Byte Level NIDS Improvement

Dr. Sameer Shrivastava
Associate Professor, Department of Computer Science and Engineering,
Global Nature Care Sanghatan Group of Institutions
Jabalpur, India

I. ***Abstract*** - **Byte sequences are used in multiple network intrusion detection systems (NIDS) as signatures to detect nasty activity. Though being highly competent, a high rate of false-positive rate is found. Here we suggest the concept of contextual signatures as an enhancement to string-based signature-matching. Instead of matching isolated fixed strings, we enhance the matching process with added context. While designing a proficient signature engine for the NIDS, we provide low-level perspective by using regular expressions for matching, and high-level perspective by taking advantage of the semantic information made available by protocol analysis and scripting language. Thereafter, we greatly augment the signature's articulateness and hence the ability to reduce false positives. Multiple examples are presented such as matching request matching with replies, using environmental knowledge, defining dependencies between signatures to model step-wise attacks, and recognizing exploit scans.**

Index Terms - intrusion detection, intrusion reply, Byte level signatures.

I. INTRODUCTION

Computer attacks can be detected by numerous kind of approach. We will be concentrating on one trendy form of misuse detection, network-based signature matching in which the system examines network traffic for matches against precisely-described patterns. The term signature generally defines the raw bytes of sequences which are the patterns. Normally, when a site deploys a NIDS the signature-matching NIDS checks the passing packets for these sequences. An alert is generated when it encounters one. Usually all commercial NIDSs follow this approach, along with the most well-known freeware NIDS, Snort. This paper shall use the term signature in the same manner as it has been used.

Signature-matching has several pleasing properties. First, the underlying conceptual notion is simple: it is easy to explain what the matcher is looking for and why, and what sort of total coverage it provides. Second, because of this simplicity, signatures can be easy to share, and to accumulate into large "attack libraries." Third, for some signatures, the matching can be quite tight: a match indicates with high confidence that an attack occurred.

On the other hand, many important limitations exists in signature-matching. In general, particularly matching using tight signatures no attack is detected unless explicit signatures are used; the matcher will in general completely miss attacks, which, unfortunately, continue to be developed at a brisk pace. In addition, often signatures are not in fact "tight." Loose signatures straight away raise the problem of false positives: alerts that do not reflect an actual attack. A second form of false positive, which signature matchers often fail to address, is that of failed attacks. Usually at many sites attacks occur at nearly constant rates that is why failed attacks are not of much concern. The only thing is that it is important to differentiate between them and successful attacks.

Something that is of interest over here is that the problem of false positives can potentially be reduced at a large scale if the matcher has extra context at its disposal: either some extra particulars regarding the exact activity and its semantics, in order to pick over false positives due to overly general "loose" signatures; or the supplementary information of how the attacked system responded to the attack, which often indicates whether the attack succeeded.

In this paper, we shall be discussing over the concept of contextual signatures, in which the traditional form of string-based signature matching is amplified by including additional context on different levels when assessing the signatures. To start with, a competent pattern matcher has to be designed and implement that is similar in spirit to traditional signature engines used in other NIDS. On this low-level we enable the use of additional context by (i) the use of full regular expressions instead of fixed strings, and (ii) giving the signature engine a notion of full connection state, which allows it to correlate multiple interdependent matches in both directions of a user session. Then, if the signature engine reports the match of a signature, we use this event as the start of a decision process, instead of an alert by itself as is done by most signature-matching NIDSs. Again, we use additional context to judge whether something alert worthy has indeed occurred. This time the context is located on a higher-level, containing our knowledge about the network that we have either explicitly defined or already learned during operation.

Further in the study, we will show several examples to demonstrate how the concept of contextual signatures can help to eliminate most of the limitations of traditional signatures discussed above. We will see that regular expressions, interdependent signatures, and knowledge about the particular environment have significant potential to reduce the false positive rate and to identify failed attack attempts. For example, we can consider the server's response to an attack and the set of software it is actually running—its vulnerability profile—to decide whether an attack has succeeded. In addition, treating signature matches as events rather than alerts enables us to analyze them on a meta-level as well, which we demonstrate by identifying exploit scans (scanning multiple hosts for a known vulnerability).

