Communications Research Centre Canada
An Agency of Industry Canada

Centre de recherches sur les communications Canada
Un organisme d'Industrie Canada

Intrusion Detection System (IDS)
Testing with a Packet Stimulator System

Frederic Massicotte

CRC Technical Note
CRC-TN-2007-003

Communications Research Centre Canada
3701 Carling Ave. Ottawa, ON Canada K2H 8S2

Ottawa, March, 2007

Canada

CRC

Abstract

The relevant commercial product and research literature shows that many techniques may be used to test Intrusion Detection Systems (IDS) that protect computer networks. There are two main techniques for testing IDS detection accuracy: the vulnerability exploitation program approach and the IDS stimulator approach. In the vulnerability exploitation program approach, real attacks are used against real target systems to generate test cases. The currently available solutions are not scalable and they are limited. For instance, the number of vulnerability exploitation programs used in test data sets is often small and the variety of the targeted systems is limited. To overcome this problem an IDS stimulator can be used to generate test cases based on the IDS signature database and to launch the packets corresponding to those signatures against different IDS for testing. However, most current IDS stimulators were developed for attacking IDS and not for IDS testing and evaluation. In this report, we will investigate how an IDS stimulator could generate test cases to identify problems in the IDS configuration or engine and to identify new IDS evasion techniques. To prove this approach, we developed a new enhanced IDS stimulator that we used against Snort and we identified configuration problems and potential evasion techniques when used against intrusion detection systems.

Résumé

Les solutions commerciales et la littérature scientifique montrent que plusieurs techniques peuvent être utilisées pour tester les systèmes de détection d'intrusion (SDI) qui protègent les réseaux informatiques. Il existe deux techniques principales pour tester la précision de détection des SDI: l'utilisation de programmes d'exploitation de vulnérabilités et l'utilisation de stimulateurs de SDI. Dans le cas des programmes d'exploitation de vulnérabilités, on utilise des attaques connues qu'on exécute contre des systèmes fonctionnels pour générer des scénarios de tests. Les solutions actuellement disponibles ne peuvent être utilisées à large échelle et sont limitées. Par exemple, le nombre de programmes d'exploitation de vulnérabilités utilisés pour générer les scénarios de tests sont souvent petits et la variété des systèmes attaquées est limités. Pour résoudre ce problème, les stimulateurs de SDI peuvent être utilisés pour générer des scénarios de tests à partir des signatures de la base de données des SDI et utiliser les paquets correspondant à ces signatures contre le SDI comme vérification. Par contre, la plupart des stimulateurs de SDI actuellement disponibles ont été développés dans le but d'attaquer les SDI et non pour les tester et les évaluer. Dans ce rapport, nous avons enquêté sur la possibilité d'utiliser un stimulateur de SDI pour générer des scénarios de tests pour des SDI tel que Snort dans le but d'identifier les problèmes dans la configuration ou l'engin des SDI et d'identifier des nouvelles techniques pour éviter les algorithmes de détection des SDI. Pour prouver la pertinence que cette approche, nous avons construit un tout nouveau stimulateur de SDI amélioré que nous avons utilisé contre Snort et nous avons trouvé des problèmes de configuration ainsi que de techniques potentielles pour éviter les algorithmes de détection lorsque nous les utilisons contre ces systèmes de détection d 'intrusion.

Table of Contents

List of Figures

1 Acknowledgement

This research project was supported in part by funding from the Director General Spectrum Engineering (DGSE), Spectrum, Information Technologies, and Telecommunications (SITT) Sector of Industry Canada. We also would like to thank François Blais from Sherbrooke University for his contribution to the success of this work.

2 Introduction

2.1 Exploits and Stimulators for IDS Testing and Evaluation

In [1] we reported on the results of an experiment during which we evaluated the response of *Snort* (version 2.3.2 [2]) [2] and *Bro* (version 0.9a9) [3] to well-known attacks on commonly used network based protocols, services and applications. Both *Snort* and *Bro* are advanced and widely used open source intrusion detection systems (IDS). The well-known attacks are actual exploit scripts and programs that run in a tightly controlled test environment whereby all the resulting network packet traffic is captured and examined.

An IDS may also be tested using crafted packets generated by an IDS "stimulator". IDS stimulators such as *Snot* [4], *Stick* [5], *IDSWakeup* [6] and *Mucus* [7] have been developed for two purposes: IDS evasion[1] testing, and IDS attack detection testing and evaluation. An IDS stimulator generates these crafted packets based on knowledge or information about the nature of an attack rather than on scripts or programmed code. For instance, an IDS stimulator may analyze an attack signature in the form of a record in a database to craft the appropriate packets for the attack. This approach is especially effective when an attack exploit signature is known, but the exploit script or program is not available.

