Using CLIPS to Detect Network Intrusions

Pedro Alípio pal@isep.ipp.pt
Paulo Carvalho pmmc@di.uminho.pt
José Neves jmn@di.uminho.pt

Departamento de Informática
Universidade do Minho, Braga, Portugal

Abstract

We describe how to build a network intrusion detection sensor by slightly modifying NASA's CLIPS source code introducing some new features. An overview of the system is presented emphasizing the strategies used to inter-operate between the packet capture engine written in C and CLIPS. Some extensions were developed in order to manipulate time-stamps, multiple string pattern matching and certainty factors. Several Snort functions and plugins were adapted and used for packet decoding and preprocessing. A rule translator was also built to reuse most of the Snort's attack signatures. Despite some performance drawbacks, results prove that CLIPS can be used for real-time network intrusion detection under certain conditions. Several attack signatures using CLIPS rules are showed in the appendix. By mixing CLIPS with Snort features, it was possible to introduce flexibility and expressiveness to network intrusion detection.

1 Introduction

Network intrusion detection sensors analyze network traffic in real-time in search of malicious activities. As a malicious event is detected, adequate responses are issued in order to stop the intruder's activity or to alert the system's administrator. Several strategies have been used to build such a sensor, although the most usual strategy follows a signature-based approach [16]. A signature-based sensor uses an algorithm that compares captured packets to signatures of known malicious events. These type of algorithms have been evolving very quickly over the recent years. On the one hand, systems such as Snort [20], use efficient pattern-matching to detect events, and support more than a thousand rules without significant performance degradation. On the other hand, simple pattern-matching techniques do not provide the required logics to describe the complete attack scenario in an expressive and flexible manner.

Most of the signature-based network intrusion detection systems are packet oriented [1], which means they do not relate events in time nor maintain information about the network state. In such systems, it is impossible to create signatures for network events that require these features. The common solution has been the use of hard-coded plugins resorting to C language programming skills and knowledge about the network intrusion detection system architecture and data structures. Using plugins leads to a performance increase in the detection process, though when compared to knowledge based systems, there is a clear loss in flexibility. To overcome this limitation, knowledge based systems such as production systems can be used as a detection engine. In this paper, we describe how the general production system tool CLIPS [19], can be used for detecting network intrusions.

The packet capture engine was written in C based on the Snort packet decoder. This choice is convenient, as rules and plugins are continually being developed for Snort lightweight network intrusion detection system [20]. Furthermore, using the same packet data structure as Snort allows to adapt any of its preprocessor plugins easily. This is particularly useful for implementing features such as IP defragmentation, TCP stream reassembly, HTTP URI decode, and several attack responses, among other. The CLIPS language was extended with several functions to perform time-stamp manipulation, Snort like string pattern-matching, and other. By controlling CLIPS agenda, it is possible to introduce multiple string pattern-matching and certainty factors. Multiple string pattern-matching increases the system performance significantly when there are several rules testing the packet payload simultaneously [**?**]. Certainty factors may be very useful to identify false positive alarms. When the certainty factor of an issued alarm is too low, it is considered not reliable. In order to increase performace, tips to create efficient rulesets are also presented. By controlling system's inference using CLIPS modules, ruleset size can be reduced, further improving the overall performance of the real-time intrusion detection process. Finally, in the appendix, examples of attack signatures are also showed and explained.

2 Related Work

There are several contibutions focusing on the use of expert systems or knowledge based systems in computer systems intrusion detection (none of those used public domain expert system shells...).

P-BEST (Production-Based Expert System Toolset) has evolved from a lineage of intrusion detection projects, which include MIDAS [22], IDES [15], NIDES [3], and more recently the EMERALD eXpert [17]. P-BEST is a forward chaining expert system shell, which was applied for both network and host based intrusion detection.

The ASAX (Advanced Security and Audit Trail Analysis on UniX) project [11] uses a rule-based language called RUSSEL (Rule Based Sequence Evaluation Language), which provides a combination of procedural and rule-based programming to reason about activity on Unix systems by analyzing audit trails.

The University of California at Santa Barbara used a slightly different approach in USTAT (Unix State Transition Analysis Tool) [12]. Intrusions were detected using state transition diagrams modeling the sequence of operations and state changes that occur during the attack instead of production rules. A similar approach was used in the IDIOT (Intrusion Detection in Our Time) system. IDIOT's detection engine was also based on a graphical view of the malicious behavior, but it used colored petri-nets to model an intrusion signature [5].

Wisdom and Sense [23] and NADIR [13], both from Los Alamos National Laboratory, are further examples of knowledge based systems oriented to malicious activity detection. In the case of Wisdom and Sense, the anomaly detection component is also implemented as a rule-base. The signature analysis component is a combination of site-specific policies, expert penetration rules and other administrative data in the same rule-base. NADIR intrusion signature rule-base uses empirical data resulting from interviews with several security experts.

3 An overview of CLIPS

CLIPS (C Language Integrated Production System) was developed using C programming language at NASA/Johnson Space Center aiming at high portability, low cost, and easy integration with external systems.

CLIPS is a multiparadigm programming language providing support for rule-based, object-oriented, and procedural programming [10]. The inference engine algorithms and the knowledge representation provided by the rule-based programming language are similar, but more powerful than those used in OPS5 production system [8]. CLIPS rules syntax is similar to rule languages such as ART, ART-IM, Eclipse, and Cognate. Only forward chaining is supported. The object-oriented programming in CLIPS is called COOL (CLIPS Object Oriented Language), which combines features of common object-oriented languages, such as Smalltalk and Common Lisp Object System (CLOS), with some new ideas. The procedural programming language has features similar to C, PASCAL, ADA and others, but it is syntactically similar to LISP. CLIPS source code is publicly available for multiple platforms.

3.1 CLIPS basic components

Like any other expert system shell, CLIPS has tree basic components: a fact list containing data on which inferences are derived; a knowledge base which contains all the rules and an inference engine that controls the overall execution.

