A novel approach for a file-system integrity monitor tool of Xen virtual machine

Conference Paper · January 2007

DOI: 10.1145/1229285.1229313 · Source: DBLP

2 authors, including:

Yoshiyasu Takefuji
Keio University
563 PUBLICATIONS **3,249** CITATIONS

SEE PROFILE

A Novel Approach for a File-system Integrity Monitor Tool of Xen Virtual Machine

Nguyen Anh Quynh, Yoshiyasu Takefuji
Graduate School of Media and Governance,
Keio university
5322 Endoh, Fujisawa, Japan 252-8520
{quynh,takefuji}@sfc.keio.ac.jp

ABSTRACT

File-system integrity tools (FIT) are commonly deployed host-based intrusion detections (HIDS) tool to detect unauthorized file-system changes. While FIT are widely used, this kind of HIDS has many drawbacks: the intrusion detection is not done in real-time manner, which might render the whole scheme useless if the attacker can somehow take over the system with privileged access in the time between. The administrator also has a lot of problems to keep the base-line database updating. Besides, the database and the FIT itself are vulnerable if the attacker gains local privileged access.

This paper presents a novel approach to address the outstanding problems of the current FIT. We propose a design and implementation of a tool named *XenFIT* for Xen virtual machines. XenFIT can monitor and fires alarms on intrusion in real-time manner, and our approach does not require to create and update the database like in the legacy methods. XenFIT works by dynamically patching memory of the protected machine, so it is not necessary to install any kernel code or user-space application into the protected machines. As a result, XenFIT is almost effortless to deploy and maintain. In addition, thanks to the advantage introduced by Xen, the security polices as well as the detection process are put in a secure machine, so XenFIT is tamper-resistant with attack, even in case the attacker takes over the whole VM he is penetrating in. Finally, if deploying strictly, XenFIT is able to function very stealthily to avoid the suspect of the intruder.

Categories and Subject Descriptors

H.2.0 [**General**]: Security, integrity and protection; D.4.6 [**Security and Protection**]: Invasive Software

General Terms

Security, Performance

Keywords

Rootkit, Intrusion Detection, Xen Virtual Machine, Linux

1. INTRODUCTION

ASIACCS'07, March 20-22, 2007, Singapore.

File-system is the heart of any computer system: it contains data files, executable files, configuration and administrative files. Almost all the security incident resulted in critical files changed: the attacker would wish to return again without any authentication, or he wants to steal precious information like password, credit-card number. To do that, he usually installs various Trojan or back-doors in the system before leaving. Further more, the sophisticated attacker always modifies or removes logging data to cover his traces. All of these activities lead to new files are generated, existent files are altered or deleted, and consequently the system's integrity is changed. That is why file-system integrity is a very important factor for computer security, and FIT should always be employed to protect critical systems. Once installed and configured properly, these monitoring tools can detect the illegal tampering and fire alarms on the incident.

Regarding the practical solutions on FIT, there are many choices. Some of the most popular tools are available with even the source code such as Tripwire ([10]), or Tripwire-alike tools like AIDE ([19]), Osiris ([20]) and Samhain ([22]). To detect the changes in the file-system, a FIT is usually equipped with a security policy: if it detects the violation against the policy (for example, a system binary file is overwritten), it will record the suspected action and signal the administrator. The administrator then must also specify a list of files or directories the FIT needs to watch out. Initially FIT is executed to collect a base-line database of file-system parameters, such as permission or ownership, file size, MAC times along with cryptographic checksums over the file content. During the scheduled operation, FIT compares the current state of the file-system with the information in the database. Any suspected change against the security policy is flagged, and the corresponding report is generated and sent to the administrator.

Unfortunately, while these solutions are widely used, they still leave much things to desire: their schemes have some drawbacks which make them vulnerable to attackers. Below is the most outstanding problems the users are experiencing:

(1) **Delay detection**: Most of the current solutions work in off-line manner, in which the tools are periodically scheduled to run and verify the integrity of the system. Clearly this leaves a window of opportunity for the attacker to penetrate the system and evade our IDS.

(2) **High deployment complexity**: Organizations with hundred machines (like ISPs or large data-center) experience a lot of problem to deploy FIT: they all require to be well prepared and planed. One of the troubles is that all systems must generate the base-line database, then these database must be safely stored so the attacker cannot damage them. However the database must be easily retrieved by the administrator when he wants to run

FIT to verify the integrity. This contrary demand alone causes a lot of headache for the administrators.

(3) **Highly maintaining overhead**: The base-line database must be maintained and updated when there are changes in the system, for example if the software are upgraded. This job might be required to be done frequently, especially because nowadays the software vulnerabilities are published almost everyday, which then come with various patches [4].

(4) **Poor information**: As almost all the FIT run in off-line manner, the information they can provide to help the forensic process are quite limited. For example they cannot answer fundamental questions such as which *proccess-name/process-id* made the modification to the data, and what happened around that event. Without these kind of clues, forensic analyzer must do a hard job to investigate the intrusion.

(5) **Highly vulnerable to attack**: Most of the current FIT are user-space applications, which makes them vulnerable if the attacker gets the root access. In that case he can kill the verifying processes, then completely disables or removes these tools, thus disarm all the intrusion system. Besides, the base-line database and the FIT itself must be protected at all cost, otherwise if the attacker compromise them, the monitoring result becomes useless. But again, the situation can be very complicated if the attacker gains privilege access right.

(6) **Exposed problem**: Most of the FITs are visible in the system: the attacker can easily find out that he is observed by an IDS. The fact that our monitoring tools fail to hide their presence might reduce its chance to detect the malicious activities of a highly-skillful attacker.

Recently some real-time FITs such as SNARE ([9]) and I3FS ([13]) are proposed to fix the problems. The common idea of these solutions is that they modify the system kernel to capture all the I/O events, then decide if the access is legal or not. Unfortunately these tools either have high deployment complexity, or simply not very suitable for some detection/prevention schemes.

