A Live Digital Forensic system for Windows networks

Roberto Battistoni[†], Alessandro Di Biagio[†], Roberto Di Pietro[‡⋆], Matteo Formica[†], and Luigi V. Mancini[†]

Abstract This paper presents *FOXP* (computer FOrensic eXPerience), an open source project to support network *Live Digital Forensics* (LDF), where the network nodes run a Windows NT family Operating System (OS). In particular, the FOXP architecture is composed of a set of software sensors, once for every network node, that log node activities and then send these logs to a FOXP collector node; this collector node analyzes collected data and manages the sensors activities. Software sensors, implementing the technique called *System Call Interposition* for Win32, intercepts all the kernel API (native API) invoked by the OS of the node. Thanks to the fine granularity of the logs, FOXP can intercept malicious activities. Centralized logs collected in the collector node, allow to detect coordinated-attacks on network nodes: attacks that would not be detectable with a single node analysis only. Note that the implemented System Call Interposition technique has allowed to intercept and redirect all of the 284 Windows XP system calls. The technique is exposed in detail and could be considered a contribution on its own. Finally, an overview of next steps to complete the FOXP project is provided.

Acknowledgements This work was partly supported by the Spanish Ministry of Science and Education through projects TSI2007-65406-C03-01 E-AEGIS and CONSOLIDER CSD2007- 00004 ARES, and by the Government of Catalonia under grant 2005 SGR 00446.

[†] "Sapienza" Università di Roma, Dipartimento di Informatica, Via Salaria n. 113, 00197 - Roma, Italy; e-mail: {battistoni, dibiagio, formica, mancini}@di.uniroma1.it

[‡] Università di Roma Tre, Dipartimento di Matematica, L.go S. Leonardo Murialdo n.1, 00146 - Roma, Italy; e-mail: dipietro@mat.uniroma3.it
[⋆] Universitat Rovira i Virgili, UNESCO Chair in Data Privacy, Dept. of Computer Engineering and Maths, Av. Països Catalans 26, E-43007 Tarragona, Catalonia; e-mail: roberto.dipietro@urv.cat

1 Introduction

Vulnerabilities are constantly growing (see Figure 1), and new vulnerabilities are the pre-requisite for new attacks. One difficulty for network administrator and police forces to deal with attacks, is that smart attackers attempt to erase or to hide proofs of their intrusion: log deleting, changing files timestamp or rootkit installation. These considerations show the need for new generations of Live Digital Forensic (LDF) tools [26] that could enforce responsibility attribution [12].

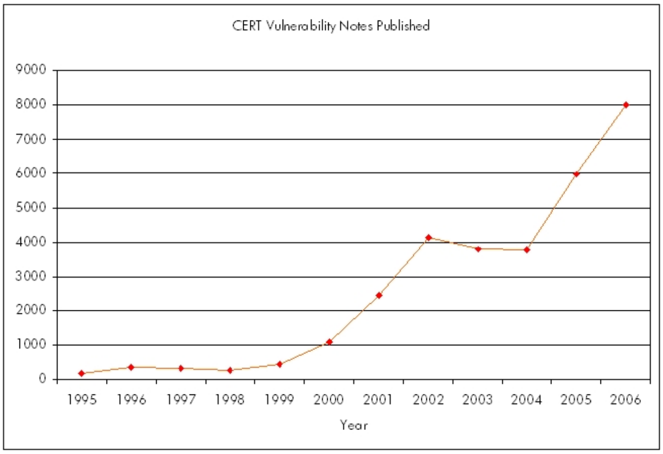

Fig. 1 CERT reported vulnerability in 1996-2006

In this context, network administrators need tools to support analysis and audit tasks and to detect intrusions and malicious activities. Suppose, for example, that in a time interval an attacker is able to exploit a node new vulnerability to attack other nodes of the network. If the administrator has not the right tool, she cannot detect neither the node generating the attack, nor when the attack started. In this type of context, tools that allow to reconstruct the entire activities of a system in a determined time interval, collecting evidences of the activities carried out in that interval, are needed. To date, there are two possibilities to accomplish this goal: traditional Computer Forensics (CF) and Live Digital Forensics (LDF). While the former approach is a static analysis of electronic supports only after a damaging event, the latter is able to represent the state of a live system for a determined time interval [2] [11].