3.1.1 Facts

In order to resolve a problem, a CLIPS program must have data or information to reason about. A "chunk" of data is called a *fact*. Facts consists of a *relation name* (a symbolic field) followed by zero or more *slots* (also symbolic fields) and their associated values, as the following example illustrates:

```
(person (name “Pedro Alípio”)
        (age 29)
        (eye-color brown)
        (hair-color dark-brown))
```

Before facts can be created, CLIPS must be informed about which slots are valid for a given relation name. Templates are specifications of facts that share the same relation name and the same structure. A template for fact *person* could be defined as:

```
(deftemplate person “A person tem-
plate”
    (slot name)
    (slot age)
    (slot eye-color)
    (slot hair-color))
```

Slots can be specified as single values or multi-values by placing the keywork *multislot* instead of *slot*. Facts can be added, removed and modified with CLIPS commands *assert*, *retract* and *modify*.

3.1.2 Rules

An expert system needs rules to perform reasoning over facts. In production systems, a rule is defined as a set of conditional elements and a set of actions. If there is a matching in all the conditional elements of the left-hand side, the rule is placed in the agenda. When the inference engine selects a rule for firing from the agenda, the right-hand side is executed. A rule in CLIPS has the following syntax:

```
(defrule <rule name> [<comment>]
    <patterns>* ; Left-
Hand Side (LHS) of the rule
    =>
    <actions>*) ; Right-
Hand Side (RHS) of the rule
```

Patterns consist of constraints applied to the fact list. Those constraints can be specified on the fact's slots. When a full LHS matching occurs, by using ?*var_name* in the pattern slot value or $?*var_name* for multifield values, variables can be bound to fact's slots values. In the pattern:

```
(data (x ?x&:(> ?x 4)))
```

CLIPS binds variable *x* to slot *x* on every instance of *data* until it finds an instance where the value of variable *x* is greater than 4. These patterns are very useful since they are simultaneously a constraint and a variable binding. All the variables that have been bound on the LHS can be used in the rule's RHS.

3.1.3 Inference engine

CLIPS inference engine uses the forward chaining approach, which means that it reasons from facts to the conclusions resulting from those facts.

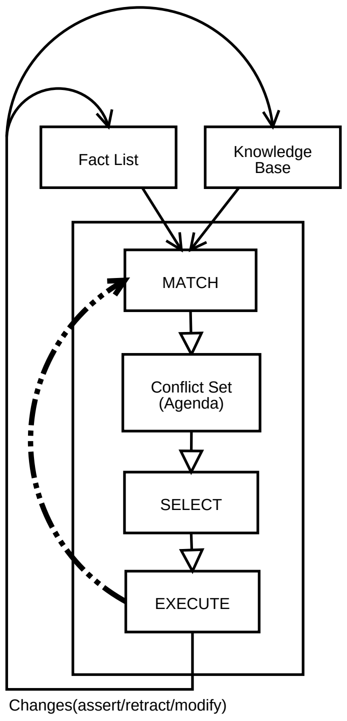

Figure 1: The inference process.

Figure 1 represents the inference process of production systems in general:

- The first step (*MATCH*) consists of updating the agenda by checking whether the LHSs of any rules are satisfied. If so, they are activated. Activations (or instantiations) are removed from the agenda if their LHSs are no longer satisfied.
- The second step (*SELECT*) consists of selecting a rule instantiation from those in the agenda, in order to execute the corresponding RHS. To perform this operation a conflict resolution criteria is needed. There are several strategies to resolve conflicts in the rule selection, either based in rule priorities (*salience* in CLIPS) or time-stamps (e.g. LEX and MEA).
- The third step (*EXECUTE*) consists on executing the RHS of the selected rule. The action nature can vary from fact list changes to regular functions or statements, such as input-output or even user-defined. If the fact list is altered, *MATCH* step is once again performed in order to determine if new activations are due.
- Finally, *SELECT* and *EXECUTE* steps are carried out until the agenda is empty.

To avoid the possibility of infinite loops, CLIPS exhibits a property called *refraction*, so that rules will not fire more than once for a specified set of facts.

To improve performance in the MATCH step the *Rete* algorithm is used [9]. The *Rete* algorithm is a fast pattern matcher which compiles information about rules in a network to obtain its speed. Instead of testing all rule's LHSs, it only looks for changes in matches on every cycle. This greatly speeds up the matching process since rules matching unaltered facts will not be tested.

4 Changing CLIPS to support network intrusion detection

CLIPS was designed to be embedded in other applications. When CLIPS is used as an embedded application the *main.c* file[1] must be changed to meet specific application requirements. This may include calling specific CLIPS statements without any user interaction, such as adding facts, reseting CLIPS, loading rules or running the system.

4.1 Structure of the sensor

The network intrusion detection sensor is conceptually divided in two major components: the packet capture engine and the CLIPS embedded expert system shell. Integration is carried out by decoding captured data into higher abstraction layers and passing it to CLIPS as facts. Figure 2 shows a more detailed view of the sensor's architecture.

[1]CLIPS's main file starts the environment, initializes user functions and calls the command line interpreter.

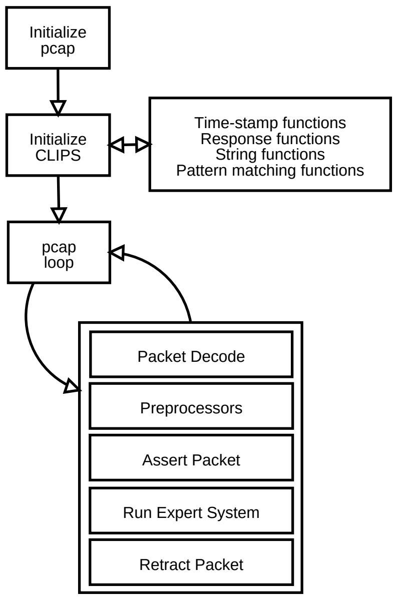

Figure 2: Sensor's structure.

The sensor is able to operate over two distinct data sources: real-time network traffic or raw packet files previously captured. If real-time network traffic data source is chosen, the packet capture engine needs to know on which network interface it will be active. BPF filters (Berckley Packet Filters) can also be applied to packets captured from both data sources.

CLIPS is then initialized. The initialization process consists of creating the CLIPS environment, initializing user functions, loading a configuration file and reseting the system. CLIPS environment are several data structures which require memory allocation and are used to maintain the system's current state. User functions (written in C) are mapped into CLIPS functions for several purposes such as packet time-stamp manipulation or pattern matching. The configuration file is used to configure the sensor, indicating which rule files and preprocessing plugins will be applied to captured packets.