Instrumenting signatures to consider additional context has to be performed manually. For each signature, we need to determine what context might actually help to increase its performance. While this is tedious for large sets of already-existing signatures, it is not an extra problem when developing new ones, as such signatures have to be similarly adjusted to the specifics of particular attacks anyway. Contextual signatures serve as a building block for increasing the expressive of signatures; not as a stand-alone solution.

The thought of contextual signatures in the framework has already been provided by the freeware NIDS Bro. In comparison to most NIDSs, Bro can neither be defined as an anomaly based system nor a signature-based system rather said to be partitioned into a protocol analysis component and a policy script component. Also The previous one feeds the second by generating a stream of events that reproduce different types of activities detected by the protocol analysis; hence, the analyzer can also be called as the event engine. For example, when the analyzer sees the establishment of a TCP connection, it generates a connection established event; when it sees an HTTP request it generates http request and for the corresponding reply http reply; and when the event engine's heuristics determine that a user has successfully legalized during a Telnet or Rlogin session, it can generate login success (otherwise, each failed attempt can result in a login failure event).

The event engine of Bro is policy-neutral i.e it does not consider any particular events as reflecting trouble but instead are available to the policy script interpreter. The predictor then defines the response to the stream of events written in Bro's custom scripting language. Here the language includes rich data types, persistent state, and access to timers and external programs; hence the reply includes a great deal of context in addition to the event itself. The response a script has to a particular event ranges from updating random state (for example, tracking types of activity by address or address pair, or grouping related connections into higher-level "sessions") to generating alerts (e.g., via syslog) or invoking programs for a reactive reply.

Usually, a Bro policy script can execute signature-style matching—for example, inspecting the URIs in Web requests, the MIME-encoded contents of email (which the event engine will first unpack), the user names and keystrokes in login sessions, or the filenames in FTP sessions—but at a higher semantic level than as just individual packets or generic TCP byte streams.

A very commanding approach of Bro is that it permits a wide range of different applications. But definitely has a significant shortcoming i.e. the policy script is capable of doing traditional signature-matching, which is burdensome for large sets of signatures, as each signature has to be coded as part of a script function. This is what makes it dissimilar from the brief, low-level languages used by most traditional signature-based systems. Also, if the signatures are matched in order, then the transparency of the matching can become exorbitant. Finally, much of community effort has already been there on developing and disseminating packet-based and byte-stream-based signatures. For example, the 1.9.0 release of Snort comes with a library of 1,715 signatures. It can be a great benefit if we can control these efforts by incorporating such libraries.

Hence an enthusiastic work would be to combine Bro's flexibility with the capabilities of other NIDSs by implementing a signature engine. While on comparing with the traditional systems, which use their signature matcher more or less on its own, we tightly incorporate it into Bro's architecture so as to provide contextual signatures. As we already know, we have two main level which uses additional context for signature matching. First, at a comprehensive level, we extend the expressiveness of signatures. Even though byte-level pattern matching is a central part of NIDSs, mostly signatures allow it to be expressed in terms of fixed strings but Bro already provides regular expressions for use in policy scripts, and can be used for signatures as well. The clarity of such patterns provides us an instantaneous way to express syntactic context. For example, with regular terminology it is easy to express the notion "string XYZ but only if preceded at some point earlier by string ABC". One significant point to be memorized regarding regular expression matching is that, once we have fully constructed the matcher, which is expressed as a Deterministic Finite Automaton (DFA), the matching can be done in O(n) time for n characters in the input, and also (n) time. (That is, the matching always takes time linear in the size of the input, regardless of the specifics of the input.) The "parallel Boyer-Moore" approaches that have been explored in the literature for fast matching of multiple fixed strings for Snort have a wide range of running times—potentially sublinear in n, but also potentially super linear in n. So, on seeing the particulars of the strings we want to match and the input against which we do the matching, regular expressions might prove

fundamentally more efficient, or might not; and hence need experimental evaluations to conclude the relative performance in practice. Also, in the making of a regular expression the matcher requires time potentially exponential in the length of the expression, clearly unaffordable.

