Formal Specification for Fast Automatic Profiling of Program Behavior

Roberto Di Pietro, Antonio Durante, and Luigi.V. Mancini

Abstract This paper illustrates a methodology for the synthesis of the behavior of an application program in terms of the set of system calls invoked by the program. The methodology is completely automated, with the exception of the description of the high level specification of the application program which is demanded to the system analyst. The technology employed (VSP/CVS) for such synthesis minimizes the efforts required to code the specification of the application. The methodology we propose has been applied to several daemons; as a case study, we discuss it in details to the Post Office Protocol, the ipop3d daemon. Though the methodology is independent from the intrusion detection tool adopted, the results have been employed to configure the REMUS intrusion detection system and are shown in this paper.

1 Introduction

Nowadays computer systems work in highly dynamic and distributed environments and require the protection mechanisms to prevent intentional or unintentional violation to the security policies. Often the attackers are able to circumvent the access control mechanisms exploiting the applications flaws. As an example, in many cases the attackers tend to hijack the control of privileged processes, such as the daemon processes. A well-known family of this kind of attack is called buffer overflow attack [2].

Our proposed methodology is aimed at mapping the normal behavior of an application program onto its allowed system calls, thus enabling the detection of attacks

Roberto Di Pietro
Università di Roma Tre, Dipartimento di Matematica, L.go S. leonardo Murialdo n. 1, 00146 - Roma, Italy, e-mail: dipietro@mat.uniroma3.it

Antonio Durante · Luigi V. Mancini
Dipartimento di Informatica, Università di Roma “La Sapienza”, Via Salaria 113, 00198 - Roma, Italy e-mail: {mancini,durante}@di.uniromal.it
that attempt to hijack the execution of privileged processes the application is possibly composed of.

The methodology we describe starts from a high level description of the daemon, such as an IETF RFC [12] and derives the set of the system calls that can be invoked by the daemon, during its normal execution. Note that the generated system calls are specific for the particular implementation of the daemon the proposed methodology is applied to. For instance, consider an FTP daemon that receives USER/PASS requests; while processing these requests, the daemon could execute different kinds of system calls depending on its implementation. To perform authentication, the FTP daemon may need to read a security-sensitive file (using regular I/O system calls), or it may access the same sensitive file via memory mapping (using the `mmap` system call), or it may access the sensitive file via NIS (using socket connects, reads, and writes), etc. Note that the particular set of system calls to perform the FTP authentication is chosen by the programmer while implementing the daemon. Hence, we do not try to synthesize the allowed system calls of all the possible implementations for a given specification, but we consider a specific implementation of the daemon that will run on the specific system under consideration.

In this paper, we assume that the specific implementation of the daemon to which the proposed methodology is applied to does not contain malicious code, though it should not be necessarily trusted. In other word, the daemon could contain potential bugs in the implementation of the high level specification (e.g. bugs that could be exploited by a buffer overflow attack), but should not contain arbitrary malicious operations (e.g. a malicious programmer adds a trojan code which creates a root account in the password file even if such operation is not strictly required by the high level specification of the particular daemon).

The main novelty of our methodology is that it represents a first attempt for the automatic definition of the daemon normal behavior profile starting from its interface definition. To derive a daemon normal profile in terms of system calls, we specify the daemon interface using a technology that has been successfully applied to protocol design and analysis [3, 4]. The specification produced can be used for every version of daemon having the same interface, also if it runs on different a OS.

The main contributions of this paper are: a methodology to speed-up the synthesis of the normal behavior of an application program. This methodology differs from that based on the source code analysis, since our approach synthesizes the program behavior starting from both a high-level specification document, such as an IETF RFC, and a specific implementation. An implicit advantage of not relying on source code, is that our approach is applicable even if the source code is not available for analysis. Moreover, the process is automated, with the only exception of the specification phase, which is a high level human-activity. Further, to show the effectiveness of such a methodology, we used it to configure a particular anomaly based IDS prototype: REMUS [1].

The paper is organized as follows: next section summarizes the related works in the field. Section 3 illustrates the proposed methodology. Section 4 offers a case study, applied to the `ipop3d` daemon, while Section 5 presents a description of how the methodology can be used to configure the REMUS prototype. In Section 6 some
concluding remarks and further research directions are exposed. Finally, the appendix reports the system calls intercepted as a result of the work developed in Section 5, and the VSP specification of the Postgres daemon. This paper is a revised and extended version of the work previously reported in [5].

2 Related works

There are several IDS proposed in literature that can be divided into two broad classes: network based and host based IDS. The former tries to detect the attempts to subvert the normal behavior of the system, analyzing the traffic of the network. The latter is intended to perform as last line of defense.

The host based IDS strives to detect intrusions analyzing the behavior of the system on which the IDS is run. The host based IDS can be further distinguished into three categories: (1) anomaly detection, (2) misuse detection, (3) specification-based. In particular, the main characterization of the three methods can be summarized as follows: (1) the anomaly detection method is based on revealing the behavior of the system that differs from a profile that depicts the normal behavior of the system that is automatically updated; (2) the misuse detection tries to classify all the possible known attacks to the system creating an association between each attack and a sort of signature. Recognizing such a signature on the system, raises an alarm; (3) the specification based approach tries to specify the intended behavior of the monitored program. Even slight variation from this behavior, raises an alarm.