Each time a packet is captured a packet handler is invoked. The packet handler decodes raw packets into higher level data structures. Preprocessor plugins are then applied to network traffic, modifying the contents of those data structures. Packet data is converted into a CLIPS's fact and asserted into the fact list. The ruleset is then invoked to perform the detection process. When CLIPS inference engine stops, the current packet is removed from the fact's list.

4.2 Packet Capture Engine

Network traffic is captured using *libpcap*, which is the packet capture library used in *tcpdump*[14]. This allows portability[2]; isolates the application from link technology; and allows the sensor to read raw packet data files captured by *tcpdump*. Other important advantage of using libpcap is that it provides an easy way of applying kernel packet filters (such as BPF) reducing the network traffic, increasing the sensor real-time performance [18].

Each time a traffic unit is captured a packet handler is called by *libpcap*. The packet handler function, among other operations, has to decode the raw packet into data structures representing the TCP/IP stack. To provide easy integration with Snort's preprocessor plugins and rules, its packet decode functions were adapted. A large data structure called *packet* is then used to integrate simultaneously each packet layer header and data in a higher level format for URIs, TCP options, payload and others.

4.3 Adding packet data to CLIPS

Packet data has to be added to CLIPS's fact list before being analyzed. A function which takes the decoded packet as an argument, performs type conversion form C language to CLIPS, so that API functions can be called to assert the packet fact. As it was discussed on section 3.1.1, a template has to be created before fact assertion. The packet fact template is not created invoking any CLIPS API functions. Instead, during the initialization process, the packet fact template is read and executed from *main.clp*.

```
(deftemplate MAIN::packet
  (slot proto) (slot nproto)
  (multislot timestamp)
  (slot srcaddr) (slot srcp)
  (slot dstaddr) (slot dstp)
  (slot flags) (slot ttl) (slot tos)
  (slot id) (slot ipopts)
  (slot fragbits) (slot seq)
  (slot ack) (slot itype)
  (slot icode) (slot icmp_id)
  (slot icmp_seq) (slot ip_proto)
  (slot fragoffset) (slot dsize)
)
```

In the template definition, *MAIN* stands for the CLIPS module, which will be explained later. Slots *proto*, *nproto* and *timestamp*, specify respectively the transport protocol, the network protocol and time-stamp which in turn is a multivalue slot containing two values: seconds and microseconds. The other slots specify packet information and have their equivalent in Snort's rule options [21].

[2]There is a windows clone of libpcap called *winpcap*.

4.4 Payload pattern matching

The template structure does not have any slot to store the packet payload data. There are two reasons why this data is not converted into a fact slot: the first is related to performance issues and the second is related to the implementation of multiple pattern matching. As payload data length maybe greater than one thousand bytes, passing it to a fact slot would be CPU an memory consuming. Payload single and multiple pattern matching is done using a test function instead of looking up for fact instantiations.

The LHS of rule can also test boolean functions. To avoid a performance decrease in looking for rule instantiations which match packet header constraints and a payload pattern, rules are written using both fact constraints and a boolean test function. Packet header data must be checked in the first place in order to reduce the number of rules that will check the packet payload contents. The payload pattern matching operation is then carried out checking a smaller set of patterns.

4.4.1 Single pattern matching

To perform single pattern matching to packet payload data, a boolean function called *payload* was added to CLIPS. This function takes one, two or three arguments. The first argument is the pattern that will be searched for. The second argument is the search offset in the packet payload area. The third argument is the search depth. Single pattern matching should only be used to test a small number of bytes, and not the whole packet payload data. As in Snort, the Boyer-Moore algorithm is used for single pattern matching[4].

4.4.2 Multiple pattern matching

Multiple pattern matching algorithm allows to simultaneously test several patterns against the packet payload, increasing the detection performance. The problem is how to do this using CLIPS rules. The solution was to add a user function called *m_payload* which takes the rule identification and the pattern to be tested as arguments. The rule identification has to be passed to the function because there is no way of knowing which rule is being tested. Recall that, in the Rete algorithm only when the leaf nodes are reached by the token (fact list change notification), the rules matching all their conditional elements are known. The pattern test is not carried out by the function. In fact, it adds the pattern to an array so that later on, when the whole conflict set is known, all patterns in the array can be simultaneously tested with Aho-Corasick Algorithm[2]. The activations without none matching patterns are removed from the agenda. The following algorithm describes how the multiple pattern matching process is accomplished:

```
function AhoCorasic(patternArray,data)
for every activation in agenda
  pattern←getPattern(activation,
                     patternArray)
  if pattern exists then
    AC_addPattern(pattern)
end for
while matchId←AC_search(data)
  activation←getActivation(matchid,
                           patternArray)
  mark activation as do not remove
end while
for every pattern in patternArray
   activation←pattern.activation
   if activation is not marked then
     remove(activation)
   end if
end for
```

The algorithm starts by checking if for each rule in the agenda there is a correspondent entry in the pattern array. If the entry exists, the pattern has to be added to the Aho-Corasick's pattern list. After the pattern list is built, *AC_search* function looks up for a matching. The associated activation is marked so that it will not be removed. These last two operations are repeated while there is a matching pattern. All the unmarked activations associated with patterns in the pattern array are removed from the agenda.

4.5 Certainty factors

Certainty factors are used as a measure of thrust in the generated alarms. There are two major sources of uncertainty in rule based systems: rule contradiction and rule subsumption. Rule contradiction occurs in the presence of two or more rules with the same LHS and with different RHSs. Rule subsumption occurs when a rule LHS is a subset of another rule LHS.

On a first approach, let $CF = \frac{1}{n}$ be the expression defining a certainty factor *CF*, where *n* is the number of rules in conflict on the inference's engine last cycle. If there is just one rule on the agenda, the certainty factor will be one hundred percent, otherwise certainty will decrease as the number of rules in conflict increases. The problem with this expression is that it does not consider subsumption. A rule subsumes another rule if both are in the agenda and one is more complex than the other. Rule complexity is evaluated by counting facts constraints and tests on the LHS of a rule. The expression $CF_i = \frac{C_i}{n * C_{Max}}$ expresses the certainty factor of rule *i* on both rule contradiction and rule subsumption, i.e, *n* reflects the number of rules in conflict, C_i the complexity of the rule *i* and C_{Max} the maximum complexity of all rules in the agenda. The certainty factor value will be lower for rules with less complexity, i.e. specific rules are more reliable than general ones.