Secondly, when on a higher level, the rich contextual state of Bro can be implemented to improve plain matching as described above. While using Bro's architecture, the engine sends events to the policy layer. Here the policy script uses all of Bro's already existing mechanisms to decide how to react.

Due to Snort's large user base, it enjoys a wide-range and up-to-date set of signatures. Hence for flexibility we have designed a custom signature language for Bro is designed, which uses Snort libraries via a conversion program. This program takes an unmodified Snort configuration and creates a corresponding Bro signature set. Although, by just using the same signatures in Bro as in Snort, we are unable to improve the resulting alerts in terms of quality. But even though if we do not go along with additional context, they immediately give us a baseline of already widely deployed signatures. Hence, Snort serves us as a reference. Throughout the paper we compare with Snort both in terms of quality and performance. But in doing so, we found several general problems for evaluation and comparison of NIDSs. We consider that these arise independently of our work with Bro and Snort, and hence described in some detail .While keeping the above limitations in mind, we can now evaluate the presentation of our signature engine and find out that it performed well.

II. CONTEXTUAL SIGNATURES

The centre of Bro's contextual signatures is a signature engine designed with three main goals in mind: (i) expressive power, (ii) the ability to improve alert quality by utilizing Bro's contextual state, and (iii) enabling the reuse of existing signature sets. We shall discuss each in turn. Afterwards, we will present our experiences with Snort's signature set, and finally show examples which reveal applications for the described concepts.

III. REGULAR EXPRESSIONS

A conventional signature usually contains a series of bytes that envoy a specific attack. An indicator of a possible attack can be seen when such a sequence is found in the payload of the packet. Hence the central part of any signature-based NIDS is the matcher. Quiet a lot of NIDSs only allow fixed strings as search patterns, but we would ask for the use of regular expressions. As we are aware that regular expressions present with several significant advantages: firstly, they are flexible than fixed strings, also the perspicuity has made them a well-known tool in many applications, and their power arises in part from providing additional syntactic framework with which to sharpen textual searches. In particular, character classes, union, optional elements, and closures demonstrate the useful for specifying attack signatures.

Astoundingly, given their power, regular expressions can be matched very competently which is performed by compiling the expressions into DFAs whose terminating states indicate whether a match is found or not. A sequence of n bytes can therefore be matched with O(n) operations, and each operation is simply an array lookup— highly efficient.

The total number of patterns contained in the signature set of NIDSs can be quite large. Snort's set, for example, contains 1,715 distinct signatures, of which 1,273 are enabled by default. Matching them independently is very expensive. Nevertheless, in case of fixed strings, there are algorithms for matching sets of strings at the same time. thus, Snort's default engine works iteratively, there has been recent work to replace it with a "set-wise" matcher [8, 12].1 Also regular expressions enable us set-wise matching for free: by using the union operator on the individual patterns, we get a new regular expression which effectively combines all of them. The result is a single DFA that again needs O(n) operations to match against an n byte sequence. Few modifications are essential to expand the interface of Bro's previously existing regular expression matcher to unambiguously allow grouping of expressions.

In spite the eloquence and effectiveness of regular expressions, we have a ground why a NIDS might avoid using them: the underlying DFA can grow very large. Depending on the particulars of the patterns on fully compiling a regular expression into a DFA leads potentially to an exponential number of DFA states. On viewing the complex regular expression built by compiling all individual patterns, this straight-forward move could easily be inflexible. In our understanding while constructing DFAs for regular expressions matching many hundreds of signatures shows that this is indeed the case. One can avoid the state/time explosion in practice, as follows.

As an alternative of pre-computing the DFA, we build the DFA "on-the-fly" during the actual matching. Every time the DFA needs to transfer into a state which is not already constructed, we calculate the new state and document it for future reuse and so restore only those DFA states which are actually needed. A significant observation is that for n new input characters, we will build at most n new states. Moreover, we find in practice that for normal traffic the growth is much less than linear.

