Evasive Attack on Stateful Signature-based Network Intrusion Detection Systems

(Technical Report CS-2008-18)

Tung Tran[1], Issam Aib[1], Ehab Al-Shaer[2], and Raouf Boutaba[1]
[1]David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada
[2]School of Computer Science, DePaul University, Chicago, USA
(t3tran, iaib)@uwaterloo.ca, ehab@cs.depaul.edu, rboutaba@uwaterloo.ca

ABSTRACT

Network Intrusion Detection Systems (NIDS) have a very important role in network security. Many NIDS evasion techniques as well as solutions were proposed in the literature. Supporting stateful signatures is a very critical function in a signature-based NIDS because many multi-stage attacks can only be detected by tracking multiple rules (signatures) matching. In order to detect these attacks, the session state corresponding to an attack is normally simulated in a NIDS. However, due to the application protocol complication and overheads, it is impossible to have a complete simulation of the session state in a NIDS. Many evasion techniques exploit this Achilles' heel of a NIDS: it does not have a complete image of the actual system it protects, which leads to the situation that the NIDS and its protected system see and explain things differently; therefore, the NIDS is possibly evaded by creative attacks.

In this paper, we propose an evasion technique to Snort rule sets using *flowbits*, making it more susceptible to false negatives. Moreover, we also suggest practical solutions with controllable false positives to prevent the proposed attack and formally proved that the solutions are complete and sound. We implemented a tool called **SFET** which can automatically parse a rule set, generate evasion sequences against the rule set and produce a patch to the rule set. A significant number of publicly available rule sets are found vulnerable to the proposed attack, which seriously affects the security of Snort users' systems. Although our proposed attack applies to Snort, the idea can be applicable to any NIDS supporting stateful signatures.

1. INTRODUCTION

Intrusion Detection System (IDS) is the generic term given to any hardware, software, or combination of the two that monitors a system or network of systems looking for suspicious activity [9]. An IDS is used to detect different kinds of malicious behaviors and attacks that can compromise the security and trust of a computer system. Alerts raised by an IDS allow the system admin, in a swift manner, to take the corresponding actions like applying patches to the system, blocking the attack by setting firewalls accordingly, etc. to minimize possible damages caused by attackers. An IDS can be either a signature-based IDS or an anomaly-based IDS based on the methodology used by the engine to generate alerts. Moreover, based on the type and location of an IDS, it can be categorized as either a network-based IDS (NIDS) or a host-based IDS (HIDS).

Snort [15] is a popular, free and open source Network Intrusion Prevention System (NIPS) and Network Intrusion Detection (NIDS) capable of performing packet logging and real-time traffic analysis on IP networks [10]. Snort might be considered a lightweight NIDS because it has a small footprint, has relatively small requirements, does not always demand a suite of specialized servers, and runs on a variety of operating systems [3]. Even though Snort supports some anomaly detection through its pre-processors, Snort is more like a signature-based IDS and famous for its intrusion detection capabilities that match packet contents against a set of rules. Snort rules are easy to write and Snort supports a strong and flexible rule language with a variety of options, which allow users to inspect all fields of a packet.

The *flowbits* option was first introduced in Snort 2.1.1 and is most useful for TCP sessions, as it allows rules to generically track the state of an application protocol [22]. Sometimes, looking at one packet at a time, one rule at a time could not tell whether an event occurs if the event is a series of actions that we need to keep track of. In fact, many times we need to inspect more than just a single packet with more than one rule to detect an attack, especially complicated ones. Before the addition of *flowbits*, Snort could not do this. The *flowbits* detection option makes snort rules stateful and allows the detection engine to track state across multiple packets in a single session. Stateful detection is so important that sometimes it is implemented separately for a specific service [25].

With *flowbits*, we can essentially set a flag that another rule can check and take into consideration. We can think of this in terms of streams and multiple rules. We can look at flowbit usage in terms of a chain of events, or a logic flow: If condition 1 happens, set a flowbit. If this flowbit is set and you see condition 2 but not condition 3, generate an alert. That second event can occur many packets later in the stream, or seconds or minutes later in the stream [3]. The *flowbits* option works by using labels to set and change the session state. The *flowbits* option is used with this format: *flowbits*: [set|unset|toggle|isset,reset,noalert][,<LABEL>];

E.g: Here is a rule set using *flowbits* from BleedingEdge [2]:

alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"BLEEDING-EDGE FTP USER login flowbit"; flow: established, to_server; content: "USER"; nocase; *flowbits*: set, login; *flowbits*: noalert;)
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg: "BLEEDING-EDGE FTP HP-UX LIST command without login"; flow: established, to_server; content: "LIST"; nocase; *flowbits*: isnotset, login;)

Table 1: Example of a Snort flowbits rule set

These rules use *flowbits* to follow a FTP session. The first rule checks if a user is trying to log in the FTP server and sets the label "login". The second rule will raise an alert if the user uses the "LIST" command, however, only if the user has not logged in yet. In other words, the second rule is triggered only if the label "login" is not set. This rule set might do a good job in detecting someone using the "LIST" command without even trying to log in the server. However, an attacker can always try to log in first (even not authorized) and then use the "LIST" command. This way allows the attacker to execute the LIST command without really logging in, but Snort still thinks that the user already logged in. As a result, the use of the "LIST" command is allowed. In this case, Snort misjudges the actual session, and of course no alert is raised.

Because of its powerfulness as well as usefulness, the *flowbits* option is now getting more attention and used more frequently. In about 20K rules we collected from many different sources on the internet, about 3K rules (15%) make use of *flowbits*. The *flowbits* concept is used in many different IDSes as well, like in Cisco IPS[4], Bro[13],etc. For example, Bro allows users to write a script to raise an alert based on a sequence of events. This is equivalent to the use of *flowbits* in Snort if we consider handling each event by a rule and set *flowbits* accordingly so that Snort will raise an alert if it see the event sequence.

Complicated attacks against services with complex protocols are normally hard to detect. The *flowbits* option allows Snort to keep track of the session state and raise an alert if an event happens when the session is in a specific state. With the use of *flowbits*, Snort keeps track of the session by maintaining its own session state, called *in-snort session state*. The in-snort session state is supposed to reflect the actual session state, in other words, the actual session state is simulated in Snort as much as possible. However, due to the protocol complication (of the session's corresponding service) and overheads caused by doing so, it is impossible to have a complete simulation of the actual session state in Snort.

Altogether, these points bring up an interesting evasion idea to the use of *flowbits* in Snort: if an attack can make Snort misunderstand the actual session state, i.e. if the attack can force a difference between the actual and the in-snort session states (Snort thinks that the session is in a state but the session is actually in a different state), it is possible to bypass Snort detection. In this paper, we propose an evasive attack on Snort rules using *flowbits*. The attack primarily deals with Snort, but in general, it can be applied to any NIDS supporting stateful signatures. The proposed attack requires access to Snort rules; however, if rules are not available, reverse engineering of network signatures techniques [11] might help.

This paper is organized as follows: Section 2 summarizes related work. Section 3 describes the flowbit-based evasion of Snort alarms. Section presents 4 formally specifies the language of all possible flowbits evasions. We then discuss the methods to trigger a rule without affecting the actual session state in Section 5. Section 6 gives a comprehensive example of a flowbits evasion attack. In Section 7, we present solutions to the proposed attack. The correctness and completeness of the solution are studied in Section 8. Implementation and evaluation results are discussed in Section 9. Finally, we provide conclusions and plans for future work in Section 10.

2. RELATED WORK

Even though there are attacks on host-based IDS like mimicry attacks introduced by Wagner and Soto [26], attacks and evasion techniques on NIDS seems to be a more interesting topic and have been very well studied in the literature. Ptacek and Newsham [14] were the first to bring up a way to evade a NIDS by using TCP Segmentation and IP Fragmentation, and FragRoute is the tool created to carry out these evasion techniques. A NIDS needs to carry out TCP segments and IP fragments reassembly to defend these evasion techniques. Besides, Handley and Paxson [9, 13] discussed evasion techniques based on inherent ambiguities of the TCP/IP protocol which leads to the difference between a NIDS and its protected system in performing TCP segments and IP fragments reassembly. Traffic normalization suggested by Handley et al. [7] tries to remove these ambiguities by patching the packet stream. Another solution to this was proposed by Shankar and Paxson [18], Active Mapping, which eliminates TCP/IP-based ambiguity in a NIDS' analysis with minimal runtime cost. Active Mapping efficiently builds profiles of the network topology and the TCP/IP policies of hosts on the network; a NIDS may then use the host profiles to disambiguate the interpretation of the network traffic on a per-host basis. This idea is implemented in the Stream5 [28] preprocessor of Snort, which makes Snort a "target-based" NIDS.

Besides NIDS evasion techniques, there are attacks on NIDS as well. Snot [20], Stick [6], IDSWakeup [1] and Mucus [12] are over-stimulation tools that cause a DOS attack to Snort by trying to overload Snort with alerts from mutated packets constructed from Snort rules. Another DOS attack to a NIDS comes from the algorithmic complexity issue [6,19], especially the authors in [19] presented a highly effective attack against Snort, and provided a practical algorithmic solution that successfully thwarts the attack.