With regard to detection evasion, an attacker may use an IDS stimulator to create false attacks that force the IDS to generate many attack event notifications (herein called "events"). These events flood the network to hide the real event that is triggered when the attacker launches the real attack. A stimulator can also be used to cause a denial of service on the IDS. If too many attack packets have to be analyzed or when too many events have to be logged, the IDS may be overwhelmed and begin to ignore packets.

In the case of *Snort*, there is an efficient denial of service presented in [8]. The authors provided well-crafted packets to *Snort*, based on *Snort* version 2.4.3 signatures that were similar enough to be analysed by a rule, but not similar enough to trigger an event. Only 4Kbps of traffic containing these packets is required to cause a denial of service on *Snort* (it looses packets).

The other use of an IDS stimulator is for IDS testing and evaluation. In [7], the authors used this technique for IDS cross-evaluation. They attempted to determine by analysis if it was possible to

[1] In this context, an evasion may include certain kinds of denial of service attacks directly on the IDS as well as specially constructed packets that are not recognized by the IDS as an attack.

use *Snort* rules (which represent attack signatures) to generate packets and to use the generated packets as test cases to test other IDS.

We believe that IDS stimulators can be used for three other IDS testing and evaluating aspects not addressed by existing IDS stimulators:

1. testing the IDS engine with its own signature database;
2. including testing strategies for the packet generation; and
3. identifying and testing the packet (or sequence of packets) "space"[2] for each attack signature.

These aspects can be used to automate the testing process and thus improve IDS technology.

2.2 Project Purpose and Methodology

Through this project, the concept of an IDS stimulator will be examined as a tool that would be effective in an engineering test and evaluation environment such as the Protocol Analysis Lab (PAL) operated by SITT. In conjunction with exploit driven testing, stimulator based testing has the potential to provide a comprehensive testing and evaluation facility.

For this project, we will only address the use of IDS stimulators for IDS testing and evaluation. We developed our own IDS stimulator that is an improvement over the known IDS stimulators [4], [5], [6] and [7]. This improved stimulator addresses 2 of the 3 aspects mentioned above by adding testing strategies to generate packets and by automating the IDS engine testing with its own signature database. Even though the third aspect is not directly addressed by this report, we also explore the importance of identifying the packet space for each IDS signature.

It is important to note that our IDS stimulator may also be used to identify IDS evasion or IDS denial of service techniques based on the problems (parameters) specified in the testing and evaluation phase. In particular, several problems were found using our IDS stimulator and the corresponding techniques could be used by attackers to evade detection by *Snort*.

Note that *Snort*, being a freely available IDS, was ideal for this project. In particular, as the *Snort* source code is available, the source code could be checked to confirm *Snort*'s processing behavior when anomalies were found during testing. Thus, the outcomes from this project can be used to improve commercial products.

The remainder of this report is structured as follows. Section 2 describes the main functionality that must be included in an IDS stimulator to accomplish testing and evaluation of IDS. Section 3 presents our IDS stimulator and its functionality. Section 4 describes the development process that still needs to be completed for this project. Section 5 presents the results we obtained by evaluating and testing five versions of *Snort*. Conclusions are drawn in Section 6.

[2] The term "space" means a packet or a group of packets that closely resemble or are related to a known packet or group of packets.

3 Test and Evaluation of *Snort* using IDS Stimulators

Several existing IDS stimulators such as *Snot*, *Stick* and *Mucus* can generate packets from *Snort* rules. However, none of them is able to do this for each and every *Snort* rule. There are three common drawbacks to these existing tools:

1. the number of Snort plug-ins[3] that can be use by these stimulators is limited;
2. the mapping of each signature within the packet space is undetermined or incomplete; and
3. the incorporation of testing strategies is incomplete or missing entirely.

The *Snort* rules are composed of plug-ins (modules) that examine certain aspects of a packet. The number of plug-ins that the exiting IDS stimulators are able to understand is limited because these stimulators were developed to generate packets from the *Snort* version 1.8 signature database. New plug-ins have since been developed for the more recent versions of *Snort*. However, *Snot*, *Stick* and *Mucus* were not updated to reflect the use of these new modules. Furthermore, it is not likely that these stimulators will ever be updated.

Moreover, depending on the type of rules tested, these stimulators do not have facilities to execute nor evaluate *Snort* using different test strategies. For example, in the case of *Snort* rules with the *flow* plug-in with the parameter *established*, we would like to test the two possible directions of the packet flow for the attack. In the case of the *flowbits* plug-in, the different combinations of attack scenarios that influence the *flowbits* variables have to be tested. In the following sections, we will explore the importance of including testing strategies and packet space mapping to signature in these tools when they are used for IDS evaluation and testing.