Still a point of concern is that given unfavorable traffic—can actually be artificially crafted by an attacker—leading to consumption of more memory by the state construction than available .Hence a memory-bounded DFA state cache was implemented. Configured with a maximum number of DFA states, it expires old states on a least-recently-used basis. In the follow-up, when we say "Bro with a limited state cache," we are addressing to such a bounded set of states (which is a configuration option for our version of Bro), using the default bound of 10,000 states.

Also one of the significant point that should be kept in mind is to combine all patterns contained in the signature set into a single regular expression. Further view suggests that most signatures contain supplementary constraints like IP address ranges or port numbers that confine their applicability to a subset of the whole traffic. On viewing these constraints, we can conclude in groups of signatures that match the same kind of traffic by collecting only those patterns into a common regular expression for matching the group, thereby able to reduce the size of the resulting DFA significantly. Further, to cope with high-volume traffic a very powerful pattern matcher can still be efficient enough.

IV. IMPROVEMENT OF ALERT QUALITY BY USING CONTEXT

Despite the fact that pattern matching is a central part of any signature-based NIDSs, there is potentially great effectiveness in incorporating more viewpoint in the system's analysis prior to generating an alert, to ensure that there is indeed something alert-worthy occurring. One can greatly increase the quality of alerts, while at the same time reducing their quantity, by using the information about the current state of the network .Hence we can say that Bro is an excellent tool for it keeps an ease of accessibility.

On the other hand the new signature engine is intended to fit nicely into Bro's layered architecture as an appendage to the protocol analysis event engine (see Figure 1). Also a custom language for defining signatures has been implemented. A new component placed within Bro's middle layer matches these signatures against the packet stream. A new event can only be inserted into the event stream when there is a match and the policy layer can then choose how to react. Furthermore, one can add information from the policy layer back into the signature engine to control its operation. A signature can denote a script function to call each time a particular signature matches further it consults additional context and indicates the subsequent event should be generated.

Usually, Bro's analyzers extort protocol-specific information by following the communication between two endpoints. For example, the HTTP analyzer extorts URIs requested by Web clients (which includes performing general preprocessing such as expanding hex escapes) and the status code and items sent back by servers in reply, while the FTP analyzer follows the application dialog, matching FTP commands and arguments (such as the names of accessed files) with their corresponding replies. Obviously, this protocol-specific analysis provides considerably more context than a simple view of the total payload as an undifferentiated byte stream.

The signature engine takes benefit of the additional information by incorporating semantic-level signature matching. For example, the signatures can include the concept of matching against HTTP URIs; the URIs to be matched are provided by Bro's HTTP analyzer. After developing the mechanism for interfacing the signature engine with the HTTP analyzer, it is now straight forward to extend it to other analyzers and semantic elements (indeed, we timed how long it took to add and debug interfaces for FTP and Finger, and the two totaled only 20 minutes). Central to Bro's architecture is its connection management. Each network packet is linked with exactly one connection. This concept of connections permits several powerful extensions to traditional signatures. Primarily, Bro reassembles the payload stream of TCP connections. Hence all pattern matching can be done on the actual stream (in contrast to individual packets). Whilst Snort has a preprocessor for TCP session reassembling, which can be accomplished by compiling several packets into a larger "virtual" packet and further passed on to the pattern matcher. Since the consequential analysis is packet-based, it suffers from discretization problems introduced by focusing on packets, such as missing byte sequences that cross packet boundaries. (See a related discussion in of the problem of matching strings in TCP traffic in the face of possible intruder evasion.)

In Bro, it is not essential for a signature match to correspond to an alert; as with other events, this decision is left to the policy script. Therefore one should memorize which signatures have matched for a particular connection. Given this information, it is then possible to specify dependencies between signatures like "signature A only matches if signature B has already matched," or "if a host matches more than N signatures of type C, then generate an alert." In a similar manner, we can for example describe multiple steps of an attack. In addition, Bro observes in which direction of a connection a particular signature has matched, which gives us the idea of request/reply signatures: we can associate a client demand with the corresponding server reply. A typical use is to differentiate between successful and unsuccessful attacks.

