Chapter 2
Background

2.1 A Popular NIDS: Snort

A popular open-source [10] (under the GNU license) NIDS, Snort, is a freely available, lightweight NIDS that is configured with a list of rules, each defining a signature and a corresponding event log description. Snort also has a plug-in architecture that allows analysis to be performed and is currently used by pre-processors to avoid some intended attacks. Snort is widely used and there is a large, freely accessible and up-to-date database of signatures available on the Internet. The Snort system is logically divided into multiple components [11], as shown in Table 2-1. These components work together to detect attacks and to generate output from the detection system, called the “Detection Engine”.

Table 2-1. Descriptions of all Snort components

Name	Description
Packet decoder	Prepares packet for processing.
Pre-processors or Input Plug-ins	Used to normalize protocol headers, to detect anomalies (payload string, URL, RE), to packet reassembly and TCP stream re-assembly.
Detection Engine	Applies rules to packets.
Logging and Alerting System	Generates alert and log messages.
Output Module	Processes alerts and logs.

The organization of these components is shown in Figure 2-1 [12]. Any data packet coming from the Internet enters the packet decoder. While making its way towards the output modules, the data packet is either dropped, logged or an alert is

generated.

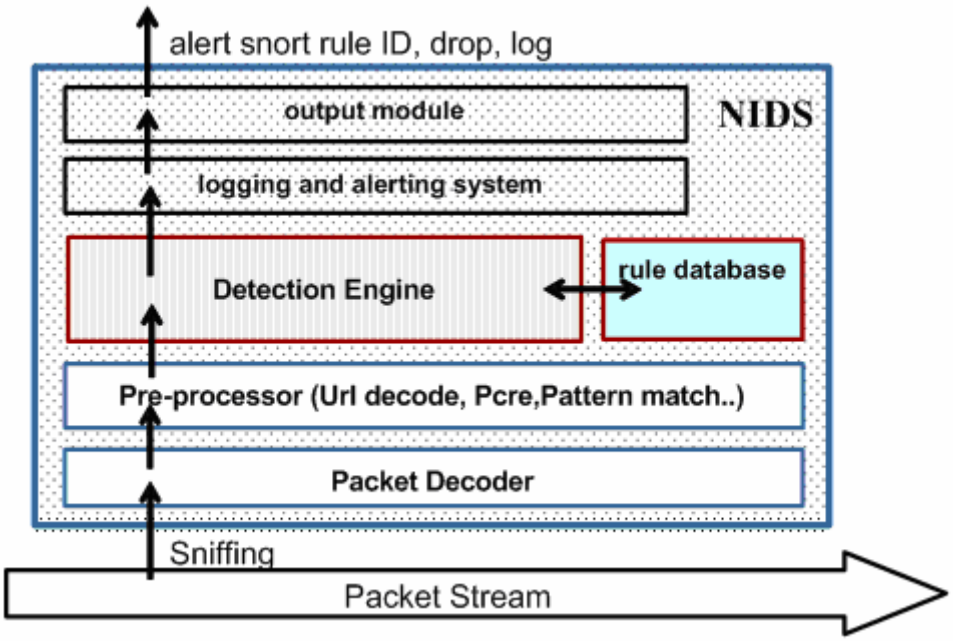


Figure 2-1. The original architecture of NIDS

Packet Decoder:

The packet decoder takes packets from different types of network interfaces and prepares the packets to be preprocessed or to be sent to the detection engine. The interface may be Ethernet, SLIP or PPP, among others.

Preprocessors

Preprocessors are components or plug-ins that can be used with Snort to arrange or modify data packets before the Detection Engine attempts to recognize whether the packet is being used by an intruder. Hackers use different techniques to fool the IDS in different ways. For example, you may have created a rule to find a signature “scripts/iisadmin” in HTTP packets. This can easily be subverted by a hacker who makes even a slight modification to this string. For example: [13]

1. “scripts/./iisadmin”
2. “scripts/examples/../iisadmin”

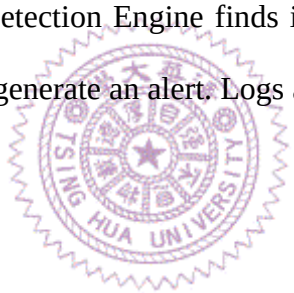
3. “scripts\iisadmin” …

To complicate the situation, hackers can also insert web Uniform Resource Identifier (URI) hexadecimal characters or unicode characters, which the web server sees as perfectly legal. Note that the web servers usually understand all these strings and are able to preprocess them to extract the intended string “scripts/iisadmin”. A preprocessor can rearrange the string so that it is detectable by the IDS.

Preprocessors in Snort can de-fragment packets, re-assemble TCP streams and so on. These functions are considered to be an essential part of the IDS.