Related to the signature (rules) testing and evaluation, Vigna et al. [24] introduced a mechanism that generates a large number of variations of an exploit by applying mutant operators to an exploit template. These mutant exploits are then run against a victim host protected by a network-based intrusion detection system. The results of the systems in detecting these variations provide a quantitative basis for the evaluation of the quality of the corresponding detection model. Besides, Mucus [12] is also a testing tool for Snort rules by using matching packets with random data in the packet fields not considered by a given rule.

Rubin et al.[17] observed that different attack instances can be derived from each other using simple transformations. TCP and application-level transformations are modeled as inference rules in a natural-deduction system. Starting from an exemplary attack instance, they used an inference engine to automatically generate all possible instances derived by a set of rules. They created AGENT, a tool capable of both generating attack instances for NIDS testing and determining whether a given sequence of packets is an attack. However, our attack is not an instance generated by AGENT, assuming that the rule set represents the original exemplary attack instance. Our attack is neither a TCP nor an application-level transformation. Existing evasion techniques might be used in our attack; however, these techniques only apply to injected packets which are not parts of the actual session. Although our paper only deals with Snort rules, which are mainly manually written by users, automatic generated semantic-aware signatures [27] or session signatures [16] are still possibly vulnerable to our proposed attack. In order to avoid false positives, these generated signatures must consider "innocent" paths (or sequences) which are not attack instances. Our attack exploits these "innocent" paths and tries to convince Snort that the session is following one of the "innocent" paths.

3. EVASION OF SNORT ALARM USING FLOWBITS ATTACK

In this section, we first analyze factors which help the establishment of the evasion technique. We then formalize the evasion problem and provide a solution to the problem.

3.1. Flowbits Evasion Establishment

- ***Packet-based property of Snort***: As we know, Snort is a packet-based NIDS and basically most packets received by Snort, after processed by preprocessors, are passed down to the detection engine to make sure all possible attacks are considered. Exploiting this feature, we can construct a packet that is not processed by the receiver's application layer but still inspected by the Snort detection engine and triggers a given rule. This will cause the inconsistency between the actual and in-snort session states.
- ***Loose rules***: A loose rule is a rule with simple matching options that potentially matches a lot of traffic. A loose rule matches a packet by only checking if its payload contains some strings and does not examine other fields and the packet

structure. It is the fact that if a rule is too tight, it might cause false negatives, and if the rule is too loose, it might cause false positives. However, a loose rule used with *flowbits* might cause false negatives. The reason people still write loose rules is in a normal connection session, without the attacker's interference, packets that match loose rules only occurs when the session is in a specific state. So loose rules rarely cause false positives and still exist in many Snort rule sets using *flowbits*. Moreover, loose rules are easier to write and reckless users normally create loose rules in their policy. Because loose rules are easy to be triggered and normally cause false positives, they might be exploited to cause a difference between the actual and in-snort session states when *flowbits* options are used to keep track of the session state. Each rule when triggered is supposed to change the in-snort session state according to the actual connection state. A loose rule might be exploited by the attacker to change the in-snort session state but preserve the actual session state.

- ***Complex sessions***: A complex session may have many possible states and many possible transitions back and forth between these states. Moreover, in order to avoid false positives, a Snort policy needs to consider "innocent" paths, which Snort should not raise an alert. Complicated states, complex state transitions, and "innocent" paths might give the attacker a chance to put the session into a desired state before finishing the attack so that the attack goes unnoticed.

- ***Insecure detection approaches***: For a given attack, even simple one, there are possibly many approaches to detect it, and more than one rule set can be created to detect the attack. Even though these rule sets semantically do the same thing (raise an alert if the attack occurs), they might make use of different session states, so it has different ways to come up with the alert. There are approaches which logically leave a path allowing the attacker to finish the attack without being detected. For example, in order to detect if a normal user (non-admin) uses a bad command in a telnet session, one straight approach is to only raise an alert if the bad command use is detected when the session is in a state that a normal user has logged in. Another approach is to always raise an alert if the use of the bad command is detected except when the session is in a state that the admin has logged in. While the first approach can always catch the attack because the attacker has to log in as a normal user before using the bad command, the second approach opens a door for the attacker: if he can pretend that he has successfully logged in as the admin, then he can execute the bad command.

While the packet-based property of Snort and loose rules allow the attacker to force a difference between the actual and the in-snort session states, complex sessions and insecure detection approaches make the evasion possible.

3.2. Flowbits Evasion Problem

Problem definition: Let S = $\{R_1, R_2, \ldots, R_n\}$ be a Snort rule set which uses flowbits. Let T be a subset of S ($T \subset S$) that raise alerts. We denote T as the set of all target rules of S. The problem is to find all possible packet sequences that successfully attack the service protected by S yet manage to not trigger any rule from T.

A target rule is normally a non-intermediate rule (usually with high priority) which indicates a successful attack when it is triggered. When *flowbits* is used in a rule set, many rules act as intermediate steps leading to the triggering of a target rule if the attack is successful. Intermediate rules normally don't raise alerts and this can be done by using "noalert" with *flowbits*. *flowbit:noalert* is a critical function. It enables us to use a rule that would hit on a lot of traffic that is not of interest, but that must occur before a packet that would be of interest in a session [3].

Definition 1 (Target rule set) *It is the set of all target rules in a rule set.*

Definition 2 (Target rule group) *It is a group of target rules that have the same match options except the flowbits conditions.*

It is conceptually possible that a rule set contains several target rules. In this case, we need to define and solve the problem for each target rule group separately so that to detect evasions that try to mislead Snort to trigger an alarm different than the one the attack really does.

A packet sequence can include packets coming from both directions: server to client and client to server. From the attacker's perspective, a packet can come from the attacker's side and the other side. However, in most of the time, the attacker plays the client role. So in this paper, we assume that the attacker always comes from the client side, and he tries to carry out an attack to a service at a server and does not want to be detected by Snort. Nevertheless, the attack concept is still applicable if the attacker is from the server side.

In this paper, we only consider the session states considered by Snort because Snort might not need to use all possible states from a real session to detect a specific attack. It is very important to understand two crucial concepts: the *actual* *session state* and the *in-snort session state*. The actual session state is the state where the session is truly currently in, and the in-snort session state is the state that Snort thinks the session is currently in.

Definition 3 (Session state) It is defined as a group of labels that are currently set. If n is the number of labels used in the rule set, then there are potentially 2^n different session states.

Definition 4 (Target state) It is a session state where a target rule in T can be triggered (to change to another state).

Definition 5 (Non-target state) It is a session state where no target rule in T can be triggered (to change to another state).

Definition 6 (Target packet) It is a packet that matches any target rule in T and presumably the last packet in the packet sequence of a real attack.

Definition 7 (Evadable rule) It is a rule that can be triggered by the attacker to change the in-Snort session state but preserve the actual session state. A rule is evadable or not depends on whether the attacker can construct a packet that can trigger the rule without affecting the actual session. This is discussed in Section 5.

An evadable rule can be triggered by two different kinds of packets: a packet (from the connection session) that is supposed to trigger the rule at a given time and correctly reflects what Snort thinks about the session, and a packet that is not supposed to trigger the rule and makes Snort misjudge the session. Given a rule R_i, let P_i represent the first kind of packets, called normal packet, and P^*_i represent the second kind of packets (if R_i is evadable), called evasion packet. P_i^* causes a change in the in-snort session state (by triggering R_i), but has no effect on the actual session state.

A packet sequence is considered a successful attack if and only if it puts the session in one of the target states right before the corresponding target packet (the last packet in the sequence) is sent. Therefore, a packet sequence is considered a successful attack but does not trigger the target rule if right before the target packet is sent, the packet sequence puts the actual session in one of the target states and puts the in-snort session in one of the non-target states.

We can assume that, when the actual session is in one of the target states and the in-snort session is in one of the non-target states, the attacker will always trigger the sending of the corresponding target packet. As a result, the problem can be redefined as finding all possible packet sequences that put the actual session in one of the target states and the in-snort session in one of the non-target states. Moreover, a rule set can detect all packet sequences that trigger a target rule in T if and only if it can detect all packet sequences that put Snort in a target state. So from now on, we don't consider a packet sequence that includes a target packet as its last packet because the target packet can be implicitly added to the end of the packet sequence when the attacker performs the real attack.

4. LANGUAGE OF ALL FLOWBITS EVASION

From the rule set S, a corresponding state diagram can be created to show all possible reachable session states and transitions between them. This state diagram can be considered as a DFA. Because we deal with two separate session states: the actual one and the in-Snort one, each of these session states correspond to a DFA, say D_s and D_a. Both D_s and D_a have the same alphabet, set of states, start state and the set of accept states. The only difference between them is the transition function. They are constructed as shown in Alg.1.