The main contribution of this paper is to detail the architecture of *FOXP*, *computer FOrensic eXPerience*, an open source project [5] to support network LDF. FOXP traces activities at a kernel level in the node OS (Windows NT family based), with the primary goal of detecting malicious activities that are trying to subvert (or subverted) the node. In compliance with the LDF approach, FOXP is able to monitor activities in the system at every moment, and it allows both a live and a *post-mortem* analysis of the system. We also detail the corner stone of this architecture: the Logger; a software module that implements the system call interposition technique at kernel level within Windows NT family OS based nodes.

Architecture components can be grouped in two distinct sets: *client side* and *server side* components (figure 2):

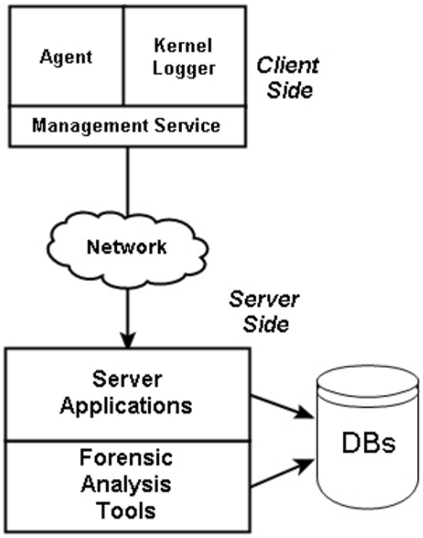

Fig. 2 FOXP architecture

1. Client side components are located on every node of the network. One component is the Logger, that allows to collect all the information needed to reconstruct the activity of the single node.
2. Server side components could be located on a single server node or on multiple server nodes, and their task is to collect nodes logs and organize them in an RDBMS for the successive forensic analysis.

The sequel of the paper is organized as follows. Section 2 reports some related works.

In Section 3 the FOXP architecture is introduced. Section 4 is focused on the Logger: the main technical component of the system. The Logger allows, extending technique called *System Call Interposition*, to intercept and monitor the entire set of Windows NT family system calls. Thanks to the ability to monitor system calls and their parameters, it will be possible to create a first dangerousness classification of Windows system call. Section 5 reports some concluding remarks and the activities currently under development to fully implement the FOXP architecture.

2 Related Work

IDSs are key to the protection of computing systems and computer networks: they allow to control systems and to react to attacks. In [3] it is shown the role of IDS in complex organization and some guidelines are provided for the development, operational conduct and management of IDS. IDS can be used as COTS or can be customized [27] to support CF or LDF.

Intrusion Detection named *anomaly based* is based on network traffic analysis. Its funding principle is that attacker behavior is different from normal user behavior and that the differences can be detected. Provided a definition of what a *normal* behavior is, anomaly based IDS could detect if the behavior is not *normal*, hence are able to detect unknown attacks or attacks for which patterns are not known yet [4] [23]. The main issue with this type of IDS is that there can be just a slight difference between *normal* and *anomalous* system behaviour. These two concepts can overlap producing a *false positive*, hence denying the access to system to a legitimate user. To reduce this problem, one could try to relax the definition of what is *normal*. However, this could produce *false negatives*: attacks could successfully run without being detected.

A subset of the IDS family are Host-based IDSs (HIDS) [4] [23]: these particular IDSs are the last line of defense in a system, because they try to detect and prevent an intrusion occurring within the system. HIDSs monitor both malicious and suspicious activities.

A subset of HIDS is constituted by the *Host Intrusion Prevention System* (HIPS) that also prevents attacks stopping invocation of malicious system calls. An example of HIPS are WHIPS [6] and REMUS [9] that use *System Call Reference Monitor* paradigm [9]. REMUS (REference Monitor for Unix System) is a prototype to monitor system calls that could be used to subvert privileged applications. REMUS uses a simple mechanism to implement the interception of the system calls at OS kernel level. Fundamentally, the execution of system calls is allowed only when the parameters of a certain set of system calls match a rule in an *Access Control Database* (ACD) in the kernel. That is, REMUS follows an *Anomaly Detection* approach.

