The ORCHIDS Intrusion Detection Tool[⋆]

Julien Olivain and Jean Goubault-Larrecq

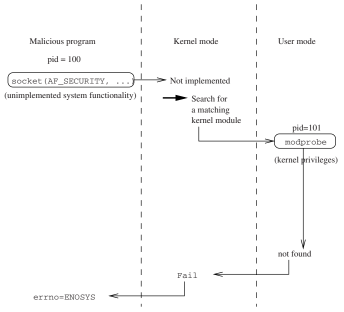
LSV/CNRS UMR 8643 & INRIA Futurs projet SECSI & ENS Cachan,
61 avenue du président-Wilson, F-94235 Cachan Cedex,
Phone: +33-1 47 40 75 50, Fax: +33-1 47 40 24 64
`olivain@lsv.ens-cachan.fr`

Abstract. ORCHIDS is an intrusion detection tool based on techniques for fast, on-line model-checking. Temporal formulae are taken from a temporal logic tailored to the description of intrusion signatures. They are checked against merged network and system event flows, which together form a linear Kripke structure.

Introduction: Misuse Detection as Model-Checking. ORCHIDS is a new intrusion detection tool, capable of analyzing and correlating events over time, in real time. Its purpose is to detect, report, and take countermeasures against intruders. The core of the engine is originally based on the language and algorithm in the second part of the paper by Muriel Roger and Jean Goubault-Larrecq [6]. Since then, the algorithm evolved: new features (committed choices, synchronization variables), as well as extra abstract interpretation-based optimizations, and the correction of a slight bug in op.cit., appear in the unpublished report [1]. Additional features (cuts, the "without" operator) were described in the unpublished deliverable [2]. Finally, contrarily to the prototype mentioned in [6], ORCHIDS scales up to real-world, complex intrusion detection.

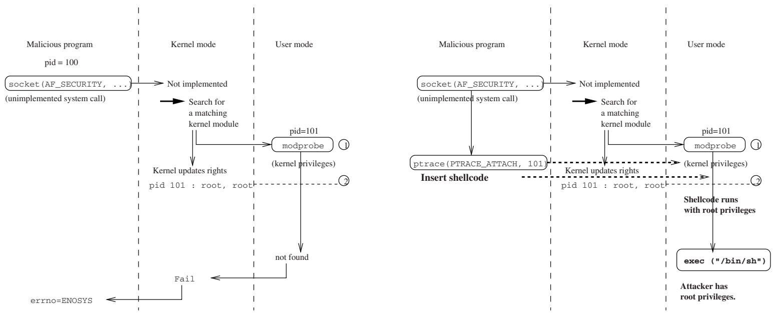
The starting point of the ORCHIDS endeavor is that intrusion detection, and specifically *misuse detection*, whereby bad behavior (so-called *attacks*) is specified in some language and alerts are notified when bad behavior is detected, is essentially a *model-checking* task. The Kripke model to be analyzed is an *event flow* (collected from various logs, and other system or network sources), and complex attack *signatures* are described in an application-specific temporal logic.

Let us give an example of a modern attack [5]. Far from being a gedankenexperiment, this really works in practice and has already been used to penetrate some systems. We also insist that, as systems get more and more secure, we are faced with more and more complex attacks, and [5] is just one representative. The schema on the right displays what a modular kernel (e.g., Linux) does when a user program (here with pid 100) calls an unimplemented functionality.



⋆ Partially supported by the RNTL Project DICO, the ACI jeunes chercheurs "Sécurité informatique, protocoles crypto. et détection d'intrusions" and the ACI cryptologie "Psi-Robuste".

The kernel will search for a kernel module that implements this functionality, calling the `modprobe` utility to search and install the desired module. If `modprobe` does not find any matching module, an error code is reported to the user program.



While this is how this is meant to work, some versions of Linux suffer from a race condition (above, left): while `modprobe` has all kernel privileges, the kernel updates the owner tables to make `modprobe` root-owned while `modprobe` has already started running. So there is a small amount of time where the malicious program has complete control over the kernel process `modprobe`: between timepoints ① and ②. The malicious program takes this opportunity to attach the `modprobe` process through the standard Unix debugging API function `ptrace`, inserting a *shellcode* (malicious code) inside `modprobe`'s code. When `modprobe` resumes execution, it will execute any code of the intruder's choosing, with full root privileges (above, right).