To implement this feature a *callback* function is required. As soon as the conflict set is obtained, a function has to be called to evaluate the maximum complexity and the number of rules in the agenda.

4.6 The agenda callback function

CLIPS has a way of calling a function each time a rule fires. This is particularly interesting to implement multiple pattern matching and certainty factors, since an agenda callback function is required. The CLIPS's API function *AddRunFunction* takes the function to be called whenever a rule is fired as an argument. Unfortunately, this is not enough to create the agenda callback function because the rule RHS is executed, which is not convinient. The solution is to create a special rule in file *main.clp*, which tells the inference engine to set the focus on a module called *EVENTS* each time a packet is detected.

```
(defrule MAIN:got_traffic "Analyser got a packet"
     (packet)
=>
     (focus EVENTS)
)
```

Module *EVENTS* is where the event signatures are defined. When rule *got_traffic* is fired, the agenda callback function is automatically invoked. The following pseudocode defines the algorithm for the callback function:

```
function agendaCallback
if length(PayloadPatterns)>0 then
```

```
  AhoCora-
sick(PayloadPatterns,Payload)
end if
if length(UriPatterns)>0
  PacketHas(Uri) then
    AhoCorasick(UriPatterns,Uri)
end if
if there are new activations then
  n←0
  for every activation in agenda
    if complexity(activation)>MaxC
    then
      MaxC←complexity(activation)
      n←n+1
    end if
  end for
end if
end function
```

The first two if structures verify if there are entries in the pattern arrays. There are two arrays for patterns: one to store information about packet payload data and the other to store information about HTTP URIs (Uniform Resource Identifiers). Multiple pattern matching is performed only if there are patterns in the arrays. In the URI case, an extra test is made to verify if the current packet has any URI. The number of activations and the maximum complexity must be calculated only when there are new rules in the agenda. If no rules were added to the agenda, recalculating the maximum complexity and the number of activations is not required. If this condition was not tested, the last rule to be fired would always have a certainty factor of one hundred percent because the number of activations would be one and the maximum rule complexity would be the rule's complexity.

5 Writing CLIPS's rules for network intrusion detection

CLIPS features, such as modules, can be used to write well structured intrusion detection signatures. Defining the ruleset as a hierarchical structure reduces the number of rules to be checked each time a packet arrives. For example, if a TCP packet is to be analyzed, it does not make sense to apply UDP signature rules to the packet. Therefore, a top level module called *MAIN* is used to define global variables, the packet template and a rule focusing on module *EVENTS* each time a packet is detected.

In module *EVENTS*, actually the main module for all intrusion signatures, the ruleset is divided into several modules according to TCP/IP protocols. If the captured packet is TCP, UDP or ICMP the inference engine should focus on corresponding module *TCP_EVENTS*, *UDP_EVENTS* or *ICMP_EVENTS*. Three levels were defined so far: level 1, with module *MAIN*; level 2, with module *EVENTS* and level 3, with modules *TCP_EVENTS*, *UDP_EVENTS* and *ICMP_EVENTS*.

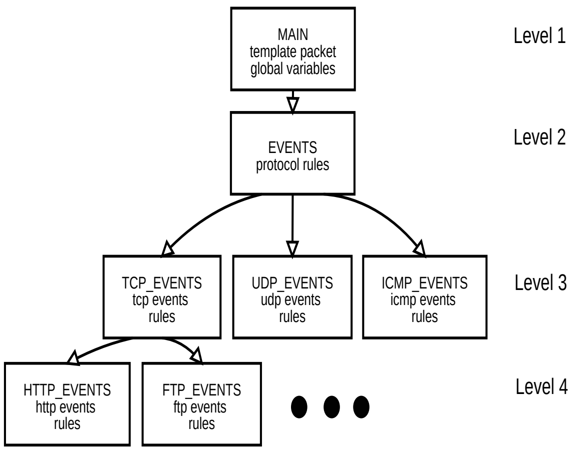

Figure 3: Module's structure for signatures rulesets.

One extra level containing rules applied to the application layer protocols was also defined to further reduce the rulesets size (see Figure 3). If the packet is HTTP, it does not make sense to apply FTP rules to it. So, on level 3 several rulesets can be specified according to the packet application protocol. The HTTP application protocol usually takes port 80, but if a proxy server is in use, port 3128 or 8080 can be used instead. Other applications use transient ports, therefore port numbers cannot be used for explicitly divide a ruleset. Transparency is obtained through the definition of global variables. Different configurations can be applied depending on the sensor's location in the network, by just changing those variables.

For example, the variables to define HTTP and FTP ports, could be written using CLIPS statement *defglobal*:

```
(defglobal MAIN
        ?*http_port* = 3128
        ?*ftp_port* = 21
)
```

where the first argument is the module's name. Variables are then defined and initialized.

Global variables in CLIPS are only visible within the module in which they are defined. When defining a module, all variables or templates to be imported or exported need to be specified. For example, in module *MAIN*, the template *packet* and all global variables have to be exported so that they can be used by the other modules. Module *MAIN* is defined by:

```
(defmodule MAIN
      (export deftemplate ?ALL)
      (export defglobal ?ALL)
)
```

In module MAIN definition, ?ALL is used to export all defined templates and variables. The following piece of code defines module *TCP_EVENTS* importing template *packet* and all global variables from module MAIN.

```
(defmodule TCP_EVENTS
   (import MAIN deftemplate packet)
   (import MAIN defglobal ?ALL)
)
```

As an example, a ruleset for FTP events must have several rules to express the modules structure. The first rule was already defined in section 4.6. All the remaining rules are:

```
(defrule EVENTS::tcp
 (packet (proto tcp))
=>
 (focus TCP_EVENTS)
)
(defrule TCP_EVENTS::ftp_event1
"ftp to server”
(packet (dstp ?dp&:
   (= ?dp ?*ftp_port*)))
=>
 (focus FTP_EVENTS)
)
(defrule TCP_EVENTS::ftp_event2
 "ftp to client"
 (packet (srcp ?sp&:
    (= ?sp ?*ftp_port*)))
=>
 (focus FTP_EVENTS)
)
```

The first rule is included in module *EVENTS* and is fired each time a TCP packet is detected. When it fires, module *TCP_EVENTS* gets the focus. In this module, two rules are required to describe FTP events. The first one detects a packet sent to an FTP server, while the second detects a packet sent to an FTP client. The constraints used compare the source and destination ports with a previously declared global variable. In the lowest level module, rules check only event specificities.