This paper proposes a solution named XenFIT, a real-time Flesystem Integrity monitor Tool, to address 6 above-mentioned shortcomings of the current FITs. XenFIT is made to run in Xen virtual machines ([6]), [15]), and it can report the security policy violations in real-time. In contrast with other tactics, XenFIT works by dynamically patching the memory of the monitored virtual machines (VM), so it get notified about potential problems of the protected VM in real-time manner, with richer information about the context of the incident. Because XenFIT scheme does not require any program or kernel module loaded inside the protected VM, it causes almost zero-effort to deploy. In addition, because there is no need to generate and update the base-line database like in legacy approaches, the maintenance cost of our solution is very low. To detect the intrusion, XenFIT have security policies for each specific VM, and the detection process is safely done in a centralized machine. Last but not least, if XenFIT is installed and configured properly, it is hard to subvert, and able to survive even if the attacker gains root access.

The rest of this paper is organized as followings. The section 2 covers some background on Xen Virtual Machine. The section 3 presents the design and implementation of XenFIT, while section 4 discusses some issues of our solution. Section 5 evaluates our approach with some rootkits and performance benchmark. Section 6 summaries the related works, and discusses in detail the outstanding problems of some popular FIT. Finally we conclude the paper in section 7.

2. BACKGROUND ON XEN VIRTUAL MACHINE

Our solution XenFIT is based on Xen, and exploits the debugging architecture of Xen to inject detection breakpoints into protected VM. In this part, we will take a brief look at Xen version 3.0.2, the latest version as of this writing. After that we discuss the kernel debugging architecture for Xen VM, which will be used by our XenFIT solution.

2.1 Xen Virtual Machine

Basically, Xen is a thin layer of software above the bare hardware. This layer is either called hypervisor or virtual machine monitor. Its main job is to expose a virtual machine abstraction that is slightly different from the underlying hardware. Xen introduces a new architecture called *xen*, which is very similar to x86 architecture. The virtual machines (VMs) executing on Xen are modified (at kernel level) to work with xen architecture[1]. All the accesses of DomUs to the hardware and peripherals must go through Xen, so Xen can keep the close eye to those VMs and control all the activities.

Running on top of Xen, VM is called Xen domain, or domain in short. A privileged special domain named Domain0 (or Dom0 in short) always runs. Dom0 controls other domains (called User Domain, or DomU in short), including jobs like start, shutdown, reboot, save, restore and migrate them between physical machines. Especially, Dom0 is able to map and access to memory of other DomUs at run-time.

2.2 Exceptions handling in Xen

In Xen, to manage other DomUs and the physical hardware, the hypervisor layer runs at the highest privilege level (ring 0 on x86 architecture). To provide a strong isolation between DomUs as well as between DomUs and the hypervisor, all the DomUs are run at lower level (ring 1 on x86 architecture). So are the interrupt handlers of DomUs: While normally the interrupt handlers are registered in the interrupt descriptor table (IDT), Xen does not allow DomU to install their handlers themselves because of the security reasons: it cannot give DomU the direct access to the below hardware. Instead, DomU's kernel are modified at source code, so the hypervisor captures the interrupts instead of letting the DomUs handle them. [2]

Specifically, in the asynchronous interrupt case, also called exception and generated when the system executes the *INTO*, *INT1*, *INT3*, *BOUND* instruction or by page faults: these exceptions are handled by the hypervisor layer first instead by the DomU's kernel. To register handlers, DomU's kernels are modified to call the hypercall named *HYPERVISOR_set_trap_table* to setup the exception handlers. The handlers are functions initialized at machine boot time, and managed by hypervisor layer.

2.3 Debugging Support Architecture in Xen

In x86 architecture, *INT3* is a breakpoint instruction which is used for debugging purpose[3]. Whenever this instruction is hit, the control is passed to the exception handler of *INT3* in kernel space.

[1]The recent hardware from Intel and AMD allow commodity OSes to run on Xen without any change, but we will not consider them here

[2]In this paper we only consider DomU as in para-virtualization case, because we do not have access to the latest VT-enable machine yet.

[3]Breakpoint instruction is an one-byte opcode with the value of *0xCC* on x86 platform

In Xen, the sequence of handling the *INT3* exceptions is as in the following steps:

- When the DomU hits the breakpoint instruction, it raises the exception #BP.
- The system makes a hypervisor switch to give control to the *INT3* handler staying in the hypervisor layer.
- The *INT3* handler in hypervisor checks if DomU is in kernel mode. If that is not the case, Xen returns the control to DomU
- If the exception comes from DomU's kernel, Xen pauses the DomU for inspection.

In fact, the Xen debugger works by exploiting the mentioned feature: When the debugger server running in Dom0 detects that the concerned domain is paused, it comes to inspect the DomU's kernel, then resume it after it finishes the job ([1]).

Besides *INT3*, *INT1* is another special interrupt made for debugging. This interrupt sends the processor into the single-step mode, in which after each construction, the handler of *INT1* is called. To make this happen, we only need to enable the trap flag (TF) of the FLAGS register. The processor switches to normal mode if the *TF* flag is turned off. And similarly to the case of *INT3*, when the system is in single-step mode, after each instruction the control is changed to the INT1 handler at hypervisor layer. The sequence of handling *INT1* is same as in *INT3*'s case.

3. XENFIT SOLUTION

XenFIT is our answer to the problem of the current FIT, and XenFIT is made to work for Xen virtual machines. Our object is to use XenFIT instead of the available FITs to protect Xen VM, to assure that the intrusions that violate the system integrity are more effectively detected. This section presents the goals, design and implementation of XenFIT.

3.1 Goals and Approaches

With XenFIT, we aim to address 6 problems of the current FIT as discussed in section 2. Therefore the design of XenFIT is driven by the following goals:

- **Real-time detection**: XenFIT should be able to detect the violation at real-time to fix the problem (1). To achieve this goal, XenFIT exploits the debugging architecture available in Xen: XenFIT runs as a daemon process in Dom0, and at run-time puts breakpoints instruction into DomU's kernel code at the right places of its file-system stack. With this approach, XenFIT is able to intercept operation of DomU's file-system, so it can see what is happening with all the I/O flow. Whenever the DomU's kernel hits these breakpoints, the control path is transferred to Dom0 via the hypervisor layer, and then to our XenFIT. XenFIT handles the exceptions similar to the way Xen debugger does: It inspects the DomU's kernel, and gather concerned data on file-system activities, which can disclose the malicious intrusion. Once we have the information, we can verify them against the security policies kept in Dom0, and the violations are reported accordingly. Then XenFIT resumes the related DomU, and waits for the next breakpoint exception.

  From DomU's point of view, XenFIT functions as an *automatical* kernel debugger. But in fact, XenFIT only handles the debugging event to access and capture information related to file-system to detect activities that breach system integrity.