3.1 Developing a Testing Strategy

There are three levels of testing that an IDS stimulator must address: the packet scenario level, the rule level and the plug-in level. Each of these testing strategies describes a testing level of abstraction that must be verified for each IDS signature.

3.1.1 Packet Scenario Level

At the packet scenario level, we want to test attacks that are modeled by the IDS as a sequence of packets that must occur before the IDS will provide an event message to the network administrator. With *Snort*, the packet scenario level is essential when several *Snort* signatures use the *flowbits* plug-in on a specific group of variables. These *Snort* rules, based on the value of the *flowbits* variables, can only be applied when these variables have certain values. Thus, we have a partial ordering of the *Snort* rules based on the variable values and we can generate all possible sequences of the *Snort* rules. These *Snort* rules can be used to construct a state machine where the states are the values of the variables and the events are the IDS events corresponding to these rules. This state machine can then be transformed into a tree. Each path from the root to each leaf corresponds to the packet scenario that has to be tested. A method to test state machines is described by transforming a state machine into a graph (tree) where each path from the root to the leaf correspond to a test case that must be conducted. Thus, for each packet scenario we generate a test case that corresponds to a possible sequence of *Snort* rules triggered based on their *flowbits* variables.

[3] *Snort* plug-ins (modules) are specific purpose additions to the basic IDS engine.

3.1.2 Rule Level

The rule level corresponds to testing each rule independently. They can be viewed as a logical expression where all the predicates have to be true to raise an event. In the case of *Snort*, the header rule and the payload plug-ins in each rule are the predicates of the logical expression. In particular, all the plug-ins in the rules must be true for the rules to raise an event. Testing techniques that used logical expressions such as *Variable Negation* and *Modified Condition Coverage* can be used to properly test each *Snort* rule.

3.1.3 Plug-in Level

The plug-in level focuses only on testing each plug-in individually to find cases where the plug-in is not able to work based on its specification. For example, the plug-in *pcre* that checks regular expressions in packets can be tested as a separate component. In this case, the regular expression can be transformed into a state machine and state machine testing techniques can be applied. The *content* plug-in that verifies whether or not a string is in a packet can be tested using different types of string sequences and illegal characters that are not allowed to be specified in a rule by the user.

