Logic in Virtualized Systems

Hedi Benzina
LSV, ENS Cachan, CNRS, INRIA
61 avenue du Président Wilson, 94230 CACHAN, France
Email: benzina@lsv.ens-cachan.fr

***Abstract*—As virtualized systems grow in complexity, they are increasingly vulnerable to denial-of-service (DoS) attacks involving resource exhaustion. A malicious driver downloaded and installed by the system administrator can trigger high-complexity behavior exhausting CPU time or stack space and making the whole system unavailable. Virtualized systems such as Xen or VirtualBox have been proposed to increase the level of security on computers. On the other hand, such virtualized systems are now targets for attacks. The weak spot of such systems is domain zero administration, which is left entirely under the administrator's responsibility, and is in particular vulnerable to attacks.**

We propose to let the administrator write and deploy security policies and rely on our policy compiler *RuleGen*, and Orchids' fast, real-time monitoring engine to raise alerts in case any policy violation, expressed in a fragment of linear temporal logic, is detected. This approach has shown its efficiency against real DoS exploits.

Keywords: Virtual machine monitors, security policies, temporal logic, denial of service attacks, intrusion detection.

I. Introduction

Virtualization is becoming an increasingly popular method to achieve security-based solutions for both personal and industrial activities. This technology presents the illusion of many smaller virtual machines, each running a separate operating system instance on the same machine. Such a virtualized environment provides isolation, security, low performance overhead, and supports heterogeneous applications. For large enterprises, where on-demand capabilities are highly desirable, such a virtualization technique is very helpful in building an ideal solution for security and application consolidations.

A virtualized system such as Xen [1], VirtualBox [2] or QEmu [3] allows one to emulate one or several so-called *guest* operating systems (OS) in one or several *virtual machines* (VM). The different VMs execute as though they were physically distinct machines. The various VMs run under the control of a so-called *virtual machine monitor* (VMM) or hypervisor, which one can think of as being a thin, highly-privileged layer between the hardware and the VMs. All the administrative tasks are done under a privileged VM called *domain zero* in Xen terminology or the *host operating system* under VirtualBox.

The weak link of almost all VMMs is *domain 0* administration where even a benevolent administrator may introduce trojans, by downloading new updates or malicious drivers, possibly leading to complete corruption, not just of the VMs, but of the VMM itself. Recently, several new attacks appeared against *domain zero* [4].

As a defense, we propose to let the system administrator describe all undesired activity flows through simple temporal formulas. Our point is that $signatures$, i.e., specifications of attack patterns, are best expressed in a logic including temporal connectives to express ordering of events. This allows one to describe attacks in a declarative way, free of implementation decisions. As in programming languages, using a $declarative$ language allows one to focus on $what$ to monitor instead of how to monitor. This caters for easier writing and easier understanding of signatures, and improves maintainability of of signature files.

We show how RuleGen, a tool for automatic generation of attacks signatures, can be used with the Orchids intrusion detection system [6] to protect the domain zero against denial of service attacks.

Outline. The plan of the paper is as follows. After reviewing related work in Section II, we present some basic components of our solution in Section III. In particular we explore the use of linear temporal logic with first-order variables for writing security policies in Section III-B. This will give the reader a flavor of what temporal logic is, and how we can generate an attack signature from a temporal formula. We shall show that the translation algorithm for this logic is NP-complete. We describe an attack scenario in Section IV. And show a line of defense against real exploits in Section V. We report on practical results in Section VI and conclude in Section VII.

II. Related work

In a previous work [7], we proposed an approach for protecting VMs using Orchids [5]. In this paper we present a new idea for protecting the VMM itself by imposing security policies expressed in temporal logic and that can be easily translated to attack signatures used by Orchids. Whereas dealing with recent attacks require expressive property specification languages, current specification techniques are restricted in the properties they can handle. They either support properties expressed in propositional temporal logics and thus cannot cope with variables ranging over infinite domains [17], do not provide both universal and existential quantification [22] or only in restricted ways [25], cannot handle unrestricted negation [24], do not provide quantitative temporal operators [23], or cannot handle both past operators [18].

This work was supported by grant DIGITEO N°2009-41D from Région Ile-de-France.