- **Stealthy functioning**: In contrast with other solution, as explained above, XenFIT only runs in user-space in Dom0. Besides, our solution does not require any code to run in the protected DomU. The only changes to DomU is some breakpoints put in its kernel memory. Therefore the intruder has very little chance to detect the presence of XenFIT.

  We will discuss few tactics in section 4 to make XenFIT more invisible and harder to detect even if the attacker has root access. All of this exercises are to improve the problem (6).

- **Low maintaining overhead**: Because XenFIT knows what is happening in the system in real-time, clearly it does not need the base-line database to compare with like other FIT solutions. All it needs is a security policy kept in Dom0, so it can verify if the gathered activities of the protected DomU are against the policy. Therefore XenFIT does not require to generate the database at initializing or to maintain the database during the run-time like others. As a result, the problem (3) is addressed.

- **Easy to deploy**: As XenFIT can gather the integrity related information at run-time only by injecting breakpoints into the protected VM, it does not need any code loaded or baseline database in the virtual domain. In fact, everything we need to deploy our IDS is to run XenFIT in Dom0, and let it monitor the protected DomU. Subsequently, this benefit fixes the problem (2)

  Another merit of our approach is that because XenFIT peeks data from DomU by using breakpoints at run-time, we do not need to patch the DomU's kernel. As a result, maintaining XenFIT for different kernel versions becomes extremely easy.

- **Enriched information**: Since XenFIT captures the hook events at run-time, it can provide valuable information such as the time when the event happens, who (user-id) generates the event, and the related process-id/ process-name. These knowledge give the administrator much better view on what happened in his VMs. Consequently XenFIT can improve the problem (4).

- **Resistant tampering**: XenFIT is proposed to run in Dom0 only, and we have absolutely no code running in protected VMs. Therefore the attacker cannot apply the strategy of shutting down the integrity checking operation like he can do with traditional FIT. Even if he takes over the whole system and has access to his kernel, everything he can do is to disable the breakpoints putting in his kernel, but cannot tamper with our IDS running in a separated and highly-secure VM (specifically, in Dom0).

  Furthermore, XenFIT keeps the security policies in Dom0, so the attacker cannot compromise them, either.

  Another advantage of our solution is that XenFIT in Dom0 does not expose to the network, so the attacker has no chance to exploit it from outside like in traditional way[4].

  Clearly with the above benefits, the problem (5) is partly addressed. Section 4 will discuss few more methods to harden the kernel of protected VM, so even if the attacker has root privilege, he cannot access and compromise the breakpoints XenFIT put inside the kernel.

[4]Actually we recommend to run Dom0 without network access, thus mitigate the security risks for the whole system.

One more advantage of XenFIT is that it is quite flexible: because of its design, we can start or stop monitoring any DomU at any time we like, just by removing the breakpoints from its kernel. Even better, all the modification can be quietly done from Dom0, and that can make XenFIT even more stealthy to the intruder inside the protected DomU.

Another advantage of XenFIT is that we can centralize all the security policies of protected DomUs in Dom0, thus managing and updating them can be done very easily. This benefit is mostly welcomed by the administrator, because otherwise the burden of keeping control on scatter policies in various VMs is very big, especially for system with hundred VMs running on it.

All the approaches above lead us to the design of XenFIT as followings.

3.2 XenFIT Design

3.2.1 XenFIT Architecture

XenFIT comes in a shape of a daemon process named *xenfitd* and runs in user-space of Dom0. It put breakpoints into the kernel memory of the protected VM, and handles the interrupt events, which happen when the breakpoints are hit. The overall and simplified architecture of XenFIT is outlined in Fig.1. In our architecture,

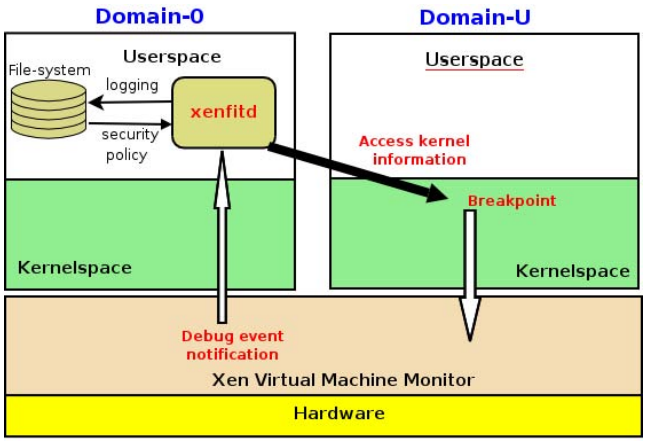

Figure 1: XenFIT architecture

whenever a breakpoint is hit inside the DomU's kernel, control is changed to *xenfitd* staying inside Dom0, and the DomU itself is paused. All of these jobs are done by the hypervisor automatically and we do not need to modify any part of the hypervisor for this "feature". The *xenfitd* daemon then accesses the DomU's kernel and gather necessary information related to the breakpoint. After that it resumes the DomU and let it continue running normally. The whole process is repeated when the next breakpoint is hit in DomU's kernel.

Regarding the collected information, *xenfitd* verifies them against a security policy of corresponding DomU, and an alarm is fired if a violation is confirmed. Besides, *xenfitd* also logs the collected information to file-system of Dom0, so the administrator can take a closer look when investigating the potential problems.

3.2.2 Performance Challenge

One of the challenges of XenFIT is the performance penalty problem: every breakpoint in DomU leads to several hyper-switchs: first is a switch from DomU kernel to hypervisor when the breakpoint is hit; second is a switch from hypervisor to Dom0 to have XenFIT handled the debug event; and finally the control is given back to the original DomU. These switchings can cause a lot of negative impact to to the overall performance of the protected DomU.

When we first investigated the problem, we thought it was a good idea to employ the same tactic of the current Xen kernel debugger to monitor DomU, because the debugger also exploits INT3 to inspect DomU's kernel at run-time ([1]). However, this approach has a major problem that can affect the system performance: the debugger in Dom0 detects the debugging event by periodically polling DomU's status [5] to see if it is paused, which is the evidence that the breakpoint was hit. By default the checking interval is 10 million nanoseconds, which means the breakpoint is not processed immediately if it comes between the checking time. For debugging purpose, this is not a major concern because performance is not a priority. But for our target, that is unfortunately unacceptable because the whole process extremely slows down.