The performance of these approaches is measured in terms of: (1) false positive, e.g. an alarm raised in correspondence to a regular situation; (2) false negative, e.g. the IDS did not raise an alarm while an intrusion occurred. The fundamental characteristic of the proposed approaches consist in defining the system behavior in terms of the sequences of the system calls invoked by the monitored application [10]. However, the approaches differ since the system behavior can be modeled in different way, e.g. formal specification [21], neural networks [9], sequences of pattern [14]. The strength and the weaknesses of each of the approach can be classified as follow: (1) the strength of the anomaly detection approaches is based on the capacity of the algorithm to generalize the model of the normal behavior of the monitored program. The higher the ability of generalization of the algorithm, the higher the probability to individuate new typology of attack. The drawback of such an approach is that when the IDS experiences for the first time a new behavior, it raises an alarm, which may be a false positive; (2) using the misuse detection approach, it is difficult to individuate new kinds of attack since this approach detects only the old ones, so false negative can occur. However, when an alarm is raised, this is because a signature has been detected, and therefore a false positive cannot occur. Note that the set of signatures could include ambiguous patterns that can be generated by an attacker as well as a legitimate user; (3) the specification techniques try to overcome the deficiencies of the anomaly detection and misuse detection approaches, defining
the intended behavior of the controlled program. Any behavior that differs from the expected one is marked as illegal and an alarm is raised.

The specification-based technique should have the precision of the misuse detection technique and also the ability of detecting new kinds of attack as the anomaly detection technique. However, on one hand, specification based techniques require a good level of technical competence: indeed, a good knowledge of the operation performed by the application program is needed because such a knowledge must be translated in a specification of the expected behavior in a format comprehensible to the IDS. On the other hand, the IDS based on anomaly and misuse detection technique are respectively self-calibrating or just calibrated. Indeed, automatic techniques that lead the learning of the IDS have been proposed [20, 26, 10]. A more feasible specification based approach is that proposed in [7]. Using this approach it is possible to implement several kinds of security mechanisms. Moreover, the described approach gives the possibility of combining in different ways various IDS mechanism.

The network intrusion-detection systems (NIDSs), i.e. [24], often report a massive number of simple alerts of low-level security-related events. Many of these alerts are logically involved in a single multi-stage intrusion incident and a security officer often wants to analyze the complete incident instead of each individual simple alert.

[15] proposes a well-structured model that abstracts the logical relation between the alerts in order to support automatic correlation of those alerts involved in the same intrusion. The basic building block of the model is a logical formula called a capability. We use capability to abstract consistently and precisely all levels of accesses obtained by the attacker in each step of a multistage intrusion. We then derive inference rules to define logical relations between different capabilities. Based on the model and the inference rules, we have developed several novel alert correlation algorithms and implemented a prototype alert correlator.

Another network based intrusion detection is proposed in [22]. The network based intrusion detection consists in a distributed multiagent intrusion detection system (IDS) architecture, which attempts to provide an accurate and lightweight solution to network intrusion detection by tackling issues associated with the design of a distributed multiagent system, such as poor system scalability and the requirements of excessive processing power and memory storage. The proposed IDS architecture consists of (i) the Host layer with lightweight host agents that perform anomaly detection in network connections to their respective hosts, and (ii) the Classification layer whose main functions are to perform misuse detection for the host agents, detect distributed attacks, and disseminate network security status information to the whole network.

Among other approaches that cannot be classified in the exposed taxonomy, it is worth noting [19, 28], which try to implement a Mandatory Access Control policy. If the security policy defined is too restrictive, the process has less privilege than the minimal ones needed to execute its functionality and then the system cannot work properly, requiring the intervention of the system administrator. However, such an
approach detects all the attempts to bypass the assigned privileges.

Attackers often try to evade an intrusion detection system (IDS) when launching their attacks. There have been several published studies in evasion attacks i.e. [29], some with available tools, in the research community as well as the hackers community. Some payload-based network anomaly detection systems can be evaded by a polymorphic blending attack (PBA). The main idea of a PBA is to create each polymorphic instance in such a way that the statistics of attack packet(s) match the normal traffic profile. [25], present a formal framework for the open problem: given an anomaly detection system and an attack, can one automatically generate its PBA instances? The framework not only expose how the IDS can be exploited by a PBA but also suggest how the IDS can be improved to prevent the PBA.

We now focus on the REMUS prototype [1]. Its design is based on the analysis of critical system calls. In particular, the overhead introduced by REMUS with respect to others IDS is negligible. The system calls have been partitioned in level of threat: the system calls of level 1 are those utilized from the hacker to gain complete control of the system. REMUS checks the system call of level 1, if the invoking process is a root daemon or if it is setuid to root; indeed, only in this case the attacker can gain access to the system as a privileged user. System calls belonging to other levels of threat are discarded by the IDS since they cannot lead to a subversion of a privileged process. REMUS allows a goods security level while intercepting only the 10% of the total number of system calls performed during execution.

