Proceedings of the 2004 IEEE
Workshop on Information Assurance and Security
United States Military Academy, West Point, NY, 10–11 June 2004

A Mechanism for Automatic Digital Evidence Collection on High-Interaction Honeypots

Martim d'Orey Posser de Andrade Carbone, Paulo Lício de Geus

Abstract—
Honeypots are computational resources whose value resides in being probed, attacked or compromised by invaders. This makes it possible to obtain information about their methods, tools and motivations. On high-interaction honeypots this is done, among other ways, by collecting digital evidence. This collection is traditionally done manually and statically, demanding time and not always generating good results.

In this paper, we describe an automatic, dynamic and transparent mechanism for collecting digital evidence from the filesystem of honeypots, eliminating the flaws found in the traditional methods. The mechanism consists of two modules: an interceptor module, that intercepts some preselected system calls on the honeypot and transmits the argument data to the honeynet; and a receiver module, that captures the transmitted data and reconstructs on the honeywall the evidence produced by an intruder during an invasion. A prototype based on the mechanism was implemented and tested in real intrusion situations. The mechanism's behavior in one of these situations is also described, followed by an analysis of the results.

I. Introduction

Ever since its proposal in the early 90's, the concept of *honeypot* has become increasingly popular among members of the IS community, being now recognized as a valuable tool in fighting blackhats [1], [2].

A honeypot can be defined as *a computational resource whose value resides in being probed, attacked or compromised by invaders* [3]. This definition is very broad, and can encompass anything from large computational environments to simple files (as proposed in [4]). In this paper, nevertheless, we shall consider honeypots as being physical computers. The value of a honeypot derives from the fact that it is not used by legitimate users in daily production activities. This makes all the activity taking place on the honeypot naturally suspicious, greatly reducing the number of false positives.

Although very simple, this idea is extremely powerful and suitable for a wide variety of goals: attack signature capture, study of the methods and psychology of blackhats, study of malicious tools, among others [4], [5], [6], [7].

Just as a honeypot can have different goals, there are also different types of honeypots, suited for different types of data one might want to collect [2], [8]. *Low-interaction* honeypots have a low level of interaction with the attacker, limiting the scope of his actions [9]. A port monitor that works by detecting and logging remote attacks is a classical example: the interaction between the attacker and the honeypot is kept restricted to the TCP/IP stack.

On the other end, *high-interaction* honeypots have a much higher level of interaction with the attacker, giving him access to the whole operating system. It's quite obvious that this second approach makes it possible to obtain a much larger and richer quantity of data, if compared to the low-interaction alternative. This data can have different origins: captured network traffic, operating system logs, keystrokes, RAM content, filesystem content, among others. These data are examples of *digital evidence*, object of study in computer forensics [10], and constitute the main source of information in the study of honeypots.

Digital evidence extraction is not a trivial procedure [11], specially when it involves honeypots. In this case, it becomes necessary to create methods that are not only efficient, but also invisible to the blackhat, who must believe he has actually compromised a production host.

Certain types of digital evidence, however, can be extracted more transparently than others. Network traffic, for example, can be captured passively through a simple traffic logger, without risking alerting the blackhat. There are other kinds of digital evidence, however, that require a more active extraction, for they are located inside the honeypot, in places such as the RAM or the filesystem. The latter is considered by forensic analysts to be the main source of digital evidence on a system [11], and thus deserves focus in the study of honeypots. This paper focuses on the evidence left inside a honeypot's filesystem, running the Linux operating system (kernel version 2.4).

Traditionally, digital evidence extraction from a honeypot's filesystem is done statically and manually, demanding active human participation. The most common technique consists of creating an image of the filesystem through the use of a copying tool, such as the Linux *dd* program . This image is then moved to a secure machine, where it is submitted to a forensic analysis. Sometimes, integrity checkers (such as *tripwire*) are used to determine the modifications inflicted upon the filesystem's data, thus determining what changes were made by the intruder [1], [12].

This methodology has many drawbacks. Not only it is manual and slow, but in many cases it also demands that the honeypot be temporarily deactivated for the image to be created. This has a negative impact on the quantity of evidence collected, besides being potentially alertive to the intruders. Besides, this method operates statically, meaning that any evidence created and erased between two checks won't be detected.

This paper describes a mechanism for automatic, dynamic and transparent digital evidence collection from honeypots' filesystems, eliminating the drawbacks found in the traditional methodologies.

Initially, in Section II, the honeynet in which the honeypot is deployed will be described. Section III will describe the evidence collection mechanism and Section IV will expose the empirical results obtained with a prototype of the mechanism in a real intrusion situation. Finally, Section V will conclude the paper with the final remarks and possible improvements and extensions to the mechanism.

