* [10] is an anomaly detection method that utilizes return addresses and that generates an abstract execution path between two program points and decides whether this path is valid according to what has been learned from the normal program executions. The *VPStatic* model [9], a variant of the VtPath model, extracts context information about stack activity from the monitored program to define a deterministic model. Finally, *Paid* [14] is a compiler-based intrusion detection system that derives a deterministic FSA model that captures system call sites, their ordering and partial control-flow information from the application source code. Moreover, it computes a set of constraints on the arguments of sensitive system calls.

6. Conclusion

PsycoTrace aims to detect when a process is attacked and exploited to execute malicious code. To this purpose, it monitors the process actual behavior to check if it differs from the specification of a context-free grammar that defines the legal traces of system calls the process can issue. At run-time, each time the process invokes a system call, PsycoTrace verifies that the current trace is coherent with the grammar. Moreover, to improve PsycoTrace detection capabilities, an invariant may be paired with each system call. The integration of a context-free grammar and invariants enables PsycoTrace to prevent and detect most attacks against the process self, i.e. attacks that modify its intended behavior. To evaluate invariants in a transparent way, the Introspection Library enables the monitoring virtual machine to access any region of memory of the monitored machine to retrieve variable values.

The current status of the work shows promising results in the accuracy of attack detection and an acceptable overhead.

References

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.

[2] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Operating system enhancements to prevent the misuse of system calls. In *Proc. of the 7th ACM conference on Computer and communications security*, pages 174–183, New York, NY, USA, 2000. ACM.

[3] S. Bhatkar and A. Chaturvedi. Sekar., R. Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments. *Proc. of the IEEE Symposium on Security and Privacy*, 2006.

[4] S. Chen, J. Xu, E. C. Sezer, P.Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proc. of the 14th USENIX Security Symposium*, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[5] C. Donnelly and R. Stallman. *The Bison manual: using the YACC-compatible parser generator*. Boston, MA: GNU, 2006.

[6] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proc. of the ACM Symposium on Operating Systems Principles*, October 2003.

[7] D.Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proc. of the 9th ACM conference on Computer and communications security*, pages 255–264, New York, NY, USA, 2002. ACM.

[8] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[9] H. H. Feng, J. T. Giffin, Y. Huang, S.Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2004.

[10] H. H. Feng, O. M. Kolesnikov, P.Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proc. of the 2003 IEEE Symposium on Security and Privacy*, page 62, Washington, DC, USA, 2003. IEEE Computer Society.

[11] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[12] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *11th Annual Network and Distributed Systems Security Symposium (NDSS), San Diego, California*, February 2004.

[13] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proc. of the 11th USENIX Security Symposium*, pages 61–79, Berkeley, CA, USA, 2002. USENIX Association.

[14] L. C. Lam, W. Li, and T. cker Chiueh. Accurate and automated system call policy-based intrusion prevention. In *Proc. of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 413–424, Washington, DC, USA, 2006. IEEE Computer Society.

[15] SecurityFocus. ghttpd Log() Function Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5960.

[16] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proc. of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.