Usually, the policy script layer can correlate arbitrary kinds of data with an association with one of its endpoints. This implies that any information we can assume from any of Bro's other components can be used to improve the quality of alerts

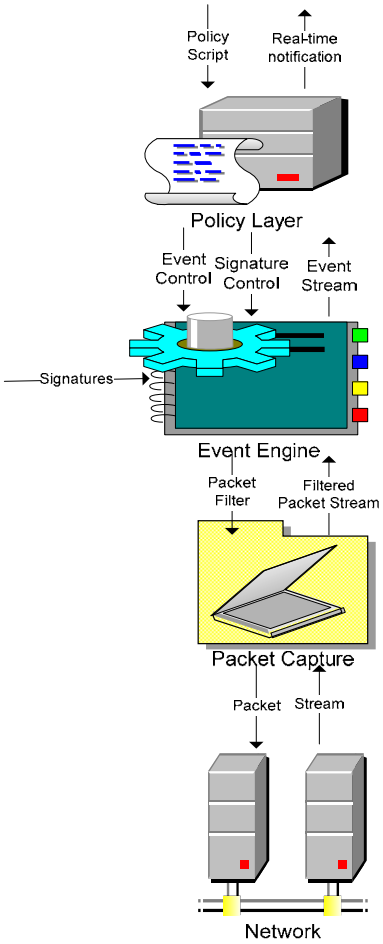

Figure 1: Integrating the Signature Engine

On observing per-connection state for signature matching one question that arises is of state management: at some instance we have to retrieve state from older connections to thwart the system from draining the available memory. But again we can leverage the work already being done by Bro .Autonomously of our signatures, Bro can performs a difficult connection-tracking using various timeouts to expire connections. The signature engine works economically even with large numbers of connections when the matching state to the already-existing per-connection state.

V. SIGNATURE LANGUAGE

A language is needed for defining signatures in any signature-based NIDS. In case of Bro, we need to select between the use of an already existing language and the implementation of a new one. A need for a new language was required because of two reasons. Firstly, it gives us more flexibility and we can incorporate the new concepts. Secondly, if we wish to utilize the existing signature sets, then it becomes simpler to write a converter in some high-level scripting language than to apply it within Bro itself.

Snort's signatures are converted into our signature language since they are wide-ranging, free and often updated.

It is rather difficult to implement a complete parser for Snort's language. We conclude that, its syntax and semantics are not fully documented, and rather often only defined by the source code. Additionally, since internals of Bro and Snort are different, it is just not possible to keep the exact semantics of the signatures.

Identifier and a set of attributes are required for definition of our signatures. Two main types of attributes are there: (i) conditions and (ii) actions. The conditions say when the signature matches, while the actions state what to do in the case of a match. Conditions are further classified into four types: header, content, dependency, and context.

Header that contains matching packet headers limits the applicability of the signature to a subset of traffic. In case of TCP, the first packet of a connection is matched. While in other protocols, this takes place for each individual packet. Generally, header conditions are defined by using a tcpdump - like syntax (for example,

tcp[2:2] == 80 matches TCP traffic with destination port 80), which is very flexible, though some shortcuts (e.g., dst-port == 80) also exists.

Regular expressions are used to define the content conditions. Again, there are two types of conditions: first, the payload statement can be used for the declaration of the expression, in which case it is matched against the raw packet payload. On the other hand, a prefix of an analyzer-specific label can be used, that extracts data and the expression is matched against it. For example, the HTTP analyzer decodes requested URIs. So, http (etc/(passwd |shadow) matches any request containing either etc/passwd or etc/shadow.

The dependencies between signatures are defined by signature conditions. Requires-signature has been implemented, which specifically says another signature has to match on the same connection first, and requires-reverse-signature, which requires the match to happen for the other direction of the connection. Both conditions are subject to negation to match only if another signature does not match.