3 Methodology

The methodology we propose takes in input a formal specification of daemon $A$, and returns as output a subset of the system calls that a specific program implementation of $A$ is allowed to invoke.

Throughout this paper, we apply our methodology to the RFC1939 (ipop3d) [12], as an example. Note that any other specification of a daemon, with the level of detail of an RFC, could have been adopted as the starting point of our methodology. However, we have based our discussion on RFC since we intend to address the implementation of any secure Internet servers, which are mainly based on the execution of standard daemons, whose expected behavior is described through RFCs. In the following, we detail the steps of the methodology and subsequently develop a simple example.

The first step of the methodology consists in modeling the daemon behavior as a that can recognize any session of commands execution of the daemon $A$, triggered by a client. This step requires a human intervention to express the RFC specification of the daemon $A$ with state transition semantic (an automaton). The states of the FSM are derived from the RFC, and the transitions between states are the possible commands that the daemon can be requested to execute.
The second step consists in formalizing the FSM using the VSP language [3]. This step must be carried out by the system analyst too.

In the third step, the VSP specification is compiled using the CVS compiler [3], detailed in Section 4.4. In particular, the result of the compilation produces a Security Process Algebra (SPA) [6] that we call FSM1.

The fourth step of the methodology consists in exploring the FSM1 to obtain the finite set of command sequences that may be invoked by an execution of the daemon. A commands execution set accepted by FSM can be equivalently represented as a subset of the command sequences produced by FSM1. Thus, executing the set of command sequences accepted by the FSM1 will invoke the same set of system calls invoked by the daemon when executing the command sequences recognized by the FSM. We assume that any command implementation invokes the same set of system calls regardless of the value and of the size on input parameters.

In the fifth step, the sequences of commands produced by CVS are translated, by a simple parsing algorithm, in the sequences of commands executable by a tool called ILSC (Invocation of Legal Sequences of Commands). The ILSC executes such command sequences on a specific implementation of the daemon. During this step the module REMUS is loaded in configuration mode to intercept and log all the system calls invoked by the daemon. Then, the logged system calls are used to update the ACL.

Note that the first and second steps above are carried out by the system analyst, while the others are automated. In the following, we illustrate the whole sequence of steps in the case study.

4 Case Study

4.1 POP3 commands

When the ipop3d daemon service is started, it listens on TCP port 110 [12]. When a client host wishes to make use of the service, it establishes a TCP connection with the server host. When the connection is established, the ipop3d server sends a greeting. The client and the ipop3d server daemon then exchange commands and responses until the connection is closed or aborted.

The commands of the post office protocol consist of a keyword followed by one or zero arguments. The response of the ipop3d daemon consists of a success indicator possibly followed by additional information. There are currently two indicators: positive ("+OK") and negative ("-ERR"). A post office protocol session progresses through a number of states during its lifetime. Once the TCP connection has been opened and the ipop3d server has sent the greeting command, the session enters the AUTHORIZATION state. In this state, the client must identify itself to the ipop3d server. Once the client has been successfully identified, the server acquires the resources associated with the client’s maildrop, and the session enters the TRANS-
ACTION state. In this state, the client requests actions to the ipop3d server. When the client has finished its transactions, the session enters the UPDATE state.

In this state, the ipop3d server releases any resource acquired during the TRANSACTION state and says goodbye. The TCP connection is then closed. For a complete description of the post office protocol see RFC1939 [12].

4.2 The FSM (step 1)

To model the interactions between a client and the ipop3d we use a Finite State Machine, FSM.

We define the FSM, where the transitions represent the commands invoked by a client and the states are those reached by the daemon as a consequence of such an interaction. Figure 1 shows the FSM derived from the RFC for the ipop3d daemon. In each state an error can occur due to a bad input command, that is BAD_INP. The errors can be divided in two kinds, as reported in Table 1.

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command error</td>
<td>the name of the command does not coincide with any-</td>
</tr>
<tr>
<td></td>
<td>one of those specified in the RFC, given the current</td>
</tr>
<tr>
<td></td>
<td>state of execution of the daemon.</td>
</tr>
<tr>
<td>parameter error</td>
<td>the parameter is omitted if required or it is wrong: out of range, mismatch.</td>
</tr>
</tbody>
</table>

Each time the ipop3d daemon receives a bad input command (BAD_INP), the software send back to the client as output an error message err-“error message”. When the client sends a “well formed” command (a command and its parameters are well formed if they respect the RFC specification) the daemon returns as output an OK+ “message”. The daemon terminates its execution when it reaches one of the two possible final states: UPDATE (U) or LOGOUT (L).

In Figure 1, the label TRANS_INP represents a set S={STAT, NOOP, LAST, RSET, LIST, DELE, RETR} of post office protocol commands that a client can independently invoke while ipop3d runs in the TRANSACTION state (T in the Figure 1).