The logic we use has the advantage that it is high-level, compact and mostly readable notation for events occurring as time passes. In addition, we propose a tool that generates automatically attacks signatures from policy formulas written in this logic. Safety properties can be described and translated to rules that can be added to the intrusion detection system (IDS). Much work have been done in this area. Sekar *et al.* brought the idea of *model-carrying code* (MCC) [12]. This technique allow one to accept and execute code even from untrusted producers. Software fault isolation (SFI) [11] instruments a program so that it cannot violate a built-in a safety policy. Security automata SFI implementation (SASI) is an SFI-based tool developed by Erlingsson and Schneider [10] for enforcing security policies encoded in a security-automata language. While our approach reduces consequently the effort done by the user, these techniques need a big deployment effort and technical intervention from the user.

III. Preliminaries

We present in this section some important components of our approach.

A. Orchids and the auditd sensor

In a previous implementation [7], we used Orchids to protect VMs. Here we want to use Orchids to protect the VMM and *domain zero*. The latter reports sequences of system calls through the standard `auditd` daemon, a component of SELinux [8], which one can even run without the need for installing or running SELinux itself. Using `auditd` as event collection mechanism, we get events in the form of strings such as:

```
1276848926.326:1234 syscall=102
success=yes a0=2 a1=1 a2=6 pid=7651
```

which read as follows: the event was collected at date 1276848926.326, written as the number of seconds since the epoch (January 01, 1970, 0h00 UTC), and is event number 1234; this was a call to the `socket()` function (code 102), with parameters `PF_INET` (Internet domain, where `PF_INET` is defined as 2 in `/usr/include/socket.h`—`a0` is the first parameter to the system call), `SOCK_STREAM` ($= 1$; `a1` is connection type here), and with the TCP protocol (number 6, passed as third argument `a2`); this was issued by process number 7651 and returned with success.

Orchids recognizes scenarios by simulating known finite automata, from a given event flow. This method allows the writing of powerful stateful rules suitable for intrusion detection.

In the next section, we will show how to avoid the complexity of writing such rules by automatically generating them from simple logic formulas.

B. The logic

Our logic is a fragment of linear temporal logic (LTL) *with past*. See the second part of [7] for a more in-depth treatment. We shall see that relying on a logical language with a well-defined semantics allows the policy compiler to benefit from several optimizations. This is entirely automated, hence safe, contrarily to more imperative languages like Russel [13], where defining and applying these optimizations must be done by hand, and is therefore error-prone. In this section we show how one can describe attacks or security policies using this logic. The without ($\diagdown$), box ($\blacksquare$) and diamond ($\blacklozenge$) past operators should be used.

For example, to describe an attack scenario where a first event *A* happens, then *B* and *C* happen successively afterwards, we write this formula : $C \wedge \blacklozenge(B \wedge \blacklozenge A)$.
Another attack scenario would be the following : the events *A* then *B* happen, and since then the event *C* never happened, the corresponding formula is : $\blacklozenge(B \wedge \blacklozenge A) \diagdown C$. We can also try to make sure that an event *A* never happened in the past by writing this formula : $\blacksquare\neg A$.

Example: To give a simple example how the reported events from `auditd` can be used to write formulas, let us consider the following formula :
$(.auditd.syscall == 3 \vee .auditd.syscall == 4) \wedge (.auditd.a0 == \text{“}/etc/passwd\text{“}) \wedge (.auditd.uid! = 0)$
This formula describes an attack where an unauthorized user ($uid! = 0$) tries to access the sensitive file ($/etc/passwd$) for reading or writing. Note that the $read$ and $write$ system calls have respectively the codes 3 and 4. We have to mention that this formula corresponds to an attack signature, however we can transform it into a formula that describes a safety property by only adding the negation operator (not) at the beginning of the formula.

C. The algorithm

In [7], we presented an algorithm that compiles formulas specifying policies into rules that can be fed to the Orchids IDS.

IV. Threat model

Since VMMs are running under usual operating systems, they are also exposed to threats and attacks. Denial of service (DoS) is one of the most serious threats. One possible scenario is this: the administrator needs to download some driver and install it; this driver contains a trojan horse, which runs a DoS attack. Even if the server is secure, the administrator can be the victim of a man-in-the-middle attack, that substitutes the authentic driver with a malicious one. Once the new driver has been installed and the DoS attack has been triggered, one cannot hope to react in any effective way. To be more precise, the VMM controls entirely the hardware abstraction layer that is presented to each of the guest OSes: no network link, no disk storage facility, no keyboard input can be trusted by any guest OS any longer. Worse, the VMM also controls some of the features of the processor itself, or of the MMU, making memory or register contents themselves unreliable.

We present in the next section our approach to avoid this kind of threats.

