Protocol Analysis in Intrusion Detection Using Decision Tree

Tarek Abbes
LORIA/INRIA-Lorraine
54602 Villers-lès-Nancy,
France
abbes@loria.fr

Adel Bouhoula
SUP'COM
2083 Cité El Ghazala,
Tunisie
bouhoula@planet.tn

Michaël Rusinowitch
LORIA/INRIA-Lorraine
54602 Villers-lès-Nancy,
France
rusi@loria.fr

Abstract

Network based intrusion detection are the most deployed IDS. They frequently rely on signature matching detection method and focus on the security of low level network protocols. Because of the large number of false positives from one side, and the incapacity to detect some attack types from another side, IDS must allow more interest to the monitoring of application level protocols.

We propose in this paper a combination of pattern matching and protocol analysis approaches. While the first method of detection relies on a multipattern matching strategy, the second one benefits from an efficient decision tree adaptative to the network traffic characteristics.

1 Introduction

Intrusion detection systems (IDS) are an essential part of the security infrastructure. They are used to detect, identify and stop intruders. The administrators can rely on them to find out successful attacks and prevent a future use of known exploits. IDS are also considered as a complementary solution to firewall technology by recognizing attacks against the network that are missed by the firewall.

Nevertheless, IDS are plagued with false positive alerts, letting security professionals to be overwhelmed by the analysis tasks. Therefore, IDS employ several techniques in order to increase the detection probability of suspect threats while reducing the risk of false positives. When using pattern matching to detect intrusions, IDS users try to refine the attack signatures and limit the search to smaller traffic intervals. On the other hand, by using protocol analysis in the detection process, IDS rely on protocol specification as explained in RFC in order to adequately analyse the traffic. So, they will be able to understand each field in the packet, and supervise the right execution of the protocols which leads to reduce the number of false positives.

We present in this paper a new method to perform protocol analysis detection. Our first contribution consists in defining a language to express detection rules. The parsing of these rules leads to the construction of a decision tree used to verify the traffic, which constitutes our second contribution. Finally, we describe an implementation of a set of widespread protocols in order to compare the performance of the 2 methods : Pattern matching and Protocol analysis.

The remainder of this paper is organized as follows : Section 2 underlines the advantages of protocol analysis in intrusion detection and Section 3 shows the theoretical basis of our protocol analysis model. In Section 4, we compare the two detection methods based on experimental results. Finally, we conclude our paper in Section 5.

2 Protocol Analysis Advantages

Intrusion detection systems (IDS) employ protocol analysis in order to understand the traffic and supervise the execution of some selected protocols. IDS can therefore employ more specific attack signatures. The immediate consequences of such analysis are immunity against evasion attempts, reduction of attacks search space and a decrease of false positives output. We detail in the following these advantages through concrete examples.

- Preventing evasion : Protocol analysis is useful to prevent evasion. In fact, by analysing the traffic, IDS become immune to path obfuscation, and hex, double hex and unicode encoding.
- Space search reduction : Protocol analysis extracts the fields of the studied protocol which leads to search for patterns in specific parts of the packet rather than the entire payload. For example, we can restrict the search of vulnerable *cgi script* at uniquely the *uri* content.
- False positives reduction : This is a consequence of the above advantage. In fact, by reducing the search

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)
0-7695-2108-8/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

space to a single field, we decrease drastically the number of false positives. For instance, unauthorized access to web pages is detected by having a notification code "403" from the web servers [3]. Filtering a "403" string in another part would raise a false alarm.

- Extra detection capability : Protocol analysis does not restrict its monitoring to network or transport protocol but it extends its inspection to high level protocols. Decoding and normalizing the received traffic with a full consideration of stacked protocols may reveal new attacks. For example, some protocols like *telnet* and *ftp* send their data in character mode. They also employ special characters like back space and erase line.
- Log effectiveness : Protocol analysis allows to record some sensitive information about the current session of protocols like authentication phase (save login name in a *telnet* session). These data are very useful when we detect attacks in further stages.
- Protocol verification : Protocol analysis permits the detection of implementation flaws and crafted packets. In fact, by applying RFC recommendation on sniffed traffic, it will be possible to look for anomalies. For instance, DNS protocol limits the length of a domain name to 255 characters [2].