We call trace a finite sequence of commands accepted by the FSM. If we consider the set of all the traces recognized by the FSM, they correspond to the set of all possible different sequences of commands invoked by a client and executed by the ipop3d daemon. Note that there is a correspondence between each command invocation and a set (possibly empty) of system calls executed at kernel level.
4.3 VSP specification for ipop3d (step 2)

To obtain the FSM1, the system analyst has to specify the daemon using the VSP language. VSP is a value-passing language like CCS value passing [18] that allows protocol specification. A VSP specification is translated in a Security Process Algebra (SPA) [6] specification using the CVS compiler. The process of describing an Internet daemon through VSP is an extension of the use for which the VSP was initially intended for, that is VSP was developed to describe protocols [3, 4]. In general, a protocol consists of a set of messages (that contain a set of values) exchanged by two or more entities to reach a common goal (e.g. authentication). Indeed, a daemon is specified in VSP via a set of messages exchange. A daemon can accept a command and give as output: (1) an error message if the command is not well formed; (2) an ok message if the command is well formed. Therefore, describing a daemon through messages is a task that can be achieved if we employ messages that contain as parameters: the name of the command that the daemon has to execute and the parameters of the invoked command.

Given the idea of how it is possible to employ the VSP to describe the behavior of a daemon, we detail below the four steps of the procedure that leads the system analyst to the specification in VSP of the FSM:

- **definition of the commands and the values of the commands parameters**: in this step the system analyst has to synthesize the set of commands that a daemon can accept and the values that the command’s parameter can assume during a daemon normal session. Table 2 describes the first step of the VSP specification.
- **definition of the messages accepted by the daemon**: as above expressed, the messages accepted by the daemon contains the name of the command that the daemon has to execute and the values of its parameters.
- **declaration and definition of the body of the daemon process**: this part specifies the “body” of the daemon server. The body of a process consists of a sequence of messages. There are two kind of messages: (1) the input messages that correspond to a command invocation; (2) the output messages, which correspond to the output of the ipop3d daemon. An output message can assume two values: (a) OK; (b) err-, according to the fact that the received command is well formed or not. In Table 4.3 the messages of the ipop3d daemon. It is not necessary to
Fast Automatic Profiling of Program Behavior

### VSP Description

- **#Names** *
  - VSP section name
  - POP3 server: commands grouped according to the state of execution and parameters number.

- **CmdA:USER,PASS,WC;** *
  - commands executable in the Authorization State

- **CmdT1:STAT,NOOP,LAST,RSET,WC;** *
  - commands executable in the Transaction state
  - with 0 parameter

- **CmdT2:LIST,WC;** *
  - commands executable in the Transaction State
  - with 1 optional parameter

- **CmdT3:DELE,RETR,WC;** *
  - commands executable in Transaction State
  - with 1 parameter

- **CmdU:QUIT,WC;** *
  - commands executable in the Update State
  - WC = wrong command in a state

- **Agent:Sam,Null;** *
  - user name

- **Pass:bianco,Null;** *
  - Passwords

- **Msg:1,2,3,4,Null;** *
  - message ids

- **NULL : null parameter value**

### Table 2 Command and parameters definition

<table>
<thead>
<tr>
<th>VSP</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#ActionDec</td>
<td>* action containing the command type and the parameter type</td>
</tr>
<tr>
<td></td>
<td>* that the POP3 server executes</td>
</tr>
<tr>
<td>cmd_1 (CmdA,Agent)</td>
<td>* action for the execution of a command of type CmdA</td>
</tr>
<tr>
<td></td>
<td>* and parameter type Agent</td>
</tr>
<tr>
<td>cmd_2 (CmdA,Pass)</td>
<td>* action for the execution of a command of type CmdA</td>
</tr>
<tr>
<td></td>
<td>* and parameter type Pass</td>
</tr>
<tr>
<td>cmd_31 (CmdT1)</td>
<td>* action for the execution of a command of type CmdT1</td>
</tr>
<tr>
<td>cmd_32 (CmdT2,Msg)</td>
<td>* action for the execution of a command of type CmdT2</td>
</tr>
<tr>
<td></td>
<td>* and parameter type Msg</td>
</tr>
<tr>
<td>cmd_33 (CmdT3,Msg)</td>
<td>* action for the execution of a command of type CmdT3</td>
</tr>
<tr>
<td></td>
<td>* and parameter type Msg</td>
</tr>
<tr>
<td>cmd_4 (CmdU)</td>
<td>* action for the execution of a command of type CmdU</td>
</tr>
<tr>
<td>OK+()</td>
<td>* action that communicates the good result of a command execution</td>
</tr>
<tr>
<td>err-()</td>
<td>* action that communicates the bad result of a command execution</td>
</tr>
<tr>
<td>Sayonara()</td>
<td>* action that communicates the session ending</td>
</tr>
</tbody>
</table>

### Table 3 POP3 messages

- specify the body of the client process, as usually required by the VSP specification, because the specification of the behavior of the daemon is comprehensive of all possible interactions that the daemon itself can perform with any client.

- Table 4 reports the VSP specification of the *ipop3d* daemon.