II. The Environment

Choosing the environment in which a honeypot will be located is an important step in its deployment, since it directly affects the nature, quantity and quality of the collected data [2]. The honeypot studied in this paper resides in a *GenII honeynet* [13], like the one illustrated in Figure 1.

A honeynet can be defined as a well delimited and highly controlled network environment containing one or more high-interaction honeypots [14]. GenII honeynets consist of an isolated network segment where a *honeywall* machine mediates the network traffic going in and out of the honeypot.

This machine basically works as a reverse firewall, capturing and analyzing the incoming traffic (malicious traffic, supposedly), and actively controlling outgoing traffic (supposedly generated by an intruder).

Incoming traffic monitoring is done through the *Tcpdump* [15] packet capturer and the *Snort* [16] intrusion detection system. The latter analyses the traffic in search for known attack signatures and generates a warning if it finds an occurrence. On the other hand, outgoing traffic must be actively controlled in order to prevent the intruder from using the honeypot as a launch base to attack other hosts located outside the network perimeter. This control makes use of several techniques that are outside the scope of this paper (further details can be found in [2]).

The GenII model makes use of another interesting technique: transparent bridging [17], activated on the *honeywall*. This technique works by relaying network frames between two of its NICs (Figure 1), without making any modifications to the content of the frames. This transparency makes the honeywall practically invisible to the intruders, a very important quality in honeypot solutions.

It is also worth mentioning that the honeywall has a third NIC, connected to the production network (Figure 1). Its

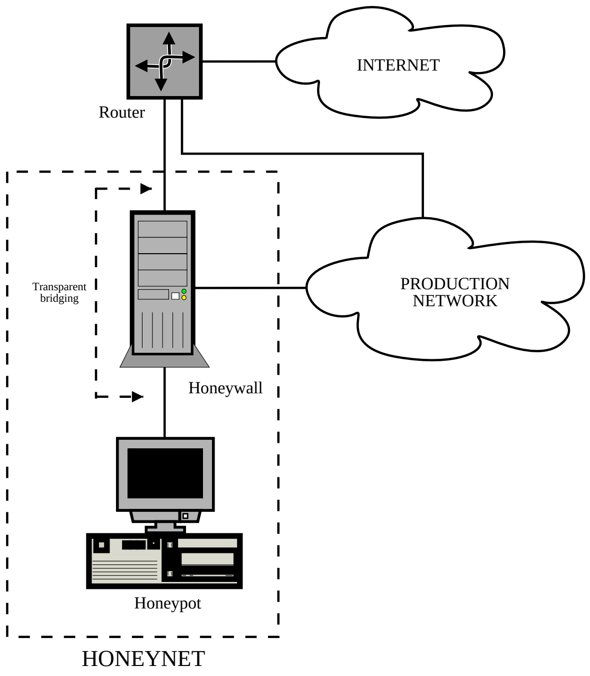

Fig. 1. The honeynet topology

goal is to simplify the honeywall's administration and the access to the collected data by making it accessible from the production network. This configuration does not break the isolation requirement established before, because the other two NICs don't have IP addresses associated with them (because of the transparent bridging technique), eliminating the risk of having an intruder breaking into the honeywall and compromising the security of the production environment.

III. Description of the Mechanism

The mechanism described in this paper is a distributed application composed by two main modules: an interceptor module, which operates on the honeypot, and a receiver module, which operates on the honeywall.

The evidence collection itself is done by the interceptor module, that dynamically intercepts the kernel system calls that somehow alter the honeypot's filesystem and transmits the context information of each intercepted call to the honeywall, a secure host to which the intruder has no access. This transmission is done in a covert manner, invisible to an intruder executing a traffic capture tool, because it communicates directly with the NIC driver, never interacting with the kernel networking subsystem (further details can be found in Section III-B.4).

On the honeywall, the data sent by the interceptor module is passively captured (since the honeywall operates using transparent bridging) by the receiver module, and logged locally in a log file. The contents of this file are

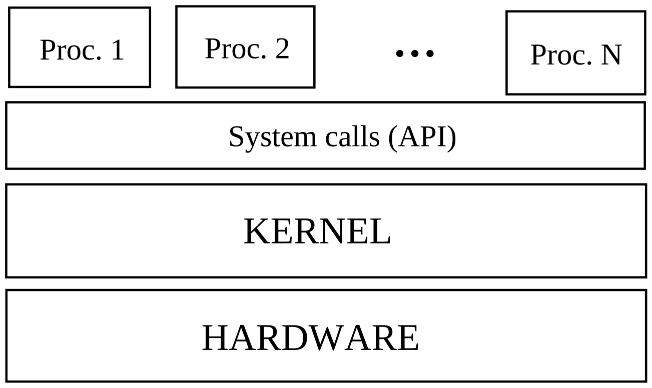

Fig. 2. System calls

fed into an evidence processor, which recovers the context information of each call intercepted on the honeypot and reproduces them sequentially. As a result, an evidence tree containing all the evidence left by an intruder on the honeypot's filesystem, from the beginning of an invasion to its end, is automatically generated in the honeywall's filesystem.