This hierarchical module structure leads to a performance increase in real-time network intrusion detection, though some modules may tend to be heavy. For instance, the *HTTP_EVENTS* module may have a large number of rules performing packet payload analysis.

6 Translating Snort rules into CLIPS rules

Why is it so important to translate Snort rules? Snort is probably the most well supported network intrusion detection system. Being a high performance lightweight open-source tool with a large number of rules, running on several platforms fostered its popularity. Consequently, the number of Snort’s contributions has been increasing, and currently there are 1868 rules defined.

The CLIPS network intrusion detection sensor was developed using some code from Snort, so that snort rules and plugins could be easily integrated. To translate Snort rules into CLIPS rules a program using *flex* and *bison* parser tools was built. The translator program reads a Snort ruleset file and converts it to a CLIPS ruleset.

6.1 Snort rules

Each snort rules has a *rule header* and *rule options*. The rule header contains the action, the protocol, the source and destination IP addresses and netmasks, and the source and destination ports. Rule options are implemented as plugins, therefore new options can easily be added to snort. These consist of a test to a packet header field, a payload pattern matching search, response directives or attack classification information. A Snort rule for a failed FTP login is:

```
alert tcp $HOME_NET 21 -
> $EXTERNAL_NET any
(msg:“FTP failed login”; flags:A+;
content “Login incorrect”;sid:100)
```

Remember that the packet template was created so that Snort packet header options could be mapped into CLIPS’s rule patterns (see section 4.3).

6.2 Converting rule headers

In CLIPS network intrusion detection sensor, there is not a rule header. Therefore, rule header components such as protocol, source and destination IP addresses and netmasks, and source and destination ports, are converted into pattern constraints applied to the *packet* fact. The last example of rule header would be converted into the following:

```
(defrule SNORT_EVENTS::s100
  (packet
    (proto tcp)
    (srcaddr ?sa) (dstaddr ?da)
    (srcp 21) (dstp ?dp)
```

A special module called SNORT_EVENTS is defined to contain the converted rules. Rule identification in CLIPS is created resorting to the signature identification option (*sid*). The Snort's rule protocol is mapped directly into *packet*'s slot *proto*. Source and destination IP addresses, and destination port need to be mapped into variables as they will be used in the RHS of the rule. The source port is converted into a constraint over the *srcp* slot.

6.3 Converting rule options

Rule options are converted mapping Snort options into constraints applied to the *packet* fact slots. Some of these constraints may be considerably complex. For example, Snort *flags* option besides describing the flags pattern may use logical operators, e.g.: + (ALL flag), match on all specified flags plus any others; * (ANY flag), match on any of the specified flags; ! (NOT flag), match if the specified flags are not set in the packet. To translate this complex option patterns, CLIPS's predicate function *check* was built. Snort's *content* option is converted into multiple pattern matching when no offset or depth are defined. For the example in section 6.2, the complete translated rule is:

```
(defrule SNORT_EVENTS::s100
  (packet
    (proto tcp)
    (srcaddr ?sa) (dstaddr ?da)
    (srcp 21) (dstp ?dp)
    (flags ?fl&:(check ?fl "A+")))
  (test (m_payload
          SNORT_EVENTS::s100
           "Login incorrect." ))
=>
  (printout t " FTP failed login to " ?sa
      " from " ?da ":" ?dp crlf)
  (printout t "cf=" (get_rule_cf) crlf))
```

Rules	Fired Rules	Alarms	Packet Drop
114	2616	1740	78%
1	3655	743	27%

Table 1: ICMP test results - Testing alarms performance

7 Real-Time performace results

Several tests were performed to measure the sensor's real-time performance. The main goal was to find how many rules could be supported by the sensor without packet drops. A real-time network intrusion detection sensor should not discard any packet. Packets may pass by the sensor without being analyzed creating a *false-negative* state. The number of *false-positive* alarms issued by the sensor is usually used as a performance measure [1]. Although this measure is not relevant to be applied to this sensor because those alarms depend on the signature's heuristics quality, and not on the system. Thus, measuring *false-positive* alarms is more oriented to less flexible sensors, with hard-coded rules. Here, the the results are expressed as a packet drop ratio resulting from measurements involving a Pentium III 700Mhz laptop, with 128MBytes of RAM, on a 10Mbps ethernet network.

7.1 Stateless rules

A first set of tests was performed using the *ping* utility with options *-f* (flood) and *-c* (count). The main goal of this test was to find out which were the sensor's major sources of performance degradation when applying simple heuristics, i.e., without state information (snort like rules). One thousand ICMP Echo packets were sent to a host running the CLIPS Network Intrusion Detection Sensor. These results are not specific for ICMP traffic as the rules in use have a similar structure to TCP and UDP rules (both test the packet header fields and payload data).

Table 1 compares the results for *icmp* and *icmp-info* snort rulesets with the results using only one rule that fires every time an ICMP ping is detected. The results (first row in table) show that using complete snort rulesets leads to a major performance decrease due to existing several rule instances firing simultaneously. In Snort, only the first rule matching its conditional elements is fired. The second row shows that using only one rule the packet drop ratio is still high (27%). The reason of such a low performance is related to the alarm function exe-

Rules	Fired Rules	Packet Drop
107	2970	26%
105	3156	21%
103	3700	7%
101	3862	3%
99	4000	0%

Table 2: ICMP test results - removing payload pattern matching rules

Rules	Fired Rules	Packet Drop
107	2970	26%
100	3084	23%
23	3152	21%

Table 3: ICMP test results - removing rules with no payload pattern matching

cution. The results also show that the number of fired rules and the number of generated alarms are distinct. This is because of module focus rules firing and due to the fact that payload pattern matching rules are always fired by CLIPS inference engine. Only then, when the agenda callback function is invoked, the non-matching rules instances are removed.