3 Protocol Analysis Method

We will explain in this section how we detect intrusions using protocol analysis and the complementarity of this method with the pattern matching. The proposed strategy consists in extracting data from each received packet and then traversing a pre-built decision tree. We devote the next subsection to explain the construction of the decision tree.

3.1 Decision Tree Construction

The decision tree is constructed by processing the specification file of each protocol to analyse. The specification is divided into 3 units. The first part defines and initializes variables that will be used in the ensuing parts. The second unit defines a set of detection rules. A detection rule contains a set of keywords that must be checked to trigger an alarm. Finally, the third part contains a set of action rules that describe the behaviour after analysing a packet. We define 4 types of actions: *"update"* will set up the generic information about the protocol session whereas *"log"* determines the output format of the raised alarm. In addition, *"exit"* interrupts the analysis of the current packet while *"continue"* verifies the next stacked protocol. We present in the following the BNF grammar of our language for specifying the protocols. Then we give in Table 1 an abstract of the second part of the *ftp* specification.

BNF grammar of the language

```
#Part1: Variable Definition
'VAR' body_1
body_1 := var_name var_value;
var_value := list_of(value) | value
#Part2: Detection Rules
'RULE' Id body_2;
Id := value
/* Id is the identifier of the rule */
body_2 := list_of(condition) | condition
condition := feature operator term
/* feature refers to one field of the protocol */
operator := contain | = | in | > | < |
term := value | list_of(value) | var_name
#Part3: Action Rules
'BHV' body_3
body_3 := condition '⇒' action argument;
condtion : boolean expression
action := update | log | exit | continue
```

We generate afterwards a decision tree from the protocol specifications of to be analysed. We found that a decision tree can speed up the detection process by checking simultaneously many detection rules.

The decision tree construction relies on 3 inference rules that are applied iteratively on an initial node (see Figure 1). A tree contains 3 types of nodes : ordinary nodes, leaves and special nodes. Each node is represented by a tuple $(c, \mathcal{R}, \mathcal{F}, \mathcal{L})$ where c is a condition, $\mathcal{R}$ is a set of candidate detection rules, $\mathcal{F}$ is a feature set and $\mathcal{L}$ is a set of detection rules. A *condition* is a tuple <feature, operator, term> where *feature* refers to a field of the protocol (see the BNF). The feature set $\mathcal{F}$ denotes the set of attributes already used to decompose the tree and $\mathcal{L}$ is the set of detection rules that are matched at that node. The initial root node contains a null condition, the whole set of detection rules, an empty set of features and an empty set of matched rules. Then, we iteratively decompose each node according to the set of possible features using the appropriate inference rules. Leaves are nodes that cannot be transformed anymore. They can be used to report attacks thanks to the detection rules contained in their last field. A special node is a node at which some attacks can be reported but can be further processed by any rules.

We denote by P the set of all features and by ξ_i any comparison operator (such as the ones given in the BNF). In addition, let e denote a feature, c denote a condition and v denote a value. Before we present our inference system, we define some auxiliary functions employed in the inference rules. We begin with the function **Param** for extracting the parameters of a rule(s).

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)
0-7695-2108-8/04 $ 20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

r_1: RULE 1 direc = to_server, cmd = site, arg = "exec", msg: Site Exec Attempt
r_2: RULE 2 direc = to_server, cmd = pass, arg contain "hack", msg: Suspect Password
r_3: RULE 3 direc = to_server, cmd = help, argSize > 900, msg: Buffer Overflow attempt

Table 1. Detection rules examples

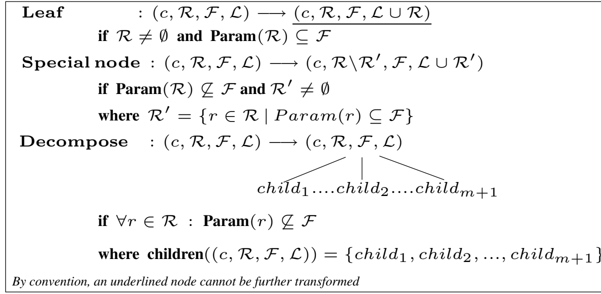

Figure 1. Inference system

Definition 3.1 *Let r be a rule of the form $r \equiv e_1\ \xi_1\ v_1 \wedge \ldots \wedge e_k\ \xi_k\ v_k$ and $\mathcal{R}$ be a set of rules. We define the **Param** function by $Param(r) = \{e_1, ..., e_k\}$. We extend this function to a set of rules $\mathcal{R}$ by $Param(\mathcal{R}) = \bigcup_{r \in \mathcal{R}} param(r)$*