The following sections will detail the internal structure of the interceptor and receiver modules presented above, after a brief introduction to Linux system calls.

A. System calls

Linux uses two distinct CPU modes: user mode and supervisor mode [18]. In the former, the processor operates under restrictions and cannot execute certain privileged instructions that directly access the computer's hardware. This is done in order to prevent malicious (or badly programmed) code from causing damage to the system. In the latter, the processor is under no restrictions, and can execute all the instructions, including the ones that directly access the hardware.

In Linux, the kernel is executed in supervisor mode and user processes are executed in user mode. That way, the presence of an *application programming interface* (API) between both becomes necessary, in order for the processes to be able to request to the kernel the accomplishment of privileged tasks, such as writing a file in the disk, or sending a packet to the network. *System calls* provide this interface, making it possible for the kernel to serve the processes' needs (Figure 2) [18].

It has already been mentioned that every hardware access by a user process must necessarily invoke a system call, because only kernel code has the necessary privileges to perform such action. This fact makes system call monitoring an attractive technique considering the goals of this work. This technique will be further detailed in Section III-B.

B. Interceptor Module

The interceptor module was implemented as a *linux kernel module* (LKM) [19], that is, an object code dynamically loaded into kernel space. This is necessary for the module to be able to manipulate the kernel's data structures, including the system call table.

Once the LKM is loaded into kernel space, it becomes necessary to disguise its presence, preventing it from being detected by intruders. In practical terms, this means one must not let the module appear in the listing provided by the *lsmod* command. This measure can be implemented through the loading of another LKM that removes the node associated with the interceptor module from the linked list of loaded modules kept in kernel space.

The interceptor module is an extension of the one implemented in *Sebek* (version 2.0.1)[20], a honeypot tool that monitors the *sys_read* system call in order to circumvent the cryptographic protection provided by the *SSH* and *SCP* tools, commonly used by intruders to encrypt their communication with the compromised host. The interceptor module extends Sebek because it intercepts not only one, but all the system calls involved directly or indirectly in the modification of the filesystem. These are listed below:

- Hard-link creation: *sys_link()*;
- Symbolic link creation: *sys_symlink()*;
- File creation and opening: *sys_open(), sys_creat()*;
- Directory creation: *sys_mkdir()*;
- File writing: *sys_write(), sys_pwrite()*;
- Memory mapping: *sys_mmap()*;
- File closing: *sys_close()*;
- File removal: *sys_unlink()*;
- Directory removal: *sys_rmdir()*;
- Renaming of files and directories: *sys_rename()*;
- File truncation: *sys_truncate(), sys_ftruncate()*;
- Permissions management: *sys_chmod(), sys_fchmod()*;
- Ownership management: *sys_chown(), sys_lchown(), sys_chgrp()*.

For the sake of simplicity, only the system calls *sys_open*, *sys_close*, *sys_write*, *sys_rename* and *sys_mkdir* calls are being intercepted in the prototype. With regard to the *sys_open* call, only the act of file creation is being considered, leaving aside the act of opening an existent file.

The operation of the module can be divided in four sequential steps, detailed in the next sections. The *sys_open* call interceptor function will be used as an example of the implementation.

B.1 Step 1: Interceptor functions installation

System call interception is a widely used and well studied technique [21], [22], [20], [23]. It consists of the alteration of the pointers stored in the system call table, in order