Table 2 compares *icmp* and *icmp-info* snort rulesets by removing rules which check the packet payload. From an initial ruleset of 114 rules, 7 rules were removed so that none would fire. There are 23 rules using payload pattern matching in their LHSs. The first row in Table 2 shows a packet drop of 26% when 107 rules are applied. By removing 2 more rules, the packet drop was 21%, and with a ruleset of 105 that value dropped down to 7%. By removing 2 more rules, packet drop was 3%. For 99 rules, there is no packet dropping. Table 3 shows the impact on packet loss, when removing rules which do not perform any packet payload pattern matching. By removing 7 rules without any payload pattern matching to the 107 rules, the packet drop was 23%. After removing a large set of rules (77) with no payload tests, the packet drop was 21%.

There are two major reasons for performance degradation. The first one is related to the alarm function execution, where its execution time interval is responsible for several packets not to be analyzed. The other one is related to the payload pattern matching process. Even using a fast multiple pattern matching algorithm[2], the performance bottleneck remains. Reducing the ruleset by removing non payload pattern matching rules from 107 to 23 rules (those that perform payload pattern matching), there is only a 5% reduction in packet drop ratio. Removing 8 payload pattern matching rules we obtained 0% of packet drop. This shows that the sensor supports about 15 (23-8) rules with payload pattern matching simultaneously in the agenda.

Attack	Max Rst	Alarms	Packet drop
Nmap	100	8	39%
Nmap	500	2	24%
Nmap	1000	1	6%
None	100	0	0%

Table 4: Port scan test

7.2 Statefull rules

Statefull rulesets use the forward chaining mechanism creating several states in the detection process. Each state is an inference cycle. It is important to know the sensor's performace using such a ruleset. As a test example, a ruleset to detect a TCP *syn scan* was created. Some utility tools were used to generate the traffic samples. *Nmap* port scanner produced the TCP *syn scan*. *Tcpreplay* was used to send arbitrary TCP traffic. *Tcpreplay* reads a *Tcpdump* raw packet file to inject traffic into a network interface. The normal TCP traffic sample was generated using the *iperf* tool.

Two distinct scenarios were considered: testing the packet drop ratio when the *Nmap* tool is used to perform a port scan to 1525 ports in a 10Mbps ethernet network; testing the packet drop rate when applying the port scan ruleset to a normal traffic pattern.

The port scan ruleset detects a TCP *syn scan* by counting the number of reset connections from a certain source to a destination host within a specified interval. Two variables had to be defined: the maximum reset connections and the time interval (set by default to 60 seconds).

The values in Table 4 result from applying different configurations to the port scan heuristics. Note that as the maximum TCP reset packets allowed is increased, the packet drop ratio decreases. There are two reasons for that. One is the number of generated alarms. As discussed before, alarm issuing is an operation responsible for significant performance degradation. The other is that by allowing a larger number of resets, the test to verify if those reset packets timestamp are within the specified time interval is executed less times, which means that, most of the time only one cycle is performed by the inference loop. By applying TCP *syn scan* rules to normal TCP traffic, there is no performance degradation due to the low number of reset connections in a normal traffic pattern, causing less rules to fire.

8 Conclusions

Building a low cost network intrusion detection sensor by using public domain tools and libraries is a possible and feasible solution. In this paper, an example of such a sensor, developed using NASA's CLIPS production system shell and adapting Snort code and data structures is proposed. The sensor supports complex heuristics, keeping event related information in a fact list and uses forward chaining reasoning to reach a conclusion. If several conclusions are reached, certainty factors are used to assess those conclusions' credibility. Multiple pattern matching was used to improve the payload analysis performance. A Snort ruleset translator was built so that those rulesets could be used by the sensor. Unfortunately, there are still some performance bottlenecks. An inference engine is much more complex than a simple pattern matching engine. More efficiency could be obtained by changing the CLIPS inference engine to support built-in multiple pattern matching. Although CLIPS is not a high performance tool oriented to real-time use, it works fine with previous Tcpdump captured traffic.

Future work includes improving the sensor's performance by using a hybrid algorithm for multiple pattern matching as in [**?**]. CLIPS may be extended just by mapping C language functions into CLIPS functions. Interoperability features can be added by building user functions for IDMEF message manipulation [6] and IDXP protocol implementation so that the alarms can be exchanged with other Intrusion Detection Systems[7].

References

[1] The evolution of intrusion detection technology. http:// documents.iss.net/ whitepapers/ TheEvolutionofIntrusionDetectionTechnology.pdf, August 2001.

[2] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–343, June 1975.

[3] D. Anderson, T. Frivold, and A. Valdes. Next-generation intrusion detection expert system (NIDES). Technical Report SRI-CSL-95-07, Computer Science Laboratory, SRI International, 1995.

[4] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, Oct. 1977.

[5] Mark Crosbie, Bryn Dole, Todd Ellis, Ivan Krsul, and Eugene Spafford. IDIOT - User Guide. Technical report, September 1996.

[6] D. Curry and H. Debar. Intrusion Detection Message Exchange Format, Data Model and Extensible Markup Language (XML). http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-07.txt, June 20 2002. (work in progress).

[7] B. Feinstein, G. Matthews, and J. White. The Intrusion Detection Exchange Protocol (IDXP). http://www.ietf.org/internet-drafts/draft-ietf-idwg-beep-idxp-05.txt, June 17 2002. (work in progress).

[8] C. L. Forgy. OPS5 User's Manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, Dept. of Computer Science, 1981.

[9] Charles Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelegence*, (19):17–37, 1982.

[10] Joseph C. Giarratano. *CLIPS User's Guide, Volume I - Basic Programming Guide*, March 31st 2002.

[11] Naji Habra, Baudouin Le Charlier, Abdelaziz Mounji, and Isabelle Mathieu. ASAX : Software Architecture and Rule- Based Language for Universal Audit Trail Analysis. In *European Symposium on Research in Computer Security (ESORICS)*, pages 435–450, 1992.

[12] Koral Ilgun. USTAT: A real-time intrusion detection system for UNIX. In *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*, pages 16–28, Oakland, CA, 1993.

[13] K. A. Jackson, D. H. DuBois, and C. A. Stallings. An expert system application for network intrusion detection. In *Proceedings of the 14th National Computer Society Conference*, pages 215–225, Washington, D.C., Oct.

1-4 1991. National Institute of Standards and Technology/National Computer Society Center.