Algorithm 1 Construction of Snort (D_s) and Actual (D_a) Session DFAs

```
1: Set of states: reachable session states constructed from the rule set
2: Start state: the state where no label is set.
3: Accept states: all target states
4: Alphabet $\Sigma = \{P_i: R_i$ is not a target rule$\} \cup \{P_i^*: R_i$ is evadable and $R_i$ is not a target rule$\}$
5: // Transition function
6: for all non target rule $R_i$ do
7:    for all state A do
8:       if $R_i$ can be triggered at A leading to state B then
9:          Add $(A, P_i) \rightarrow B$ to both $D_s$ and $D_a$
10:         if $R_i$ is evadable then
11:            Add $(A, P_i^*) \rightarrow B$ to $D_s$
12:            Add $(A, P_i^*) \rightarrow A$ to $D_a$
13:         end if
14:      end if
15:   end for
16: end for
```

1.Construct D_s (S) and D_a (S) based on the rule set S and evadable rules in S
2.Construct $\neg D_s(S)$
3.Construct D_e (S) = D_a (S) ∩ $\neg D_s(S)$

Table 2. Steps to find all possible evasion sequences

Let $L_s(S)$ and $L_a(S)$ be the languages corresponding to D_s and D_a respectively. While $L_s(S)$ represents all packet sequences that Snort thinks to put the session in a target state, $L_a(S)$ represents all possible packet sequences that truly put the session in a target state.

The goal is then to find all possible packet sequences that truly put D_a in a target state but not D_s. These packet sequences must hence be accepted by D_a and rejected by D_s. In other words, these packet sequences are accepted by both the D_a and $\neg D_s$. If we consider these packet sequences as a language, say $L_e(S)$, then:

Lemma 1 (Language of all flowbits evasions) *The language of all packet sequences (or evasion sequences) that successfully attack the service protected by a rule set S using flowbits evasion is equal to:*

$$L_e(S) = L_a(S) \cap \neg L_s(S) = L(D_e(S)) \quad (1)$$

$$D_e(S) = D_a(S) \cap \neg D_s(S) \quad (2)$$

$L_e(S)$ therefore represents all possible packet sequences (or ***evasion sequences***) that successfully attack the service without the Snort's detection. Table 2 shows steps to find all possible evasion sequences.

5. TRIGGERING A SNORT RULE WITHOUT AFFECTING THE ACTUAL SESSION STATE

An important task the attacker needs to do in the *flowbits* evasion is to construct a packet which causes the triggering of a given rule to change the in-snort session state but does not have any effect on the actual session state. In practice, there are many possible conditions that can be exploited to construct such a packet. For example, if the Snort IDS is misconfigured, existing NIDS evasion techniques [15,8,13] can be used to accomplish this task. However, in this paper, we concentrate on exploiting two existing factors to create evasion packets: the packet-based property of Snort and loose rules.

Besides, we also discuss the possibility of constructing evasion packets for two kinds of rules: rules triggered by packets coming from the client side and rules triggered by packets coming from the server side. It is always easier for the attacker to control packets coming from the client side (the attacker's side) than the server side.

5.1. Packet-based property of Snort

As mentioned above, Snort is a packet-based NIDS and most packets received by Snort are checked by the detection engine. This feature allows the creation of evasion packets. There might be many ways to achieve this, but one method is to construct a packet that matches a given rule, however, with "out of order" sequence number. This packet is not processed by the receiver's application layer but still examined by the Snort detection engine and then triggers the rule. For example, if the expected sequence number of the next packet (from the client) is X, we can construct a packet with sequence number X-1. Snort with stream5 preprocessor enabled knows that the packet does not have an expected sequence number for the session stream and is overlapped with a previous packet (assuming that this previous packet has some payload). Snort does exactly as the protected host does: not reassemble this packet into the session stream. However, the packet is still passed down to the detection engine because the packet might match TCP-based attacks where sequence number is not important (e.g: Nmap[29] uses TCP packets with random sequence number to probe a host's OS). This issue was tested with Snort 2.8.1 [21] (the newest version at this time).

Packets constructed by exploiting this feature are injected into the connection session with the purpose of faking interactions (requests and responses) between the client and the server to make Snort misunderstand the session. There are two cases according to two kinds of rules:

Rules matching traffic from the client side: It is always possible to construct a packet (with "out-of-order" sequence number) to fake a request from the client and inject it into the connection session. Therefore, any rule matching traffic from the client side can be triggered without causing the actual session state change.

Rules matching traffic from the server side: It is easy to fake a response from the server but it is hard to deliver the packet to the Snort IDS. In order to deliver a packet to the Snort IDS, the attacker needs to have control over a computer from a network where the crafted packet can reach the Snort IDS (e.g.: a computer from the same network with the server the attacker is trying to attack). Notice that even though the attacker is in the same network with the server, carrying the attack directly from local is still detected if the Snort IDS is configured to detect attacks carried out in the local network. Therefore, constructing fake responses from the server is still necessary.

The scenario where the attacker can control a computer from which the packet can be delivered to the Snort IDS is not rare. For example, an employee from a company has his own computer or a student has an account to access a server in the university network, or the attacker has compromised a computer in the network before, etc. In this case, the Snort IDS thinks that the packet comes from the server and processes it normally.

In addition, when dealing with rules matching packets from the server side, sometimes the injected packet needs to be repeatedly sent to make sure that it comes to the Snort IDS before the real response from the server. When the real response reaches the Snort IDS, the in-snort session has already been in a state where the real response is no longer important, however, the injected response is ignored by the client and the real response is processed normally at the client side. This situation occurs when there are more than one possible response from the server for a given request, and based on the response from the server, the in-snort session state is changed accordingly.

5.2. Loose rules

When a rule is loose, it does not thoroughly match a packet by using tight options like *dsize*, *depth*, *offset*, etc. A loose rule can wrongly explain the intention of a packet if the packet just happens to match the rule but logically does something else. Moreover, it is very possible to create or trigger the sending of such a packet. The packet can be created from the connection session itself (no need to be crafted and injected). There are 2 cases according to two kinds of rules:

Rules matching traffic from client: Although depending on the service protocol, most of the time the attacker can easily make a request from client that matches a loose rule but logically does something else rather than the rule expected. For example, a loose rule checks if a user currently in a FTP session is trying to quit the session by examining packets from the client to see if any packet has “QUIT/n” in its payload. The attacker can make a request to create a directory named “QUIT”, which happens to have ““QUIT/n” in the packet payload and causes Snort misjudge the session.

Rules matching traffic from server: It is harder to evasively trigger a loose rule matching traffic from the server side than the client side because traffic from the server side is not always controllable. However, when dealing with interactive protocols, there are many tricks the attacker can use to cause the server to send a packet containing desired strings and then trigger the rule. Let’s say if a rule simply checks for any packet from the server in a telnet session which contains the string “Granted” in the payload, the attacker can issue an invalid command containing “Granted”. The server will send back a complain of unknown command, which happens to contain the command name “Granted”, and hence triggers the rule. Another trick is the attacker can create a folder with name “Granted” and then try to list all the folders.

5.3. Summary

```
Evadability(Rule R)
1- If R matches traffic from client → return EVADABLE
2- If R contains only content options → return HIGHLY EVADABLE
3- Otherwise → return PROBABLY EVADABLE
```

Table 3. Algorithm to determine the evadability of a rule

In summary, a rule’s content and the scenario context decide whether a packet can be constructed to trigger the rule without affecting the actual session state. In other words, a rule is evadable or not depends on its content and the scenario context.

If a rule matches traffic from client: it is always evadable because the attacker can exploit the packet-based property of Snort to construct a corresponding evasion packet. If the rule is also loose, the attacker has another choice (exploiting the rule looseness) to accomplish the task. On the other hand, if a rule matches traffic from server: the probability the rule is evadable depends on several aspects: whether the attacker has access to a computer where fake responses (from the server) can reach the Snort IDS, service protocol, and the rule’s looseness. If the rule is loose, it is very likely that the rule is evadable. Otherwise, the rule is evadable with small possibility. Table 3 shows the algorithm to determine the evadability of a rule.

6. Example

The rule set in Table 4 is created to follow up an FTP session. It will raise an alert if a non-admin user tries to do anything related to an important file which should only be accessed by the Admin. Rule 1 and Rule 2 determine if a non-admin user is logging in, Rule 3 indicates that the user is denied to login, Rule 4 checks if the user is successfully logged in, Rule 5 indicates that the user has logged out of the FTP session, and Rule 6 checks if the logged-in user tries to do anything with a restricted important file and raises an alert. Only rule 6 is the target rule.