To fix the problem, we decide not to use the mentioned polling tactic of the Xen kernel debugger. Instead we exploit the fact that whenever the breakpoint as well as when the processor is in step-trace mode, the hypervisor sends an event to Dom0 to notifies potential debugger. While the standard debugger does not use this feature, we employ the trick, and have XenFIT handled the debugging event. To do that, XenFIT only needs to put the protected DomU into the debugging mode[6], and binds to the virtual interrupt dedicated for debugging event to get notified from the hypervisor[7] . Thanks to this strategy, XenFIT is instantly aware when the DomU hits the breakpoints, and does not need to poll the DomU for the paused status. The experiements prove that the solution significantly improves the overall performance.

3.2.3 Breakpoints

To capture the information that can be used to detect intrusion, it is very important to put the breakpoints at the right places in DomU's kernel. Because all the I/O from user-space to devices must go through system-calls, it is natural that we should put our breakpoints to I/O system-calls. Specifically we pay attention to I/O system-calls that creates, removes or change attributes of file,directory and device. See Table 1 for the full list of selected system-calls. By intercepting 15 system-calls[8], we can get all the interesting events about system integrity: for example we are notified when a file is *created/written to/renamed/unlinked*, a directory is *created/removed*, etc.

Regarding the breakpoints, one of the major concerns is that how can we know exactly where we must put the breakpoints into the kernel? An intuitive answer for this question is to rely on the kernel source code, and decide to put the breakpoints at related lines of code. Clearly this is an convenient way, because we can inspect the code and see where is the best place to intercept the system flow. So if we know the address in the memory of related lines of code, we can put the breakpoints there. But then, we have another question: how to determine the address of related lines of code?

Fortunately, this problem can be solved quite easily thanks to debugging information coming with kernel binary. In fact, we can exploit a feature made for kernel debugger: If the kernel is compiled with debug option, the kernel binary stores detail information in *DWARF* format about the kernel-types, kernel variables and, most importantly to our purpose, the kernel address of every source code

[5] This can be done thanks to the *libxc* function *xc_waitdomain()*
[6] This can be done with a domain control hypercall, with the special command *XEN_DOMCTL_setdebugging*
[7] The virtual interrupt is named *VIRQ_DEBUGGER*
[8] Thus 15 breakpoints are put into the kernel memory of protected VM

System-call	Gathered information
open	A file is open
close	A file is close
mkdir	A directory is created
rmdir	A directory is removed
link	A link file is created
unlink	A file is removed
rename	A file is renamed
setattr	File-permission is changed
mknod	A device file is created
chmod	Access mode of a file is changed
chown	Owner of a file is changed
setuid	A file is set uid, thus run with higher privilege
write	A file is written to
setxattr	Extra attributes are set on a file
removexattr	Extra file attributes are removed

Table 1: List of system-calls adopted by XenFIT

line ([7]). As a result, we only need to compile DomU's kernel with debug option on, and analyze the kernel binary to get the kernel addresses of the source code lines we want to insert the breakpoints to. Note that this option only generates a big debugged kernel binary file besides the normal kernel binary, and this debugged kernel saves all the information valuable for debugging process. We can still use the normal kernel binary, thus the above requirement does not affect our system at all.

To make clear on where to insert breakpoints, let us look at one example: the *unlink* system-call in the Fig. 2. In this function, the *unlink* system-call successfully removes a file in line *2086*. Because other error path does not remove the file, thus does not affect the file-system integrity, we should focus on this line of code and put the breakpoint there. When *XenFIT* is notified of this breakpoint, it comes to get the value of the variable *name*, which stores the removed file-name. Besides, we should also get the value of the variable *error*, which tells us the result of the system-call. We can get the address of these variables *name* and *error* from the DWARF debug information that comes with the kernel binary.

As a result, by intercepting this system-call, *XenFIT* knows which file is removed.

Likewise, each breakpoint has a separate breakpoint-handler, which is called when the corresponding breakpoint is hit. Each breakpoint-handler does different thing, but with the common object: to collect the useful information about the activities at the breakpoints, so XenFIT can rely on these information to detect the intrusion.

3.2.4 Correlate for further information

In the above example of the *unlink* system-call, we can see that XenFIT can only get the file-name as well as the result of the system-call, but nothing else. Even worse, in case of other system-calls such as *write*, if we rely on the function's code as in the above case, we can only get the file-descriptor of the affected file, but not its file-name[9]. To gather more useful information, we must correlate with information from other system-calls, as well as get information from other kernel objects. That is why we must pay attention to *open* and *close* system-calls.

- *Open system-call*: By intercepting this this system-call, we knows which file-name corresponds to which file-descriptor. XenFIT saves these kind of information, so it can correlate with other system-calls that only work with file-descriptor like *write* to figure out which file (in term of path-name) is affected.

- *Close system-call*: A process can open and close many files in its lifetime, and the old file-descriptor can be reused. By watching the *close* system-call and correlate with *open* system-call, we can know exactly which files are still open.

[9]Many file-system related system-calls such as *write* work on file-descriptor instead of file-name.

```
2056 static long do_unlinkat(int dfd,
                const char __user *pathname)
2057 {
2058   int error=0;
2059   char * name;
...
2063
2064   name=getname(pathname);
2065   if(IS_ERR(name))
2066     return PTR_ERR(name);
2067
2068   error=do_path_lookup(dfd, name,
                  LOOKUP_PARENT, &nd);
...
2075   dentry=lookup_hash(&nd);
2076   error=PTR_ERR(dentry);
2077   if (!IS_ERR(dentry)) {
...
2084     error=vfs_unlink(nd.dentry->d_inode,
                                dentry);
2085     exit2:
2086     dput(dentry);
2087   }
...
```

Figure 2: The *unlink* system-call source code of Linux kernel 2.6.16

Besides, it is essential to have information about related *user-id* and *process-id* which generated the event. To achieve these kind of information, we can access other kernel object, such as *task_struct* structure of the current process. Thanks to the kernel types information got from the DWARF data coming in the kernel binary, we can access to all kernel objects, as well as extract out all the structure fields (More on this problem will be discussed in the Implementation section below).