[14] V. Jacobson, C. Leres, and S. McCanne. tcpdump. www.tcpdump.org.

[15] H. S. Javitz and A. Valdes. The SRI IDES statistical anomaly detector. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 316–326, Oakland, California, May 20-22 1991. IEEE Computer Society Press.

[16] Sandeep Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue, IN, 1995.

[17] Ulf Lindqvist and Phillip A. Porras. Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy, Oakland, California*, 1999.

[18] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the 1993 Winter USENIX Conference*, San Diego, CA, 1993.

[19] Gary Riley. CLIPS A tool for building expert systems. http:// www.ghg.net/ clips/ CLIPS.html.

[20] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of LISA '99: 13th Systems Administration Conference*, 1999.

[21] Martin Roesch. *Snort Users Manual*. http://www.snort.org, snort release: 1.9.x edition, April 26th 2002.

[22] M. M. Sebring, E. Shellhouse, M. E. Hanna, and R. A. Whitehurst. Expert systems in intrusion detection: A case study. In *Proceedings of 11th National Computer Security Conference*, pages 74–81, Baltimore, Maryland, Oct. 17-20 1988. National Institute of Standards and Technology/ National Computer Security Center.

[23] H. S. Vaccaro and G. E.Liepins. Detection of anomalous computer session activity. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 280–289, Oakland, California, May 1-3 1989. IEEE Computer Society Press.

Appendix

A Heuristics for TCP *syn scan*

```
;**********************
;   TCP - PORT SCANS  *
;**********************
(defglobal TCP_EVENTS
       ?*maxRsts* = 50
       ?*maxRstsInterval* = 60
)
(deftemplate TCP_EVENTS::rst
       (multislot ts)
       (slot src)
       (slot dst)
       (slot count (default 0))
)
(defrule TCP_EVENTS::rst_counter_init
   ?fid<-(packet (proto tcp) (timestamp $?tm1) (srcaddr ?sa)
             (dstaddr ?da)
             (flags ?fl&:(check ?fl "R*")) )
   (not (rst (src ?sa) (dst ?da) (ts $?tm2) (count ?c) ))
=>
   (assert (rst (src ?sa) (dst ?da) (ts $?tm1) (count 1)))
   (retract ?fid)
)
(defrule TCP_EVENTS::rst_counter_count
   ?fid<-(packet (proto tcp) (timestamp $?tm2) (srcaddr ?sa)
             (dstaddr ?da)
             (flags ?fl&:(check ?fl "R*")) )
   ?fid2<-(rst (src ?sa) (dst ?da) (ts $?tm1) (count ?c))
   (test (between $?tm2 $?tm1
         (ts_add $?tm1 (create$ ?*maxRstsInterval* 0)))
   )
=>
   (modify ?fid2 (count (+ ?c 1)))
   (retract ?fid)
)
(defrule TCP_EVENTS::port_scan_detected
   ?fid<-(rst (src ?sa) (dst ?da) (ts $?tm1) (count ?c&:(> ?c ?*maxRsts*)))
=>
   (retract ?fid)
   (alarm $?tm1 ?da n ?sa n "TCP Syn scan" "Scan" (get_rule_cf)
      (create$ "http://www.insecure.org"))
)
```

TCP syn scan is a port scan technique often referred to as "half-open" scanning, because it does not open a full TCP connection. It sends a *syn* packet, as if it was opening a real connection, and waits for a response. A *syn-ack* indicates an active port. A *rst* is indicative of a non-listener. If a *syn-ack* is received, a *rst* is immediately sent to tear down the connection (actually the OS kernel does this). The primary advantage to this scanning technique is that few sites will log it.

The heuristic to detect TCP syn scan is very simple. There are two threshold variables corresponding to the maximum number of resets and the maximum time interval length. The number of resets within the interval are counted for a certain source and destination. If the counter exceeds the maximum number of resets, a TCP syn scan is assumed by the sensor and an alarm is generated.

B Heuristics for Brute Force FTP Attack

```
(defglobal FTP_EVENTS
        ?*maxFailedLogins* = 3
        ?*maxFailedLoginsInterval* = 60
)
(deftemplate FTP_EVENTS::failed_logins
        (multislot ts)
        (slot src)
        (slot dst)
        (slot count (default 0))
)
(defrule FTP_EVENTS::failed_ftp_login_count_init
        ?fid<-(packet (proto tcp) (srcaddr ?sa) (dstaddr ?da) (dstp ?dp)
                        (srcp 21) (flags ?fl&:(check ?fl "PA")))
        (test (m_payload failed_ftp_login_count_init "Login incorrect." ))
=>
        (assert (failed_logins (src ?sa) (dst ?da) (ts $?tm1) (count 1)))
        (retract ?fid)
)
(defrule FTP_EVENTS::failed_ftp_login_count
        ?fid<-(packet (proto tcp) (srcaddr ?sa) (dstaddr ?da) (dstp ?dp)
                        (srcp 21) (flags ?fl&:(check ?fl "PA"))
        ?fid2<-(failed_logins (src ?sa) (dst ?da) (ts $?tm1) (count ?c))
         (test (between $?tm2 $?tm1
                (ts_add $?tm1 (create$ ?*maxFailedLoginsInterval* 0)))
         (test (m_payload failed_ftp_login_count "Login incorrect." ))
=>
         (modify ?fid2 (count (+ ?c 1)))
         (retract ?fid)
)
(defrule FTP_EVENTS::ftp_brute_force
        ?fid<-(failed_logins (src ?sa) (dst ?da) (ts $?tm1)
                (count ?c&:(> ?c ?*maxFailedLogins*)))
=>
        (retract ?fid)
        (alarm $?tm1 ?da n ?sa n "FTP brute force attempt" "FTP"
            (get_rule_cf)
            (create$ none)
        )
)
```

A FTP brute force attack is an attempt of illegal access to an FTP server by trying several combinations of passwords. The heuristic to detect it is very similar to the TCP syn scan heuristic. There are two variables: maximum tolerated failed logins and the time interval length. If the number of failed logins within the interval exceeds the maximum value allowed, an alarm is issued.