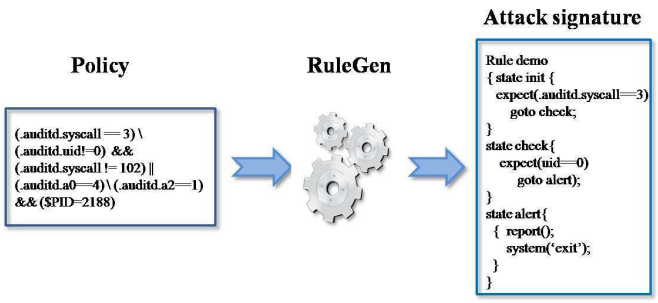

Fig. 1. The RuleGen tool

V. Protecting VMMs with logic, RuleGen and Orchids

Our approach is simple but efficient : we show how the administrator of a virtualized architecture can protect his system against DoS attacks by writing policy formulas and relying on our tool RuleGen that compiles these formulas to rules that can be fed to the Orchids IDS. The latter can easily detect and stop complex attacks.

Our case study is the following: we run a malicious version of FUSE [14], a generic filesystem driver, under the *domain zero* of a Xen hypervisor (which is equivalent to *the host OS* of the VirtualBox VMM. This modified version of FUSE contains two real-world DoS attacks that can be triggered easily after installing the driver.

Then we show how the administrator can prevent these two attacks using RuleGen. The latter is an implementation of the algorithm presented in [7]. RuleGen can efficiently generate attack signatures from temporal formulas written in our logic. RuleGen helps the administrator avoid the complexity of writing Orchids' rules. This is important since the attack base of Orchids needs to be updated frequently and sometimes quickly.

A. The multiple listen DoS attack

This attack [15] is quite simple, but can cause a lot of damage, in particular when triggered in the host where the VMM is running. It makes the system unavailable and the administrator becomes unable to react since his administration platform has crashed. Consequently, all running VMs will be unavailable. The attack consists in two calls to the $listen$ function (linux/socket.h) on the same ATM (Asynchronous Transfer Mode) socket descriptor. Linux 2.6.x kernels and many Linux distributions are vulnerable to this attack.

We show below how we insert the code of this attack in the source code of FUSE:

```
//FUSE driver file : fusermount.c
switch (ch) {
    case 'u':
      unmount = 1;
      /******* malicious code ******/
      int sock = socket(PF_ATMSVC, 0, 37);
      listen(sock, 7);
      listen(sock, 2);
      system("/bin/cat /proc/net/atm/pvc");
      /******************/
      break;
    case 'h':
      usage();
```

We modify the source code of FUSE by adding the code of the exploit to the file "fusermount.c". When the client tries to unmount the filesystem (using the command : "fusermount -u"), the attack is triggered and the system crashes.

1) Expressing the attack in our logic: This attack can be expressed as follows :

$$\blacklozenge(\$PID == .auditd.pid \wedge .auditd.syscall == 102 \wedge .auditd.a0 == 4) \wedge (.auditd.pid == \$PID \wedge .auditd.syscall == 102 \wedge .auditd.a0 == 4)$$

This formula describes two events correlated by the variable $\$PID$ (the process identifier) and connected with the "$\wedge$" (and) operator. The first event is a $socketcall$ system call (code 102) with the first argument $a0$ equals 4 (the $listen$ function). The pid of the process is catched from the $.auditd.pid$ field and put in the variable $\$PID$. The second event is similar to the first one, but should be triggered by the same process, and should come later since the first one is preceded by the diamond $\blacklozenge$ operator (which means that it happened once in the past).

2) The generated rule: RuleGen parses the formula and generates the attack signature in respect of the syntax of Orchids' rules. The first state should be named $init$, this state waits for a $listen$ function call ($socketcall$ system call with first parameter equals 4) and at the same time catches the pid of the process triggering this event. Once the second state is reached, we are sure that time has elapsed, and the expected event was triggered. The second state named $check$ waits for a second call to the $listen$ function and goes to the $alert$ state if the event is triggered by the same process (the two events are correlated by the $\$PID$ variable). The third state is the $alert$ state which is responsible for killing the offending process and making a report that helps the security administrator in his work. The generated rule is below :

```
rule listen_atm_vcc
{
  state init
 {
    $PID = .auditd.pid;
    expect (.auditd.syscall == 102 &&
            .auditd.a0 == 4)
       goto start;
  }
  state check
  {
```

```
    expect (.auditd.pid == $PID &&
            .auditd.syscall == 102 &&
            .auditd.a0 == 4)
        goto alert;
  }
 state alert
 {
  system("kill -9"+str_from_int($PID));
  report();
  }
}
```

B. The lock_lease DoS attack

This is another DoS attack [16]. In brief, the attack goes in an infinite loop trying to obtain numerous file-lock leases, which will consume excessive kernel log memory. Once the leases timeout, the event will be logged, and kernel memory will be consumed. Many Linux 2.6.x kernels are vulnerable to this attack.