R1	alert tcp 192.168.0.101 any -> 192.168.0.100 21 (msg:"FTP Normal User Login Attempt - Send username"; flow:established, to_server; content:"USER";depth:5;nocase;content:!"Admin";within:5;content:"|0D0A|"; *flowbits:set,NonULogA; flowbits:noalert;)*
R2	alert tcp 192.168.0.101 any -> 192.168.0.100 21 (msg:"FTP Normal User Login Attempt - Send password"; flow:established, to_server; content:"PASS"; depth:5; *flowbits:isset, NonULogA ; flowbits:set, NonULogA2;flowbits:noalert;)*
R3	alert tcp 192.168.0.100 21 -> 192.168.0.101 any (msg:"FTP login denied";flow:established,to_client;flags:A; *flowbits:isnotset, NonULogedIn; flowbits:isset, NonULogA ; flowbits:isset, NonULogA2; flowbits:unset, NonULogA ; flowbits:unset, NonULogA2; flowbits:noalert;)*
R4	alert tcp 192.168.0.100 21 -> 192.168.0.101 any (msg:"FTP login granted"; flow:established,to_client; content:"230 Login successful.|0D0A|";nocase;flags:AP; *flowbits:isset, NonULogA2;flowbits:set, NonULogedIn ;flowbits:noalert;)*
R5	alert tcp 192.168.0.101 any -> 192.168.0.100 21 (msg:"FTP user exits";flow:established, to_server;content:"QUIT|0D0A|"; nocase; *flowbits: isset,NorULoggeddIn; flowbits:unset,NonULogedIn;flowbits:unset, NonULogA ; flowbits:unset, NonULogA2;flowbits:noalert;)*
R6	alert tcp 192.168.0.100 any -> 192.168.0.101 21 (msg:"Normal User accesses important file"; flow:established, to_client; content:"Windows";content:"system32";content:"sam"; nocase; *flowbits:isset, NonULogedIn;)*

Table 4. A rule set to detect a non-admin user accessing an important file from a FTP session

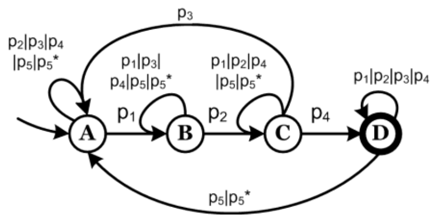

Figure 1: D_s of the FTP rule set

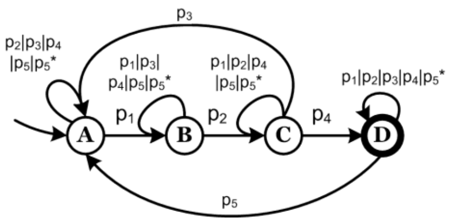

Figure 2: D_a of the FTP rule set

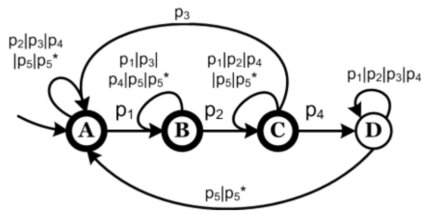

Figure 3: $\neg D_s$ of the FTP rule set

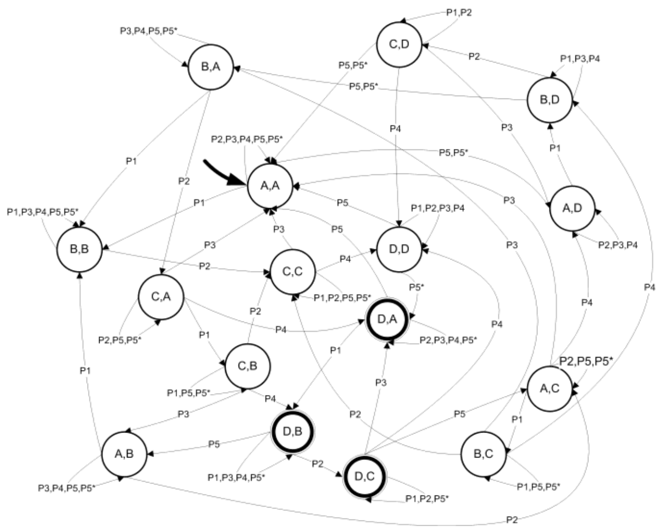

Figure 4: D_e of the FTP rule set

Theoretically, any rule is evadable, but in this example, for simplicity, we assume that only R5 is evadable from the attacker's perspective. It means that the attacker can come up with a way to trigger Rule 5 without affecting the actual session state. The attacker can accomplish this in many different ways which are discussed in Section 5.

There are 3 labels used in this rule set which put the session in $2^3 = 8$ possible states. However, there are only 4 reachable states including the empty state (no label is set): A ={}, B={NonULogA}, C={NonULogA, NonULogA2} and D={ NonULogA, NonULogA2, NonULogedIn}

The in-Snort DFAD$_s$ and actual session state DFAD$_a$ are depicted in Fig.1 and Fig.2. $\neg D_s$ is constructed as in Fig.3. The intersection of D_s and D_a, which is D_e is constructed in Fig.4, where (A,A) is the start state and (D,A), (D,B), and (D,C) are accept states. The language corresponding to this D_e represents all possible packet sequences that successfully attack the FTP server.

For example, P_1 P_2 P_4 P_5* is a packet sequence accepted by this D_e. The attacker can apply this packet sequence to perform a real attack. The attack is carried out as follow: The attacker logs in as a normal user (not Admin) with correct username and password. This action needs P_1 and P_2 to be sent from the attacker and leads to the sending of P_4 from the server to indicate that the user is successfully authorized. The next step the attacker needs to do is to cause the sending of P_5*. There are 2 options to create P_5^*. The first option is to manually construct and inject into the connection a packet that matches Rule 5 but has out-of-order sequence number. The second option is to send a packet that matches Rule 5 but logically does something else rather than exiting the session as Snort thinks. The attacker can create a directory named "QUIT", which makes Snort misjudge the session and think that the user has logged out. After that, the attacker can take the last step of the attack which is downloading or accessing the restricted important file at the server. This action won't trigger the target rule.

7. FLOWBITS EVASION RECTIFICATION

7.1. Manual Solution

The admin needs to re-examine rules that he thinks evadable. If the admin is not sure which rule is really evadable, it is safe to assume that all rules are evadable and constructs all possible evasion sequences as shown previously to see which rules can be exploited to evade Snort if they are evadable. These rules should be rewritten in a tighter manner: matching different packet fields (from different layers like TCP, IP) by using non-payload options; the packet payload should be carefully matched by using tight options like *offset, depth,* etc. Moreover, if possible, the admin should limit rules matching packets coming from the attacker's side as much as possible because rules that match packets coming from the protected side (normally from server) are harder to evade, while rules matching packets coming from the client side are always evadable as discussed in Section 4.

7.2. Ideal Solution

An ideal solution is the solution that automatically changes the rules so that the new rule set semantically does the same thing as the old rule set but is not vulnerable to the proposed attack. However, the difficulty to find such an ideal solution is to automatically understand the semantics of the rule set. Moreover, even if the rule set's semantics is understood, it is possible that any rule set which follows exactly the semantics is always vulnerable to the proposed attack. Therefore, finding an ideal solution is very hard and sometimes impossible.

7.3. Proposed Solutions

In this section, we suggest solutions to patch vulnerable rule sets. The idea is to create a new rule set that detects not only all attack sequences but also all evasion sequences. If the number of evasion sequences is small, extra rules can be added to detect each of these sequences without causing much overhead (number of added rules). Normally, a small vulnerable rule set has a small number of evasion sequences. Besides, there might be an exponential number of evasion sequences, so patching each evasion sequence only works effectively for small rule sets. On the other hand, if the rule set is large (the number of evasion sequences is also large), another approach is used to patch the rule set based on the common features among all evasion sequences: always contain at least one evasion packet. Moreover, a target packet always follows an evasion sequence in a real attack is an aspect that can be considered as well.

7.3.1. Solution for small rule sets

Our proposed solution for small rule sets is trying to detect all possible evasion sequences by adding extra rules to the old rule set with acceptable overhead. We can simultaneously consider the D_e as a directed graph, where each state is a node and each transition is a directed link. Theoretically, we need to add a rule set to detect each path from the start state to an accept state (called an evasion path), however, the number of evasion paths is possibly infinite and we can not add infinite number of rules to Snort. We need to find a way to add minimum number of rules but still possibly detect all evasion paths. It turns out that it is enough to consider only simple evasion paths (a simple path is a path that does not have a circle) because rules added to detect all simple evasion paths can detect all evasion paths. Moreover, it is sufficient to consider only subset paths over all simple paths. Algorithm 2 details the procedure.