Example 3.1 *Let $RS = \{r_1, r_2, r_3\}$ be a set of three detection rules defined on the ftp service and shown in Table 1. Then Param(r_1)=Param(r_2)={direc, cmd, arg}; Param(r_3)={direc, cmd, argSize}; Param(RS)={direc, cmd, arg, argSize}.*

We next define a projection function **Proj**, which given a set of detection rules and a feature, extracts all possible values associated to that feature.

Definition 3.2 *Let $\mathcal{R}$ be a set of rules and e be a feature. We define the projection function **Proj** as $Proj(R, e) = \{v \mid \exists(e = v \wedge r) \in \mathcal{R}\}$.*

Example 3.2 $Proj(\{r_1, r_2, r_3\}, direc) = to_server$.

Definition 4.3 explains the candidate function **Cand**, which given a set of detection rules and a condition c, extracts all rules that verifies the condition.

Definition 3.3 *Let $\mathcal{R}$ be a set of rules and c be a condition. We define the candidate function **Cand** as $Cand(\mathcal{R}, c) = \{r \in \mathcal{R} \mid r \equiv c \wedge r'\}$.*

Example 3.3 $Cand(\{r_1, r_2, r_3\}, direc = to_server) = \{r_1, r_2, r_3\}$.

We denote by **Next** a function that selects the next suitable feature to perform a decomposition at some node n. The function is based on some feature selection criteria that will be explained in Subsection 3.1.1. Finally, we define the **Children** of a node n.

Definition 3.4 *From any node n of the tree labelled by $(c, \mathcal{R}, \mathcal{F}, \mathcal{L})$ such that $\forall r \in \mathcal{R}\ :\ \textbf{Param}(r) \not\subseteq \mathcal{F}$ let:*

- $e = Next(c, \mathcal{R}, \mathcal{F})$
- $\{v_1, ...v_m\} = Proj(\mathcal{R}, e)$
- $\forall\ i \in [1..m]\ \mathcal{R}_i = Cand(\mathcal{R}, e\ \xi_i\ v_i)$
- $\mathcal{R}_{m+1} = \{r \in \mathcal{R} \mid e \notin Param(r)\}$
- *$child_i$ ($1 \leq i \leq m$) the nodes $(e\ \xi_i\ v_i, \mathcal{R}_i, \mathcal{F} \cup \{e\}, \mathcal{L})$*
- *$child_{m+1}$ the node $(not(e\ \xi_1\ v_1 \vee \ldots \vee e\ \xi_m\ v_m), \mathcal{R}_{m+1}, \mathcal{F} \cup \{e\}, \mathcal{L})$*

Then we define the **Children** *of the node n as $Children(n) = \bigcup_{1 \leq i \leq m+1} child_i$.*

Example 3.4 $Children(root) = (direc = to_server, \{r_1, r_2, r_3\}, \{direc\}, \emptyset); (not\ direc = to_server, \emptyset, \{direc\}, \emptyset)$.

The 3 inference rules for building the decision tree are in Figure 1. Starting from an initial node, the rules are iteratively applied to the nodes of the tree until no more application is possible. The *decompose* rule creates children nodes from a node n if some parameters of the candidate detection rules $\mathcal{R}$ of n are not in the features set $\mathcal{F}$. We select such a parameter according to the Next function and create for each case analysis one child. If all the features in some nodes have been considered so far then the node cannot be processed further: it becomes a leaf (denoted by underlining it). Moreover, if we have selected all the features of only a subset of the detection rules of one node then we apply the *special node* inference rule to add new matched detection rules. Finally, we stop the tree construction when all reduced nodes are leaves. We note that the inference rules are exclusive of each other.

3.1.1 Feature selection

The feature selection is very important for building efficient decision tree. Krugel and Toth [4] have used the information gain criteria to construct a decision tree for classifying the detection rules. They have considered that all the rules are equiprobable. Our problem is different from theirs. In fact, we do not try to find the adequate cluster of rules that fits the packet under analysis but we try to directly classify

IEEE COMPUTER SOCIETY

the flow in either the attack or not_attack classes. In our approach, we define for each protocol a set of features and we use the characteristics of the traffic to build an optimal decision tree. We have implemented an independent program that, giving a training file, a feature set and one criterion for feature selection, builds the adequate decision tree. We have integrated 3 possible criteria : the information gain, the gain ratio and the Gini index [6].