Logging and Alerting System

Depending on what the Detection Engine finds inside a packet, the packet may be used to log an activity or to generate an alert. Logs are kept in simple text files.



Output Modules

Snort output module or plug-ins, which control the type of output generated by the system, can produce different results depending on the desired format of the output generated by the logging and alerting systems.

2.2 Snort component-Detection Engine

The Detection Engine [14] is the most important part of Snort. The Detection Engine’s role is to detect any intrusion activity in a packet. The Detection Engine employs Snort rules to do this. The rules are read into internal data structures or chains where they are matched against all packets. If a packet matches a rule, an appropriate action is taken. Taking an appropriate action means either logging the packet or generating an alert. Depending on the strength of the engine and the number

of rules that are defined, the amount of time to needed respond to different packets may vary.

Snort uses rules stored in text files and rules are grouped into categories. For example, the file “web-iis.rules” stores all rules about Windows IIS attacks. Snort reads these rules during start-up and then builds internal data structures or chains in order to apply the rules to captured data.

All Snort rules have two logical parts: the rule header and the rule options. When snort initializes and parses the rules, it creates a separate rule tree for TCP, UDP, ICMP and IP. Within each rule tree, there is a separate three-dimensional linked list of RTNs (dimension one), OTNs (dimension two) and function pointers (dimension three). The RTNs include the IP address and port. An example of an RTN list and the three-dimensional links are shown in Figures 2-2 and 2-3, respectively [15].

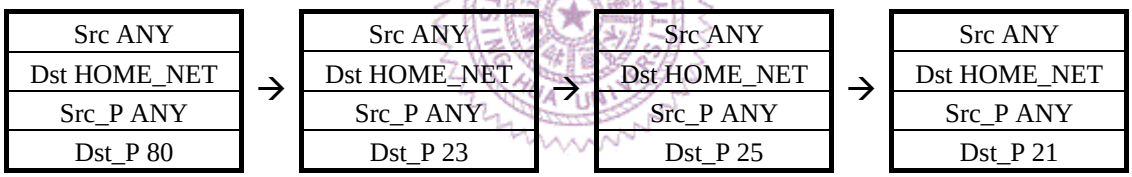


Figure 2-2. The three-dimensional linked list (RTN list)

When Snort sends a packet through the Detection Engine, its first checks the type of IP protocol of the packet, so that the packet can be sent to the correct rule tree. Once the packet is sent to the correct tree for evaluation, it is checked against each RTN, from left to right, until a match is found. When checking the RTNs, Snort first looks at the IP addresses and then the port information, if necessary. If an RTN is found that matches the current packet, it is then checked against the OTNs one by one to see if a further match is found. Each OTN has a linked list of function pointers (dimension three) to the tests that need to be carried out for a particular OTN.

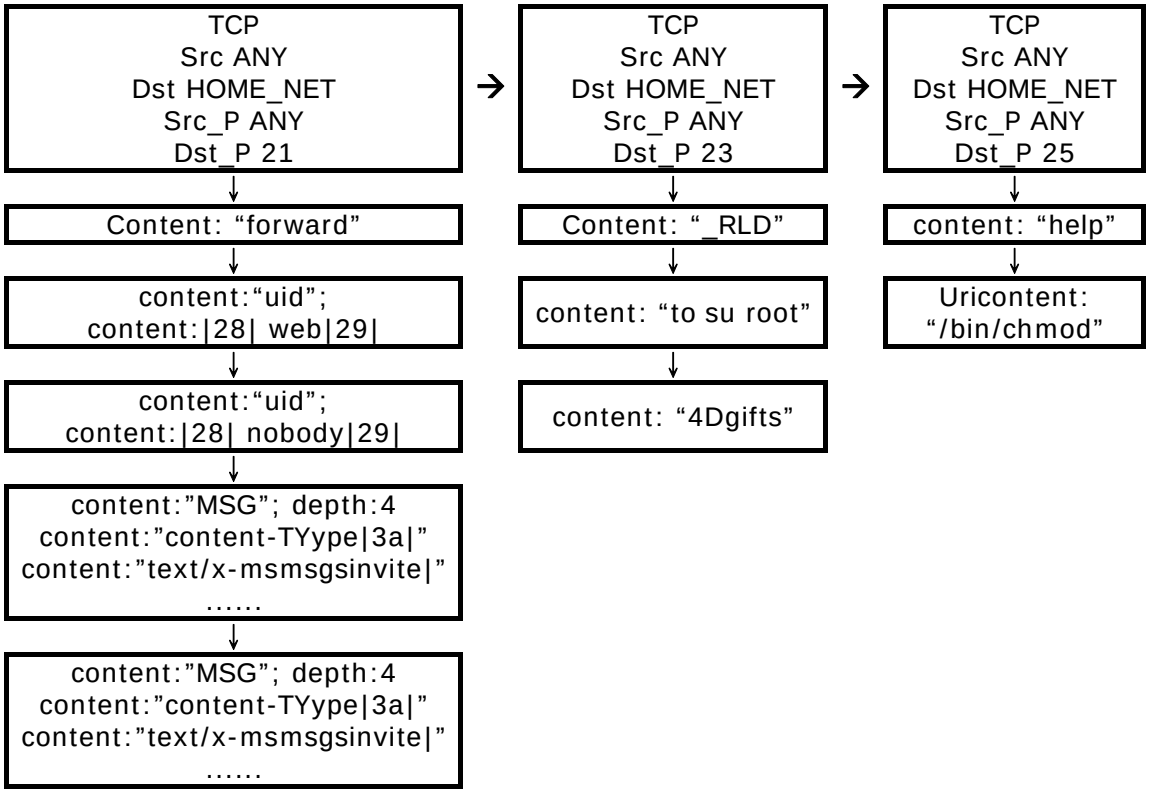


Figure 2-3. The three-dimensional linked list (RTN &OTN)

2.3 Snort rule payload keyword

In order to redesign the Detection Engine, we must understand all the important keywords, especially those related to packet payload. The public Snort payload keywords are *content*, *uricontent*, *pcre*, *offset*, *depth*, *within* and *distance*. These are introduced below [16].

The content keyword

One of Snort’s important features is its ability to find data patterns inside packets. The pattern may present as an ASCII string or as binary data in the form of hexadecimal characters. Like viruses, intruders also have signatures and the *content* keyword is used to find these signatures in the packet.

The following rule does the same thing but the pattern is listed in hexadecimal. Hexadecimal number 47 is equal to ASCII character G, 45 is equal to E, and 54 is

equal to T. You can also match both ASCII strings and binary patterns in hexadecimal form inside one rule.

```
alert tcp any any → any any (
    msg: “GET matched”;
    content: “|47 45 54|”;   ...
)
```

The *uricontent* keyword

The *uricontent* keyword is similar to the content keyword except that it is used to look for a string only in the URI part of a packet.

The *PCRE* (perl compatible regular expression) keyword

It allows rules to be written using PERL compatible regular expressions.

The *offset /depth* keyword

The *offset* keyword is used in combination with the *content* keyword. Using this keyword, we can start the search at a certain offset location relative to the start of the data portion of the packet. We can use the *depth* keyword to define the point after which Snort should stop searching for patterns in the data packets. The *depth* keyword is also used in combination with the *content* keyword to specify an upper limit to the pattern matching. The following rule tries to find the word “HTTP” between the 4th character and 40th character of the data portion of the TCP packet:

```
alert tcp any any → any any (
    content: "HTTP"; offset: 4; depth: 40;   ...
)
```

This keyword is very important since we can use it to limit searching inside the packet.

The *distance/within* keyword

The *distance* keyword allows the rule writer to specify, within the packet, that the distance between the pattern and the previous pattern is at least *N* bytes. The

within keyword is a *content* modifier that makes sure at most *N* bytes are between previous pattern matches using the *content*.

alert tcp any any → any any (
content: "ABC"; content: "DEF"; distance :1; within:10 ;
)

The payload form is as follows: (* is any 8-bits data).

	Payload			
offset		1~10		
data	ABC	*	DEF	*

2.4 SoC (System On Chip) platform

Today, System-on-Chip (SoC) [17,18,19] devices target high performance applications in which rapid time-to-market is of great importance. On these platforms, there is a tradeoff among performance, flexibility and time-to-market. A typical platform usually combines a micro-processor, memory, user-defined logic circuits and peripherals around standard bus architecture. Specific IP cores could be added to form derivatives targeting specific applications. This procedure reduces design time and increases flexibility.

SoC systems are more powerful than pure software systems, because the user-defined logic is the product of pure a software system, it can easily increase the performance. The micro-processor simultaneously takes charge of the system. A typical design flow for a SoC system is shown in Figure 2-4.

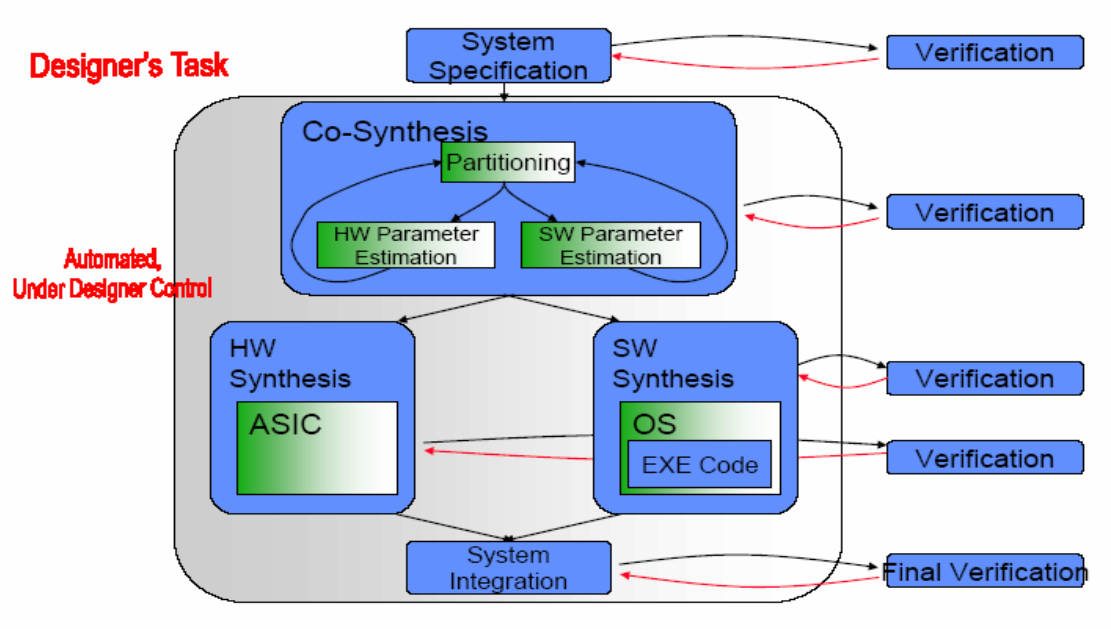


Figure 2-4. SoC system design flow

2.5 Multi-pattern matching algorithms

Snort has a pattern-match approach that uses a multi-pattern matching algorithm which is capable of finding different patterns simultaneously. Two multi-pattern matching algorithms, one Aho-Corasick-based and the other TCAM-based, are discussed below.

Aho-Corasick algorithm:

Roughly speaking, the Aho-Corasick [20] algorithm uses a finite automaton structure that accepts all strings in the set. The automaton is structured so that every prefix is represented by only one state, even if the prefix is part of multiple patterns. When the next character in the text is not one of the expected next characters in the pattern, a failure link is made to a state representing the longest prefix of a pattern that is also the proper suffix of the current state. An example of an AC tree for patterns {he, she, his, hers} is shown in Figure 2-5. The time complexity of the Aho-Corasick algorithm is $O(n)$; where n is total length of input data.

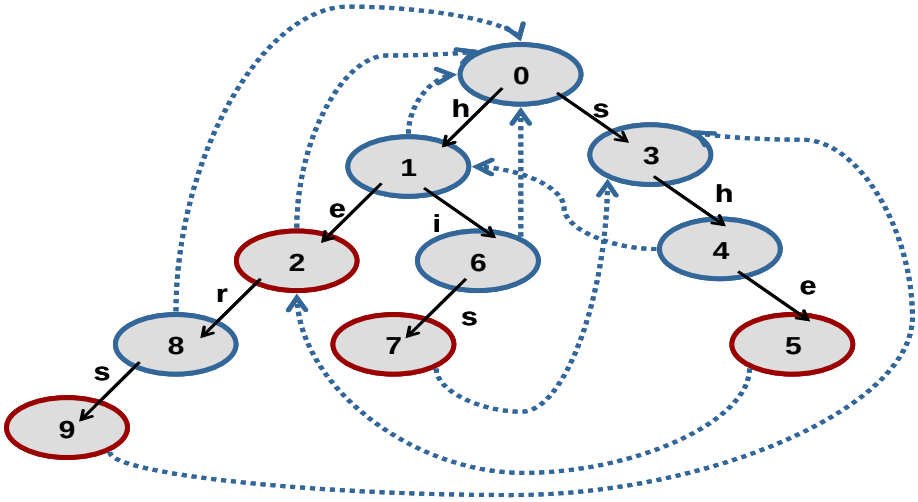


Figure 2-5. An example of AC tree for pattern {he, she, his, hers} [21]

TCAM:

Ternary Content Addressable Memory (TCAM) is a type of memory that can perform parallel searches at high speeds [22, 23, 24, 25]. TCAM consists of a set of entries. The top entry in the TCAM has the smallest index and the bottom entry has the largest. Each entry is a one-bit vector of cells, where each cell stores one bit. Therefore, a TCAM entry can be used to store a string. TCAM works as follows: given an input string, it compares this string against all entries in its memory, in parallel, and reports one entry that matches the input [26, 27, 28].