Algorithm 2 Flowbits Rectification for Small Rule Sets

```
1: // preprocessing
2: for all target state $t \in D_e$ do
3:     remove outgoing transitions of $t$
4: end for
5: create $SP$ the set of all simple paths from the start state of $D_e$ to a target state.
6: $k \leftarrow 0$;
7: for all simple path $P \in SP$ do
8:     $k \leftarrow k + 1$;
9:     Let $P = q_1 q_2 \dots q_m$, where $q_i$ is the signature (or an evasion) of rule $R(q_i)$
10:    Create a new flowbits rule set $S_k$.
11:    Let $A_0^k$ be the start state of $S_k$ (empty flowbits state)
12:    for $i \leftarrow 1$ to $m$ do
13:        Create flowbits label $A_i^k$
14:        Add to $S_k$ the rule consisting of:
15:        begin
16:        flowbits: isset, $A_{i-1}^k$; //the flowbits condition
17:        all options in $R(q_i)$ in the original rule set (header and body) except the flowbits options
18:        flowbits: set $A_i^k$;
19:        flowbits: noalert;
20:        end
21:    end for
22:    // target rule
23:    Create flowbits label $A_{m+1}^k$
24:    Let $R(q_a)$ be the target rule associated with target state $R(q_m)$
25:    Add to $S_k$ the rule consisting of:
26:    begin
27:    flowbits: isset, $A_m^k$; //the flowbits condition
28:    all options in $R(q_a)$ in the original rule set (header and body) except the flowbits options
29:    flowbits: set $A_{m+1}^k$
30:    end
31: end for
```

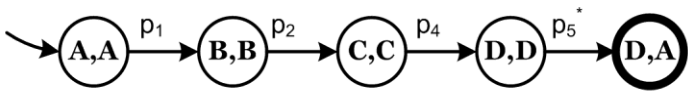

Figure 5: Simple evasion path(s) collected from D_e

R₁(1)	*flowbits*:set,**A-12**;*flowbits*:noalert;
R₁(2)	*flowbits*:isset, **A-12**; *flowbits*:set, **A-13**; *flowbits*:noalert;
R₁(3)	*flowbits*:isset,**A-13**;*flowbits*:set,**A-14**;*flowbits*:noalert;
R₁(4)	*flowbits*:isset,**A-14**;*flowbits*:set,**A-15**;*flowbits*:noalert;
R₁(5)	*flowbits*:isset,**A-15**;

Table 5. Rules added for the simple path in Fig.5

The set of simple paths SP (Alg.2) collected from the D_e of Fig.4 has one simple path (after removing subset paths) as shown in Fig.5. There are five rules added for this simple path as shown in Table 5.

The fact that Alg.2 searches for all simple paths that reach target states (line 5) makes its running time potentially prohibitive. We prove in Appendix D that the complexity of this phase ranges from M^2 (D_e consisting of a single simple path) to $|T| \times (M^2 - |T| - 1) \times |\Sigma|^{M^2 - |T| - 1}$, where M is the number of states of D_s and |T| is the size of the target rules set.

Because of this complexity, Alg.2 is only suitable for rule sets of a small size. In the following, we present a different solution that is feasible for large rule sets.

7.3.2. Solution for large rule sets

We want to detect all evasion sequences but sometime we can not find all of them or maybe there are too many of them to deal with, which cause too much overhead. We need a way to cover all evasion sequences without considering each of them separately. We know that any evasion sequence needs to exploit at least one evadable rule, i.e. it contains at least one evasion packet. Furthermore, in a real attack, a target packet is always sent after an evasion sequence.

The idea is to set a flag whenever an evadable rule is triggered. After that, if Snort sees the target rule and the flag is set, it will raise an alert. However, we do not need to do this with all evadable rules. An evadable rule needs to set the flag when it is triggered only if the rule causes the rule set vulnerable ($L_e(S)! = \emptyset$) even if all the other rules are not evadable. This kind of evadable rule is said vulnerable. The reason is if two or more evadable rules together cause the rule set vulnerable, then at least one of them is vulnerable (this is proved in Appendix C). Therefore, an evasion sequence always contains an evasion packet whose corresponding evadable rule is vulnerable.

Algorithm 3 Flowbits Rectification for Large Rule Sets

```
input $T_m$ : a target rule set
construct $D_s$ and $D_a$ corresponding to $T_m$
// Find vulnerable rules of $D_s$
set $V_m = \{\}$; // set of vulnerable rules
for all $R_i$ in S do
    construct $D_s^i$ (resp. $D_a^i$) from $D_s$ (resp. $D_a$), where $D_s^i$ and $D_a^i$ have as evadable transitions only those related to $R_i$.
    $D_e^i \leftarrow D_a^i \cap \neg D_s^i$
    if $D_e^i$ has a reachable target state then
        $V_m \leftarrow V_m + \{R_i\}$
    end if
end for
if $V_m \neq \phi$ then
    create new label $F_m$
    Add rule $R_m$ consisting of:
    begin
     flowbits: isset, $F_m$; //the flowbits condition
     all the common options and flowbits in the rules of $T_m$ (header and body) except the condition set of flowbits.
    end
    for all $R_i \in V_m$ do
        Add to $R_i$ "flowbits:set, $F_m$
    end for
end if
```

Alg.3 has a polynomial time complexity. In Appendix E, we prove that its worse case running time is $(|R| \times (|\Sigma| M^2 + 1)$, where |R| is the size of the rule set, M is the number of states of D_s, and |T| is the size of the target rules set.

R_3	*flowbits*: isnotset, NonULogedIn; *flowbits*: isset, NonULogA; *flowbits:* isset, NonULogA2; *flowbits*: unset, NonULogA; *flowbits*: unset, NonULogA2; **flowbits: set, F_6**; *flowbits*: noalert;
R_5	*flowbits*: isset, NonULoggeddIn; *flowbits*: unset,NonULogedIn; *flowbits*: unset, NonULogA; *flowbits*: unset, NonULogA2; **flowbits: set, F_6**; *flowbits*: noalert;
R_7	msg:"Normal User accesses important file"; **flowbits: isset, F_6;**

Table 6. Modified and added rules using the large rule sets approach

Example: Instead of giving an example of a large rule set, we use the rule set in Table 2 for simplicity because the approach works for small rule sets as well. Assume that all rules in Table 2 are evadable. The first step (lines 1 to 9 pf Alg.3) indicates that only rule R_3 and R_5 are vulnerable (note that Fig.4 is the D_e created for R_5). So R_3 and R_5 are modified by inserting the flowbits option *flowbits:set*, F_6. R_7 is the added rule and all other rules are the same. Table 6 shows the modified and added rules to patch the rule set.

7.3.3. False positives discussion

Even though the approach used for large rule sets also works for small rule sets, it potentially causes more false positives then the one used for small rule sets. While the latter only raises an alert if a complete evasion sequence is seen, the former raises an alert on important packets in an evasion sequence. However, the overhead caused by the latter is larger than the former. Therefore, depending on the size of a rule set and the acceptable overhead, an appropriate solution can be applied.

A patching solution without completely changing the rule set (which requires the understanding of the rule set semantics) always potentially causes false positives. In the extreme case, when all rules are vulnerable, the only solution is to always raise an alert when an evasion packet and a target packet are seen. This reflects the situation when Snort can not trust any packet and it has to raise an alert most of the time in the favor of not missing the attack.

However, our proposed solutions should not cause false positives in general because the patched portion covers possible events seen from the attacker and normally not occurred from normal users. For example, in an FTP session, a normal user will not access a file after he quits the session. Moreover, in practice, the extreme case does not seem to exist, as shown in the evaluation section.

7.3.4. False positives control patch

Although our solutions are only potential to cause false positives, there are situations (like the extreme case) where false positives are very likely to occur. In this section, we propose a false positives control patch in addition to the proposed solutions. False positives occur when alerts are raised on normal users' actions. Therefore, in order to avoid false positives, Snort needs to consider session packets it will see after it is put into a target state (by patched rules). These packets might give Snort some clues about who is running the session. If Snort sees a target packet right away, it is most likely that the attacker is running the session. Otherwise, if following packets cause transitions in D_s as a normal user will do, it is possible that the session is run by a normal user. The more session packets Snort considers afterward, the more accurate decision Snort will make.

Solutions for small and large rule sets put Snort into a target state when an evasion sequence is seen or a vulnerable rule is triggered respectively. Now, instead of raising an alert whenever a target packet is seen after Snort is put into a target state, if a number of packets corresponding to normal users' actions are seen (before a target packet), Snort will assume that the session is run by a normal user and put the session back to a non-target state.

In order to determine all possible actions a normal user might do after Snort is put into a target state, we need to know all the states in D_s after an evasion sequence is seen (for small rule sets solution) or after a vulnerable rule is triggered (for large rule sets solutions). Then all possible actions of a normal user are equivalent to all paths starting from any of these states.

Let L be the length of actions (or length of a path) Snort considers afterward.

Here is the procedure to add new rules (to the patched rule set from proposed solutions) to control potential false positives:

A. Find all states in D_s that are used to determine possible normal users' actions:

- Solution for small rule sets: each evasion sequence puts D_s into a state and this state can be found by tracing through D_s. States are collected for all evasion sequences.

- Solution for large rule sets: for each vulnerable rule, find all states in D_s where the rule can be triggered (to change to a different state), and then find all states right after the vulnerable rule is triggered. States are collected for all vulnerable rules.

B. For each state X_t found at stage A:

1) Find all paths from D_s of length L started at state X_t

2) Assign each path a different number.