- **definition of a generic session;** it is sufficient to consider a single instance of the daemon VSP process because the VSP coding of the daemon process generates all the sequences of the messages that could be executed during a session with a generic client. Table 5 reports the VSP invocation of a POP3 session performed by the daemon *ipop3d*. 

Table 4  The VSP specification of POP3

```plaintext
#Processes
Def POP3ser(u:Agent)
Var
Agent:name;
CmdA:c1,c2;
CmdT1:c31;
CmdT2:c32;
CmdT3:c33;
CmdU:c4;
Pass: p;
Msg:m,m1,m2;
Begin
  cmd_1(c1,name).
  if ((c1=USER) & (name=Sam))
    'OK+().
  cmd_2(c2,p).
  if ((c2=PASS) & (p=bianco))
    'OK+().
  cmd_31(c31).
  if (c31!WC)
    'OK+().
  cmd_32(c32,m1).
  if (c32!WC)
    'OK+().
  cmd_33(c33,m2).
  if ((c33!WC) & (m2!Null))
    'OK+().
  cmd_4(c4).
  if (c4=QUIT)
    'Sayonara().
  else
    'err-().
  endif
  else
    'err-().
  endif
  else
    'err-().
  endif
  else
    'err-().
  endif
  else
    'err-().
  endif
  else
    'err-().
  endif
  else
    'err-().
  endif
  else
    'err-().
  endif
End
```
Fast Automatic Profiling of Program Behavior

Table 5 the session invocation for VSP daemon

4.4 Compiling VSP (step 3)

The CVS compiler takes in input the VSP specification of the daemon and generates the FSM1. The body of the VSP process is made up of a linearly ordered sequence of input and output messages. The FSM1 obtained using the CVS compiler can be modeled as a tree. The corresponding tree model for the generation of the FSM1 can be obtained according to the rules in Figure 2.

```
TreeGeneration (node *m, CartesianP *P)
Begin
If (m==root) {
  GenerateSPA(m);
  m=m->next;
} else if (m==InputMessage) {
  If (checkInBound (m)) {
    GenerateSPA(m);
    m=m->next;
    TreeGeneration(m,P);
  } else {
    while (P!=Null) {
      P = GenerateCartesianP(m);
      TreeGeneration(m,P);
    }
  }
} else if (m==OutMessage) {
  if (checkInBound (m)) {
    GenerateSPA(m);
    m=m->next;
    TreeGeneration(m,P);
  } else {
    print(err);
    exit
  }
} else if (m==Null)
  exit;
End
```

Fig. 2 The SPA generation code.

In the routine for the generation of the FSM1 code we call m the message that we want to translate in SPA code. P is a possible instance of the message m. Each time the routine generates a FSM1 message, the routine moves to the next message
via the statement $m = m \rightarrow \text{next}$. When a message is translated from the VSP to the FSM1, we say that a VSP message is *expanded* in a FSM1 messages. The routine in Figure 2 works as follows:

1. the root of the tree is the FSM1 name of the process;
2. if the next examined message $m$ is an input message, this message must consist of a set of parameters, say $\text{par1, par2, ..., parK}$ usually not instanced. The routine checks out if the current message has parameters that can assume only one possible value with the function $\text{checkInBound}$. If this is not the case, the CVS compiler, starting from this message, generates the Cartesian product of the value of the command parameters. Each element of the Cartesian product constitutes a different son of the root if the expanded message is the first. Otherwise, the generated messages are the sons of the previous *expanded* message;
3. if the examined message $m$ is an output message, then its parameters are usually instanced. If the parameters are bounded, that is they assume just a value, the compiler generates a son of the previous *expanded* message. Such a node is labeled with the output message and the actual values of the parameters. If the parameters are not instanced, that is they assume more than one value, an exception is raised and the routine is stopped;
4. the routine terminates when there are no more messages to expand.

Note that the representation of FSM1 is indeed a tree, since it does not contain neither links to other nodes at the same level, nor links to the ancestor. Moreover, each node has one parent only. Finally, there cannot be isolated messages, since the compiler always links a generated node to one and exactly one of the previously generated nodes. Therefore, we are assured that the generated FSM1 graph is a tree.

4.5 Visiting the FSM1 (step 4)

Producing all the traces of the FSM1 consists of a depth first search in the process algebra tree produced by the CVS compiler. We use the algorithm $\text{GetTraces}$ which takes as input: the first line of the SPA code $\text{firstline}$; the root node, e.g. the name of the SPA process $\text{firstnode}$; and an $\text{emptybuffer}$ that will contain the execution traces, e.g. the command sequences. Note that the algorithm $\text{GetTraces}$ follows a classical depth-first visit, getting all the paths root-leaf of the FSM1.

4.6 Executing Traces (step 5)

The traces produced by the $\text{GetTraces}$ algorithm are translated in command sequences that can be invoked by the ILSC module. The ILSC executes such command sequences on a specific implementation of the daemon (that we want to profile) when the IDS is loaded in the OS kernel.
5 Using the Methodology to Configure REMUS

To exemplify the application of our methodology to an IDS based on the analysis of the system calls invoked by a program, we have used the methodology to configure REMUS. In REMUS, the system calls that are considered critical for the security of the system, (see Table 6), are intercepted by a LINUX kernel module specifically designed for this purpose. This module operates as reference monitor that denies or allows the execution of a particular system call invoked by a daemon or by a setuid software program. The decision of the reference monitor is based on a kernel data structure, the ACL, that maintains the set of authorized system calls and their relative parameters. The content of such a data structure can be seen as a classification of the behavior of a program.