Our experimentation with the 3 different feature selectors gives the same result. This is due to the small number of classes (only 2 : attack and not_attack classes). Indeed the Gini prefers splits that put the largest class into one pure node, and all others into the other. Entropy used in gain computation favors size-balanced children nodes. The difference appears when the number of classes is large. In this case the Gini produces splits that are too unbalanced close the root.

3.2 Pre-processing phase

Once the decision trees are constructed, we can start the analysis process. We first store in a dedicated data structures useful information from the captured traffic. We have defined three types of data structure : the header node, the container node and the trace node.

Header node

Header nodes represent various connections under inspection and are organized in a Splay tree [7]. They contain information about the transport and the network protocols. They also point to container nodes which contain information about the application protocol used in these connections. In order to organize the heading node in the Splay tree, it does not suffice to take into account the 4-tuple (source address, destination address, source port, destination port) as we have done it in [1]. In fact, doing so leads us to obtain two heading nodes for each connection : one for client/server flow and another one for the server/client flow which prevent us from analyzing the interaction between client and server. Hence, we have defined another 4-tuple to describe each connection between two machine ip_1 and ip_2. The 4-tuple has the form (α = min(ip_1,ip_2), β = max(ip_1,ip_2), port used by α, port used by β). We then use a lexicographical order to perform comparisons.

Container and trace nodes

Our goal with the protocol analysis is to supervise the execution of application layer protocols and understand the nature of the traffic in order to extract meaningful fields. Therefore, the two phases of identification and decoding of the protocols are essential to perform the analysis. We classify the extracted data in 2 categories : permanent data

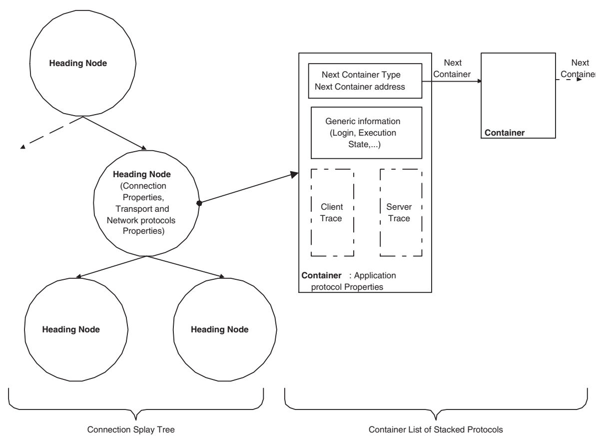

Figure 2. Data Structure

and temporary ones. Permanent data refer to all information useful during the total execution of the protocol (for example the login name in a telnet session) whereas temporary data are those needed only to perform some tests on the packet. We store the permanent data in a specific data structure called container. This structure is allocated when we identify a new session of the specified protocol and will be released at the end of the connection. The container points to a temporary data structure called "trace" that saves the temporary data of each packet. The trace will be then released when we finish the processing of the packet. Finally, as application layer protocols can be stacked one on the other, we define in each container the type and the address of the next container which will refer to the next stacked protocol. For instance, if we use https and we try to supervise the two protocols HTTP and SSL, then we link the container of HTTPS to the container of SSL. To seek for the container of SSL, it suffices to look for the header node of that connection. We summarize in Figure 2 the different data structures needed to perform the analysis.

3.3 Detection phase

The detection process exloits the results of the preprocessing phase to traverse the decision tree. If we reach a leaf, then a detection rule matches and we trigger the corresponding alert. Among the operators appearing with the keywords of the detection rules, let us outline the “contain” operator. It serves to match patterns in the extracted protocol fields. Hence, the search will be very fast and generates less false positives.