3) For the simple path number *k*:

a) A label is created for each node. The first node is labeled as the last node of the simple path (from D_e) corresponding to the evasion sequence that puts D_s into X_t (for small rule sets solution) or *TARGETRULE_**M*** (for large rule sets solution). The rest are labeled “A-t-k2”, “A-t-k3”,…,“A-t-k*n*” respectively as the order of the nodes on the path.

b) For the first link on the path: assume the link’s attribute is P_J (or P_J^*). Create a rule $R^t(1)$ uses all options in R_J in the original rule set (header and body) except the *flowbits* options. $R^t(1)$’s *flowbits* options check if the first node’s label is set, and then set label “A-t-k2”. “*noalert*” option is used in this rule.

c) For the last link on the path: assume the link’s attribute is P_J (or P_J^*). Create a rule $R^t(n-1)$ that uses all options in R_J in the original rule set (header and body) except the *flowbits* options. $R^t(n-1)$’s *flowbits* options check if label “A-t-k(*n-1*)” is set, and then unset the first node’s label. “*noalert*” option is used in this rule.

d) For each link on the path (except the first and the last links): assume that the link connects a node with label “A-t-k*i*” to a node with label “A-t-k(*i+1*)”, and the link’s attribute is P_J (or P_J^*). A new rule $R^t(i)$ is created by using all options in R_J in the original rule set (header and body) except the *flowbits* options. $R^t(i)$’s *flowbits* options check if label “A-t-k*i*” is set, and then set label “A-t-k(*i+1*)”. “*noalert*” option is used in the rule.

L can be chosen by Snort users to balance between false positives and overheads (number of rules added). The longer L, the less false positives, but the more rules to be added.

There is always a trade off between vulnerability and false positives. In the extreme case where Snort needs to raise an alarm most of the time, the new rule set is totally secure. On the other hand, if the rule set is not patched and insecure, it will not cause any new false positive. This false positives control patch makes the rule set vulnerable again because a smart attacker can always send packets corresponding to all possible actions a normal user might do before sending the target packet. However, this patch at least gives Snort users an option to control potential false positives and is useful when missing some attacks is better than having too much false positives. Moreover, Snort users can apply the heuristic method to reduce number of rules added to control false positives. Instead of adding rules to cover all simple paths found in step 1 (stage B) above, only simple paths that reflect regular normal users’ actions should be considered (these actions can be found from the application session log files).

R^1(1)	msg:"FTP Normal User Login Attempt – Send username"; *flowbits*:isset,**A-15**; *flowbits*:set,**A-1-12;** *flowbits*:noalert;
R^1(2)	msg:"FTP Normal User Login Attempt - Send password"; *flowbits*:isset, **A-1-12**; *flowbits*:unset, **A-15**; *flowbits*:noalert;

Table 7. False-Positives-Control Rules added to the solution in Table 3 when L=2

Example: There are two rules (Table 7) added to the solution in Table 5 (solution for small rule sets) to control false positives where L=2.

8. PROOF OF CORRECTNESS

There are two things that we need to prove for each solution approach:

1) The new rule set is complete: it can still detect all packet sequences detected by the old rule set and also detect all possible evasion sequences.

2) The new rule set is sound: The new rule set itself is not vulnerable to the proposed attack.

8.1. Solution for small rule sets

In this section, we introduce a crucial concept: *independent rule set.* Two rule sets are independent if the triggering of any rule in one rule set does not depend on the existence of the other rule set. This concept is important to argue the soundness and completeness of the new rule set.

There are 3 ways in Snort where the triggering of one rule has effect on the triggering of another rule: rules using *flowbits, dynamic/activate* rules and *pass* rules. However, we can assume that rules using flowbits are all alert rules. Moreover, another assumption we can make is: *flowbits:reset* is not used in a rule set. This option *flowbits:reset* is used without specifying any label and equivalent to unsetting all labels already set for a flow. This option can be replaced by using *flowbits:unset* on all labels of a rule set.

With these assumptions, two rule sets which use the *flowbits* option on two non-overlapping label sets are independent

Assume S_1, $S_{2\ \ldots}$,S_n are added rule sets for n simple paths in ***SP.*** We have $S_{new} = S \cup S_1 \cup S_{2\ \ldots} \cup S_n$.

8.1.1. *Completeness of the new rule set*

In order to prove that the new rule set can detect all packet sequences detected by the old rule set, we need to prove that $L_s(S) \subset L_s(S_{new})$ and this is proved in Appendix A.1 Moreover, we need to prove that the new rule set also detect all possible evasion sequences. In other words, we need to prove that, given any evasion sequence X_{es} accepted by $D_e(S)$, the new rule set can detect X_{es}. i.e., $X_{es} \in L_s(S_{new})$. This is proved in Appendix A.2.

8.1.2. *Soundness of the new rule set*

In order to show that the new rule set S_{new} is safe against the proposed attack, we need to prove $L_e(S_{new}) = \varnothing$. In order to prove this, we first need to prove if $S = S_1 \cup S_2$ (S_1 and S_2 are independent), $L_s(S) = L_s(S_1) \cup L_s(S_2)$ and $L_a(S) = L_a(S_1) \cup L_a(S_2)$.

Then we prove that each added rule set corresponding to a simple path is not vulnerable to the proposed attack. Finally, by using induction, we can prove $L_e(S_{new}) = \varnothing$. The proof can be found at Appendix B.

8.2. Solution for large rule sets

Let S_{new} be the new constructed rule set. We'll prove that S_{new} is complete and sound.

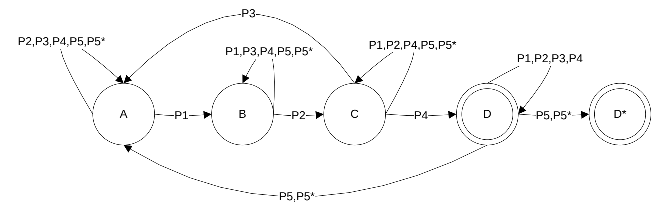

Figure 6. NFA$_s$ of the new rule set S_{new}

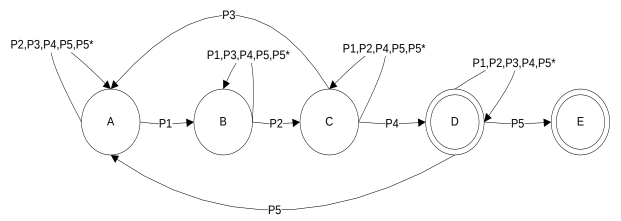

Figure 7. NFA$_a$ of the new rule set S_{new}

8.2.1. *Completeness of the new rule set*

The way the new rule set is constructed can be viewed from the DFAcreation. It is equivalent to adding extra transitions and states to the $D_s(S)$: for any state X, for any vulnerable rule R_i, if (X,P_i^*) != X then add a state X* and transitions $(X,P_i^*) = X^*$, $(X,P_i) = X^*$. Set X* as a target state. Now we have an NFA$_s$ of the new rule set. It is easy to see that any sequence accepted by $D_s(S)$ is accepted by this NFA$_s$, i.e, $L_s(S) \subset L_s(S_{new})$. Fig.6 shows an example of the NFA$_s$ of the new rule set S_{new} for the vulnerable rule set in Table 2 with the assumption that only rule 5 is evadable (and the rule is vulnerable).

To prove that all evasion sequences can be detected by the new rule set, we first need to prove a theorem that if two or more evadable rules together cause the rule set vulnerable, then at least one of them is vulnerable. This theorem is proved in Appendix C. Therefore, any evasion sequence needs to trigger at least one vulnerable rule, otherwise, there exists an evasion sequence that does not contain any evasion packet whose corresponding rule is vulnerable, and this contradicts the theorem. From the way the new rule set is constructed, after a vulnerable rule is triggered, Snort is always put in a target state, hence given any evasion X_{es}, we have $X_{es} \in L_s(S_{new})$. So, all evasion sequences can be detected by the new rule set.

8.2.2. *Soundness of the new rule set*

We can convert $NFA_s(S_{new})$ to $NFA_a(S_{new})$ by replacing transitions $(X,P_i^*) \rightarrow Y$ by $(X,P_i^*) \rightarrow X$ for all vulnerable rules R_i's. Fig.7 shows the corresponding NFA_a of the new rule set S_{new} in Table 4.

We see that any sequence accepted by $D_a(S)$ is accepted by $NFA_a(S_{new})$, i.e, $L_a(S) \subset L_a(S_{new})$.

Given any packet sequence $M \in L_a(S_{new})$.

- If M contains evasion packets corresponding to a vulnerable rule, $M \in L_s(S_{new})$ as the way S_{new} is constructed.

- If M does not contains evasion packets corresponding to any vulnerable rule, then $M \notin L_e(S)$. If $M \notin L_s(S_{new})$, then $M \notin L_s(S)$ because $L_s(S) \subset L_s(S_{new})$. So $M \notin L_a(S)$ because $M \notin L_e(S)$. Then $M \notin L_a(S_{new})$ because $L_a(S) \subset L_a(S_{new})$ which is a contradiction. Therefore $M \in L_s(S_{new})$.

Hence, there does not exists any packet sequence M so that $M \notin L_s(S_{new})$ and $M \in L_a(S)$. So the new rule set is not vulnerable.