Finally, context conditions allow passing the match decision on to various components of Bro. If all other conditions match then they are evaluated. For example, we state a tcp-state condition that poses restrictions on the current state of the TCP connection, and eval, which calls an arbitrary script policy function.

Upon meeting of all conditions, the actions related with a signature are executed: a signature is inserted by the event match into the event stream, which includes the value of the event and the signature identifier, corresponding connection, and other context. The signature match is then analyzed by policy layer.

VI. THE POWER OF SIGNATURE

This section discusses, about several examples that convey the power provided by our signatures. First, we exhibit how regular expressions can be used to define more “tight” signatures. Then, we show how to identify failed attack attempts by taking into account the set of software a particular server is running as well as the response of the server. We next demonstrate how to model an attack in multiple steps to avoid false positives, and finally how alert counting is used for identifying exploit scans. We conclude that the examples presented are not supported by Snort without extending its core significantly.

A. Using Regular Expressions

Regular expressions allow far more flexibility than fixed strings. Snort signature generates a large number of false positives. While a corresponding Bro signature using a regular expression for identification of the exploit is more reliable. If an attacker constructs a string of the form “...; <shell-cmds>”, and passes it on as argument of the recipient CGI parameter, vulnerable form mails will execute the included shell commands. Arbitrary order can be used to give CGI parameters, the Snort signature has to rely on identifying the form mail access by its own. But by using a regular expression, we can explicitly specify inclusion of a particular character in the recipient parameter.

B. Vulnerability Profiles

Only some versions of the software are actually vulnerable while most exploits are aimed at particular software. If overwhelming number of alerts is given a signature matching NIDS can be generated, we hence consider the view that the only attacks of interest are those that actually have a chance of succeeding. One doesn’t care if, for example, an IIS exploit is tried on a Web server running Apache, one may not even care. Hence to prioritize alerts based on this kind of vulnerability information is done. Bro’s concept says that call the set of software versions that a host is running its vulnerability profile. The profiles of hosts on the network are collected after protocol analysis, using version/implementation information that the analyzer observes. Signatures can then be restricted to certain versions of particular software.

As a proof of principle, vulnerability profiles are there implemented for HTTP servers, and for SSH clients and servers. We intend to extend the software identification to other protocols.

Future work is to extend the notion of developing a profile beyond just using protocol analysis. We can passively fingerprint hosts to determine their operating system version information by observing specific idiosyncrasies of the header fields in the traffic they generate, or in addition employ active techniques to explicitly map the properties of the site’s hosts and servers. Finally, an add-on to automated techniques, we can implement a configuration mechanism for manually entering vulnerability profiles.

C. Request/Reply Signatures

In addition of pursuing the idea to avoid alerts for failed attack attempts, signatures can be defined that take into account both directions of a connection. In operational use, we see a lot of attempts to exploit CVE-2001-0333 to execute the Windows command interpreter cmd.exe. For a failed attempt, the server typically answers with a 4xx HTTP reply code, indicating an error. These failed attempts can be ignored by, first defining one

signature, http-error, which recognizes such replies. Then a second signature can be defined, cmdexe success, which matches only if cmd.exe is contained in the requested URI and the server, does not reply with an error. In Snort it's not possible to define this kind of signature, as it lacks the notion of associating both directions of a connection.

D. Attacks with Multiple Steps

An example of an attack executed in two steps is the infection by the Apache/mod ssl worm (also known as Slapper), released in September 2002. The worm first probes a target for its potential openness by sending a simple HTTP request and inspecting the response. It turns out that the request it sends is in fact in violation of the HTTP 1.1 standard, and this peculiarity provides a somewhat "tight" signature for detecting a Slapper probe.