WHIPS (Windows NT family Host based Intrusion and Prevention System) [6] is a system similar to REMUS that implements detection and prevention in Windows OSs and uses the *Reference Monitor* paradigm too.

In figure 3 we report the scheme of WHIPS, while details on the techniques implemented in WHIPS can be found in [14]. WHIPS and REMUS are on *SourceForge* as Open Source projects and it is possible to download them[7] [8].

An example of CF architecture is *Forensix* [21] for UNIX system. Similarly to FOXP, Forensix allows to control all the activities in the OS kernel space sending logs to a central server to structure logs in an RDBMS to implement CF high level analysis.

BluePipe [18] is another LDF system for *NIX platforms that is an alternative to a classic *post-mortem* analysis. It first performs a so called *on-the-spot* analysis, then results go to a central server for detailed analysis (as FOXP and Forensix, Bluepipe implement a client/server paradigm too).

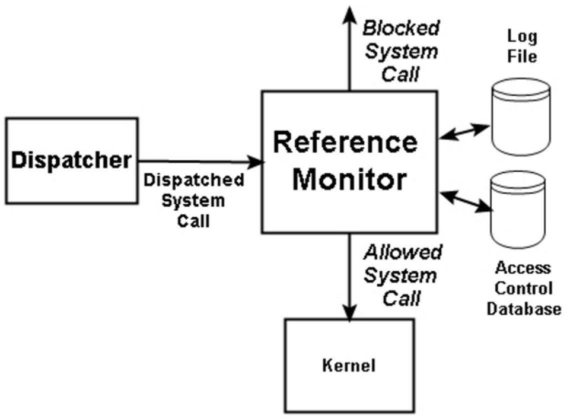

Fig. 3 WHIPS module

FOXP is similar to previous mentioned systems, but it has also some important differences:

- As Forensix and BluePipe, FOXP is based on a distributed agent (Logger) architecture, whereas WHIPS and REMUS are made of a single kernel module implementing system call interposition technique.
- Development of Forensix, Bluepipe and REMUS are just suitable for open source (well documented) OS. FOXP (as WHIPS) is developed for Windows NT family based OS. This introduces a critical level of complexity: Windows is a closed source OS, with little documentation available, especially for the native API realm.
- As Forensix and Bluepipe, FOXP uses an RDBMS too. This allows to store lot of logs in a structured way, useful to perform efficient analysis thanks to the expressivity of SQL.

3 FOXP Architecture

The aim of the FOXP architecture is to monitor the activities of the nodes in a network; when the client running on a node detects a malicious or suspicious activity, it will record the activities running on that node for a suitable time interval. Recorded data will be then sent to a centralized analysis system for CF purpose. In the following we provide a detailed discussion of the FOXP architecture.

Note that it is out of the scope of this paper to address the issue of communication security. Indeed, in this paper we focus on the forensics components of our proposed architecture only. However, note that the cited issue, together with the issues needed to provide a complete and deployable architecture, such as authenticity and non-repudiability of collected logs, are currently under investigation and will be presented in a different paper.

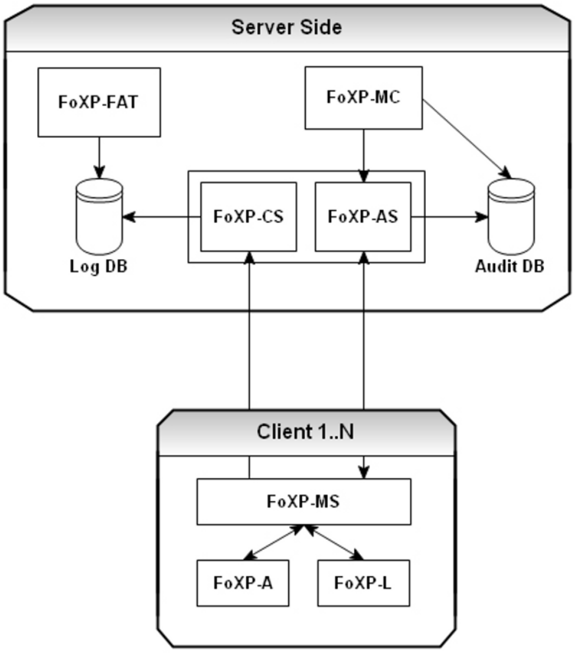

Fig. 4 Detail of FOXP Architecture

As shown in Figure 4, we assume a network of N nodes, where on each node it is installed a software module called *FOXP Agent* (FOXP-A): when this module detects an anomaly in the behavior of the system, it activates the *FOXP Logger* (FOXP-L) that starts collecting and sending data to a dedicated server machine. FOXP Architecture is composed by client-server subsystems (Figure 2): the client subsystem is located on every node of the network, and it controls the activities of the single node in order to detect anomalous behaviors.