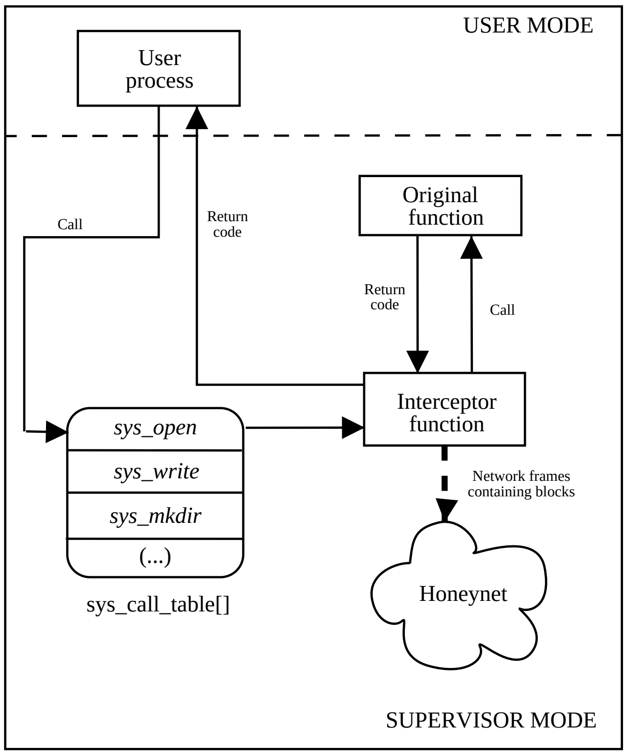

Fig. 3. *Sys_open* system call interception

to execute, before the original code, an interceptor code (Figure 3). The implementation of this technique for the *sys_open* call is illustrated below:

```
int (*old_open)(const char *pathname, int flags, mode_t mode);
asmlinkage int new_open
(const char *pathname, int flags, mode_t mode);

asmlinkage int new_open
(const char *pathname, int flags, mode_t mode)
{
  [...]
}

int init_module(void)
{
  (unsigned long *)old_open = sys_call_table[__NR_open];
  sys_call_table[__NR_open] = (unsigned long *)new_open;
}
```

In the code above, inside the module's main function (*init_module()*), the system call table entry associated with the *sys_open* call is overwritten with the address of the interceptor function *new_open*. Before that, however, the old value stored in this position is transfered to the variable *old_open*, to be used in the invocation of the original function.

B.2 Step 2: Original call invocation and related checks

In the interceptor functions, the first task is the invocation of the original function with the parameters given by the process, as can be seen below:

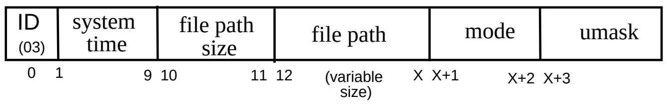

Fig. 4. Block structure for the *sys_open* call

```
  res = old_open(pathname, flags, mode);
  if((res != -1) && ((flags & O_CREAT) != 0)){
    [...]
  }
  return res;
}
```

This invocation is done right in the beginning of the function so that the interceptor code knows beforehand if the original function will be successfully executed, before it executes the rest of the intercepting code. In case it is successful, the remaining of the intercepting code is executed; if not, the interceptor function exits and the return value (stored in the variable *res*) is returned to the invoking process (Figure 3). In the example above, the file opening flags are also checked. The remaining of the interceptor code is executed only if the file referenced by *pathname* is being created (`O_CREAT`). This is because the prototype will only consider file creation, ignoring the opening of pre-existent files. For the other system calls, other verifications are made.

B.3 Step 3: Additional data retrieval and block assembly

In this step, it is necessary to retrieve all the necessary data to create a *block*. A block is a logical data unit that contains all the data necessary for a system call to be reproduced on the honeywall. Therefore, not only it must contain the parameters given to the call, but also process and system context information.

For the *sys_open* system call, these data include the process' current working directory (to be used in the composition of the absolute file path), system time, the process' permission mask, the name of the file given as a parameter and the creation permissions. Another important information is the identifier of the system call. All intercepted calls are associated to an integer that uniquely identifies them in a certain block. This identifier is used by the receiver module to determine the type of system call to which the block corresponds, so it can interpret the block correctly. All the blocks associated with the *sys_open* call, for example, have an ID number of **03**, and other blocks associated with other system calls have different IDs.

After the retrieval of the data, the block is assembled. The structure of the block representing an occurrence of a *sys_open* call is shown in Figure 4.

B.4 Step 4: Ethernet frame creation and transmission

After the assembly of the blocks, they are encapsulated inside Ethernet frames and sent to the network. In this sending, Ethernet, IP and UDP are used as data-link, network and transport protocols, respectively.

The frame generation is done by the *gen_pkt* function, used in Sebek for the same purpose. This function initially allocates a *socket buffer* [19] (kernel structure used as a storage place for network frames), and creates the UDP, IP and Ethernet headers. The block is then copied to the socket buffer's payload area. At first, the content of the fields of the Ethernet, IP and UDP headers is irrelevant, since the honeywall's NICs have no IP addresses associated, and the network frames must be captured passively. There must be, however, some method to uniquely identify the frames generated by the interceptor module, so that the receiver module can identify them amidst the universe of frames passing through the honeynet. In this work, the *Destination port* field of the UDP header was chosen for this matter, with the value of *1666*.