Table 6 Critical system calls

<table>
<thead>
<tr>
<th>system calls</th>
<th>dangerous parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>chmod, fchmod</td>
<td>a system file or a directory</td>
</tr>
<tr>
<td>chown, fchown, lchown</td>
<td>a system file or a directory</td>
</tr>
<tr>
<td>execve</td>
<td>an executable file</td>
</tr>
<tr>
<td>mount</td>
<td>on a system directory</td>
</tr>
<tr>
<td>rename, open, mkvod</td>
<td>a system file</td>
</tr>
<tr>
<td>link, symlink, unlink</td>
<td>a system file</td>
</tr>
<tr>
<td>setuid, setresuid, setfsuid, setreuid</td>
<td>UID set to zero</td>
</tr>
<tr>
<td>setgroups, setgid, setfsgid, setregid</td>
<td>GID set to zero</td>
</tr>
<tr>
<td>init_module</td>
<td>modules not in /lib/modules</td>
</tr>
</tbody>
</table>

During the execution of the ILSC, the module REMUS stores the system calls invoked by the daemon in a file with the following format: system call name - parameters - invoking program. The first field consists of the name of the system call that the application program can invoke; the second field consists of the argument values of the system call; the third field is the name of the monitored application program. The content of this file is subsequently used to update the ACL. A schema of the methodology applied to the REMUS IDS is showed in Figure 3.

The first two steps in Figure 3 are carried out by the system analyst while the others are automatic. During the fifth step the module REMUS is loaded in configuration mode to intercept and log all the system calls invoked by the ILSC. Then, the logged system calls are used to update the ACL. After the fifth step completes, the system is ready to provide its intended services. During this production mode, REMUS allows a system call execution if and only if the invoking process and the value of the arguments comply with the contents of the ACL previously build in the configuration mode. We could have both false positive and false negative. As
for false positive: the set of system calls intercepted by REMUS can be an underestimated approximation of the normal behavior of the analyzed daemon, because there could be sequence of command not specified in the VSP (due to system analyst error). As for false negative, REMUS does not take in account the order whom the system calls has to be invoked, for instance a mimicry attack could have success [27].

However, the majority of penetration techniques that allows an attacker to hijack the control of a privileged process will be blocked by the IDS configured in this way (buffer overflow technique is among the blocked ones).

### 5.1 Results

The described methodology has been applied to a set of daemons. We report the results in Table 7. From the above table, it is remarkable that: (a) only a limited number of daemons requires the execution of critical system calls; (b) the number of critical system calls intercepted, for those daemons that invoke them, is quite low. These findings can be explained with the growing attention that has been paid to security. Indeed, a basic step to minimize the possibility of system subversion is to reduce the number of critical system calls invoked by daemons in privileged mode.

It is worth noting that the objective of the REMUS prototype is to prevent the subversion of the system. Henceforth, if a non-super user application is compromised, this is not detected by REMUS. This point is clearly stated by the REMUS developers [1] and leveraged throughout this paper. However, this assumption may
Table 7  Methodology results

<table>
<thead>
<tr>
<th>OS</th>
<th>Daemon</th>
<th>FSM1 sessions</th>
<th>Number of Critical Sys Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>red hat 7.2</td>
<td>Qmail 1.03 (smtp service)</td>
<td>5922</td>
<td>0</td>
</tr>
<tr>
<td>mandrake 9.0</td>
<td>wu-sftpd 2.62</td>
<td>6014</td>
<td>0</td>
</tr>
<tr>
<td>mandrake 9.0</td>
<td>vs-ftpd 1.1.3</td>
<td>4890</td>
<td>0</td>
</tr>
<tr>
<td>mandrake 9.0</td>
<td>Postgres</td>
<td>77272</td>
<td>0</td>
</tr>
<tr>
<td>mandrake 9.0</td>
<td>sendmail 8.11.6</td>
<td>5348</td>
<td>5</td>
</tr>
<tr>
<td>mandrake 9.0</td>
<td>postfix</td>
<td>4920</td>
<td>21</td>
</tr>
<tr>
<td>redhat 7.3</td>
<td>pop3d</td>
<td>5230</td>
<td>3</td>
</tr>
</tbody>
</table>

raise some objection in adopting REMUS together with the proposed methodology; for instance, the subversion of the Postgres daemon, run by a non-privileged user, could possibly compromise a sensitive Database while not compromising the entire system. This point can be addressed intercepting the critical system calls the Postgres daemon invokes when run with normal user privileges. The normal behavior profile obtained can thus be used to prevent the daemon subversion also when it is run as a non-privileged process. Note that this operation allows a complete re-use of the VSP daemon specification.