Client subsystem is composed by:

- *FOXP Agent* (FOXP-A): it performs the analysis of the node activities. If an anomaly is detected, than the logging is activated (Section 3.1).
- *FOXP Logger* (FOXP-L): it intercepts the system calls invoked on the node and keeps track of them in a logging file (Section 3.1).
- *FOXP Management Service* (FOXP-MS): it manages the Agent and the Logger on every node as well as their communications with the centralized server of the architecture (Section 3.1).

Server-side subsystems are represented by one or more dedicated servers: these systems store the logs sent from every network node and collect these logs into a database for a successive forensic analysis. The server-side subsystems are:

- *FOXP Collector Server* (FOXP-CS): it receives and stores logs from every network node (Section 3.2).
- *FOXP Audit Server* (FOXP-AS): it receives and stores the state of the nodes (Section 3.2).
- *FOXP Management Console* (FOXP-MC): it remotely manages network nodes communicating with the FOXP-MS on every node (Section 3.2).
- *FOXP Forensic Analysis Tool* (FOXP-FAT): it executes the analysis of the collected logs and states (the Section 3.2).

3.1 Client-side Components

Client-side components reside on every node of the network and have the task to intercept and to log system calls invoked on the analyzed node. Recorded logs are sent to a dedicated server node (*Collector Server*) that stores their data in an RDBMS. As shown in Figure 5, the main components are:

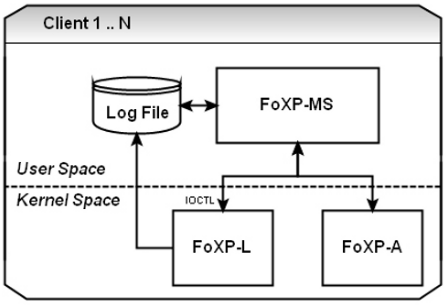

Fig. 5 Client-Side Components

- FOXP Agent
  it is an IDS (based on the Anomaly Detection [4] [23] paradigm) installed on every network node, that executes basic analysis of the node activities. All the FOXP Agents realize a Distributed IDS (DIDS) [1]: in this type of IDS, the role of the traditional IDS is delegated to a group of agents equipped with some intelligence (at the moment still not formalized and implemented in the system [14]). FOXP Agent will be able to detect malicious activities; the consequent action is then to send an alarm to the *Management Service* that will activate the *Logger*.

- FOXP Logger
  it is the main module of the FOXP architecture; it can be manually activated from the system administrator through the (server-side) *Management Console* or automatically by the (client-side) *Agent*. Through the system call interception in the Windows kernel, FOXP Logger captures and records every system generated event. Such module is implemented as a kernel driver: it is loaded and run in a reserved (and not paged) zone of kernel memory. FOXP Logger is loaded in the kernel by the *Management Service* and then it is launched together with the OS. Because the kernel module resides in the kernel space, it is not directly accessible from the user space programs, and it only accepts commands from the *Management Service*: module loading or unloading from the kernel memory, starting or stopping the interception of the system calls, and log writing on a file. Further details of this module will be given in Section 4.
- FOXP Management Service
  it performs the following tasks:

  - it receives commands from the *Management Console* for the *Agent* rules update;
  - it forwards commands directly to the *Logger* (through IOCTL channels [28]) without using the *Agent*.
  - it sends to the *Audit Server* periodic messages on the node live state (*heartbeat*), as well as the notifications of all the actions completed on the node from the *Agent* and the *Logger*.
  - it receives messages from the *Agent* and consequently sends commands to the *Logger*;
  - it sends to the *Collector Server* the data collected from the *Logger*.