We show below where we inserted the code of the exploit in FUSE source code to make sure that it will be executed when the kernel starts using the driver.

```
//FUSE file : fusermount.c
static int open_fuse_device(char **devp)
{
  int fd = try_open_fuse_device(devp);
  /***** malicious code ****/
  int r;
  while(1)
  {
    //lock
    r = fcntl(fd, F_SETLEASE, F_RDLCK);
    //unlock
    r = fcntl(fd, F_SETLEASE, F_UNLCK);
  }
  if (fd >= -1) return fd;
  fprintf(stderr,"%s: fuse device error");
  return -1;
}
```

When the filesystem is mounted, the $fusermount$ program (fusermount.c) tries to open *"/dev/fuse"* (the *open_fuse_device()* function). At this time, the attacker can be sure that his exploit is being executed.

1) Expressing the attack in our logic: This attack can be written in our logic as follows :

$$(\blacklozenge(.auditd.syscall == 5 \wedge \$PID == .auditd.pid) \wedge (loop \wedge (.auditd.syscall == 221 \wedge .auditd.a2 == "f\_setlease" \wedge .auditd.pid == \$PID))) \setminus (.auditd.syscall == 6 \wedge .auditd.pid == \$PID)$$

Here is how this formula can be read: every process that makes a call to the $open$ function (code 5) and then makes numerous locks ($fcntl64$ system call with code 221, and with the parameter "f_setlease") on a descriptor without closing it ($close$ system call has the code 6), is a suspicious one and should be stopped.

The keyword $loop$ is used when we need to describe looping events : $loop \wedge A$ means that the event A will loop. The administrator, and after getting experimental results, should fix the permitted number of locks that can be made to a descriptor without making the system crash (10 in the case of our experiments).

2) The generated rule: As shown in the previous attack, RuleGen transforms this formula into an attack signature that can be added to the Orchids' rules base. The generated rule is:

```
#define MAX_CALLS 10
#define SYS_OPEN 5
#define SYS_CLOSE 6
#define SYS_FCNTL 221
#define LEASE "F_SETLEASE"

rule lock_lease_dos
{
  state init
 {
    $counter = 0;
    $PID = .auditd.pid;
    expect (.auditd.syscall == SYS_OPEN )
       goto start;
  }
 state start
  {
   $counter = 1 ;
   expect (.auditd.pid == $PID  &&
       .auditd.syscall == SYS_FCNTL
       .auditd.a2 == LEASE)
       goto listen_loop;
  }
 state listen_loop
  {
    $counter = $counter + 1;
    if ($counter == MAX_CALLS)
       goto alert;

    expect (.auditd.pid == $PID &&
       .auditd.syscall == SYS_CLOSE)
       //abort the surveillance
       // of this process
       goto init;

    expect (.auditd.pid == $PID &&
         .auditd.syscall == SYS_FCNTL &&
         .auditd.a2 == LEASE &&
         $counter < MAX_CALLS)
       goto listen_loop;
  }
 state alert
```

```
 {
   //kill the attack process
   system("kill -9"+str_from_int($PID));
   report();
 }
}
```

VI. Experiments

We deployed our solution on a 1000 MHz Intel Core Duo machine with 4096 KB cache running Xen 3.3.1 as hypervisor. Dom0 is a 32-bit Fedora 11 Linux with 2 GB of RAM. We also use two guest VMs: Fedora 10 and Ubuntu 8 with 1 GB and 512 MB RAM, respectively.

We perform a set of experiments to evaluate RuleGen and Orchids performance on the target platform using a malicious FUSE driver. Practical results look promising: Orchids can detect simultaneously the two DoS attacks presented earlier and stop them before the system crashes. We do not claim that this approach is the final solution to DoS attacks. There are cases where the DoS attack is stealthy; after all, this is a starting point for writing complete security policies that can be able to protect the system from families of DoS attacks.

VII. Conclusion and future work

Our import, as we hope to have demonstrated, is that it is possible to describe policies through simple temporal logic without caring about technical details related to the intrusion detection mechanism. This temporal logic has a clean semantics and a clear notion of correlation of events. We have also presented RuleGen, a tool for compiling security policies written in a temporal logic into attack signatures expressed as EFSA (extended finite-state automata). RuleGen offers to virtualized systems administrators the ability to write their own policies. We have shown that RuleGen and Orchids can be used to protect VMMs from DoS attacks.

This work opens several directions for further investigation. First, we would like to raise the abstraction level of the generated attack signatures to reduce the description effort required when writing policy formulas. Next, we aim to apply the use of RuleGen and Orchids to a wider range of applications not limited to VMMs. Finally, we feel that using other low-level sensors to detect kernel-level malware would be a plus.

References

[1] Xen, 2005–2011. http://www.xen.org/.

[2] VirtualBox, 2011. http://www.virtualbox.org/.

[3] Qemu, 2011. http://www.qemu.org/.

[4] R. Wojtczuk. Subverting the Xen hypervisor. In *Black Hat'08*, Las Vegas, NV, 2008.

[5] J. Olivain and J. Goubault-Larrecq. The Orchids intrusion detection tool. In K. Etessami and S. Rajamani, editors, *17th Intl. Conf. Computer Aided Verification (CAV'05)*, pages 286–290. Springer LNCS 3576, 2005.

[6] J. Goubault-Larrecq and J. Olivain. A smell of Orchids. In M. Leucker, editor, *Proceedings of the 8th Workshop on Runtime Verification (RV'08)*, Lecture Notes in Computer Science, pages 1–20, Budapest, Hungary, Mar. 2008. Springer.

[7] H. Benzina and J. Goubault-Larrecq. Some Ideas on Virtualized Systems Security, and Monitors. In DPM/SETOP'10, LNCS 6514, pages 244-258. Springer, 2010.

[8] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. Technical report, NSA, 2001.

[9] M. R. Garey and D. S. Johnson. Computers and Intractability, a Guide to the Theory of NP-Completeness. W.H. Freeman and Co., 1979

[10] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In RSP: 21th IEEE Computer Society Symposium on Research in Security and Privacy, 2000.

[11] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, pages 203216, Asheville, NC, December 1993.

[12] R. Sekar, C. Ramakrishnan, I. Ramakrishnan, and S. Smolka. Model-carrying code (MCC): A new paradigm for mobile-code security. In *Proceedings of the New Security Paradigms Workshop (NSPW 2001)*, Cloudcroft, NM, Sept. 2001. ACM Press.

[13] A. Mounji. Languages and tools for rule-based distributed intrusion detection. PhD thesis, FUNDP, Namur, Belgium, 1997.

[14] FUSE, 2011. http://fuse.sourceforge.net/.

[15] CVE-2008-5079, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5079.

[16] CVE-2005-3857, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-3857.

[17] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In Proc. of the 26th Conference on Foundations of Software Technology and Theo- retical Computer Science (FSTTCS), vol. 4337 of LNCS, pp. 260-272. Springer, 2006.

[18] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst., 20(2):149-186, 1995.

[19] K. Havelund and G. Rosu. Efficient monitoring of safety properties. Int. J. Softw. Tools Technol. Trans., 6(2):158-173, 2004.

[20] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In Proc. of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI), vol. 2937 of LNCS, pp. 44-57. Springer, 2004.

[21] S. Hallé and R. Villemaire. Runtime monitoring of message-based workflows with data. In Proc. of the 12th International IEEE Enterprise Distributed Object Com- puting Conference (EDOC), pp. 63-72. IEEE Computer Society, 2008.

[22] J. Hakansson, B. Jonsson, and O. Lundqvist. Generating online test oracles from temporal logic specifications. Int. J. Softw. Tools Technol. Trans., 4(4):456-471, 2003.

[23] M. Roger and J. Goubault-Larrecq. Log auditing through model-checking. In Proc. of the 14th IEEE Computer Security Foundations Workshop (CSFW), pp. 220-234. IEEE Computer Society, 2001.

[24] A. P. Sistla and O. Wolfson. Temporal triggers in active databases. IEEE Trans. Knowl. Data Eng., 7(3):471-486, 1995.

[25] O. Sokolsky, U. Sammapun, I. Lee, and J. Kim. Run-time checking of dynamic properties. Elec. Notes Theo. Comput. Sci., 144(4):91-108, 2006.