C Example of rules translated from Web-cgi Snort ruleset

```
; RULE to detect "WEB-CGI HyperSeek directory traversal attempt"
(defrule SNORT_EVENTS::s803
   (packet (proto tcp) (timestamp $?ts) (srcaddr ?sa) (dstaddr ?da)
            (srcp ?sp) (dstp 80) (flags ?fl&:(check ?fl "A+"))  )
   (test (m_payload s803 "%00"))
   (test (m_uri_check s803 "/hsx.cgi"))
=>
   (alarm $?ts ?sa ?sp ?da $s
      "WEB-CGI HyperSeek directory traversalattempt"
       "web-application-attack" (get_rule_cf)
       (create$ "cve,CAN-2001-0253" "bugtraq,2314" )
   )
)
; RULE to detect "WEB-CGI SWSoft ASPSeek Overflow attempt"
(defrule SNORT_EVENTS::s804
     (packet (proto tcp) (timestamp $?ts) (srcaddr ?sa) (dstaddr ?da)
               (srcp ?sp) (dstp 80) (flags ?fl&:(check ?fl "A+"))
               (dsize ?ds&:(> ?ds 500))
      )
     (test (m_payload s804 "tmpl="))
     (test (m_uri_check s804 "/s.cgi"))
=>
     (alarm $?ts ?sa ?sp ?da $s
             "WEB-CGI SWSoft ASPSeek Overflow attempt"
             "web-application-attack" (get_rule_cf)
             (create$ "bugtraq,2492" )
     )
)
; RULE to detect "WEB-CGI webspeed access"
(defrule SNORT_EVENTS::s805
   (packet (proto tcp) (timestamp $?ts) (srcaddr ?sa) (dstaddr ?da)
             (srcp ?sp) (dstp 80) (flags ?fl&:(check ?fl "A+"))
   )
   (test (m_payload s805 "WSMadmin"))
   (test (m_uri_check s805 "/wsisa.dll/WService="))
=>
   (alarm $?ts ?sa ?sp ?da $s
            "WEB-CGI webspeed access"
            "attempted-user" (get_rule_cf)
            (create$ "arachnids,467" )
   )
)
; RULE to detect "WEB-CGI yabb access"
(defrule SNORT_EVENTS::s806
   (packet (proto tcp) (timestamp $?ts) (srcaddr ?sa) (dstaddr ?da)
             (srcp ?sp) (dstp 80) (flags ?fl&:(check ?fl "A+"))
   )
   (test (m_payload s806 "../"))
   (test (m_uri_check s806 "/YaBB.pl"))
=>
   (alarm $?ts ?sa ?sp ?da $s
             "WEB-CGI yabb access"
```

```
                 "attempted-recon" (get_rule_cf)
                 (create$ "arachnids,462" )
    )
)
; RULE to detect "WEB-CGI wwwboard passwd access"
(defrule SNORT_EVENTS::s807
    (packet (proto tcp) (timestamp $?ts) (srcaddr ?sa) (dstaddr ?da)
                 (srcp ?sp) (dstp 80) (flags ?fl&:(check ?fl "A+"))
    )
    (test (m_uri_check s807 "/wwwboard/passwd.txt"))
=>
    (alarm $?ts ?sa ?sp ?da $s
                "WEB-CGI wwwboard passwd access"
                "attempted-recon" (get_rule_cf)
                (create$ "bugtraq,649"
                              "cve,CVE-1999-0953"
                              "arachnids,463"
                )
    )
)
```

The current number of rules translated is over 1200. Only few are showed hero so that they can be syntactically compared with other network intrusion detection systems rules.

D The *main.clp* file

```
(defmodule MAIN
        (export deftemplate ?ALL)
)
(deftemplate MAIN::packet
        (slot proto (default ""))
        (slot nproto (default ""))
        (multislot timestamp (default 0 0))
        (slot srcaddr (default ""))
        (slot srcp    (default 0))
        (slot dstaddr (default ""))
        (slot dstp    (default 0))
        (slot flags   (default "0"))
        (slot ttl     (default 0))
        (slot tos     (default 0))
        (slot id      (default 0))
        (slot ipopts  (default ""))
        (slot fragbits (default ""))
        (slot seq     (default 0))
        (slot ack     (default 0))
        (slot itype   (default 0))
        (slot icode   (default 0))
        (slot icmp_id (default 0))
        (slot icmp_seq (default 0))
        (slot ip_proto (default 0))
        (slot fragoffset (default 0))
        (slot dsize     (default 0))
)
(defrule MAIN:got_traffic "Analyser received a traffic unit"
        (declare (salience 3000))
        (packet)
=>
        (focus EVENTS)
)
```

This is the main rule file. A template specifying the packet structure and a special rule which fires each time a packet is detected by the analyzer are defined. CLIPS agenda callback function is invoked whenever a rule fires. This special rule (*got_traffic*) was built to force a rule firing so that all the activations in conflict could be known.

E The configuration file *conf.clp*

```
; sets the ip defrag preprocessor (based on the snort preprocessor)
(frag2)
; sets the URI preprocessor (based on the snort preprocessor)
(http_decode_ports "80,3128")
; sets multipattern matching case insensitive
(set_mp_matching_nocase)
; Main rule file
 (load "main.clp")
; Protocol structured rules
(load "events.clp")
(load "icmp_events.clp")
(load "tcp_events.clp")
(load "ftp_events.clp")
(load "http_events.clp")
; TCP syn scan
(load "port_scan.clp")
; Snort translated rules
(load "snort_events.clp")
(load "ddos.clp")
(load "dos.clp")
(load "dns.clp")
(load "exploit.clp")
(load "misc.clp")
(load "policy.clp")
(load "rpc.clp")
(load "scan.clp")
(load "shellcode.clp")
(load "tftp.clp")
(load "icmp-info.clp")
(load "icmp.clp")
(load "backdoor.clp")
(load "finger.clp")
(load "info.clp")
(load "netbios.clp")
(load "rservices.clp")
(load "smtp.clp")
(load "sql.clp")
(load "telnet.clp")
```

```
(load "virus.clp")
(load "porn.clp")
(load "x11.clp")
(load "ftp.clp")
(load "web-misc.clp")
(load "web-attacks.clp")
(load "web-cgi.clp")
(load "web-coldfusion.clp")
(load "web-frontpage.clp")
(load "web-iis.clp")
(load "web-iis2.clp")
```

This file defines which preprocessors and rulesets will be used by the sensor.