In Appendix 7 we report: (1) the name of the system calls intercepted and the appropriate commands to be added to the ACL; (2) the VSP specification of the Postgres daemon. In particular, note that the specification of the Postgres daemon is compact and almost self-explanatory. To specify the daemon behavior according to the VSP language took one day only. This should testify the feasibility of the proposed methodology.

6 Concluding remarks

In this paper, we have drawn a methodology to derive, starting from a high level specification of an application program, the set of the system calls an application can invoke. Our methodology does not need to access the source code, thus, it can be adopted even in those environment in which only the executable is available. Moreover, the methodology is completely independent from the IDS tool adopted. Note that when a new release of a daemon implementation becomes available, it is necessary only to execute the ILSC to upgrade the ACL (step 5), while preserving the efforts spent in the steps 1-4 of the methodology. Finally, except for the first two step of the methodology described in Section 3, which are at a high level of design, the process is completely automatic.

The adoption of a specific technology (VSP/CVS), which is internally based on automaton representation, allows us to obtain a good profile of the normal behavior of the application program.

We tested the proposed methodology to derive the normal behavior profile of a set of daemons critical for the deployment of a secure WEB server. The results
we obtain encourage us to explore the program normal behavior space using the specification driven methodology we propose. As a future work we plan to extend the methodology in order to deal with the so called evasion attacks [29].

7 Appendix

In the following we report two test cases. The first is related to the pop3d, sendmail and postfix daemon; the second is related to the Postgres daemon. The first test case shows the effectiveness of the proposed approach used with the REMUS IDS. The second test case is intended to apply our methodology to a more complex daemon.

7.1 The critical system calls

Table 8 reports the critical system calls intercepted by REMUS, set in DEBUG mode, when the sessions produced by the CVS compiler are executed, for the pop3d the sendmail and postfix daemon.

The second column reports as first field the system call name, then the system call parameters, and finally the name of the program that could invoke that system call.

7.2 Postgres VSP Specification

Table 9 reports the VSP specification of the Postgres daemon version 7.2. Once compiled CVS produce a .trc file, which comprises 77272 Postgres sessions. The sessions executed, with REMUS set in DEBUG mode, did not produce any critical system calls. Hence, the Postgres daemon either does not execute critical system calls or when it execute them, it runs without privileged rights.

References

Table 8. The critical system calls

<table>
<thead>
<tr>
<th>OS &amp; Version</th>
<th>Critical System calls in the ACL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandrake 9.0</td>
<td>unlink /var/spool/mail/root.lock gnu-pop3d</td>
</tr>
<tr>
<td>Gnu-pop3d 0.9.8</td>
<td>open /var/spool/mail/root.lock gnu-pop3d</td>
</tr>
<tr>
<td></td>
<td>open /var/spool/mail/root.gnu-pop3d</td>
</tr>
<tr>
<td>Mandrake 9.0</td>
<td>exec /usr/bin/procmail/ sendmail</td>
</tr>
<tr>
<td>Sendmail 8.12.5</td>
<td>open /var/spool/mqueue/ sendmail</td>
</tr>
<tr>
<td></td>
<td>link /var/spool/mqueue/ sendmail</td>
</tr>
<tr>
<td></td>
<td>unlink /var/spool/mqueue/ sendmail</td>
</tr>
<tr>
<td></td>
<td>rename /var/spool/mqueue/ sendmail</td>
</tr>
<tr>
<td>Mandrake 9.0</td>
<td>open /var/spool/postfix/pid/master.pid master</td>
</tr>
<tr>
<td>Postfix</td>
<td>open /var/spool/postfix/pid/unix.local.local local</td>
</tr>
<tr>
<td></td>
<td>open /var/spool/postfix/pid/unix.showq showq</td>
</tr>
<tr>
<td></td>
<td>open /var/spool/postfix/pid/unix.flush flush</td>
</tr>
<tr>
<td></td>
<td>exec /usr/lib/postfix/cleanup master</td>
</tr>
<tr>
<td></td>
<td>fchmod /var/spool/postfix/maildrop/ postdrop</td>
</tr>
<tr>
<td></td>
<td>open /var/spool/postfix/maildrop/ postdrop</td>
</tr>
<tr>
<td></td>
<td>exec /usr/lib/postfix/bounce master</td>
</tr>
<tr>
<td></td>
<td>exec /usr/lib/postfix/smtpd master</td>
</tr>
<tr>
<td></td>
<td>exec /usr/lib/postfix/trivial-rewrite master</td>
</tr>
<tr>
<td></td>
<td>open /var/spool/postfix/pid/unix.cleanup cleanup</td>
</tr>
<tr>
<td></td>
<td>exec /usr/lib/postfix/nqmgr master</td>
</tr>
<tr>
<td></td>
<td>exec /usr/lib/postfix/smtp master</td>
</tr>
<tr>
<td></td>
<td>exec /usr/lib/postfix/pickup master</td>
</tr>
<tr>
<td></td>
<td>exec /usr/lib/postfix/local master</td>
</tr>
<tr>
<td></td>
<td>open /var/spool/postfix/public/pickup postdrop</td>
</tr>
<tr>
<td></td>
<td>exec /usr/lib/postfix/showq showq master</td>
</tr>
<tr>
<td></td>
<td>open /var/spool/postfix/pid/unix.rewrite trivial-rewrite</td>
</tr>
<tr>
<td></td>
<td>exec /usr/lib/postfix/flush master</td>
</tr>
<tr>
<td></td>
<td>open /var/spool/postfix/pid/unix.bounce bounce</td>
</tr>
</tbody>
</table>