3.2.5 Security Policy

Once we know which file is processed, to detect the intrusion we must rely on the security policy. For each protected DomU, *XenFIT* has a separate policy, and it must verify the collected events against the pre-defined policy and report the violations accordingly. At the moment the available policies are:

- ReadOnly policy: All modifications except access times will be reported for these files. The binary files in */usr/bin* and system configuration files (for example files in */etc* directory) are good candidates to have this policy.

- IgnoreAll policy: No modifications will be reported. This policy can be applied for */tmp*, for example.

- IgnoreNone policy: All modifications such as owner and access-mode will be reported. Critical files such as kernel

binaries (*/boot/**) and system files (*/sbin*) can have this policy.

- Attributes: Only modifications of ownership and access permissions will be checked. The typical files are */etc/mtab*, */etc/resolv.conf*, which can be changed dynamically at runtime by system.

With each policy file, XenFIT assigns a separate file, and this file lists all related the files and directories that are applied that policy to monitor. To be flexible, XenFIT supports wild-card paths, so the path such as */etc/httpd/** can be specified.

At run-time, *xenfitd* verifies an event sent from a domain against the corresponding policies, and if it finds the violation, a report is generated. The content of the report provides detailed information to help the administrator carry out the investigation. For example Fig.3 demonstrates our experimental: our test program named *test_xenfit* tried to overwrite remove the file */bin/ps*, which is listed in the *ReadOnly* policy. Such an action violates the defined policy, so despite the action successed, we still get the violation report from XenFIT.

```
[2006-9-31 05:34:25][small] WRITE /bin/ps,
process 4013, user-id 0
```

Figure 3: An example report on the security violation

The above report tells us that the domain had the integrity violation is *small*, and the user caused the alarm is *root*, which has user-id of *0*. This user tried to overwrite to the system file *ps*, which is widely used to show all the active processes in the system. The event is quite suspicious, and might indicate that he is an intruder trying to trojan the *ps* to hide his malicious backdoor process. This is a popular trick of user-level rootkit. The administrator should take a closer look at other log records to confirm this incident.

3.2.6 Report the Intrusion

By default XenFIT writes down the report on security violations to separate logging files for each domain, but it can also log to the system log (which is usually *syslogd*). Besides, to give the administrator more details, *xenfitd* can optionally save all the event data it gets from DomUs (including "benign" event) to separate logging files. Once the incident happens, the administrator can inspect the full logging to have better view on what the attacker has done after he broken in. However, the administrator might wish to get notified immediately on the intrusion. To achieve this goal, we suggest to use another log-file monitoring utilities, such as *swatch* ([3]). These tools can monitor the logging reports generated by *xenfitd* and signals the administrator in real-time by various ways: sends an email, plays an alarm sound, or even send a SMS to his hand-phone. The overall scheme to chain *xenfitd* and a log-file monitoring tool is described in Fig.4

3.3 XenFIT Implementation

At the moment XenFIT is only implemented in Linux. The reason is that other Os-es (like FreeBSD and NetBSD) are not ready for Xen 3.0.2, the most advanced Xen version we are working on, yet. So in this part we will present XenFIT's implementation specifically for Linux environment. The same techniques can be applied for others, however.

3.3.1 Handling Breakpoints

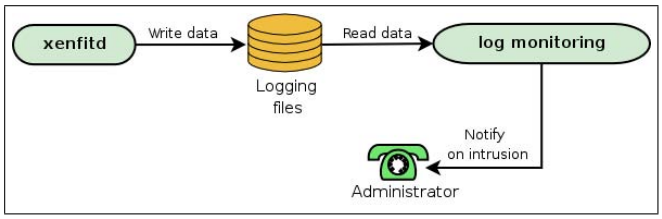

Figure 4: Tool chain scheme to report the security violation in real-time

xenfitd is a daemon process running in user-space of Dom0. It watches for the new DomU to boot, then patches the kernel memory of DomU by writing the breakpoint instructions into pre-determined addresses. To extract the address of related line of kernel source code, which is used as the address of the breakpoint, we developed a small tool that deals with DWARF information in kernel binary file. Before patching breakpoints, *xenfitd* saves the original bytes at breakpoint addresses, so it can recover them later. Because each breakpoint must be processed in a different way, each is attached with a breakpoint-handler specifically defined for it. After patching step, *xenfitd* enables the debugging mode of the DomU, then waits for the debugging events.

When a breakpoint is hit inside the kernel of DomU, the hypervisor pauses the DomU for *xenfitd* to examine and it sends a debugging event to notified *xenfitd*. Receiving the signal, *xenfitd* comes up to handle the breakpoint in the following sequences:

(1) First, *xenfitd* executes the corresponding breakpoint-handler. Overwrite the breakpoint with the original byte we saved when DomU first booted up. Then *xenfitd* forces DomU to execute the original instruction at the breakpoint by decreasing the instruction pointer [10] by 1. The reason is that the breakpoint instruction is exactly 1 byte length. Besides, *xenfitd* puts DomU into the single-step mode by turning on the *TF* flag. Finally, *xenfitd* resumes the DomU

(2) When the DomU resumes, it execute the original code at the place of the breakpoint. However, because it is in single-step mode, the control is again transferred to *xenfitd* and DomU is paused again. This time *xenfitd* recovers the breakpoint by overwriting the breakpoint instruction on the breakpoint address. After that, *xenfitd* disables the single-step mode by turning off the *TF* flag, then resumes DomU. DomU continues to execute normally until it hits another breakpoint, and the same procedure repeats.

3.3.2 Access DomU's Kernel Memory

In XenFIT architecture, we need to read and write to DomU's kernel memory. In order to access to a specific virtual address of DomU, we must first translate it into physical address. Currently Xen support several kinds of architecture: *x86_32*, *x86_32p* and *x86_64*, and each of these platforms has different schemes of paging memory. Hence XenFIT must detect the underlying hardware, and then translates the virtual memory accordingly by traversing the page table tree.

To traverse the page table tree, it is imperative to know the physical address of the page directory. In Xen, we can have the virtual control register *cr3* of each virtual CPU of the VM by getting corresponding CPU context via Xen function *xc_vcpu_getcontext()*

[10] The *EIP* register in Intel platform.

([24]). Besides, as Xen supports several architectures such as *x86*, *PAE* and *x86_64* (thus different page-table formats), XenFIT must handle the page-table accordingly to convert the virtual address to physical address.

After that, XenFIT accesses the memory of DomU by mapping the physical address with the function named *xc_map_foreign_range()* ([24]). Then it goes on reading or writing to the mapped memory [11]

3.3.3 Parsing Kernel Objects

A key challenge in inspecting the memory of VM is how to bridge the semantic gap between the raw memory and kernel objects. To do that XenFIT must be able to have a good knowledge about OS structure. And to understand in detail the layout of DomU's kernel and kernel objects, XenFIT must know exactly their address and structure.

* **Object's address**: Each object in the kernel is located at a certain memory address, and kept unchanged during its lifetime[12]. To watch the integrity of the object, it is mandatory to know its address. XenFIT finds the address of Linux kernel objects via the kernel symbol file *System.map* coming with the kernel binary.

* **Object structure**: To know only the object address is far from enough. For example, if we want to get the list of kernel modules, we can first reach the address of the first kernel module, the Linux variable *modules*. But then to get the next kernel module pointed by a field named *list.next* in the *module* structure, we must know the relative address of this field in module structure. This job is not trivial, as the *module* structure depends on kernel compiled option, and it might also change between kernel versions[13].

  To extract data about kernel-types, we leverage part of code of LKCD project ([17]). LKCD is an open source tool to save and analyze the Linux kernel dump. LKCD can parse the dump thanks to an internal library *libklib*, which extract all the information it needs from the DWARF data in the kernel binary as well as from the kernel symbol file. This library parses the kernel symbols and extracts kernel-types from debugged kernel binary, then caches the data in the memory for its tool, *lcrash* to use. Besides, *libklib* also interprets *lcrash* user command, and serves as a disassemble engine for various hardware platforms. Because of these reasons, *libklib* is a very big and complicated code, thus cannot be employed as it is for XenFIT. Another problem is that *libklib* is designed to analyze kernel dump, but not to cope with hostile data. So if somehow the attacker modifies the kernel structure in malicious way, *libklib* might crash.

  In our implementation, we only reused part of *libklib*, in which we only keeps the code that extracts and parses kernel-type information from kernel binary. The library is also hardened to resist potential attacks. Finally, our kernel parse code is around only 14000 lines of C source code, which is about 30% size of the original *libklib*.

[11]This depends on the mapped right is *PROT_READ* (read) or *PROT_WRITE* (write).

[12]Note that Linux kernel memory is never swapped out.

[13]Linux kernel never tries to keep compatible between different versions. The Linux kernel developers argue that backward compatibility might block its continuous innovation.

4. DISCUSSION

To make XenFIT work, the VM's kernel (DomU's kernel) must be compiled with kernel debugging information. That is simply the only requirement for VM's kernel, and can be easily done by enable an option at compile time.

We suppose that in our scheme, Dom0 is securely protected [14] such that the attacker has no chance to exploit our immune data, even if he totally takes over his virtual box.

As we propose XenFIT to be able to function stealthily, we should remove all the evidence of its existence. In DomU, there might be one more place the intruder can investigate to discover XenFIT's presence: the kernel binary and kernel symbol files. Fortunately, in Xen architecture DomU is run by loading the kernel from Dom0, so we will not need to have kernel binary file, together with kernel symbol files in DomU's file system.

Last but not least, all the path to the kernel memory should be prohibited, as the intruder might somehow get the root access in DomU and use that privilege to access the kernel internal and modifies the breakpoints to disable XenFIT. In order to prevent this problem, DomU's kernel should be compiled with */dev/*{*kmem,mem,port*} removed ([16]), and the ability of loading kernel module at runtime should be eliminated, too.

In the current solution, *xenfitd* has a difficulty in understanding the domain-level semantic got from DomU. For example, the event information provides who (represented by user-id), generated the event. Obviously it is better if we have the information about user-name, not user-id (which is only a number). Unfortunately the kernel is only aware of user-id, while user-name is something only available in user-land. In addition, the user-id is only meaningful in the domain that produces the event, but not in Dom0. Consequently, from Dom0 the administrator can only identify the possible attacker by his user-id, but not by his user-name. This trouble might be solved by letting Dom0 keeps the user database of DomU (the */etc/passwd* file of DomU is suitable for this purpose). However this solution is far from perfect, because it is hard to keep the database in Dom0 updated. So at the moment, we are temporarily content with the current situation, and look forward to improving it in the future.

5. EVALUATION

This section first presents the security evaluation results of XenFIT, then measures the performance impact of XenFIT.

5.1 Security Evaluation

This section presents the security evaluation results of XenFIT: the evaluation aims to test 6 popular rootkits and back-doors to see if they are detected by XenFIT. These rootkits and backdoors are installed on a protected DomU. Rootkits are categorized as user-level type if they do not modify the system kernel, and kernel-level type if they require to alter the kernel.

The user-level rootkits we choose are: *t0rn* ([12]), *lrk5* ([18]), *dica* ([8]) and *SAdoor* ([5]) These rootkits infect quite a few critical files the system, among of them are */usr/bin/*{*ls,netstat,ps*}.

To detect these rootkits, we put the above system files in the *ReadOnly* policy. With this setting, XenFIT is able to detect immediately the illegal modifications to the above binaries and alerts the administrator in real-time.

Regarding the kernel-level rootkit, we choose *knark* ([11]) and *adore-ng* ([11]). These rootkits infect the kernel and open hidden

[14]Thus Dom0 might be considered to belong to the Trusted Computing Base (TCB), the core component required to enforce the system security.

ports to wait for the unauthorized connections from outside. Because these malware do not modify any files on the file-system, XenFIT cannot detect them. However, XenFIT is still able to spot the intrusion if the attacker penetrates the system remotely and modifies a protected file.

5.2 Performance Evaluation

As XenFIT only intercepts some file-system related system-calls to gather information, the impact on the system only focuses on the file-system activities. Therefore our performance evaluation only carries out on the file-system benchmarks.

To measure the performance penalty, we choose a classical benchmark: decompress the Linux kernel. The reason we take this benchmark because Linux kernel contains a lot of data, and the decompress process creates and removes a great number of files and directories: for example unzipping the kernel 2.6.16 generates more than 27000 files and directories, including temporary data.

The benchmark decompresses Linux kernel 2.6.16 with the command "*tar xjvf linux-2.6.16.tar.bz2*". We run this test 10 times on the native kernel and the kernel with XenFIT installed, then gets the average results. The configuration of the domains in the benchmarks are as below:
Dom0: Memory: 384MB RAM, CPU: Pentium3 600MHZ, IDE HDD: 40GB, NIC: 100Mbps
DomU: Memory: 128MB RAM, file-backed swap partition: 512MB, file-backed root partition: 2GB

All the domains in the tests run Linux Ubuntu distribution (version Breezy Badger), with the latest updates.

Table 2 shows the result of the benchmarks - all the numbers are in seconds:

	Native VM	VM protected by XenFIT
real	1m57.561s	11m50.680s
user	1m23.760s	9m35.540s
sys	0m15.382s	1m43.470s

Table 2: Measurements on unzip the Linux kernel

We can see that the kernel overhead costs around 5.8 times more than the native kernel, and the overall overhead is around 607%. XenFIT's high performance impact can be explained by a lot of hyperswitchs it must execute when handling breakpoints.

6. RELATED WORKS

Our work is a combination of two areas: dynamically monitoring system and HIDS. Our work is inspired by the work of K.Arigos et.al in [2], but while this paper proposed to use breakpoints for honeypot purpose with Xen, but we exploit the idea for a HIDS. The way we handle debugging events is also completely different from [2]: their authors pushed the security policy into the Xen layer (via an add-in hypercall) and let the hypervisor analyzes the policy there. To do that, the authors made quite a big modification to Xen layer (around 2700 lines of code), which they also mentioned it as a TCB. We would argue that it is not desired to make such a major change to such an important component, because it makes the whole system less stable, as well as increase the maintenance cost. In our solution, *XenFIT* makes absolutely no modification to the hypervisor as well as to the protected DomU.

Besides, in [2] all the security policy analyzes are done inside the hypervisor. Thus we suspect that some modification to the policy or security engine would require to modify and recompile the hypervisor and the system must be reboot for the change to take effect. Meanwhile, our solution puts all the policy and decision making process on policy in a user-space process in Dom0, and that makes it very easy to modify policy or even *xenfitd* itself without having to reboot the whole system.

Reliable access to data is a prerequisite for all the computer systems. Data might be corrupted due to hardware or software malfunction, malicious activities or inadvertent. The most popular integrity assurance mechanism occur at file-system level: all the on-disk file-systems perform off-line consistency checking with user-space programs like *fsck*, which is usually run at start-up to fix unclean shutdown. Recent advanced file-systems support journaling feature to avoid the unnecessary checkup. However this class of integrity checking purely aims to fix the inconsistency errors, but not to cope with security incidents, so they are not the research topic of this paper.

Regarding the Intrusion Detection System (IDS), there are few ways to classify them. One is to consider the source of data used for the detection: if the tool uses the information derived from the local host, it is called host-based IDS, or HIDS in short. In contrast if it captures data from the local network, it is of network-based IDS type, or NIDS.

The HIDS resides on a system and provides protection for a that particular computer. The HIDS relies on tracking the operating system activities. Meanwhile the NIDS gathers and analyzes the network packets of the local network to detect the malicious traffic, in order to protect all the machines in the same network segment. The most popular NIDS in the academia community is the open-source Snort IDS ([23]).

Our XenFIT solution is made to detect the tampering on the file-system level, so it is of the HIDS type. Hence in this paper we only focus on the file-system level HIDS solutions together with their problems and discuss the solutions to improve them.

Because file-system integrity is important to the overall security, it is not surprised that there are many paper-works and resulted tools on the topic. In general we can categorize them into 2 kinds: off-line FIT and on-line FIT.

6.1 Off-line FIT

The off-line FIT must be scheduled to periodically run to check for the problems. The most well-known tool of this kind is Tripwire: Tripwire relies on a base-line database of file-system checksum, which is generated when it is first installed. At run-time, Tripwire verifies the current system and compares with the checksum in the database, then reports the violations according to the predefined security policy. Other kinds of FIT more or less follow the scheme of Tripwire are AIDE, Samhain and Osiris.

Unfortunately all of these softwares suffers from 6 problems of FIT we discussed above. In attempt to address their weaknesses, they try to propose and implement special methods as followings:

- All the off-line FIT is periodically run to inspect the malicious changes to the file-system, so they all have the "delay detection" trouble (problem 1). This is the designed scheme, so there is no perfect solution for it. To mitigate the problem, Samhain suggests to run in daemon mode to minimize the security window between run-times.

- To avoid the suspect of the attacker (especially in the case he has local access), Samhain put in a lot of effort to hide its presence. Samhain proposed few weird methods to hide its presence: the database can be append at the end of a JPEG file, the configuration file is steganographically hidden in a postcript image file, while the binary tool (named *samhain*) and boot script file-name are changed to hide from suspicious eyes. Last but not least, Samhain even provides a tool

(*samhain.pk*) to pack the executable, so it is hard to reverse engineer the binary ([21]).

- To avoid the possible compromise on the database and to improve the scalability, Osiris and Samhain use the client/server architecture, and save the database and policies on the server (with the assumption that the server is secure). Osiris does not store the scan data on the scanned host, the scanning result is instead sent back to the management host (server), and the comparison is done there. In contrast Samhain pushes the database to the client and verifies the integrity there.

- To mitigate the risk of tampering the binary tool itself, Osiris and Samhain advise to keep the scan agent software on read-only media (CDROM), but it becomes cumbersome if the administrator wants to upgrade the software. Other than that, Samhain offers a couple of self-integrity checks including signed data and configuration files, a key compiled into the executable, and the ability to hide itself from the system process list. Osiris makes use of a run-time session key that acts as a mean to authenticate to the management host as well as to detect tampering.

- Recently Hal Pomeranz ([14]) suggested another method to secure the database and the binary tool: put them on the secure sever, and only download them and run the verified process when needed on the client. Accordingly, the Tripwire/AIDE database and FIT are saved on the server, and periodically pushed to the client to run via a secure channel (SSH, specifically). The author recommends to makes the whole operation stealth by rename the binary tool (thus the FIT process name), so if the attacker is present in the system, he is not aware of the checking process.

However, no proposed method can fool the skillful attacker. Sooner or later he is able to figure out that the system is running a FIT, and if he somehow gets the root access, he can shutdown the tool. After that the attacker can do anything he wishes on the system without anybody's awareness.

One more noticeable problem of the client/server architecture adopted by the advanced tools like Osiris and Samhain is: the server must exposed to the network, so it could be the target of the intruder. The attacker might exploit the server remotely first to disable it, then comfortably penetrates the client.

6.2 On-line FIT

In contrast with off-line solution, an on-line FIT is able to monitor and react on the security violations in real-time. The strategy is to stay in the critical path of file-system activities, such as *open/read/write/rename/unlink*,... All of these activities can disclose the integrity violation in real-time. However this method comes with some trade-offs: to inspect the low-level activities, the tool must usually work at OS kernel level, which requires to patch the system kernel for it to work. This may lead to conflict with other third-party patches approved by the same machine, especially because the kernel used by different OS vendors are quite different. Moreover, not all the systems would allow to push new code into their kernel.

One example of online FIT is I3FS ([13]): I3FS is a file-system layer that intercepts and tracks down the related VFS functions, so it can detect when a specific file is changed in real-time. Nevertheless, I3FS requires to change the kernel of the protected machine, which is not always possible in production system.

Another popular on-line type tool is SNARE, a complete software suite that can be used to monitor system events. SNARE employs the client/server architecture, in which the client collects and send the system-call audit events to the server, and the sever carries out the analyzing process. Since the event log contains a great deal of information about the system activities, SNARE is also proposed to be a HIDS solution.

Unfortunately, we are not convinced that SNARE is suitable as the integrity detection tool, because of some following drawbacks:

- SNARE is not designed to evade the attacker: SNARE is vulnerable to problem (6) discussed above. For example on Linux system, the attacker could detect SNARE by looking for the agent process (named *auditd*), or checks if the file */proc/audit* is existed. In addition, SNARE can also be shutdown by the attacker if he gains the root access: in case the agent process is killed, no more events can be reported to the server. This is the same problem experienced by the above off-line FIT.

- SNARE is applied to the client as a kernel patch (not as a kernel module). That might complicates the usage, because the kernel must be recompiled, then rebooted to make SNARE work, which is not always desired on a production system.

Another reason we do not prefer the solution is that SNARE is proposed as a patch, and this may force the SNARE developers to release separate patches for different kernel versions, especially because the kernel API is likely changed any time. That demand can significantly slow down the support for the newer kernel. For example at the time of this writing, the latest Linux kernel version is 2.6.17, but the most updated patch of SNARE is only for the kernel 2.6.11.7, and unsurprisingly this patch is not cleanly applied on 2.6.17 kernel.

7. CONCLUSIONS

This paper proposes the design and implementation of XenFIT solution to eliminate some problems of current file-system integrity solutions. XenFIT introduces some unique features: real-time monitoring, centralized policy, no need the base-line database (so the administration overhead is much more mitigated), easy to deploy, and highly tamper resistant. We propose to use XenFIT in a chain tool with another log-file monitoring tools to notify about the intrusion in real-time. Moreover, if being installed in a strict manner, XenFIT is stealthier, harder to detect even with privileged user. All of these advantages make XenFIT a valuable HIDS solution.

As Xen will be available in the mainline Linux kernel very soon, we believe that this solution will benefit everybody. For the time being, XenFIT only works for Linux-based virtual domains. We plan to provide support for other Os-es such as FreeBSD, NetBSD once these ports are working stably on Xen.

8. REFERENCES

[1] N. A.Kamble, J. Nakajima, and A. K.Mallick. Evolution in kernel debugging using hardware virtualization with xen. In *Proceedings of the 2006 Ottawa Linux Symposium*, Ottawa, Canada, July 2006.

[2] K. Asrigo, L. Litty, , and D. Lie. Virtual machine-based honeypot monitoring. In *Proceedings of the 2nd international conference on Virtual Execution Environments*, New York, NY, USA, June 2006. ACM Press.

[3] T. Atkins. SWATCH: The Simple WATCHer of Logfiles. `http://swatch.sourceforge.net/`, July 2004.

[4] CERT Coordination Center. CERT/CC Overview Incident and Vulnerability Trends. Technical report, Carnegie Mellon Software Engineering Institute, May 2003.

[5] CMN. SAdoor: A non listening remote shell and execution server. http://cmn.listprojects.darklab.org/, 2002.

[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 *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[7] DWARF Workgroup. DWARF Debugging Format Standard. http://dwarf.freestandards.org/Home.php, January 2006.

[8] R. Hock. Dica rootkit. http://packetstormsecurity.nl/UNIX/penetration/rootkits/dica.tgz, 2002.

[9] Intersect Alliance. System iNtrusion Analysis and Reporting Environment. http://www.intersectalliance.com/projects/Snare/, January 2005.

[10] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[11] T. Miller. Analysis of the Knark rootkit. www.ossec.net/rootkits/studies/knark.txt, 2001.

[12] T. Miller. Analysis of the T0rn rootkit. http://www.sans.org/y2k/t0rn.htm, 2002.

[13] S. Patil, A. Kashyap, G. Sivathanu, and E. Zadok. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, pages 69–79, Atlanta, GA, November 2004.

[14] H. Pomeranz. File Integrity Assessment via SSH. http://www.samag.com/documents/s=9950/sam0602a/0602a.htm, February 2006.

[15] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick. Xen 3.0 and the art of virtualization. In *Proceedings of the 2005 Ottawa Linux Symposium*, Ottawa, Canada, July 2005.

[16] sd. Linux on-the-fly kernel patching. http://www.phrack.org/show.php?p=58&a=7, July 2002.

[17] SGI Inc. LKCD - Linux Kernel Crash Dump. http://lkcd.sf.net, April 2006.

[18] L. Somer. Linux Rootkit 5. http://packetstormsecurity.nl/UNIX/penetration/rootkits/lrk5.src.tar.gz, 2000.

[19] The AIDE team. AIDE: Advanced Intrusion Detection Environment. http://sourceforge.net/projects/aide, November 2005.

[20] The Osiris team. Osiris host integrity monitoring. http://www.hostintegrity.com/osiris/, September 2005.

[21] The Samhain Labs. Samhain manual. http://la-samhna.de/samhain/manual/index.html, 2004.

[22] The Samhain Labs. The SAMHAIN file integrity/intrusion detection system. http://la-samhna.de/samhain/, January 2006.

[23] The Snort team. Snort - the de-facto standard for intrusion detection/prevention. http://www.snort.org, January 2006.

[24] Xen project. Xen interface manual. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/interface/interface.html, August 2006.