If the server spots itself as Apache, the worm then tries to exploit OpenSSL susceptibility on TCP port 443. Two signatures are required that reports an alert only if these steps are carried out for a destination that runs a vulnerable OpenSSL version. Slapper-probe, the first signature, checks the payload for the illegal request. The script function is vulnerable to slapper is called, if found. Using the susceptibility profile described above, the function evaluates to true if the destination is known to run Apache as well as a weak OpenSSL version. If so, the signature matches. Slapper-exploit, the header conditions of the second signature, matches for any SSL connection into the specified network, and the signature calls the script function has slapper probed. The function generates a signature match if slapper-probe has already matched for the same source/destination pair. Thus, Bro generates alert if the combination of probing for a susceptible server, plus a potential follow-on exploit of the vulnerability, has been seen.

VII. CONCLUSIONS

In this work, we tried developing the general notion of contextual signatures as an enhancement on the traditional form of string-based signature-matching used by NIDS. Instead of matching fixed strings in seclusion, contextual signatures enhance the matching process with both low-level context, by using regular expressions for matching rather than simply fixed strings, and high-level context, by taking advantage of the semantic context made available by Bro's protocol analysis and scripting language.

We achieved some major improvements over other signature-based NIDSs such as Snort, by tightly integrating the new signature engine into Bro's event based architecture, which often suffered from generating a huge number of alerts. To leverage Bro's context and state-management mechanisms and to improve the quality of alerts a signature-match only as an event was interpreted, rather than as an alert by itself. Multiple examples have been specified to show the power of this approach: matching requests with replies, recognizing exploit scans, making use of open profiles, and defining dependencies between signatures to model attacks that span multiple connections. Additionally, the freely available signature set of Snort has been converted into Bro's language, to build upon existing community efforts.

As a baseline, we tried evaluating our signature engine using Snort as a reference, that compared the two systems in terms of both run-time performance and generated alerts using the signature set archived at. But in this process, we came across several general problems when matching NIDSs: conflicting internal semantics, incompatible tuning options, the difficulty to devise "representative" input, and extreme sensitivity to hardware particulars. The last two are mainly challenging, because there are no prior signs when comparing performance on one particular trace and hardware platform that we might obtain diverse results using a special trace or hardware platform. Thus, great caution is necessary in interpreting comparisons between NIDSs.

Based on our work, now we are in the course of deploying Bro's contextual signatures operationally in numerous educational, research and commercial environments.

REFERENCES

[1]. Farooq Anjum Dhanant Subhadrabandhu and Saswati Sarkar,"Signature based Intrusion Detection for Wireless Ad-Hoc Networks: A comparative study of various routing protocols", Seas, 2008.

[2]. Hongmei Deng; Xu, R.; Li, J.; Zhang, F.; Levy, R.; Wenke Lee, " Agent-based cooperative anomaly detection for wireless ad hoc networks", Parallel and Distributed Systems, Volume 1, Issue , 0-0 0 Page(s):8, 2008.

[3]. Haiguang Chen, Peng Han, Xi Zhou, Chuanshan Gao, "Lightweight Anomaly Intrusion Detection in Wireless Sensor Networks", Intelligence and Security Informatics, Springerlink, 2007.

[4]. WANG Ding-cheng, JIANG Bin. Review of SVM-based Control and Online Training Algorithms [J]. Chinese Journal of System Simulation, 2007

[5]. CHEN You, SHEN Hua-Wei, LI Yang, CHENG Xue-Qi.An Efficient Feature Selection Algorithm Toward Building Lightweight Intrusion Detection System [J]. Chinese Journal of Computers, 2007

[6]. Desai N. Intrusion Prevention Systems: the Next Step in the Evolution of IDS. Feburary 2003. http://www.securityfocus.com/infocus/1670. Accessed 30 November, 2008.

AUTHORS PROFILE

Dr. Sameer Shrivastava is an Associate professor in the Department of Computer Science and Engineering, in Global Nature Care Sangathan group of institutions, Jabalpur, M.P. India. He received his Ph.D.(Computer Science and Engineering) from Bhagwant University, Ajmer, Rajasthan in 2011. He has 14+ years of experience in teaching and research. His areas of specialization include Network Security. He is a Cisco Certified Network Associate, SUN certified and Microsoft Certified Professional. He has published papers in many International and national level Journals on Network Security and Computer Networks.