  This module has been implemented as an OS service: it runs in background and supplies functionality not tied to the single user (services of Windows are similar to UNIX daemons). The requirement to use a service in order to communicate with driver when in user mode, derives from the fact that only an application with specific privileges can interact with a kernel module. The Management Service is initially installed by the system administrator and configured in automatic mode at start-up, so it will be started automatically at OS startup.

3.2 Server-side Components

On the Server-side, as synthesized in Figure 6, the tasks are the following: to receive, to store and to analyze logs received by the nodes of the network. It is fundamental that the reception is not compromised in some way (for example with DoS attacks), that the sent data are neither accessed nor alterable from unauthorized entities, and that the log storage support is adequately protected. These requirements, that could be achieved with standard network security techniques, are out of the scope of this paper and we assume them fulfilled. In detail, Server-side components are:

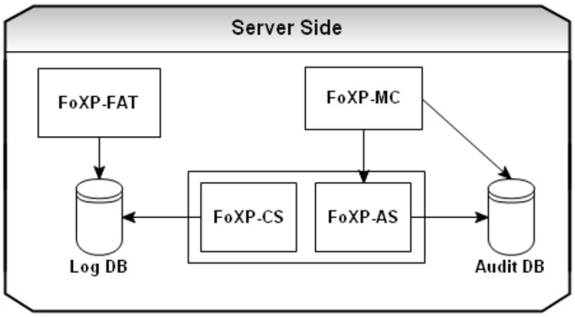

Fig. 6 Server-side Components

- FOXP Collector Server
  it is a server application that receives, over a TCP channel, the logs from the *Management Service* installed on every node and stores these logs into an RDBMS (*Log DB*). The server is a multithreaded application and the number of the thread is correlated to the number of the network nodes that are sending their log; it implements load balancing techniques with other server processes or with other *Collector Server* in cluster configuration.
- FOXP Audit Server
  it is a server application that receives, over a TCP channel, commands from the *Management Console* and forwards them to the *Management Service* of the destination nodes. Moreover, the *Audit Server* receives the notifications about the state and the completed operations from the *Agent* and the *Logger*; each of these pieces of information is to be inserted in a relational database (*Audit DB*).
- FOXP Management Console
  it is an administration console that is used to manage the entire FOXP architecture. It is used by the network administrator to monitor the state of the nodes (query on *Audit DB*), to configure (or to update) the FOXP *Agent* rules, and to manage the *Logger*.
- FOXP Forensic Analysis Tool
  it is a tool for the forensic analysis of the data stored in the Log DB. The main task of this component is the detection and the acquisition of the evidences to be used in a legal procedure to prosecute computer crimes perpetrators. This analysis methodology has not been defined yet at the moment, but will be addressed in future work. Some guidelines on Forensic Analysis methodology can be found in [16] and [15]. In particular, in [24] it is defined a formal model for the definition of forensic analysis procedures.

4 Logger

The *Logger* is implemented as a kernel driver. We chose this technique because in Windows NT family based OS (as well as in *NIX) any code can be injected into the kernel through a driver. However, note that this is critical for the stability and the security of the system. Indeed, such modules operate in Ring 0 [1] and constitute an extension of the kernel. For this reason, third party drivers — usually employed to control hardware devices — should be installed only if certified from *trusted* entities. In the following we detail the techniques used to implement this feature.

4.1 Interception of system call

The purpose of the FOXP Logger is the interception of all the Windows system calls, in order to monitor the entire activity of the system. The interception technique, that constitutes a contribute on its own, is described in detail in the following.

The paradigm of *system call interposition (SCI)* has already been used, for instance for the development of: sandboxing systems [20], process confinement, intrusion detection, auditing and privilege elevation [25] [17]. Although the SCI paradigm is a powerful method to monitor application behavior, it is a very error prone technique. In particular, Garfinkel [19] shows some of the difficulties and provides some guidelines to cope with them.

The interception technique was introduced by M.Russinovich in [13] (based on the SCI paradigm). Basically, this technique replaces the pointers to the original system calls in the SSDT table [2] with pointers to new functions. This substitution is not a trivial task: in fact, in the most recent Windows versions, it is not possible to write in certain regions of the kernel space, in particular the region where the SSDT resides. This makes the traditional technique of SCI useless: if someone tries to modify the SSDT, writing in a protected region, this would cause a Blue Screen of Death [3] [28].