Generally, there is a biunivocal relation between blocks and network frames, but there is an exception: the *sys_write* call. Not rarely, this call manipulates large volumes of data, making it necessary to divide the block into smaller frames (4KB was the value stipulated). After the creation of the Ethernet frame, the only remaining task is to send it to the network, transferring it to the appropriate device driver. This sending is done through the kernel function *hard_start_xmit*.

Because it is self-sufficient in the generation of the Ethernet frames, and talks directly to the NICs device driver (the same way Sebek does), the sending mechanism goes around all the hooks existent in the kernel networking subsystem (*netfilter*). This makes the transmission invisible to an intruder executing a packet capturer (such as *Tcpdump*) on the honeypot.

C. Receiver Module

The receiver module, shown in Figure 5, operates on the honeywall as a user process. Unlike the interceptor, the receiver module is executed on a secure machine, and therefore does not need to be hidden.

It works by passively capturing the network frames that pass through the honeynet, followed by the verification of the UDP destination port, in order to identify the ones sent by the interceptor module. The blocks are extracted from the identified frames and stored sequentially in a *block log*. This log contains nothing less than the complete chronology of the alterations inflicted upon the honeypot's filesystem, from its activation to the most recent instant. With these data at hand, a great myriad of possible types of evidence reconstruction opens up. One may want, for instance, to obtain all the files created by an intruder during an invasion. It could also generate a report containing all the files altered by the intruder, with the content of the alteration. There are some even more exciting possibilities: the highly temporal granularity of the data allows the creation of a timeline, containing the complete evolution of a set of files (and, why not, the entire filesystem). However, this last possibility demands the existence of a local copy of the honeypot's original filesystem (before it was altered by the intruder), to be used as an initial state in the evidence reconstruction.

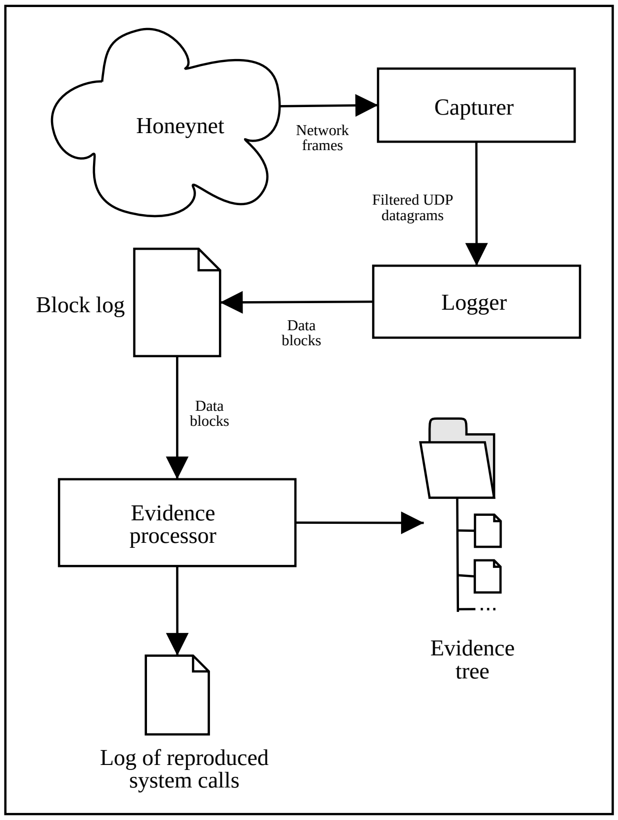

Fig. 5. The Receiver Module

In all the situations mentioned above, the evidence reconstruction is done by an *evidence processor*, a part of the receiver module. Its operation varies depending on the type of the reconstitution, but in all cases, the basic event chronology is the same: the received blocks are sequentially analyzed, its contents interpreted and the system calls represented are reproduced locally. This reproduction relies on the identifier field of each block, using it to determine the system call to which each block refers. Next, the other fields of the block are passed as arguments to a specific function that processes them and executes the system call.

All the evidence reconstructed by the evidence processor is confined to a directory hierarchy (mirroring the honeypot's filesystem hierarchy), whose root is specified in the receiver module's configuration. Therefore, this root directory contains an *evidence tree*, within which are all the evidence (files and directories) created by the intruder on the honeypot and reconstructed by the evidence processor on the honeywall.

Besides the evidence tree, the evidence processor also generates a complete record of all the reproduced system calls. The importance of this record becomes clear if we consider the situation in which the intruder removes files or directories. These items would not be included in the evidence tree and, without this record, we would never notice the removal.