Table 9  Postgres VSP specification: declarations

#Names
Agent: A,B;
CmdDB: CREATEDB;
CmdTB1: CREATETABLE;
CmdTB2: INSERT;
CmdTB3: SELECT;
CmdU: QUIT;
Namedb: DataBase;
Nametb: Table;
Field: ID, surname, Born;
Options: PrimaryKey, NotNull, NULL;
OptionsBy: Group, Order, NULL;
ValuesID: 1, 2, NULL;
ValuesSurn: Alice, Bob, NULL;
ValuesDN: 1975, 1976, NULL;
Parameter: Star, surname, NULL;
Condition: Condition1, Condition2, NULL;

#ActionDec
cmd_1 (CmdDB, Namedb)
cmd_21 (CmdTB1, Nametb, Field, Options, Field, Options, Field, Options)
cmd_22 (CmdTB2, Nametb, ValuesID, ValuesSurn, ValuesDN)
cmd_23 (CmdTB3, Parameter, Nametb, Condition, OptionsBy)
cmd_3 (CmdU)
OK+()
er-()

#RolesDec
first
second

#ProcessesDef

Table 10  Postgres VSP specification: first agent

first(s1:Agent)

Var
CmdDB: c1.a;
CmdTB1: c21.a;
CmdTB2: c22.a;
CmdTB3: c23.a;
CmdU: c3.a;
Namedb: DBase;
Namedtb: tab_1a;
Field: cp1_a, cp2_a, cp3_a;
Options: opz1_a, opz2_a, opz3_a;
OptionsBy: opzby_1a;
ValuesID: v_1a;
ValuesSurn: surn_1a;
ValuesDN: DN_1a;
Parameter: p_1a;
Condition: cond_1a;

cmd_1(c1_a, DBase).
  if ((c1_a=CREATEDB) & (DBase=DataBase))
    'OK+().
  cmd_21(c21_a, tab_1a, cp1_a, opz1_a, cp2_a, opz2_a, cp3_a, opz3_a).
  if (((c21_a= CREATETABLE) & (tab_1a=Table) & (cp1_a|cp2_a) & (cp2_a|cp3_a) & (cp1_a|cp3_a))
    'OK+().
  cmd_22(c22_a, tab_1a, v_1a, surn_1a, DN_1a).
  if ((c22_a=INSERT) & (tab_1a=Table) & (v_1a!NULL) & (surn_1a!NULL) & (DN_1a!NULL))
    'OK+().
  cmd_23(c23_a, p_1a, tab_1a, cond_1a, opzby_1a).
  if ((c23_a=SELECT) & (tab_1a=Table) & (p_1a!NULL))
    'OK+().
  else
    'err-().
  endif
else
  'err-().
endif
else
  'err-().
endif
else
  'err-().
endif
else
  'err-().
endif
End
Table 11  Postgres VSP specification: second agent and session invocation

second(s2:Agent)
Var
CmdDB: c1_b;
CmdTB1: c21_b;
CmdTB2: c22_b;
CmdTB3: c23_b;
CmdU: c3_b;
Name: tb_2b;
Field: cp1_b, cp2_b, cp3_b;
Options: opz1_b, opz2_b, opz3_b;
OptionsBy: opby_2b;
ValuesID: v_2b;
ValuesSurn: surn_2b;
ValuesDN: DN_2b;
Parameter: p_2b;
Condition: cond_2b;
Begin
  cmd_21(c21_b, tab_2b, cp1_b, opz1_b, cp2_b, opz2_b, cp3_b, opz3_b).
  if ((c21_b=CREATETABLE) & (tab_2b=Table) & (cp1_b!cp2_b) & (cp2_b!cp3_b) & (cp1_b!cp3_b))
    'OK+().
  cmd_22(c22_b, tab_2b, v_2b, surn_2b, DN_2b).
  if ((c22_b=INSERT) & (tab_2b=Table) & (v_2b!NULL) & (surn_2b!NULL) & (DN_2b!NULL))
    'OK+().
  cmd_23(c23_b, p_2b, tab_2b, cond_2b, opby_2b).
  if ((c23_b=SELECT) & (tab_2b=Table) & (p_2b!NULL))
    'OK+().
  else
    'err-().
  endif
else
  'err-().
endif
else
  'err-().
endif
else
  'err-().
endif
else
  'err-().
endif
End
#Session *invocation
first(A)
second(B)
#RestrictionOn
channel(Cmd_1,Cmd_21,Cmd_22,Cmd_23,Cmd_3)
Intrusion Detection Systems
Di Pietro, R.; Mancini, L.V. (Eds.)
2008, XIV, 250 p. 20 illus., Hardcover