To avoid experiencing a BSOD due to the violation of the memory protection mechanism, the solution is based on applying a patch to the SSDT [10] [22] [29]. In particular, this technique leverages the use of a *Memory Descriptor List* (MDL), to allocate a zone of non paged memory with the same interval of addresses of the virtual memory of the original SSDT. The fundamental difference is that this new zone of memory can be accessed in write mode.

A function in the MDL replacing an original system call (in agreement with the SCI technique) is called *wrapper*. Each wrapper is coded to perform logging and control operations. These activities carried out, the wrapper completes its last

[1] Windows has two modes (or rings) [28]: user mode (Ring 3) and kernel mode (Ring 0)

[2] System Service Dispatch Table, a kernel data structure where every system call has associated an index and a pointer to the memory zone that hosts the system call code

[3] BSOD, an error not recoverable that requires to reboot the system

function: it invokes the original system call. This way, the intended functionalities associated to the invocation of the system call are performed. A limited disadvantage of this technique is that, for every system call that must be intercepted, it is necessary to write a new wrapper.

To save (in *OldAPIPtr*) the pointer to the original system call (the *APIName*) and to exchange this one with the pointer to the wrapper (*NewAPIPtr*), it will be used the *HOOK* macro:

```
HOOK(APIName, NewAPIPtr, OldAPIPtr)
OldAPIPtr=ExchangePointers(
                &SSDT[Index(APIName)],
                NewAPIPtr)
```

Listing 1 HOOK macro

For example, if we want to intercept the Windows NT *ZwOpenFile* system call, we would use the HOOK macro in the following way:

```
HOOK(ZwOpenFile, NewZwOpenFile, OldZwOpenFile);
```

Listing 2 ZwOpenFile API

After the completion of this operation the OS will execute, when invoked, the new function *NewZwOpenFile* and not the *ZwOpenFile*; in *OldZwOpenFile* will be stored the memory address of the original system call (necessary for its restoration when the wrapper returns).

4.2 The Wrapper functions

In this subsection we detail the structure of the wrapper, focusing on the *NewZwOpenFile* introduced above:

```
NewZwOpenFile(OUT PHANDLE phFile,
                ..,IN ULONG OpenMode) {
  doLog("ZwOpenFile", phFile, .., OpenMode);
  OldZwOpenFile(phFile, .., OpenMode);
}
```

Listing 3 NewZwOpenFile API

Remind that the goal of the FOXP system is to trace the invocation of every system call. The main operation executed by each wrapper will be to write into a file some information about the intercepted system call: the name, the parameters, the date and the time of the invocation, the information on the current user and, at last, the complete path of the process that invoked the system call. The last operation of every wrapper will be to recall the original system call not to interfere with the normal evolution of the computation.

4.3 Interruption of the interception

It can happen that the administrator requires to unload the Logger. In this case, the original system calls must be restored. This action, issued via the management module of the Logger (*Agent* or *Management Console*), invokes the UNHOOK macro:

```
UNHOOK(APIName, OldAPIPtr)
ExchangePointers(&SSDT[Index(APIName)],
                 OldAPIPtr)
```

Listing 4 UNHOOK macro

After this operation the driver will be unloaded from memory by the *Management Service* using the Logger *UnLoad* function.

In the following we report an issue we had to deal with to implement the unload function. In the first stage of the implementation, the restoration of the original pointers of the system call (*UNHOOK* macro) and the unload of the logger (the *Unload()* function) was executed in one phase only, often generating (but in a non-deterministic fashion) a BSOD with error:

```
DRIVER_UNLOADED_WITHOUT_CANCELLING_PENDING_OPERATIONS
```

Listing 5 BSOD error

We figured out the reason: some system calls delay their own execution till the moment they are called again. This behavior can be due to the fact that the release of the resources held by the driver overlaps with the instant when these resources are requested by the pending system call. This problem has been superseded by separating the restoration phase from the unload phase.