Just as the interceptor module, the prototype's receiver module deals only with the *sys_open*, *sys_close*, *sys_write*, *sys_rename* and *sys_mkdir* calls. Thus, the prototype works only with the creation and renaming of files and directories, leaving aside more complex types of evidence reconstruction.

TABLE I
Histogram of the blocks received during the invasion

Syscall	**Blocks**	**%**	**Bytes**	**%**
sys_write	6792	96,38%	6757481	99,68%
sys_open	86	1,22%	11192	0,16%
sys_close	86	1,22%	9314	0,14%
sys_rename	22	0,31%	1152	0,02%
sys_mkdir	12	0,17%	337	<0,01%
Total	7047	100%	6779476	100%

IV. Experimental Results

The prototype was submitted to an empirical validation process, through the analysis of its behavior in live intrusion situations. The interceptor and receiver modules were properly installed and activated on the honeypot (running kernel version 2.4.19) and the honeywall (ditto), respectively. Both machines were confined in a honeynet such as the one described in Section II. The Sebek's keylogger was also installed in the honeypot, capturing the intruder's keystrokes in kernel space and covertly sending them through the network, so as to be captured by the honeywall.

Twenty days after its deployment, the honeypot was attacked and compromised. The intruder remotely exploited a vulnerability in the honeypot's FTP server, and obtained a shell prompt with root privileges. The occurrence of the attack was immediately notified by Snort:

```
1/04-17:24:49.684235  [**] [1:1630:5] FTP EXPLOIT CWD overflow [**]
[Classification: Attempted Administrator Privilege Gain] [Priority: 1]
```

A few seconds after the exploitation, the traffic logger loaded on the honeywall started capturing UDP datagrams originated from the keylogger, containing the shell commands typed by the intruder. The commands follow:

```
# unset $HISTFILE
# id
# ls
# wget
# wget ftp://whitez:whiter0x@217.215.111.13/www/vmkit2k.tar.gz
# tar xvzf vmkit2k.tar.gz
# cd vmkit2k
# ./install spike68
```

The intruder started by removing the environment variable $HISTFILE, hoping that the commands he typed would not be logged by the shell. He then executed the *id* command, confirming that he indeed had root privileges. Following that, he downloaded the file `vmkit2k.tar.gz` from a remote server with the *wget* tool, uncompressed it, entered the newly created directory `vmkit2k` and executed the *install* program.

After this sequence of commands, the intruder tried to connect to the port 15000 TCP, where he presumably had installed a backdoor. After checking it, he disconnected from the honeypot.

As soon as the intruder started transferring the file `vmkit2k.tar.gz`, the receiver module began capturing network frames sent by the interceptor module. The first received frame follows:

```
00 02 b3 00 00 00 01 00  00 0f c4 a9 08 00 45 0d   ..............E.
00 49 00 50 00 00 01 11  21 2c 0A 0A 0A 01 0A 0A   .I.P....!,......
0A 02 06 82 06 82 00 35  29 06 03 00 87 fd a7 3f   .......5)......?
11 b6 09 00  17 00 2f 76 61 72 2f 66 74 70 2f 76   ....../var/ftp/v
6d 6b 69 74  32 6b 2e 74 61 72 2e 67 7a b6 01 12   mkit2k.tar.gz...
00 00 00                                           ...
```

The first payload byte (following the initial 42 bytes containing the Ethernet, IP and UDP headers) is **03**. This byte represents the *id* field of the encapsulated block; in this case, the *sys_open* system call *id* number (Figure 4). As said before, the interceptor module implemented in the prototype deals only with file creation, thus the frame illustrated above contains a block referring to the creation of the `/var/ftp/vmkit2k.tar.gz` file on the honeypot.

After this initial block, many others followed, referring to the creation, writing and renaming of files and directories. The block receival statistics are shown in Table I

The received blocks were stored sequentially in the block log, totalizing 6.46 MB of data. These blocks were then input to the evidence processor, that in turn generated an evidence tree inside the directory `/EVIDENCE` of the honeywall's filesystem. This tree contains every directory and file created by the intruder on the honeypot, organized in a way that mirrors the honeypot's filesystem. This means that the directory `/EVIDENCE/var/ftp` of the honeywall contains the evidence created in the directory `/var/ftp` of the honeypot:

```
root@honeywall / $ ls -l /EVIDENCE/var/ftp
total 1444
drwxr-xr-x    4 root root        4096 Nov  7 20:09 vmkit2k
-rw-r--r--    1 root root     1467505 Nov  7 20:09 vmkit2k.tar.gz
```

As expected, the `vmkit2k.tar.gz` file downloaded by the intruder was correctly rebuilt. Its integrity and every other reconstituted file's integrity was confirmed through the use of MD5 checksums.