Challenges in On-line, Real-Time Model-Checking. Intrusion detection requires specific logics to describe attack signatures, and specific model-checking algorithms.

Compared to standard misuse detection tools, a temporal logic allows one to describe behavior involving several events over time: standard misuse detection tools (e.g., anti-virus software or simple network intrusion detection systems) match a library of patterns against single events, and emit an alert once single so-called *dangerous* events occur. More and more attacks nowadays involving complex, correlated sequences of events, which are usually individually benign. In the `ptrace` attack, *no* individual event (calling an unimplemented system call, or `ptrace`, etc.) is dangerous per se.

The signature language of ORCHIDS extends [6–Section 4]. Among other things, it allows one to write temporal formulas of the typical form $F_1 \wedge \Diamond(F_2 \wedge \Diamond(F_3 \ldots) \vee F'_2 \wedge \Diamond(F'_3 \ldots))$, where $\Diamond$ is the strict "there exists in the future" operator. In general, more complex formulae can be written, using operators resembling Wolper's ETL [7]—except going through a transition denotes either no time-passing at all (ϵ-transitions), or $\Diamond$ (not $\bigcirc$ as in ETL). Such formulae are described internally as automata; we just give a signature for the `ptrace` exploit as an illustration. (Some other attacks such as the `do_brk` exploit [3] require committed choices, or other features of ORCHIDS not described in [6]. To save space, we don't recount them here.) A formula matching the `ptrace` exploit is the following automaton, described in slightly idealized form:

$$\begin{tikzcd}
\mathtt{Attach}(X,Y,Z) \arrow[r, "\Diamond"] & \mathtt{Exec}(Y) \arrow[r, "\Diamond"] & \mathtt{Syscall}(X,Y) \arrow[r, "\Diamond"] \arrow[d, "\Diamond"] & \mathtt{Getregs}(X,Y) \arrow[dl, "\Diamond"'] \\
& & \mathtt{Poketext}(X,Y) \arrow[r, "\Diamond"] & \mathtt{Detach}(X,Y)
\end{tikzcd} \quad (1)$$

where X, Y, Z are existentially quantified first-order variables meant to match the attacker's pid, the target's pid (i.e., modprobe's pid), and the attacker's effective uid respectively; where $\texttt{Attach}(X, Y, Z)$ abbreviates a formula (not shown) matching any single event displaying a call to ptrace by process X owned by Z, on process Y, with the ATTACH command, $\texttt{Exec}(Y)$ matches single events where /sbin/modprobe is execed with pid Y, and the remaining formulas match single events where process X issues a call to ptrace on target Y, with respective commands SYSCALL, GETREGS, POKETEXT (used to insert the shellcode), and DETACH.

Compared to other more standard uses of model-checking, the logic of ORCHIDS is constrained to only specify *eventuality* properties. This is because the model-checker needs to to work *on-line*, that is, by always working on some finite (and expanding over time) prefix of an infinite sequence of events. Compared to standard model-checking algorithms, e.g., based on Büchi automata for LTL, the model-checker is not allowed to make multiple passes over the sequence of events (e.g., we cannot rebuild a product automaton each time a new event is added); in general, intrusion detection tasks are submitted to very stringent efficiency requirements, both in time and in space.

Second, the logic of ORCHIDS includes some first-order features. As witnessed by the use of variables X, Y, Z in (1), this logic can be seen as an existential fragment of a first-order temporal logic.

Finally, such a model-checker cannot just report the *existence* of matches, but must enumerate all matches among a given representative subset, with the corresponding values of the existential variables, build an alert for each match and possibly trigger countermeasures. This is the *raison d'être* behind the $\texttt{Getregs}(X, Y)$ formula in (1); if we only wanted a yes/no answer, this would just be redundant, and could be erased from the automaton; here, this is used to be able to report whether the attacker issued at least one call to ptrace(PTRACE_GETREGS) or not during the attack.

The model-checking task for the logic of ORCHIDS is NP-hard (it includes that of [6–Section 4]), but can be done using an efficient, on-line and real-time algorithm [2, 1]. Moreover, this algorithm is *optimal* in the following sense: for every attack signature (formula F), if at least one attack (sequence of possibly non-contiguous events) is started at event e_0 that matches F, then exactly one attack is reported amongst these, the one with the so-called *shortest run*. The latter is usually the most meaningful attack among all those that match. The notion of shortest run was refined in ORCHIDS, and now appears as a particular case of *cuts* [2]; this gives more control as to which unique attack we wish to isolate amongst those that match.

Related Work. There are many other formalisms attempting to detect complex intrusion detection scenarios, using means as diverse as Petri nets, parsing schemata, continuous data streams, etc. Perhaps one of the most relevant is run-time monitoring (or cousins: F. Schneider's security automata and variants, and security code weaving), where the event flow is synchronized at run-time with a *monitor* automaton describing paths to bad states. The ORCHIDS approach is certainly close to the latter (although arrows in e.g., (1) are more complex than simple automaton transitions); shortest runs and cuts, which introduce priorities between paths in the monitor, and the fact that only one optimal path among equivalent paths is reported, is a useful refinement.

Implementation. The ORCHIDS engine is implemented in C. At the core of ORCHIDS lies a fast virtual machine for a massively-forking virtual parallel machine, and a byte-code compiler from formulae (such as (1)) to this virtual machine. ORCHIDS uses a hierarchy of input modules to subscribe to, and to parse incoming events, classified by input source and/or event format. A main event dispatcher reads from polled and real-time I/O, reads sequences of events in `syslog` format, `snare`, `sunbsm`, `apache` and other various formats, coming from log files or directly through dedicated network connections, and feeds the relevant events to the core engine. ORCHIDS is able to do both system-level and network-based intrusion detection, simultaneously.

Here are a few figures of ORCHIDS on an instance of the `ptrace` attack:

Time :	Real time :	1267s
	CPU Time :	370.810s
	CPU usage :	29.27%
Resources :	Memory (peak) :	2.348 MB
	Signalisation network load :	1.5 GB
Analyzer :	Loading and rule compilation :	< 5 ms
	Processed events :	4 058 732

To stress the detection engine, the attack was hidden in the middle of a huge amount of normal `ptrace` debugging events, generated by tracing the compilation of the whole *GCC C Compiler* (with the command line `tar xzvf gcc-3.3.2.tar.gz ; cd gcc-3.3.2 ; ./configure ; cd gcc ; strace -F -f make`).

Conclusion. The `ptrace` attack above is one of the typical attacks that ORCHIDS can detect. Experiments are going on at LSV to test ORCHIDS on actual network traffic and system event flows.

From the point of view of security, a good news is that, contrarily to most misuse intrusion detection systems, ORCHIDS is able to detect intrusions that were not previously known (contrarily to popular belief on misuse IDSs). E.g., the signature we use for the `do_brk` attack [3], which tests whether some process managed to gain root privilege without calling any of the adequate system calls, detected the recent (Jan. 2005) Linux `uselib` attack.

For more information, see the Web page [4].

References

1. J. Goubault-Larrecq. Un algorithme pour l'analyse de logs. Research Report LSV-02-18, Lab. Specification and Verification, ENS de Cachan, Cachan, France, Nov. 2002. 33 pages.
2. J. Goubault-Larrecq, J.-P. Pouzol, S. Demri, L. Mé, and P. Carle. Langages de détection d'attaques par signatures. Sous-projet 3, livrable 1 du projet RNTL DICO. Version 1, June 2002. 30 pages.
3. A. Morton and P. Starzetz. Linux kernel `do_brk` function boundary condition vulnerability. `http://www.securityfocus.com/bid/9138`, Dec. 2003. References CAN-2003-0961 (CVE), BugTraq Id 9138.
4. J. Olivain. ORCHIDS—real-time event analysis and temporal correlation for intrusion detection in information systems. `http://www.lsv.ens-cachan.fr/orchids/`, 2004.

5. W. Purczyński. Linux kernel privileged process hijacking vulnerability. `http://www.securityfocus.com/bid/7112`, Mar. 2003. BugTraq Id 7112.
6. M. Roger and J. Goubault-Larrecq. Log auditing through model checking. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW'01), Cape Breton, Nova Scotia, Canada, June 2001*, pages 220–236. IEEE Comp. Soc. Press, 2001.
7. P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.