9. IMPLEMENTATION & EVALUATION

We created a program called **SFET** (Snort *Flowbits* Evasion Tool) to parse a rule set, check if the rule set is vulnerable to the proposed attack, generate the corresponding D_e (or evasion sequences) and patch the rule set accordingly depending on its size and the number of evasion sequences.

SFET can be run in 3 modes: specified mode , automatic mode and cautious mode. In the specified mode, **SFET** allows users to specify which rule is evadable and which rule is a target rule. In the automatic mode, **SFET** itself decides the possibility of a rule to be evadable based on the rule's matching options (like content options and traffic direction the rule matches) and chooses rules with no *flowbits:noalert* option as target rules. Lastly, in the cautious mode, **SFET** assumes all rules in a rule set are evadable and chooses only one rule from all possible target rules (no *flowbits:noalert* option) as the target rule. A rule set is considered vulnerable if there exists an evasion sequence for any chosen target rule.

We collected all possible rule sets publicly available from the internet (most of them from BleedingEdge [2] and SourceFire [23]). There is about 60% of the rules using *flowbits* matches traffic coming from the client's side (presumably from the attacker's side), hence these rules are considered evadable. All together (considering different rule options as well), there is about 68% out of the rules using *flowbits* determined by **SFET** evadable and highly evadable. In addition, there are about 6% and 4% out of 400 rule sets (using *flowbits*) detected vulnerable to the proposed attack when **SFET** was run in the cautious mode and the automatic mode respectively.

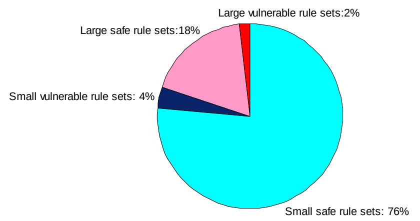

Figure 8. Vulnerable and safe rule sets percentage when SFET is run in the cautious mode

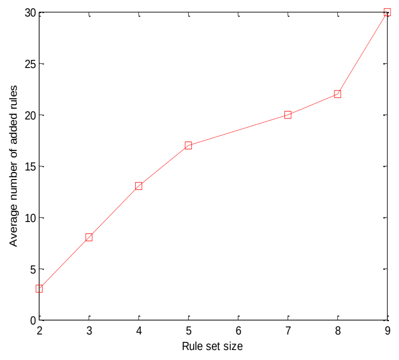

Figure 9. Overheads to patch small rule sets

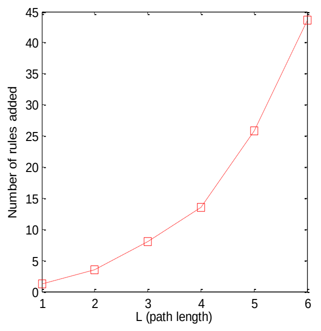

Figure 10. Overheads from false positives control patch

When running **SFET** in the specified mode with some chosen rule sets (we know exactly which rule is evadable), all evasion sequences generated by **SFET** can be converted to a real attack (this is not true for other modes).

Even though large rule sets (the number of rules is greater than 9) make up only 20% out of the considered rule sets, they are more susceptible to the attack than small rule sets. While 10% of large rule sets are vulnerable to the attack, only 5% of small rule sets are vulnerable. This is shown in Fig.8.

When applying the proposed solution to small vulnerable rule sets, the number of added rules to patch a rule set in average is triple the number of rules in the rule set (for both automatic and cautious modes). Fig.9 shows the average number of added rules for each rule set size (note: we do not find any vulnerable rule set of size 6).

For large vulnerable rule sets, the number of modified rules is the same as the number of vulnerable rules. Even though some large rule sets have many evadable rules, in average, only 10% of evadable rules are vulnerable. In addition, the number of added rules for each large rule set is as most the number of target rules in the rule set. The average number of added rules is only 3.5 for both automatic and cautious modes.

We chose some patched rule sets (including small and large rule sets) to apply the false positive control patch for different values of L. In average, the number of rules added to control false positives increases exponentially as L increases (as expected) and this is shown in Fig.10.

10. CONCLUSION AND FUTURE WORK

Besides the increase of sophisticated attacks, in many cases, considering a single rule and a single packet at a time is not sufficient to detect an attack, therefore stateful rules are necessary and important to any NIDS. The *flowbits* option in Snort is getting used more frequently to fulfill the need of detecting many different attacks. However, writing rules to completely simulate a session protocol is impossible, therefore, under some circumstances; it is possible to fool Snort into misjudging the actual session and hence evade Snort detection.

In this paper, we proposed an evasion technique to Snort rule sets using flowbits based on several factors: the packet-based nature of Snort, loose rules, complicated sessions and insecure detection approaches. Moreover, we suggested practical solutions to prevent the evasive attack for small and large vulnerable rule sets, and formally proved that the solutions are complete and sound. In addition to these solutions, we proposed a false positives control method which allows Snort users to reduce potential false positives caused by the patches.

We implemented a tool called **SFET** which can automatically calculate the possibility that a rule is evadable based on the rule's content and generate all possible evasion sequences to a given rule set. Besides, **SFET** is also possible to generate a new rule set which not only does the job of the old one, but also thwarts the proposed attack.

The evaluation showed that a significant number of available rule sets are vulnerable to the proposed attack, which seriously affects the security of Snort users' systems. Our solutions are shown very practical and do not cause much overhead in both cases: small and large vulnerable rule sets.

Despite we already suggested a method to control false positives, it is still necessary to measure correctly how much false positives caused by the patched rule sets. However, this is left for the future work because it requires Snort to run a long period of time. Besides, even though finding an ideal solution is intractable and sometime impossible, it is worth to look for one if it exists and we plan to do this in the future work. Finally, although our proposed attack applies to Snort, the idea can be applied to any NIDS with stateful signatures. So we consider applying the attack to different NIDS in the future work as well.

REFERENCES

[1] S.Aubert.Idswakeup. http://www.hsc.fr/ressources/outils/ids wakeup/, 2000

[2] BleedingEdge Inc. [Online]. Available: http://www.bleedingthreats.net/

[3] Brian Caswell and Jay Beale and Andrew R.Baker, Snort Intrusion Detection and Prevention Toolkit, 2007, Syngress Publishing

[4] Cisco Systems, Inc., Installing and Using Cisco Intrusion Prevention System Device Manager 6.0. OL-8824-01, 2006-2008.

[5] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In Proceedings of the 12th USENIX Security Symposium, August 2003.

[6] C. Giovanni. Fun with packets: Designing a stick, March 2001. Endeavor Systems, INC.

[7] Mark Handley, Christian Kreibich and Vern Paxson, "Network Intrusion Detection: Evasion, Traffic Normalization," Proc. 10th USENIX Security Symposium,2001.

[8] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In Usenix Security, Aug. 2001.

[9] http://www.informit.com/articles/article.aspx?p=21778

[10] http://en.wikipedia.org/wiki/Snort_%28software%29

[11] C. Kruegel, D. Mutz, W. Robertson, G. Vigna, and R. Kem-merer. Reverse Engineering of Network Signatures. In Pro-ceedings of the AusCERT Asia Pacific Information Technol-ogy Security Conference, Gold Coast, Australia, May 2005

[12] D. Mutz, G. Vigna, and R. Kemmerer. An experience developing an IDS stimulator for the black-box testing of network intrusion detection systems. In Annual Computer Security Applications Conference, Las Vegas, NV, December 2003.

[13] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time", In Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, January 1998.

[14] T.H. Ptacek, T.N. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Secure Networks, Jan. 1998.

[15] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In Proceedings of the 13th Large Systems Administration Conference (LISA '99), November 1999

[16] S. Rubin, S. Jha, and B. Miller. Language-Based Generation and Evaluation of NIDS Signatures. IEEE Symposium on Security and Privacy, Oakland, California, May, 2005.

[17] Shai Rubin, Somesh Jha, Barton P. Miller, "Automatic Generation and Analysis of NIDS Attacks," acsac, pp. 28-38, 20th Annual Computer Security Applications Conference (ACSAC'04), 2004

[18] U. Shankar and V. Paxson. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. In Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, CA, May 2003.

[19] R. Smith, C. Estan , S. Jha , "Backtracking Algorithmic Complexity Attacks against a NIDS," Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual , vol., no., pp.89-98, Dec. 2006

[20] Sniphs. Snot, January 2003. http://www.l0t3k.org/tools/IDS/ snot-0.92a.tar.gz

[21] Snort 2.8.1 http://www.snort.org/dl/current/snort-2.8.1.tar.gz

[22] Sourcefire, Inc., Snort Users Manual. The Snort Project, 2003-2008.

[23] SourceFire Inc. [Online]. Available: http://www.sourcefire.com/

[24] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS), October 2004.

[25] G. Vigna, W. Robertson, , Vishal Kher; R.A Kemmerer, "A stateful intrusion detection system for World-Wide Web servers," Computer Security

[26] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In ACM Conference on Computer and Communications Security, Washington, DC, November 2002.