Many other files and directories were rebuilt by the evidence processor, providing some very interesting data concerning the tool installed by the intruder. For example, inside the directories `/usr/bin`, `/usr/sbin` e `/sbin` of the honeypot's filesystem, many system binaries, such as `netstat`, `sshd`, `ps`, `ifconfig`, `syslogd`, among others, were replaced by new binaries named equally, suggesting the possibility of a rootkit. The evidence analysis that followed confirmed this hypothesis, even though we were unable to identify it precisely (there is evidence suggesting it is a variant of the Illogic rootkit). A listing of the files replaced by the rootkit in one of the system's directories (`/usr/bin`) follows:

```
root@honeywall / $ ls -alR /EVIDENCE/usr/bin
FS1/usr/bin:
total 1100
drwxr-xr-x    2 root root        4096 Mar  4 16:26 .
drwxr-xr-x    4 root root        4096 Mar  2 17:38 ..
-rwxr-xr-x    1 root root       88064 Mar  4 16:26 crontab
-rwxr-xr-x    1 root root      163083 Mar  4 16:26 dir
-rwxr-xr-x    1 root root      123540 Mar  4 16:26 du
-rwxr-xr-x    1 root root      213195 Mar  4 16:26 find
-rwxr-xr-x    1 root root       21765 Mar  4 16:26 killall
-rwxr-xr-x    1 root root       22535 Mar  4 16:26 pstree
-rwxr-xr-x    1 root root      205288 Mar  4 16:26 ssh2d
-rwxr-xr-x    1 root root       67941 Mar  4 16:26 top
-rwxr-xr-x    1 root root      163084 Mar  4 16:26 vdir
```

Another issue that deserves discussion is the performance impact created by the presence of the interceptor module on the honeypot. It's quite natural to experience an overhead with system call interception, due to the additional code that must be executed at each call. On a honeypot, however, this overhead must be kept under control, in order not to raise suspicions on the intruder.

Table I reveals that the *sys_write* call constitutes the main bottleneck in the mechanism, due to its fraction in the received blocks (96,38%) and data (99,68%). The overhead was measured through experiments consisting of copying files (a procedure that makes intensive use of the *sys_write* call) of different sizes (from 100KB to 10MB). The results showed that, depending on the size of the file, the increase in the copying time ranges from 70% to 90%. These numbers are very acceptable if we consider that, for small and medium files (<10MB), the total copying time doesn't take more than a fraction of a second. This overhead only becomes a problem in the writing of large chunks of data (tens or even hundreds of MB), creating a noticeable delay.

V. Conclusions

This paper described a mechanism based on system call interception for evidence collection from honeypot filesystems. This mechanism eliminates the flaws found in the traditional methodologies, providing automatization, dynamism and transparency.

In the experiment described in Section IV, the prototype behaved as expected, generating a tree containing every file and directory created by the intruder during the invasion. This process was carried out automatically, saving some precious time that would otherwise be spent in a forensic analysis with no guaranteed results. The evidence tree was submitted to an analysis, which determined that the tool installed by the intruder was a rootkit.

The potential of this approach, however, goes far beyond what was shown by the prototype. The dynamism and high temporal and informational granularity with which the evidence collection is done allows the reconstitution of the evidence tree at different moments of the invasion, making it possible to create a complete evidence timeline. Therefore, even if the intruder is careful enough to erase all the evidence before disconnecting, the processor module will be able to rebuild them: it must only travel back in the block log and process the blocks associated with the creation of the erased evidence. This temporal overview provides a unique insight into the intruders *modus operandi*, that can be rarely obtained with the traditional methodologies.

Also worth noticing is the mechanism's transparency. It intercepts the data, sends the blocks and reconstitutes the evidence without ever letting the intruder notice it. The only drawback has to do with the overhead created by the interceptor module's presence on the honeypot, that, on specific situations, can raise suspicion. However, performance measures show that this overhead is only noticeable in the writing of big chunks of data. Nevertheless, the *sys_write* interceptor's code optimization constitutes an important extension to this work.

Another limitation has to do with the integrity of the system call table. It is widely known that an intruder can modify it through the installation of an LKM rootkit, which could result in the neutralization of the interceptor module. Another possible improvement to the mechanism, therefore, would be the creation of a supervisor module that, somehow, would keep the integrity of the table. With the actual security paradigm, however, it is impossible to build a totally robust solution. This happens because the intruder generally has root privileges on the honeypot. With such privileges, the intruder can defeat any surveillance system; switching the kernel, for example, or even wiping the filesystem clean. Nothing can be done to stop him. The efforts must be turned to the creation of more transparent surveillance systems, that do not capture the intruder's attention. In this context, the use of virtual machines represents a great improvement, and the porting of the mechanism described here to a VM manager, such as User-Mode Linux [24], constitutes another relevant extension to this work.

VI. Acknowledgments

The authors would like to thank FAPESP for the financial aid given to this project, and also the other members of the Institute of Computing's System Security and Administration Laboratory of the State University of Campinas for their support in achieving the goals of this project.

References

[1] The Honeynet Project, *Know Your Enemy: Revealing the Se-*

curity Tools, Tactics, and Motives of the Blackhat Communit. Indianapolis, IN: Addison-Wesley, 2001.

[2] L. Spitzner, *Honeypots: Tracking Hackers.* Boston, MA: Addison-Wesley, 2002.

[3] L. Spitzner, "Honeypots: Definitions and values." Available in: <http://www.tracking-hackers/papers/honeypots.html> (Accessed on Jan. 2004), May 2003.

[4] L. Spitzner, "Honeytokens: The other honeypot." Available in: <http://www.securityfocus.com/infocus/1713> (Accessed on Jan. 2004), July 2003.

[5] The Honeynet Project and The Honeynet Research Alliance, "Know your enemy: A profile." Available in: <http://www.honeynet.org/papers/profiles/cc-fraud.pdf> (Accessed on Jan. 2004), June 2003.

[6] L. Oudot, "Fighting internet worms with honeypots." SecurityFocus - Available in: <http://www.securityfocus.com/infocus/1740> (Accessed on Jan. 2004), Oct. 2003.

[7] L. Oudot, "Fighting spam with honeypots." SecurityFocus - Available in: <http://www.securityfocus.com/infocus/1747> (Accessed on Jan. 2004), Nov. 2003.

[8] Recourse Technologies, "The evolution of deception technologies as a means for network defense." Sans Institute - Available in: <http://www.sans.org/rr/wp/recourse.pdf> (Accessed on Jan. 2004), Feb. 2002.

[9] N. Provos, "Honeyd: A virtual honeypot daemon," in *10th DFN-CERT Workshop*, (Hamburg, Germany), Feb. 2003.

[10] M. A. dos Reis and P. L. de Geus, "Standardization of computer forensics protocols and procedures," in *Proceedings of the 14th Annual Computer Security Incident Handling Conference*, (Waikoloa Village, Hawaii), June 2002.

[11] M. A. dos Reis, "Forense computacional e sua aplicação em segurança imunológica," Master's thesis, Instituto de Computação - UNICAMP, Campinas, SP, Brazil, 2003. (In Portuguese).

[12] Honeynet.BR Team, "Honeynet.BR: Desenvolvimento e Implantao de um Sistema para Avaliao de Atividades Hostis na Internet Brasileira," in *Anais do IV Simpsio sobre Segurana em Informtica (SSI'2002)*, (So Jos dos Campos, SP, Brazil), nov. 2002. (In Portuguese).

[13] The Honeynet Project, "Know your enemy: Genii honeynets." Available in: <http://project.honeynet.org/papers/gen2/> (Accessed on Jan. 2004), June 2003.

[14] L. Spitzner, "Learning the tools and the tactics of the enemy with honeynets," in *Proceedings of the 12th Annual Computer Security Incident Handling Conference*, (Chicago, IL), June 2000.

[15] "Tcpdump." http://www.tcpdump.org.

[16] "Snort." http://www.snort.org.

[17] A. S. Tanenbaum, *Computer Networks.* Upper Saddle River, NJ: Prentice Hall, 4. ed., 2003.

[18] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts.* New York, NW: John Wiley & Sons, 6. ed., 2002.

[19] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel.* Sebastopol, CA: O'Reilly, 2. ed., Dec. 2002.

[20] The Honeynet Project, "Know your enemy: Sebek." Available in: <http://www.honeynet.org/papers/sebek.pdf> (Accessed on Jan. 2004), Nov. 2003.

[21] K. Jain and R. Sekar, "User-level infrastructure for system call interposition: A platform for intrusion detection and confinement," in *Proceedings of Network and Distributed System Security (NDSS 2000)*, (San Diego, CA), Feb. 2000.

[22] N. Provos, "Improving host security with system call policies," Tech. Rep. 02-3, University of Michigan, Nov. 2002.

[23] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *Proceedings of Network and Distributed System Security (NDSS 2003)*, (San Diego, CA), Feb. 2003.

[24] "User-mode linux." http://user-mode-linux.sourceforge.net.