3.2 Packet Space versus IDS Signature Space

In many situations, multiple packets can match an IDS signature. In the case of *Snort* a very large number of packets correspond to only one signature. It is important to investigate for flaws in the packet space represented by the signature. The *Snor*t rule in Figure 1 states that an alarm will be raised if a packet that comes from the EXTERNAL_NET to the HTTP_SERVERS on port HTTP_PORTS and contains the text ".printer" in the URL. There are many TCP packets that could match this signature that are not attack packets nor are they even legal packets.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-IIS ISAPI .printer access"; flow:to_server,established;
uricontent:".printer"; nocase; reference:arachnids,533;
reference:bugtraq,2674; reference:cve,2001-0241; reference:nessus,10661;
reference:url,www.microsoft.com/technet/security/bulletin/MS01-023.mspx;
classtype:web-application-activity; sid:971; rev:10;)
```

Figure 1 Snort Example

Let P be the set of all IP packets that can be generated. This set is finite because IP packets have a length bound and a finite set of options fields. Suppose it is possible to create two subsets of P that describe the validity of a packet. $P_{protocol}$ represents the set of packets that can be understood by the transport protocol and the application protocol of a target system. We know that some packets will only be understood in some contexts, but to keep this explanation in two dimensions, let us say that this set contains all *legal* packets that can be understood. P_{other} represents the packets that cannot be understood by any protocol. We assume, but it still needs to be verified, that $P_{protocol} \cup P_{other} = P$ and $P_{protocol} \cap P_{other} = \varnothing$. Let $P_{KnownAttack}$ be the set of packets that represents known attacks in the IP packet space. We know that $P_{protocol} \cap P_{KnownAttack} \neq \varnothing$ and $P_{other} \cap P_{KnownAttack} \neq \varnothing$ because they are attacks that used well-defined protocol packets to exploit a vulnerability and some others that exploit a vulnerability using packets outside the protocol specification. Let $P_{signature\ i}$ be the packets that correspond to the IDS signature i. A good IDS is one that $P_{signature\ 1} \cup \ldots \cup P_{signature\ n} = P_{KnownAttack}$ where n is the number of signatures in the IDS signature database. When considering

Snort, some signatures correspond to packets outside $P_{KnownAttack}$ that are in P_{other}. Thus, *Snort* is vulnerable to the false attack IDS evasion techniques described in the previous section. An evaluation of the IDS's rules is required to identify these problems.

It is a difficult task to evaluate the packet space of an IDS signature, but an IDS stimulator could be used to address part of this problem. For instance, a set of packets for $P_{signature\_i}$ is generated (where the corresponding attack packets must meet the protocol specification to succeed) with packets that are outside the protocol specification and these packets are provided to the IDS to verify that the IDS does not generate any events or alarms.

3.3 IDS Stimulator Missing Implementation

It is difficult to find an algorithm that will generate the corresponding packet for all the *Snort* 2.x plug-ins. This method or algorithm is the opposite of the method *f(packets,signature)* in an IDS that takes packets and checks whether or not they match a signature. The problem is to find an algorithm which, for each signature, is able to find the corresponding groups of packets. Thus, an IDS stimulator can be viewed as a system with two components. A parser component to create the object that will be used for generating the packets such as *Snort* rules and the packet generating component. There are mainly four IDS stimulators available: *Stick*, *Snot*, *IDSWakeup* and *Mucus*. We did a survey of the functionality and capability of these IDS stimulators and decided to not reuse these tools for this project, but to develop our own solution to the problem for several reasons.

First, none of these tools seems to be in ongoing development and there is no plan that would indicate that the functionality and capability of these tools will be updated by their owner. In fact, three of them (*Stick*, *Snot* and *Mucus*) were developed to generate *Snort* rules for version 1.8 (2003) and earlier and are not able to generate packets for the *Snort* 2.x plug-ins such as *byte_test*, *byte_jump*, *flow* and *flowbits*. These IDS Stimulators have a limited number of *Snort* plug-ins that they are able to use, thus, they are not able to generate all packet sequences for all *Snort* rules. To test the plug-ins for rule and attack scenario levels, all plug-ins must be implemented in the IDS stimulator to generate the packets. In the case of *IDSWakeup*, it used simulated attacks already in its databases, thus is it not relevant to this project because it cannot be used to test the IDS engine within its signature databases. We cannot analyze the packet space for each of its IDS signatures because its IDS signatures are not the source of information needed to create its simulated attacks. Second, because the packet generation components of these tools rely on the *Libnet*[4] library and that we already have a *Snort* rules parser in our toolkit the reusability of one of these tools is minimal. In fact, the *Libnet* library is the only important component to reuse from these tools.

4 Proof-of-concept Design

For this first version of the proof-of-concept for the IDS stimulator, called *Stimulator2*, we decided to focus on rule level testing. By verifying when the logical expression described by the *Snort* rule is true, we can partially address the packet space problem by creating packets outside the protocol specification, if possible, and to implement all the plug-ins missing from the other IDS stimulators. We used the rule level testing strategy as a starting point because an IDS engine is as only good as

[4] http://www.packetfactory.net/libnet/

its knowledge, in this case, its rule database. The proof-of-concept currently only works with *Snort*, but it is able to work with all versions before *Snort* 2.4.3. It was decided to only use *Snort* for this project because it is the only IDS with an open rules database with enough rules to evaluate this approach for IDS testing and to identify IDS evasion attacks. This proof-of-concept IDS stimulator is used to test whether *Snort* is able to detect the packets corresponding to a *Snort* rule for each *Snort* rule. The proof-of-concept was designed to generate, for each *Snort* rule, a corresponding group of test cases that will trigger the rules based on several testing strategies addressing different functional components of the IDS. We developed a framework in Java to allow easy incorporation into the framework of different testing strategies with a minimum of effort. The proof-of-concept IDS Stimulator works with an IDS evaluation framework to test the IDS.

The IDS Stimulator is responsible for generating the group of packets for each *Snort* rule based on the current testing strategy. It is responsible for generating the data set that will be used for the IDS evaluation. The IDS evaluation framework is responsible for submitting the test cases (packets generated for each IDS rule) to the IDS and verifying if *Snort* is able to detect the attack using the corresponding rules for the packets in the test cases. Figure 2 presents our proof-of-concept IDS stimulator.

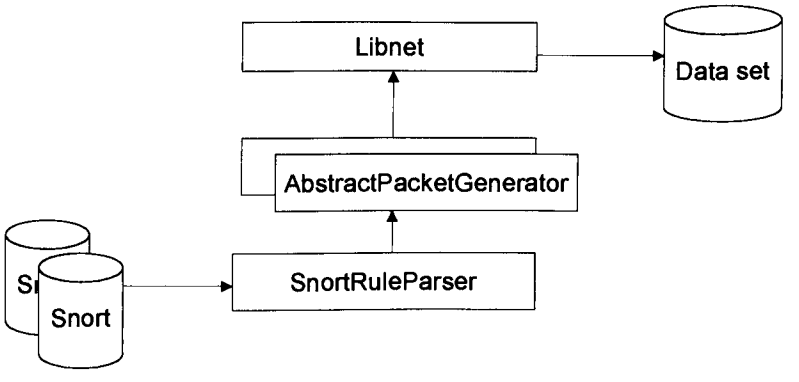

Figure 2 IDS Stimulator Overview

First, all the *Snort* rules from the version under test are analyzed and stored using *SnortRule* objects into memory by the *SnortRuleParser* module. The objects are given to each *AbstractPacketGenerator* that generates the packet corresponding to each *SnortRule* object using the corresponding testing strategy. There are more than 10 *AbstractPacketGenerator* classes that implement different testing strategies to identify problems within the *Snort* IDS engine in relation to its configuration and signature database. Each *AbstractPacketGenerator* uses the *Libnet* module to generate the packets corresponding to that signature in a *tcpdump* file format. If there are multiple groups of packets that have to be tested for this signature, a file is generated for each group of packets. Each of these files represents a test case that will be used to test the *Snort* IDS version. The files are then documented and integrated into the data set.

The IDS evaluation framework is an enhanced version of the framework presented in the paper [1] that we used to evaluate *Snort* and *Bro* with real attack data. Figure 3 presents the IDS evaluation framework that we enhanced to test each *Snort* version. It consists of four components *IDS Evaluator*, the data sets, the *IDS Results Analyzer* and the IDS.

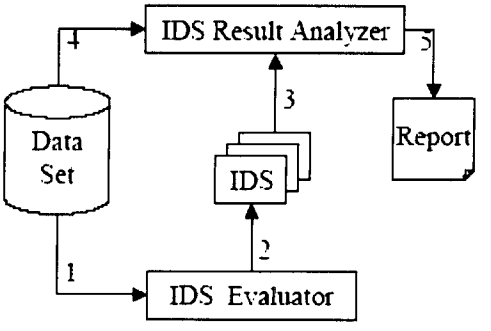

Figure 3 IDS Evaluation Framework Overview

First, the *IDS Evaluator* takes each test case from the data set and then provides it to the IDS. In the case of IDS stimulation testing, only the data set generated with an IDS signature database is submitted to that IDS. Thus, the data set generated with *Snort* 2.4.3 rules is submitted only to *Snort* 2.4.3. The *IDS Results Analyzer* is responsible for automatically analyzing the events of the IDS. The system relies on the corresponding signature and documented success of the detection to know if the IDS has properly detected the attack or not. For example, a group of packets generated using signature *i* based on test strategy *t* is documented to be detectable by the IDS. Thus, if the IDS raises the event from signature *i* when this group of packets is submitted to the IDS, we know that the IDS has passed this test case. In the case of a test that is not supposed to be detected by the IDS, it is successful when the IDS did not provide any event notification from the corresponding signature that is evaluated in that test scenario.

5 Missing Functionality

In our IDS Stimulator, there are still minor improvements to be made before being able to generate all test cases for each *Snort* rule. However, the *Snort* versions tested in this project contain between 2500 to 3200 rules and the missing functionality only affects a few rules. These improvements can be combined into two groups: the missing *Libnet* library interaction functionality and the missing *Snort* plug-ins.

5.1 Missing Libnet Interaction Functionality

In the case of *Libnet* library four tasks must be completed for our program to be able to process each *Snort* rule:

1. IP Options interface implementation;
2. RPC interface implementation;
3. all ICMP types interface implementation; and
4. the resolution of the destination broadcast address bug.

5.1.1 IPOptions

There is a *Snort* plug-in *ipoption* that is used to check the IP Options in a packet. An interface must be developed between our proof-of-concept and the *Libnet* library function that sets the IP Options within a packet. This task has been left outside of this proof-of-concept because only a few *Snort*

rules (3 rules in all tested versions) use this plug-in. Also, the *Libnet* library interface for the IP Options would have been time consuming during the development process because the *Libnet* library relies on strings and not on IP Options objects to set the IP options in the packet structure.

5.1.2 RPC

There is a *Snort* plug-in *RPC* that is used to look at certain remote procedure calls (RPC) fields in a packet. Once again an interface between our proof-of-concept and the *Libnet* library must be developed for generating test cases using this functionality. This plug-in has been left aside like the IP Options based on the number of *Snort* rules using it (4 rules for only two *Snort* versions) and the level of specification of the function that set RPC packet in the *Libnet* library.

5.1.3 ICMP

To detect attacks, *Snort* decodes three protocols: TCP, UDP and ICMP. One of the *Snort* rule files (ICMP.rules) contains rules to detect each type of ICMP packet. In the *Libnet* library, there are only a limited number of ICMP types that have been implemented (echo, network mask and unreachable). It is not clear that we could use these methods to set other types of ICMP packets because some ICMP packets such as *unreachable* have the same field construction as many other types of ICMP packets. Thus, it was decided for the proof-of-concept to only generate echo, network mask and unreachable ICMP packets in the data set.

5.1.4 Broadcast Destination IP Address

A problem was found in the *Libnet* library when we created an IP packet with a destination address of 255.255.255.255. The function in this situation returns an error. The rules using this type of IP address for a destination have been removed from the data set of test cases generated.

5.2 Missing *Snort* Plug-ins

Snort has 52 plug-ins (rule options) classified into five classes:

1. the header plug-ins;
2. the post-detection rule plug-ins;
3. the non-payload plug-ins; and
4. the payload plug-ins and the reference plug-ins.

There are 6 "header" plug-ins that define the protocol, the source and destination IP and the protocol source and destination port that are used by a packet. There are 6 "reference" plug-ins that are used to describe the rules such as the message sent to the user and the reference to vulnerability databases. There are 15 "payload" plug-ins that describe the packet payload. There are 20 "non-payload" plug-ins that describe the protocol fields in the packet. There are 5 "post-detection" plug-ins that describe what needs to be done when an alarm is raised by a signature against a packet. Only the "reference" plug-ins are not relevant to packet generation. Most of the *Snort* plug-ins available up to and including *Snort* version 2.4.3 were implemented in our IDS stimulator.

Four *Snort* plug-ins that were not implemented or were not working properly in our proof-of-concept software were: *fragbits*, *ANSI*, *ftpbounce* and *flowbits*.

5.2.1 Fragbit

The *fragbits* plug-in was implemented, but some of the packets generated with the *Snort* rules that contain this plug-in are not detected by all tested versions of *Snort*. It is unclear whether this is due to a problem in our proof-of-concept or in the *Snort* engine. More study is needed to find the source of the problem. This plug-in only represents a few *Snort* rules in the version we tested in this project (6 rules for 2 versions and 5 rules for 3 versions) and thus, this should not have an impact on the overall results.

5.2.2 ASN1

The plug-in *ASN1* was not implemented in this proof-of-concept version. This plug-in allows us to look at problems in this encoding and identifies some problems in attack packets. This plug-in was not implemented because of the effort needed to implement it and potential results ratio. This plug-in is only used in a few *Snort* rules in the version that we tested in this project (4 rules in one version and 10 rules in 4 versions).

5.2.3 FTPBounce

For the same reasons as for the previous plug-in, the *ftpbounce* plug-in was not implemented in the proof-of-concept. This plug-in is used in only a few *Snort* rules (1 rule in 1 version) and its impact on the results is limited to only one version.

5.2.4 Flowbits

The plug-in used for verifying multiple packets in one session, called *flowbits,* was implemented, but only one test strategy was developed for this plug-in. To test this plug-in, a series of attack packets must be included in the test cases. However, the proof-of-concept only creates one attack packet per test case so the *flowbits* plug-in could not be completely tested in this project. The rule level testing abstraction can only test part of this plug-in. For example, the rules that use methods such as *isset* in *flowbits* on a variable are classified as not being able to be detected by *Snort* (this is the expected behavior for *Snort*) by the testing strategy. We are able to verify, using the rule level testing strategy that *Snort* should not detect these packets as attacks using the test cases generated with these rules. To be able to identify this packet as an attack, *Snort* must identify a packet that matches a rule that sets the same variable checked using *isset* in these rules before the rule is triggered. Thus, by only using the rule level testing strategy, the attack packet that raises the bit monitored in the rule by *isset* is not included in the test case. However, the single attack packet testing strategy for *flowbits* was verified and implemented within the proof-of-concept. But, to accurately test this plug-in, a combination of attack packets follow the attack scenario specified by the rules using *flowbits* has to be included in the test case. To properly test this plug-in, our proof-of-concept would have to construct test cases with complex attack scenarios that involve multiple packets that change the state of the variable values in a session.

We estimate that these missing *Libnet* and *Snort* plug-in implementations are a minor factor for each tested version of *Snort* in this project. In fact, they represent at most only 5% of all the *Snort* plug-ins for each version of *Snort* tested.

6 Results and Outcomes

Five versions of *Snort* were tested with our proof-of-concept program: versions 2.2.0, 2.3.0, 2.3.1, 2.3.2 and 2.4.3. For each version problems were found in *Snort* using our IDS stimulator with different testing strategies. These problems can be classified into two classes: the configuration problems and the design problems. The configuration problems arise when a specific configuration of *Snort* disables certain rules that are not triggered based on the configuration. In this case, we also identified unexpected behavior that is probably related to the configuration problem. We also only used the default *Snort* configuration for testing all versions. The design problems are related to the design of the tools and not to its configuration. These problems can only be resolved by modifying the design and implementation of the *Snort* software.

6.1 Configuration Problems

In the case of configuration problems, all problems identified are related to the *http_inspect* preprocessor. There are 8 different configuration problems identified by our proof-of-concept as shown in the following sub-sections.

6.1.1 Uricontent Distance

The Uricontent distance problem arises when a *Snort* rule contains the *uricontent* plug-in followed by the plug-in *content* and the *distance* plug-in. The packet generated by our proof-of-concept is not detected by the corresponding *Snort* rule. We were not able to determine in the time available why this problem occurs but it seems to be related to the configuration or behavior of the *http_inspect* preprocessor.

6.1.2 Special Character

For an obscure reason, when the hexadecimal character 0x09 or 0x0a is used in the *uricontent* plug-in of the *Snort* rules, the packet generated with this rule is not detected by the corresponding *Snort* rule. The value 0x09 corresponds to the horizontal tab and the value 0xa0 corresponds to the new line feed. Again, this problem seems to be related to the configuration or behavior of the *http_inspect* preprocessor because a property setting can be activated that changes these characters to other values before giving the packet to the rule engine of *Snort*.

6.1.3 Client Port 80

When a client system (browser) requests a connection using port 80 or 8080 as a source port, which is normally used as a HTTP server port, the generated packet is not detected by its corresponding *Snort* rule. Once again this problem seems to be related to the configuration or behavior of the *http_inspect* preprocessor.

6.1.4 HTTP GET

The HTTP GET problem is also related to the *http_inspect* plug-in. When there is a *Snort* rule with the word HTTP or GET in the *uricontent Snort* plug-in, the packet generated by our proof-of-concept is not detected by *Snort* with its corresponding rule. The *http_inspect Snort* preprocessor normalizes the packet payload and removes the HTTP and GET keyword of the packet before submitting it to the *Snort* rule engine. A discussion on this subject can be found in http://www.snort.org/archive-3-233.html.Further studies must be conducted to check if there are

other *Snort* rules in the rule database that are there to replace these rules when they are used with this preprocessor.

6.1.5 Port 80

Some of the *Snort* rules use the *uricontent* plug-in but look at packets targeting ports other than 80 and 8080. The default configuration of *Snort* for the *http_inspect* preprocessor is 80 and 8080. Thus, the packets generated with these rules are not detected by their corresponding *Snort* rule.

6.1.6 Triple Slashes

The triple slashes problem is similar to the HTTP GET problem. The packet generated with the *Snort* rules that has two consecutive slashes were not detected because the *http_inspect* preprocessor normalizes the packet payload and automatically removes the slashes that are not needed. For more information about this, you may find a discussion on this subject at http://www.snort.org/archive-3-233.html.

6.1.7 Traversal

The traversal problem is also similar to the HTTP GET problem and the triple slashes problem. The packet generated with *Snort* rules that contain directory traversal (\..\ or \.\) were not detected by *Snort*. The *http_inspect* preprocessor once again normalizes the packet payload by removing the directory traversal before providing it to the *Snort* rule engine. Discussion on this subject can be found in http://www.snort.org/archive-3-233.html.

6.1.8 Unicode

The packets generated with *Snort* rules that contain Unicode encoding in the *uricontent* plug-in were not detected. In this case, we suspect that the *http_inspect* plug-in is responsible for this situation and transformed the packet payload again. It is not clear if there are *Snort* rules that correspond to all the *Snort* rules not detected because of the problems noted above.

6.2 Design Problems

There are four design problems that were identified by our proof-of-concept: rule verification, rule ordering, packet space and the upper layer protocol unaware problems.

6.2.1 Rule Verification

The *Snort* rules verification problem is related to the *Snort* configuration, but when we realized the impact of this problem for the Snort engine, we classified it as a design problem. *Snort* does not check each packet with all the rules in certain scenarios. For instance, if a packet has triggered three rules (*Snort* default configuration), no more rules will be checked using this packet and the three events will be logged. This default value can be changed to another number in the *Snort* configuration file to examine more rules in this type of scenario. This behavior was enabled by the developers of *Snort* to optimize (minimize) event logging for the *Snort* rule engine. However, this technique could lead to IDS evasion. For instance, for each *Snort* version, the proper value for this parameter is related to the signatures contained in the *Snort* signature database and should be the number of rules that could intersect a particular attack scenario. Based on this design specification

of looking at only the first *n* rules that match a packet, we see at least two problems with this approach.

First, if the value in the configuration is very high (near 100) and if the *Snort* signature database allows it, an attacker could, based on the *Snort* rule specification, create a packet that would intersect a large number of rules (around 100). If we were to send this packet on a network monitored by *Snort* at a certain interval, it could cause the *Snort* engine to drop packets because *Snort* will use processor/system resources to log a large number of events even if these packets do not represent a real attack.

Second, if the value in the configuration is low (near 3) it could be possible, based on the *Snort* rules, to initiate an attack that would trigger three rules, none of which correctly identify the attack.

Based on the current state of the *Snort* rules, we believe that there is potential for these two types of attack to be conducted successfully against *Snort*. In fact, the limit of three rules seems insufficient because some of the proof-of-concept generated packets with *Snort* rules triggered more than three rules.

6.2.2 Rule Ordering

The order of the packets is also important because some of the packets that triggered many *Snort* rules could have an effect on the events raised by *Snort*. In one case, the order of two rules changed the events raised by *Snort* because the first rule changed the state of a *flowbits* required by the other rule. If the rules were inverted, the results would have been different because the effect of the first rule required by the second rule could not have been propagated and none of the rules would have been triggered. It is not clear whether the *Snort* developers carefully ordered the rules to ensure no logic error in the *Snort* rule database. However, this could be an easy mistake to make when someone is using *flowbits* to write *Snort* rules. Further study must be made of this subject to check whether this is a good design or whether it could lead to new IDS evasion techniques against *Snort*.

6.2.3 Signature versus Packet Space

We were able to determine that *Snort* signatures represent packets that are not attack packets. The packet space identified from a *Snort* signature contains packets that do not meet the protocol specification and that do not represent valid attack attempts. Many *Snort* signatures are not built to distinguish among packets meeting the protocol specification. In fact, most of the packets generated with the *Snort* rules do not meet the protocol specification, but are still detected by *Snort* as attack packets that could affect a target. This leads to another problem, described in the next section, we found with *Snort* being protocol unaware of certain transport, session and application protocols.

6.2.4 Upper Layer Protocol Unaware

In most cases, *Snort* is not able to verify the validity of transport, session and application protocols. The protocol layer above the protocol specified in the *Snort* rule header has to be specified using the payload plug-ins. In the case where the plug-ins are not used to check certain aspects of the upper layer protocols any packet that matches the rule header and the payload plug-ins will match the rule even if the encoding of the protocol above the one specified in the header is not correct

based on its specification. In fact, the *Snort* plug-ins implementations seem to be deficient when decoding the session and the application protocol monitored by the rule.

7 Conclusion

In this report, we presented a proof-of-concept of a new enhanced IDS stimulator and a data set of false attacks that can be used to test and evaluate the *Snort* IDS as well as identify vulnerabilities in the system. We found that *Snort* has configuration and design problems that could lead to IDS evasion techniques that attackers may be able to use. In the case of *Snort* configuration problems, some rules could never be triggered in the default configuration and it is not clear whether there are other rules that are in the *Snort* signature database that detect the corresponding attacks in the default configuration. The design problems in *Snort* allow attackers to use an IDS stimulator for IDS evasion and denial of service because the packet space of certain *Snort* signatures contains non-attack packets. Moreover, the rule order and the number of triggered rules can be used by an attacker to evade detection by *Snort* using crafted packets.

Some improvements are still needed for this IDS stimulator to be used to its full potential. For now, we identified three improvements that need to be completed: the missing plug-ins and interfaces to *Libnet*, developing an algorithm to determine the packet space of each *Snort* rule and adding more testing strategies.

Based on the results obtained in this project, we plan to develop a prototype that will be able to implement what is missing in the proof-of-concept to create more test cases and that will help to find other issues with *Snort* IDS.

References

[1] Massicotte, F., Gagnon F., Labiche, Y., Couture, M., Briand, L.: Automatic Evaluation of Intrusion Detection System Proceedings of the Annual Computer Security Applications Conference (ACSAC) Miami, Florida December 2006

[2] Beale,J., Fistor J.C.: Snort 2.0 Intrusion Detection. Syngress Publishing (2003).

[3] Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Network 31 (1999) 2435-2463

[4] Sniph: Snot www.securityfocus.com/tools/1983 (2006)

[5] Aubert, S.: IDSWakeup www.hsc.fr/ressources/outils/idswakeup/ (2006)

[6] Giovanni, C.: Fun with Packets: Designing a Stick. packetstormsecurity.nl/distributes/stick.htm (2006)

[7] Vigna, G. Robertson, W. Balzarotti, D.: Testing network-based intrusion detection signatures using mutant exploits. Proc. ACM conference on Computer and communications secuiryt (2004) 21 – 30.

[8] Smith R., Estan C., Jha S., Backtracking Algorithmic Attack Complexity in NIDS Proceedings of the Annual Computer Security Applications Conference (ACSAC) Miami, Florida December 2006