4.4 Extension of the System Call Interposition technique

The system calls that can be intercepted with the macro previously described are only 104 out of a total of 284: for each of them, it exists an exported symbol [4] in the *ntoskrnl.lib* library, that allows to invoke the system call using its name. Note that trying to invoke a system call from a driver, a system call that has an unexported symbol (as for example *ZwAddAtom*) would generate the linking error:

```
error LNK2019:
unresolved external symbol
__imp__ZwAddAtom@24
referenced in function
_Hook@0sys\i386\Logger.sys:
error LNK1120: 1 unresolved externals
```

Listing 6 Linking error

[4] The symbols are special strings that identify a function within a program and link it to the respective code in a library

For the *Logger* module to intercept also those system calls for which the symbol from *ntoskrnl.lib* is not exported, the original technique has been extended as follows.

First, observe that, in the SSDT, a system call is clearly identifiable not only through its name, but also via its numerical index within the same SSDT. Then, the basic idea is to recall a system call just using its corresponding index in the SSDT (and not its name). The HOOK macro has been therefore extended through new *HOOK_NE* macro:

```
HOOK_NE(IndexAPIName,NewAPIPtr,OldAPIPtr)
OldAPIPtr=ExchangePointers(
                    &SSDT[IndexAPIName],
                    NewAPIPtr)
```

Listing 7 HOOK_NE macro

Using this macro the Logger will be able to intercept, without generating any error, the not exported system call (for example the *ZwCreateProcess*):

```
HOOK_NE(0x002f,NewZwCreateProcess,
        OldZwCreateProcess);
```

Listing 8 ZwCreateProcess API

Consequently, we define the *UNHOOK_NE* macro:

```
UNHOOK_NE(IndexAPIName,OldAPIName)
ExchangePointers(&SSDT[IndexAPIName],
                    OldAPIName)
```

Listing 9 UNHOOK_NE macro

4.5 Getting the image path of a process invoking a system call

Remind that, among the items that are collected by the logger, the complete process path of the process invoking the system call is an important one. Indeed, this would greatly help in both LDF and post-mortem analysis, to attribute responsibilities.

However, it is necessary to emphasize as finding this path has not been an easy task. Indeed, in principle there exist several ways to obtain this piece of information, when programming within the user space. However, many of these methods fail when adapted to the kernel mode, or just cause instability of the system, carrying to non deterministic BSOD generation.

We were able to devise an effective method, reported in the following. It is based on the usage of the *ZwQueryInformationProcess* system call, that takes three formal parameters. We have to pass to it, as parameters, both (*ProcessInformationClass* type) and *ProcessImageFileName* value (defined in *PROCESSINFOCLASS*). The third parameter is used to collect the return value. Hence, invoking the *ZwQuery-InformationProcess* system call with the described parameters, returns —within the third parameter— the string containing the complete path calling process.

5 Conclusions and Future Work

In this paper we have presented FOXP, an open source project to support network LDF where nodes run a Windows NT family based OS. In particular, we have detailed the architecture components of FOXP. Further, we have shown the first implementation achievement: the Logger module of the *FOXP agent*. This module is the one that presents the major technical difficulties within the project, and its implementation could be considered a contribution on its own. In particular, the Logger *extends* the system call interposition technique. As a result, all of the 284 system call in Windows XP OS have been mapped, and therefore FOXP is able to reconstruct all the system activities on such system. Note that the completeness of system call interception is a very important aspect for a system to support computer forensic activities.

As for future work, we are undergoing the following steps: to classify the system calls according to their level of dangerousness, to assess the overhead introduced by our Logger. We are also striving to extend our SCI technique on VISTA 32-bit OS. Preliminary results are quite encouraging.

Finally, note that two of the described architecture components have not been implemented yet: FOXP Agent and FOXP Management console. These development activities will be carried out as soon as the previously described steps complete; note that a preliminary feasibility study supported out intuition that their implementation should not present technical difficulties.