In addition, because we may find similar signatures, we modify the *decompose* inference rule to take into account the keywords that use pattern matching operations. We group all similar signatures and we generate several sub-

IEEE COMPUTER SOCIETY

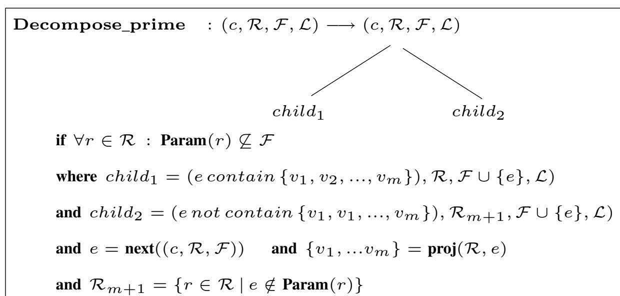

Figure 3. Decompose_prime rule

tree based on the result of the match operation. The new inference rule called *decompose_prime* is shown in Figure 3. We have used the Aho Corasick method to perform the multipattern matching since our experiments in [1] shows its superiority on the Forward DAWG and Match DAWG algorithms.

4 Experimentation

We have implemented several protocols in order to compare the performance of the protocol analysis model from a pure pattern matching based method. The set of specified protocols include FTP, TELNET, SNMP, SMTP, RPC DNS and HTTP protocols. We focus our discussion on the RPC portmapper service.

In order to carry out experiment on RPC protocol, we simulated 414 RPC packets representing 46 different forms of attack. Out of them, Snort version 1.9 could detect only 3 types of attacks namely, use of dump procedure over UDP (Sid 429), TCP (Sid 590) and RPC portmap request rstatd (Sid 583). This is because Snort heavily relies on pattern matching without any definition of offset or depth for search. On the other hand, the remaining RPC rules of Snort which failed to detect some attacks, are restricted to search patterns only in some area in the payload leading to its failure. Therefore, when relying on only pattern matching detection, the decrease of signature search space decreases the detection time but increases the number of false negative. This can be very well attributed to lack of protocol analysis prior to pattern matching.

By using the protocol analysis method, we succeed to detect all these attacks. In fact, the detection process becomes a simple identification of the called procedures or a pattern search in an extracted argument of these procedures. While the method presents an extra cost for decoding the protocol, the overhead of preprocessing is rapidly balanced by a faster detection. The gain is more relevant when dealing with a large number of detection rules which is the common case nowadays given the increasing number of attacks.

Our experiment on DNS protocol shows the possibility of detecting several DoS attacks that cannot be matched by only pattern matching. For example, client flooding occurs when a client sends a single DNS query but receives thousand of DNS responses from attackers. Our system is capable of detecting it by associating each reply to the corresponding request. Similary, presence of circular reference in DNS caches, which could crash a session when requesting a zone transfer, can be detected by our system using a protocol analysis approach.

5 Conclusion

Most intrusion detection systems rely on pattern matching operations to look for attack signatures. Nevertheless, the number of signatures is constantly growing following the increasing types and varied forms of attacks. The pattern matching becomes a computational expensive task.

We show in this paper that an arbitrary definition of the domain search for signatures causes the generation of false negatives. To overcome this problem, we rely on a protocol analysis approach that leads to the construction of decision trees in the initial phase of the IDS deployment. The built tree is adaptative to the network traffic characteristics since the features chosen to split the tree ensure the highest reduction of entropy or the lowest Gini impurity. In addition, pattern matching operations are now integrated inside decision tree. They are triggered after achieving light verifications and benefit from a refined domain search of signatures.

References

[1] T. Abbes, A. Bouhoula, and M. Rusinowitch. Filtrage efficace pour la détection d'intrusions. In *Conférence Sécurité et Architecture Réseaux SAR 2003*, Jully 2003.

[2] R. Elz and R. Bush. RFC 2181: Clarifications to the DNS specification, July 1997. Updates RFC1034, RFC1035, RFC1123. Status: PROPOSED STANDARD.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, The Internet Society, June 1999. See `http://www.ietf.org/rfc/rfc2616.txt`.

[4] C. Kruegel and T. Toth. Using decision trees to improves signature-based intrusion detection. In *Proceedings of the 6th International Workshop on the Recent Advances in Intrusion Detection (RAID'2003)*, LNCS v. 2820, pages 173–191, November 2003.

[5] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the IEEE Symposium on Research on Security and Privacy*, Oakland, CA, May 2002. IEEE Press.

[6] S. Rasoul and D. Landgrebe. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics*, Mars 1991.

[7] S. S. Skiena. *The algorithm design manual*. Springer-Verlag New York, Inc., 1998.

0-7695-2108-8/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY