Increasing Performance in High Speed NIDS

A look at Snort's Internals

Neil Desai
ndesai01@tampabay.rr.com

Introduction

The increasing of network utilization and the weekly increase in the number of critical application layer exploits means Network Intrusion Detection Systems (NIDS) designers must find ways to speed up their attack analysis techniques when monitoring a fully-saturated network and maintaining a good false positive to false negative ratio.

While increasing the CPU speed and RAM of the NIDS will help deal with more content analysis there is a point where the amount of money spent on the new hardware will not be proportional to the increase in speed of the content analysis. In general there are only a few things a user can do to help the NIDS keep up with today's demands, such as limiting the amount of attacks that the NIDS looks for or load balancing via a layer 7 switch. The rest is up to the NIDS developers.

Bottlenecks

There are four main areas that take up a considerable amount of time in the current version (1.8.3) of snort[i], but only one of them gives a considerable amount of performance increase while maintaining the portability of snort.

1. Getting the packet off the wire[ii]. == Snort uses libpcap to get the packets off the wire. Libpcap is almost the standard for grabbing packets off the wire and is used by many protocol-decoding applications. The libpcap library is good for most applications but does not lend itself well to high-speed data acquisition because only one libpcap function can be used at any one time. The snort community could develop custom drivers that would be OS- and possibly NIC-specific, but that would severely hinder the portability of snort.
2. Clearing out data structures. == Every packet that comes in has to be stored in some type of data structure. This also means that all of these data structures need to be cleared out to make room for other packets. According to Marty Roesch, some of this could be tweaked by a little "code tightening and rethinking some of our base assumptions."[iii]
3. Pattern matching. == The snort community has looked at implementing different pattern matching algorithms to improve the speed of snort on saturated networks at high speeds. By implementing a different algorithm snort 2.0 will have about a 500 percent increase in performance[iv].
4. Checksum verification. == To help verify the integrity of the packet snort verifies all protocol checksums. This means that for every packet snort must

compute the checksum and then verify that it matches the current packet's checksum.

Performance can also be affected by preprocessors and the parameters that they are loaded with. The parameters that are specified to the preprocessor can also affect the false-positive ratio and the effectiveness of NIDS evasion techniques. We will take a brief look at how preprocessors can affect performance. An in-depth discussion of how to effectively configure preprocessors is outside the content of this paper. Each preprocessor has a different area of responsibility (i.e. frag2 => IP fragmentation reassembly, stream4 => TCP reassembly/stateful inspection, http_decode => normalize HTTP requests, etc.) the performance gained or lost would be specific to:

1. The type of traffic that the NIDS monitors. This would also include the amount of traffic for a particular protocol.
2. The parameters that are specified to the preprocessor. The parameters will determine things like what port(s) to look for, how much memory to use, how long to hold onto the information, etc.

Proper configuration of the preprocessors will take some time to fine tune. This would involve the person installing the NIDS to have:

1. A good understanding of the network that is being monitored. This would include protocols, applications and traffic patterns.
2. An in-depth understanding of the protocols on the network. The NIDS installer should be able to do protocol analysis on the traffic to determine thresholds, false-positives, etc.
3. Knowledge of the preprocessors and how they work and how they can be configured. Some of the preprocessors are as easy as just defining them (i.e. frag2, telnet_decode). Some preprocessors only take a few arguments that won't require much protocol analysis of the network because they are straight forward (i.e. rpc_decode, http_decode, unidecode). Then there are the ones that will take a lot of time and research to fine tune (i.e. stream4, stream4_reassemble, spade).

Brief History of Snort and Pattern Matching

In earlier days, snort used brute force pattern matching which was very slow and was seen as a place where performance could be improvement[v]. The first thing done to boost performance was implementing a partial Boyer-Moore pattern matching algorithm. After a couple of months a full implementation of Boyer-Moore was implemented[vi]. Next was the implementation of a "2-dimentional linked list with recursive node walking," which gave snort a 200 to 500 percent performance increase[vii]. Then snort developers rewrote the detection engine to include a "linked-list-of-function-pointers", also called a "three-dimensional linked list",[viii] which is where snort is today.

Overview of Snort Rules

Figure 1. Snort rule.

```
alert udp $EXTERNAL_NET any -> $HOME_NET 177 (msg:"MISC xdmcp query";
content: "|00 01 00 03 00 01 00|";reference:arachnids,476;
classtype:attempted-recon; sid:517; rev:1;)
```

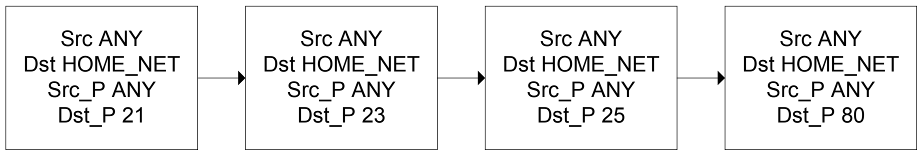
Snort rules are broken into two pieces; the rule header and the rule option(s). The rule header is everything up to the first parentheses. The rule option(s) are everything within the parentheses. The rule header can be loosely mapped to the RTN (Rule Tree Node) and the rule options can be loosely mapped to the OTN(s) (Optional Tree Node).

There are 35 keywords in snort 1.8.3 that can be used in the rule option(s), 20 of which will be used in the OTN's. Of the 20 items that will be in the rule options, 17 will be either a true/false (i.e. equal to or not equal to) value or greater than/lower than value. The snort engine can easily pass this information through the linked list with little overhead. Most of the computational overhead comes from the use of the following three keywords: content, content-list and uricontent. Each of these keywords calls the pattern-matching engine to parse the data portion of the packet for a particular pattern. Because of the overhead that the pattern-matching engine causes it is the last part of the rules option(s) that is checked. Of the 1270 rules, 1086 rules contain either the "content" or "uricontent" keyword[1].

Rule Parsing and Detection Engine

When snort initializes and parses the rules it creates a separate rule tree for TCP, UDP, ICMP and IP. Within each rule tree there will be a separate three-dimensional linked list of RTNs (dimension one) and OTNs (dimension two) and function pointers (dimension three). The RTNs will include the IP address information and port information.

Figure 2. An example of the chain header (RTN).

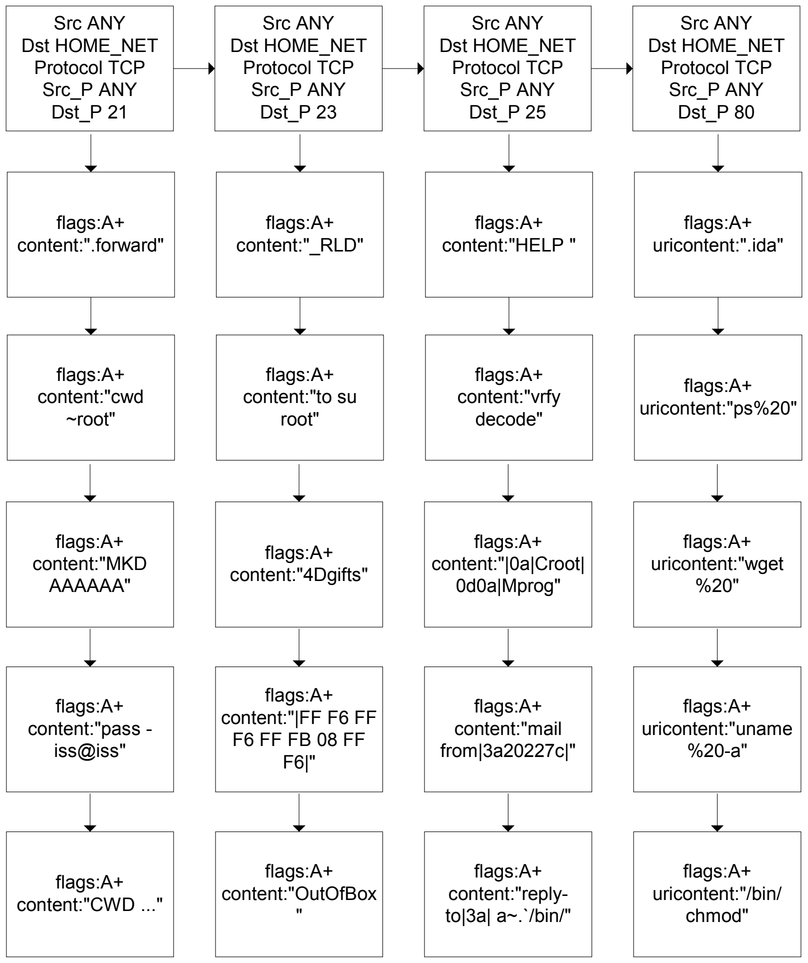


When snort sends a packet through the detection engine it first sees what IP protocol the current packet is so that it can send it to the correct rule tree[ix]. Once the packet is sent to the correct tree for evaluation it will be checked against each RTN, from left to right, until a match is found. When checking the RTNs, snort will first look at the IP addresses and then the port information, if necessary[x]. If an RTN is found that matches the current packet, then it goes down the OTNs one by one to see if a match can be found. Each OTN is not checked for every option

[1]
```
$ls *.rules | wc -l
33
$egrep -v "^#" *.rules | egrep [a-z] | wc -l
1270
$egrep -v "^#" *.rules | egrep content | wc -l
1086
```

that is available because it would be a waste of resources to check for things that do not exist (i.e. checking for content on a non-content packet). Instead each OTN has a linked list of function pointers (dimension three) to the tests that need to be carried out for that particular OTN.

Figure 3. An example of chain headers (RTNs) and chain options (OTN's).



Src ANY Dst HOME_NET Protocol TCP Src_P ANY Dst_P 21
Src ANY Dst HOME_NET Protocol TCP Src_P ANY Dst_P 23
Src ANY Dst HOME_NET Protocol TCP Src_P ANY Dst_P 25
Src ANY Dst HOME_NET Protocol TCP Src_P ANY Dst_P 80
flags:A+ content:".forward"
flags:A+ content:"_RLD"
flags:A+ content:"HELP "
flags:A+ uricontent:".ida"
flags:A+ content:"cwd ~root"
flags:A+ content:"to su root"
flags:A+ content:"vrfy decode"
flags:A+ uricontent:"ps%20"
flags:A+ content:"MKD AAAAAA"
flags:A+ content:"4Dgifts"
flags:A+ content:"|0a|Croot|0d0a|Mprog"
flags:A+ uricontent:"wget %20"
flags:A+ content:"pass -iss@iss"
flags:A+ content:"|FF F6 FF F6 FF FB 08 FF F6|"
flags:A+ content:"mail from|3a20227c|"
flags:A+ uricontent:"uname %20-a"
flags:A+ content:"CWD ..."
flags:A+ content:"OutOfBox "
flags:A+ content:"reply-to|3a| a~.`/bin/"
flags:A+ uricontent:"/bin/ chmod"

Snort uses the Boyer-Moore pattern-matching algorithm when attempting content matching on the packet payload.[xi]. This pattern-matching algorithm is one of the most efficient algorithms for string matching and is often used for the "search" and/or "replace" commands within a text editor[xii]. The Boyer-Moore algorithm is good for a single string search, but when dealing with a NIDS a single packet can partially match many different rules and for each rule the algorithm will have to be run. For example, a packet is matched for the pattern */cfdocs/cfmlsyntaxcheck.cfm* (web-coldfusion.rules shown below) and the next 15 OTNs all contain the same */cfdocs/* directory in the beginning of the pattern. After searching the current packet it is certain that */cfdocs/* does not appear anywhere in the packet. The next 15 OTN searches will all fail but are performed anyway.

A new pattern-matching algorithm was needed to help overcome this shortcoming. Silicon Defense did some initial research on this matter and tested a new algorithm that uses the best aspects of the Boyer-Moore algorithm and the Aho-Corassick algorithm to gain a significant performance boost over the current Boyer-Moore algorithm[xiii].

Figure 4. Partial contents of web-coldfusion.rules:

```
/cfdocs/cfmlsyntaxcheck.cfm
/cfdocs/exampleapp/
/cfdocs/exampleapp/email/application.cfm
/cfdocs/exampleapp/email/getfile.cfm
/cfdocs/exampleapp/publish/admin/addcontent.cfm
/cfdocs/exampleapp/publish/admin/application.cfm
/cfdocs/examples/cvbeans/beaninfo.cfm
/cfdocs/examples/mainframeset.cfm
/cfdocs/examples/parks/detail.cfm
/cfdocs/expeval/
/cfdocs/expeval/displayopenedfile.cfm
/cfdocs/expeval/exprcalc.cfm
/cfdocs/snippets/
/cfdocs/snippets/evaluate.cfm
/cfdocs/snippets/fileexists.cfm
/cfdocs/snippets/gettempdirectory.cfm
```

Overview of the Boyer-Moore Algorithm

Before going into the basics the following is a layout of the terminology.

1. Pattern to match will be noted as *P*.
2. Text to match against (payload) will be noted as *T*.
3. The length of pattern (*P)* will be noted as *LP*.
4. The length of text (*T)* will be noted as *LT*.
5. The first and last character of *P* will be noted as P_1 and P_{LP} respectively.
6. The first and last character of *T* will be noted as T_1 and T_{LT} respectively.
7. When initially matching up the *P* and *T* the last character in *P*, P_{LP}, will match up with T_{LP}.

Pattern to look for: `EXAMPLE`
Text to look in: `HERE IS A SIMPLE EXAMPLE`[2]

With a naïve pattern-matching algorithm *P* would be searched in *T* as follows:

1. Align the left end of *P* with the left end of *T* so that P_1 and T_1 are aligned (Figure 5).

[2] Moore, Strother J. The Boyer-Moore Fast String Searching Algorithm, http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/index.html, (18 Feb 2002).

Figure 5.

P_1 P_{LP}
EXAMPL**E**
HERE I**S** A SIMPLE EXAMPL**E**
T_1 T_{LP} T_{LT}

2. Match the characters of *P* against *T* from left to right until either a mismatch of characters occurs or *P* is exhausted. In this case P_1 = E and T_1 = H and E ≠ H (Figure 5).
3. If a mismatch occurs the algorithm will shift *P* one character to the right and start the matching process again (Figure 6). This time P_1 = "E" will be aligned with T_2 = "E". Since "E" = "E" it will check P_2 = "X" against T_3 = "R". Since "X" ≠ "R" it will shift *P* one character to the right and start all over again.

Figure 6.

P_1 P_{LP}
EXAMPL**E**
H**E**RE IS A SIMPLE EXAMPL**E**
T_2 T_{LT}

4. The above process will continue until either a complete match of *P* if found in *T* or until P_{LP} shifts past the right end of T_{LT}.

In the above case it took the naïve algorithm 28 attempts to find a match. As you can see this is a "brute force" matching algorithm that can take a long time to either make a match or determine that a match cannot be made. This is similar to the way that snort started off with its pattern matching techniques.
The Boyer-Moore algorithm has three strengths that are not contained in the naïve algorithm that make it efficient and very good even in a worst-case scenario.

1. Right to left scan. This is in contrast to the naïve method that scans from left to right. The left end of *P* is still aligned with the left end of *T* but the matching starts on the right end of *P* and moves left until a mismatch occurs (Figure 7).

Figure 7.

P_1 P_{LP}
EXAMPL**E**
HERE I**S** A SIMPLE EXAMPL**E**
T_1 T_{LP} T_{LT}

2. Bad character shift. The first match attempt will start at P_{LP} = "E" and T_{LP} = "S" (Figure 7). Like many advanced pattern-matching algorithms Boyer-Moore preprocesses the pattern and gains heuristic information. It will use this information

to compute the amount to shift *P* to the right. The algorithm has to determine the right-most character in *P* that it can match in *T*. In this case it would be the letter "E" (Figure 8).

Figure 8.

P_1 P_{LP}

EXA**MPLE**

HERE IS A SI**MPLE** EXAMPLE

T_1 T_{LP} T_{LP+9} T_{LT}

3. Good Suffix shift. Once this match is made the algorithm will start matching at the right end of *P*. This time P_{LP} = "E" will match T_{LP+9} = "E" (Figure 8). Now it will match P_{LP-1} = "L" against T_{LP+8} = "L". Again we have a match so P_{LP-2} = "P" will be matched against T_{LP+7} = "P". This is also a match so it will attempt P_{LP-3} = "M" against T_{LP+6} = "M". Since it is still matching it will attempt P_{LP-4} = "A" against T_{LP+5} = "I". This time a mismatch occurs. Because of the preprocessing of the pattern it has found that *T* contains the string "MPLE" and it will look for the next occurrence of that string in *T* . Then it will shift *P* to the right so that the string "MPLE" of *P* will be aligned with the next occurrence of the string "MPLE" in *T* (Figure 9).

Figure 9.

P_1 P_{LP}

EXA**MPLE**

HERE IS A SIMPLE EXA**MPLE**

T_1 T_{LP} T_{LT}

4. Once this shift is complete the matching of characters will start again from the right of *P*. In this case when every character of *P* is matched to *T* starting at P_{LP} = "E" and T_{LT} = "E" a full match occurs.

With the Boyer-Moore algorithm the above search took only 12 attempts before a successful match was detected. This is more than twice as fast as the brute force method.

Aho-Corasick_Boyer-Moore Hybrid

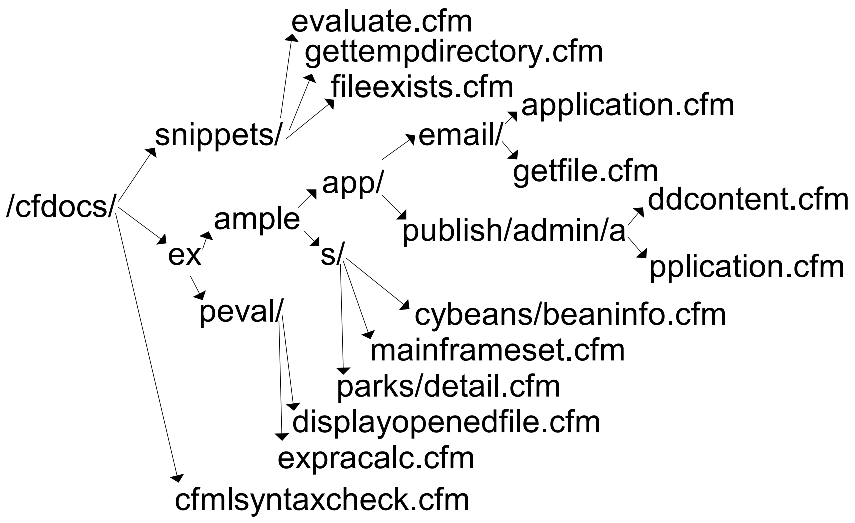
While the name would imply that the new algorithm is a mix between the Aho-Corasick and Boyer-Moore algorithms it really is not. It is a "Boyer-Moore like algorithm applied to a set of keywords held in an Aho-Corassick like keyword tree that overlays common prefixes of the keywords."[3]. This new algorithm takes the best characteristics of both the Boyer-Moore and Aho-Corasick algorithms.

[3] Coit, Jason and Staniford, Stuart and McAlerney, Joe. (21 June 2001), http://www.silicondefense.com/software/acbm/speed_of_snort_06_21_2001.pdf page 3, (19 Feb 2002)

1. Similarities:
    a. Boyer-Moore -> Bad character shift.
    b. Aho-Corasick -> Keyword tree.
2. Variances:
    a. Boyer-Moore -> Instead of using the original good suffix shift the new algorithm will use the good prefix shift.
    b. Boyer-Moore -> While the packet (text, *T*) will be searched from right to left, the tree (pattern, *P*) will be searched from left to right.
    c. Aho-Corasick -> Instead of building a tree based on suffixes this tree will be built on prefixes.

By looking at our original problem (Figure 4) we have 16 different rules that share some common information. If we had loaded just those 16 rules in the AC_BM keyword tree it would look very different (Figure 10).

Figure 10. Partial web-coldfusion.rules.



At first glance this tree may look more confusing than Figure 4, but once you study this tree you can see how efficient it is. A mismatch can eliminate many rules from being searched that will eventually fail. With a normal Aho-Corasick keyword tree the pattern would be searched for one character at a time like the naïve algorithm. With the addition of the Boyer-Moore good prefix shift and bad character shift, the algorithm can quickly determine if a match occurs or the current packet does not match any current patterns. This algorithm is outlined in Dan Gusfield's "*Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*" as a "Boyer-Moore Approach to Exact Set Matching."

The AC_BM algorithm shows a slight performance increase when used in non-content matching rules, but difference really shows when it comes to content matching rules. In snorts current implementation the number of content matching rules will significantly affect the performance and will not scale well. This is why new pattern matching algorithms have been researched and show promise.

Current Issues

Even though the new algorithm has better performance for content matching rules it does have some areas that either should be improved upon or taken into account when implementing the AC_BM version of snort.

First, since AC_BM starts the matching at the right end of the packet and moves to the left end of the packet it may trigger a different rule than the original Boyer-Moore algorithm when the same packet is sent to the respective algorithm for analyzing. For example, if you have two rules that only differ by the content that is searched for:

Rule 1 content: `Firstpartofpacket`
Rule 2 content: `Lastofpacket`
Packet Data (Text): `FirstpartofpacketandthentheLastofpacket`

When the packet data is sent to the Boyer-Moore algorithm it will trigger on rule 1. When the same packet data is sent to the AC_BM algorithm it will trigger on rule 2. This is a result of the different directions in which each algorithm examines the packet data. If the AC_BM algorithm were set up to examine the packet data from left to right, like Boyer-Moore, it would get rid of the anomaly. The only reason that it was not done in the initial implementation of snort with AC_BM is that it was a proof of concept implementation.

Like many other products (Cisco access-lists, CheckPoint FireWall-1 rules, etc.) snort is a "first rule match wins" type of architecture. The problem that arises is with the AC_BM keyword tree is the first match will be the shortest match, which may not be the right match. When AC_BM builds the keyword tree it loses the ordering of the rules. The ordering of the rules is what allows the current version of snort to find the longest match.

This longest match rule is preferred for applications like IP routing (EIGRP, OSPF, BGP-4) and regular expressions. With the longest match rule, the most specific rule will always be triggered. Hank[xiv] takes it a bit further. Hank alerts to all matches. This type of alerting gives the NIDS administrator, or whoever looks at the alerts, more information than most NIDS today. This will change in snort 2.0, which will move toward a "last-exit"[xv] match. Basically this will make snort look for the longest match and make it more accurate.

The only thing kept in the AC_BM keyword tree is the content for which to search. They had to figure out a way to keep the various non-content options. Since the AC_BM implementation was only meant as a proof of concept, they (Silicon Defense) had to either change the preprocessing of the rules or change the way that snort organized the RTNs and OTNs. To keep snort as close to the original architecture as possible they chose to change the way that snort imported the rules. To do this they separated the content and non-content rules and handled the option rule checking differently depending on the type of rule. In doing the rule separation they also changed the way that content rules are handled. In the original snort all options are checked first before Boyer-Moore is called to check the content. In snort with AC_BM all the other options are checked after AC_BM is called to the check the content of the rules.

Protocol Analysis

When most people think of a NIDS they think of pattern matching. There is another way to implement NIDS though, and that is through protocol analysis. Protocol analysis products have been around for many years and have the ability to analyze the data (real time) for the user so that they can quickly determine what problems are occurring on their network. By taking this to the next level you would get a protocol analysis NIDS.

The architecture for a protocol analysis NIDS is very different from a pattern matching NIDS. The protocol analysis NIDS will decode each packet according the protocol specification. It will then it will check each field to make sure that it conforms to the standard. If it doesn't then the NIDS will flag the packet accordingly. For packets that conform to the standards but are still an exploit (i.e. showcode.asp, /bin/sh) the NIDS can do a pattern match in the particular field that needs to be checked instead of the entire packet. BlackICE takes a different approach to the pattern matching issues that could be a bottleneck. In the case of a HTTP packet the user can specify the text to match for in the URI. For example in the packet below BlackICE would split the URI into separate components, in this case "SAMPLE" and "showcode.asp". Then it would do an exact match for each component in a list of exploitable components. Since it does an exact match and not a pattern match it can determine if the component is a match or not quicker.

Snort started to become protocol aware when the keyword "uricontent" was added. This gave snort users the ability to search only the URI portion of a HTTP packet instead of the entire packet/payload (Figure 11).

Figure 11.

```
HTTP - Hyper Text Transfer Protocol
Command:           GET
URI:               /SAMPLE/showcode.asp
Version:           HTTP/1.1..
Accept:            */*..
Referer:           http://www.victim.com/..
Accept-Language:  en-us..
Accept-Encoding:  gzip, deflate..
User-Agent:        Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt)..
Host:              www.victim.com..
Connection:        Keep-Alive..
Cookie:            RoxenUserID=0x673b30....
```

The performance increase from this methodology will all depend on many factors, like packet size, field size, components to check for, etc.

One of the big drawbacks of a protocol analysis NIDS is that every vendor implements the protocol according to how they interpret the RFC. This can cause false positives if the NIDS developers are validating the packets differently than a vendor that has traffic that is seen by the NIDS.

The second drawback is that if someone finds a way to evade the protocol analysis NIDS the decoding engine will have to be rewritten. For example when Rain Forest Puppy came out with Whisker[xvi] it had some very new and interesting IDS evasion techniques. While users who ran snort could easily update the signatures so that they could catch someone scanning them with Whisker, NetworkICE's BlackICE had to be rewritten to catch these exploits. In the end though the developers of BlackICE addressed the issue (evasion techniques) and not the program (Whisker). By addressing the techniques and not the program the developers will be able to develop a NIDS that can not only alert to the current NIDS evasion techniques but also to other similar techniques that have taken advantage of the same issues within that protocol.

The third drawback for a protocol analysis NIDS is how it deals with a packets that it does not have a decode for. This could be as simple as sending the user an alert that the NIDS has seen a packet that it does not know how to decode or it could attempt to run some heuristics on the packet to see if it is a protocol that is knows about, but is listening on a different port.

The strength of a protocol analysis NIDS is that exploits that are new should be easier to catch since it does not rely on matching a known pattern. Evasion techniques like polymorphic shellcode, URL encoding, session splicing, etc. are still easy for the protocol analysis NIDS to detect for the same reason.

Depending on the network architecture, regulations and level or paranoia you might want to run a NIDS inside of your network. This could be a simple as a single NIDS watching a major portion of your network or as complicated as a distributed NIDS environment with each NIDS configured to watch traffic that is specific to that particular segment. By placing the NIDS on the LAN you run into many more situations where a protocol analysis NIDS would be the appropriate tool. Below I will outline some areas that would be best suited for a protocol analysis NIDS.

1. Most large companies run some sort or interior routing protocol to keep the network up and running. Most of these routing protocols update via multicast (OSPF, EIGRP) or broadcast (RIP, IGRP). In the past few years there have been utilities (irpas[xvii], nemesis[xviii], nmap[xix]) written that will either gain information from these types of protocols or spoof these protocols. With these types of utilities an attacker could seriously disrupt the network by injecting bad routes or DoSing nodes with bad packets. Even worse an attacker could spoof a gateway and capture the packets as it redirects the packets to the correct gateway. Network engineering departments are usually not concerned with security and see security as a nuisance. Because of this attitude, plus the false sense of security of being behind a firewall, attacks on the routers and routes can be easy to execute and hard to detect. A NIDS might be able to detect routing protocols, but it does not have any way of knowing of what types of protocols are suppose to be running or how they are suppose to be configured. For example if you have EIGRP as your only routing protocol and you run only one AS, 50, you would have a hard time detecting an EIGRP packet that had a different AS. Even harder would be to look for route injection or modification.
2. Not all security professionals take layer two spoofing seriously. Some people are still under the assumption that if you are in a 100% switched environment that you are safe from sniffers and other packet capture software. Enter dsniff[xx] and ARP0c[xxi]. These two tools give the attacker the ability to fool switches into sending the attacker packets that it is not suppose to. Since these two tools only work on the LAN not many people focus on detecting these attack methods. While the protocol analysis NIDS would not see anything wrong with the way that the packets are constructed it may have some sort of anomaly detection function built into it to help with these types of attacks.

Stateful Inspection

The term stateful inspection has long been associated with firewalls. When dealing with firewalls, stateful inspection usually deals only with layer 4 protocols (i.e.TCP, UDP). Firewalls have having to become more protocol aware as new protocols are developed that take advantage of a setup channel and a data channel. An early example of this is the FTP protocol. FTP utilizes a two separate connections, one for control information (TCP port 21) and one for data transfer (TCP 20). When a firewall sees a FTP connection (TCP port 21) to a FTP server it knows to open up a dynamic rule that would allow the data connection (TCP port 20) from the server back to the client. This dynamic rule will be torn down depending on the information in the control connection (i.e BYE command).

In the early days NIDS developers did not worry about the state that a connection was in to determine how a packet should be handled. Starting with fragrouter[xxii] NIDS developers have had to worry about packet reassembly at the IP layer. When fragrouter first came out it was able to evade many of the NIDS on the market. Snort added what is now know as the frag2 preprocessor to handle these types of evasion techniques. Some commercial NIDS only alert the NIDS administrator that fragmented IP packets have been detected but it doesn't reassemble the fragmented packets.

Then hackers came out with tools like stick[xxiii] and snot[xxiv]. These tools could be used to overload the NIDS with packets that match NIDS rules and overload the NIDS administrator with a lot of false positives. This took advantage of the fact that most pattern matching NIDS only looked in the current packet to see if it matched a rule. These tools don't even setup a TCP session properly (i.e Three Way Handshake) before sending the data. They just send just enough data to trip the NIDS by using existing snort rules to generate the forged packets. To take care of this DoS against the NIDS itself the snort community added the stream4 preprocessor with the "-z est" option. This allows snort to track the state of the TCP connection before sending the packets to the detection engine. Since packets generated via stick and snot don't establish a TCP connection they would not get sent to the detection engine. Depending on how the stream4 preprocessor is setup snort could alert the NIDS administrator that it has seen a TCP packet that is not part of an existing connection.

Protocol Analysis NIDS developers take the term stateful inspection differently. They are aware how certain protocols are suppose to act and react and can watch each protocol "statefully". By tracking not only the stimuli but also the response, the protocol analysis NIDS can do a better job at determining if the attack is successful. In the case of a HTTP exploit a pattern matching NIDS would alert you to the fact that is saw a packet that matched a rule, but a protocol analysis NIDS would be able to tell you with a certain amount of accuracy if the attack was successful or not by watching the HTTP message generated by the server.

Where are we now and where do we go from here?

The research done by Silicon Defense is also closely followed by research done by a joint effort of Mike Fisk and George Varghese. They implemented a new algorithm called Setwise Boyer-Moore-Horspool. Their results were different from those of Silicon Defense, but still showed that with a different pattern matching algorithm snort and other devices that need high speed pattern capabilities would be less susceptible to a DoS of the device due to the slow pattern matching algorithm. They noted that a special algorithm that would change from a standard Boyer-Moore-Horspool to a Setwise Boyer-Moore-Horspool to an Aho-Corasick_Boyer-Moore depending on the size of the patterns to search for performed better than any other variant.

Todd Lewis, author of Hank, liked the idea of AC_BM and took it a bit further. Instead of implementing what had already been done by Silicon Defense he took a step back and looked at what areas of AC_BM could be improved upon and implemented the changes. Silicon Defense noted their issues with memory consumption and shortest pattern match and these issues are implemented differently in Hank. Hank triggers all rules that match and has a more memory-efficient AC_BM implementation.

We have only touched the surface of what can be done to increase the performance of NIDS in regards to pattern matching there is still a lot more room for research in this area.

NIDS developers are still at odds over which NIDS technology is better (pattern-matching (snort 1.x, ISS RealSecure) or protocol analysis (ISS Sentry –formerly NetworkICE Sentry). Newer NIDS implementations (ISS 7.0, snort 2.0[xxv]) seem to be combining the best of both worlds. The combination of a protocol analysis engine and a pattern matching engine would capture the strengths of both worlds and make a NIDS that is fast, hard to evade and able to catch new (zero day) exploits.

Acknowledgments

Thanks to the following people who made sure that the paper is technically correct:

1. Marty Roesch: www.snort.org , Made sure that everything about snort was correct Helped with the understanding of snort at the code level.
2. Jason Coit: www.silicondefense.com , Made sure that the information and examples describing the pattern matching algorithms was correct.
3. Fyodor Yarochkin: Helped with the understanding of snort at the code level and in describing the preprocessors.
4. Robert Graham: www.robertgraham.com , Helped in understanding protocol analysis and it's place in NIDS.

[i] Roesch, Marty.(30 Jul 2001),http://archives.neohapsis.com/archives/sf/ids/2001-q3/0228.html(2 Feb 2002)
[ii] Roesch, Marty.(16 Aug 2001),http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/MartyRoesch/bh-usa-01-Marty-Roesch.ppt slide 38, (16 Feb 2001).
[iii] Roesh, Marty.(30 Jul 2001),http://archives.neohapsis.com/archives/sf/ids/2001-q3/0228.html(2 Feb 2002)
[iv] Roesch, Marty.(30 Jul 2001),http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/MartyRoesch/bh-usa-01-Marty-Roesch.ppt slide 33, (16 Feb 2001).
[v] Roesch, Marty. (06 Mar 1999) ChangeLog distributed with snort 1.8.3, www.snort.org (26 Feb 2002).
[vi] Roesch, Marty. (13 Oct 2001), mstring.c version 1.15, www.snort.org. (26 Feb 2002).
[vii] Roesch, Marty. (01 Aug 1999) ChangeLog distributed with snort 1.8.3, www.snort.org, (26 Feb 2002).
[viii] Roesch, Marty.(30 Jul 2001), http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/MartyRoesch/bh-usa-01-Marty-Roesch.ppt slide 23, (16 Feb 2001).
[ix] Roesch, Marty. (26 Oct 2001),EvalPacket function in rules.c version 1.86, www.snort.org, (26 Feb 2002).
[x] Roesch, Marty. (26 Oct 2001), EvalHeader function in rules.c verson 1.86, www.snort.org, (26 Feb 2002).
[xi] Roesch, Marty. (6 Nov 2001), SnortUsersManual.pdf page 17 section 2.3.9 version 1.8.3 www.snort.org, (26 Feb 2002).
[xii] Charras ,Christian and Lecroq, Thierry, Handbook of Exact String-Matching Algorithms, http://www-igm.univ-mlv.fr/~lecroq/string/node14.html#SECTION00140, (28 Feb 2002).

[xiii] Roesch, Marty.(30 Jul 2001), http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/MartyRoesch/bh-usa-01-Marty-Roesch.ppt slide 33, (16 Feb 2001).
[xiv] Lewis, Todd. http://hank.sourceforge.net/, (18 Feb 2002).
[xv]Roesch, Marty.(30 Jul 2001), http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/MartyRoesch/bh-usa-01-Marty-Roesch.ppt slide 35, (16 Feb 2001).
[xvi] Rain.Forest.Puppy, http://www.wiretrip.net/rfp/p/doc.asp/i2/d21.htm , (19 Feb 2002).
[xvii] FX, http://www.phenoelit.de/irpas/docu.html, (20 Feb 2002).
[xviii] Nathan, Jeff. http://www.mirrors.wiretapped.net/security/packet-construction/nemesis/nemesis-README.txt, (20 Feb 2002).
[xix] Fyodor. http://www.insecure.org/nmap/, (20 Feb 2002).
[xx] Song, Dug. http://www.monkey.org/~dugsong/dsniff/, (27 Feb 2002).
[xxi] FX. http://www.phenoelit.de/arpoc/index.html, (27 Feb 2002).
[xxii] Song, Dug. http://www.w00w00.org/files/sectools/fragrouter/, (27 Feb 2002).
[xxiii] Giovanni, Coretez, http://www.eurocompton.net/stick/projects8.html, (27 Feb 2002).
[xxiv] Sniph, (18 Aug 2001), http://www.sec33.com/sniph/, (27 Feb 2002).
[xxv]Roesch, Marty.(30 Jul 2001),http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/MartyRoesch/bh-usa-01-Marty-Roesch.ppt slide 32, (16 Feb 2001).