References

1. Abraham, A., Thomas, J.: Distributed intrusion detection systems: A computational intelligence approach. Applications of Information Systems to Homeland Security and Defense (Chapter 5), 105–135 (2005)
2. Adelstein, F.: Live forensics: Diagnosing your system without killing it first. Communications of the ACM **49**(2), 63–66 (2006)
3. Allen, J., McHugh, J., Christie, A.: Defending yourself: The role of intrusion detection systems. IEEE Software **17**(5), 42–51 (2002)
4. Axelsson, S.: Intrusion detection systems: A taxomomy and survey. Tech. rep. (2000)
5. Battistoni, R., Di Biagio, A., Di Pietro, R., Formica, M., Mancini, L.V.: The foxp project. SourceForge.net, http://foxp.sourceforge.net/
6. Battistoni, R., Gabrielli, E., Mancini, L.V.: A host intrusion prevention system for windows operating systems. In: Computer Security ESORICS 2004, vol. 3193, pp. 352–368. LNCS (2004)
7. Battistoni, R., Mancini, L.V.: The whips project. SourceForge.net, http://whips.sourceforge.net/
8. Bernaschi, M., Gabrielli, E., Mancini, L.V.: The remus project. SourceForge.net, http://remus.sourceforge.net/
9. Bernaschi, M., Gabrielli, E., Mancini, L.V.: Remus: A security-enhanced operating system. ACM Transactions on Information and System Security pp. 36–61 (February 2002)
10. Butler, J., Hoglund, G.: Rootkits: Subverting the Windows Kernel. Addison Wesley Professional (2005)

11. Carrier, B.D.: Risks of live digital forensic analysis. Communications of the ACM **49**(2), 56–61 (2006)
12. Casey, E.: Investigating sophisticated security breaches. Communications of the ACM **49**(2), 48–55 (2006)
13. Cogswell, R., Russinovich, M.: Windows nt system call hooking. Dr. Dobb's Journal (January 1997)
14. Di Pietro, R., Durante, A., Mancini, L.: Formal specification for fast automatic ids training. In: Ali Abdallah, Peter Ryan, and Steve Schneider, editors, article from the BCS-FACS International Conference on Formal Aspects of Security 2002, vol. 2629, pp. 191–204. LNCS (Spring 2003)
15. Di Pietro, R., Mancini, L.V.: A methodology for computer forensic analysis. article of the 3rd Annual IEEE Information Assurance Workshop pp. 41–48 (2002)
16. Di Pietro, R., Me, G., Mochi, M., Strangio, M.A.: An effective methodology to deal with slack space analysis. article of the International Conference on E-Crime and Computer Evidence (ECCE'05) (2005)
17. Forrest, S., Pearlmutter, B., Warrender, C.: Detecting intrusions using system calls: Alternative data models. In: article of 1999 IEEE Symposium on Security and Privacy, pp. 133–145. IEEE (1999)
18. Gao, Y., Richard III, G.G., Roussev, V.: Bluepipe: A scalable architecture for on-the-spot digital forensics. International Journal of Digital Evidence **3**(1) (2006)
19. Garfinkel, T.: Traps and pitfalls: Practical problems in system call interposition based based security tools. article of the ISOC Symposium on Network and Distributed System Security Symposium (2003)
20. Garfinkel, T., Pfaff, B., Rosenblum, M.: Ostia: A delegating architecture for secure system call interposition. Internet Society's 2003 Symposium on Network and Distributed System Security (2004)
21. Goel, A., chang Feng, W., chi Feng, W., Maier, D., Walpole, J.: Forensix: A robust, high-performance reconstruction system. International Conference on Distributed Computing Systems Security Workshop (SDCS-2005) (1999)
22. Hoglund, G., McGraw, G.: Exploiting Software: How to Break Code. Addison-Wesley (2004)
23. Jones, A.K., Sielken, R.S.: Computer system intrusion detection: A survey. Tech. rep. (1999)
24. Leigland, R., Krings, A.W.: A formalization of digital forensics. International Journal of Digital Evidence **3**(2) (2004)
25. Provos, N.: Improving host security with system call policies. Tech. rep. (2002)
26. Richard III, G.G., Roussev, V.: Next-generation digital forensics. Communications of the ACM **49**(2), 76–80 (2006)
27. Ruighaver, A.B., Tan, K.M.C., Thompson, D.: Intrusion detection systems and a view to its forensic applications. Tech. rep. (2000)
28. Russinovich, M., Solomon, D.: Microsoft Windows Internals. Microsoft Press, 4th edition (2004)
29. Schreiber, S.: Undocumented Windows 2000 secrets : A programmers cookbook. Addison-Wesley (2001)