[27] V. Yegneswaran, J. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In USENIX Security, 2005. Applications Conference, 2003. Proceedings. 19th Annual , vol., no., pp. 34-43, 8-12 Dec. 2003

[28] Judy Novak, Steve Sturges. Target-Based TCP Stream Reassembly. http://www.snort.org/docs/stream5-model-Aug032007.pdf

[29] Nmap. http://nmap.org

APPENDIX

Note: all lemmas and proofs in appendix **A** and **B** deal with rule sets which use non-overlapping label sets in their *flowbits*. Assumptions in Section 7 are still made here: which means all rule set are pairwise independent.

A. The new rule set is complete (proof)

Lemma 1: If $S = S_1 \cup S_2$, $L_s(S) = L_s(S_1) \cup L_s(S_2)$.

Proof:

Because $S = S_1 \cup S_2$, so target rules of S = target rules of $S_1 \cup$ target rules of S1. Because S_1 and S_2 are independent, then if a packet sequence puts S_1 in a target state, it still puts S_1 in a target state when S_2 is added to S_1. The same argument applies for S_2. A target state of S is a state where a target rule of S can be triggered, which is either S_1's target rule or S_2's target rule. It means that a target state of S is either S_1's target state or S_2's target state. Therefore, a packet sequence that puts S in a target state if and only if it puts S_1 or S_2 in a target state. So we have $L_s(S) = L_s(S_1) \cup L_s(S_2)$.

Lemma 2: If $S = S_1 \cup S_2$, $L_a(S) = L_a(S_1) \cup L_a(S_2)$.

Proof:

We'll prove that $D_a(S) = D_a(S_1) \cup D_a(S_2)$

- By Lemma 1, we have $L_s(S) = L_s(S_1) \cup L_s(S_2)$. So $D_s(S) = D_s(S_1) \cup D_s(S_2)$.

- Let $D^* = D_a(S_1) \cup D_a(S_2)$.

- Let $F_{s\text{-}S}$, $F_{s\text{-}S1}$, $F_{s\text{-}S2}$, $F_{a\text{-}S}$, $F_{a\text{-}S1}$, $F_{a\text{-}S2}$, and F^* be transition functions for $D_s(S)$, $D_s(S_1)$, $D_s(S_2)$, $D_a(S)$, $D_a(S_1)$, $D_a(S_2)$, and D^* respectively.

- Both D* and $D_a(S)$ have the same alphabet which is $\cup\{P_i, P_i^*(\text{if } R_i \text{ is evadable}) : R_i \text{ in S and } R_i \text{ is not a target rule}\}$

- Given any state A in $D_s(S_1)$, any state B in $D_s(S_2)$, any R_i in S:

  + By the way union of two Ds is constructed, the transition function of $D_s(S)$ is as follow:

    $F_{s\text{-}S}([A,B],x) = [F_{s\text{-}S1}(A,x) , F_{s\text{-}S2}(B,x)]$

  + By the way union of two Ds is constructed, the transition function of D* is as follow:

    $F^*([A,B],x) = [F_{a\text{-}S1}(A,x) , F_{a\text{-}S2}(B,x)]$

  + Based on the construction of D_s and D_a of a rule set, we have:

  $F_{a\text{-}S}([A,B],x) = F_{s\text{-}S}([A,B],x)$ if x is P_i

  $F_{a\text{-}S}([A,B],x) = [A,B]$ if x is P_i^* and R_i is evadable

  $F_{a\text{-}S1}(A,x) = F_{s\text{-}S1}(A,x)$ if x is P_i

  $F_{a\text{-}S1}(A,x) = A$ if x is P_i^* and R_i is evadable

  $F_{a\text{-}S2}(B,x) = F_{s\text{-}S2}(B,x)$ if x is P_i

  $F_{a\text{-}S2}(B,x) = B$ if x is P_i^* and R_i is evadable

- We have:

  + if x is P_i:

  $F_{a\text{-}S}([A,B],x) = F_{s\text{-}S}([A,B],x) = [F_{s\text{-}S1}(A,x) , F_{s\text{-}S2}(B,x)] = [F_{a\text{-}S1}(A,x) , F_{a\text{-}S2}(B,x)] = F^*([A,B],x)$

  + if x is P_i^* and R_i is evadable:

  $F_{a\text{-}S}([A,B],x) = [A,B] = [F_{a\text{-}S1}(A,x) , F_{a\text{-}S2}(B,x)] = F^*([A,B],x)$

  $\rightarrow F^* = F_{a\text{-}S}$

- Let $A_{s\text{-}S}$, $A_{s\text{-}S1}$, $A_{s\text{-}S2}$, $A_{a\text{-}S}$, $A_{a\text{-}S1}$, $A_{a\text{-}S2}$, and A^* be sets of accept states of $D_s(S)$, $D_s(S_1)$, $D_s(S_2)$, $D_a(S)$, $D_a(S_1)$, $D_a(S_2)$, and D^* respectively. We have:

  $A_{a\text{-}S} = A_{s\text{-}S}$ (the construction of D_s and D_a) and $A_{s\text{-}S} = \{[e_1, e_2]$ in which either e_1 is in $A_{s\text{-}S1}$ or e_2 is in $A_{s\text{-}S2}\}$ (the construction of union of two Ds)

  $A^* = \{[e_1, e_2]$ in which either e_1 is in $A_{a\text{-}S1}$ or e_2 is in $A_{a\text{-}S2}\} = \{[e_1, e_2]$ in which either e_1 is in $A_{s\text{-}S1}$ or e_2 is in $A_{s\text{-}S2}\}$

  $\rightarrow A_{a\text{-}S} = A^*$

- With the same argument, we also have that $D_a(S)$ and D* have the same start state.

=> We have shown that $D_a(S)$ and D* have the same alphabet, state transition functions, start state and accept states. So $D_a(S) = D^* = D_a(S_1) \cup D_a(S_2)$, in other words, $L_a(S) = L_a(S_1) \cup L_a(S_2)$.

A.1 The new rule set can detect all packet sequences detected by the old rule set

Proof:

We have $S_{new} = S \cup S_1 \cup S_{2\,\dots} \cup S_n$ where $S, S_1, S_{2\,\dots}, S_n$ use non-overlapping label sets. So by Lemma 1,

$L_s(S_{new}) = L_s(S \cup S_1 \cup S_{2\,\dots} \cup S_n) = L_s(S) \cup L_s(S_1 \cup S_{2\,\dots} \cup S_n)$ (because S and $S_1 \cup S_{2\,\dots} \cup S_n$ use non-overlapping label sets and are independent)

$\rightarrow$ All packet sequences detected by S are detected by S_{new}.

A.2 The new rule set can detect all possible evasion sequences

Proof:

- We need to prove that, given any path X from the start state to an accept state in the $D_e(S)$, the new rule set can detect the packet sequence corresponding to X, say X_{es}.

- Let Z be the simple path after all circles are removed from X. Let Y be the simple path in ***SP*** such that Y's corresponding packet sequence is a prefix of Z's corresponding packet sequence. Let S_Y be the rule set added for Y and Y_{es} be Y's corresponding packet sequence. We will prove that X_{es} is detected by S_Y or accepted by $D_S(S_Y)$. Assume S_Y uses label set $\{A_1, A_2,\dots,A_n\}$ and $Y_{es} = Y_1\ Y_2\ \dots\ Y_{n-1}$. So $X_{es} = Y_1\ X_1\ Y_2\ X_2\ \dots\ Y_{n-2}\ X_{n-2}\ Y_{n-1}X_{n-1}$ where X_i is any packet sequence.

- Let T_i be the state where $A_1,\dots,A_i$ are set and $A_{i+1},\dots,A_n$ are not set. Let F_s be the transition function of $D_s(S_Y)$.

- By the construction of S_Y, we have

$F_s\,(T_i, Y_J) = T_{i+1}$ if i=j and T_i if i!=j (i<n)

$F_s\,(T_i, x) = T_i$ if x != Y_J for some j or i=n

$\rightarrow F_s\,(T_i, X_J) = T_k$ where k>=i

- Now let's consider how X_{es} is processed by $D_s(S_Y)$:

We'll prove that the state right before $D_s(S_Y)$ sees Y_i is T_c where c>=i. This can be proven by using induction.

It is true for i=1. Assume it is true for 1<=i<=k.

Let T_e and T_f be the states right before and after $D_s(S_Y)$ sees Y_k repsectively. We know e>=k because of the inductive hypothesis.

If e=k, $T_f = F_s\,(T_k, Y_k) = T_{k+1} \rightarrow$ f=k+1>k.

If e>k, $T_f = F_s\,(T_e, Y_k) = T_e \rightarrow$ f=e >k.

So f>=k+1

We have $T_w = F_s\,(T_f, X_k)$ (w>=f) is the state right before $D_s(S_Y)$ sees Y_{k+1}. We have w>=f>=k+1.

So we have proven that the state right before $D_s(S_Y)$ sees Y_i is T_c where c>=i. It means that the state right before $D_s(S_Y)$ sees Y_{n-1} is T_{n-1} or T_n. So $F_s(T_1, X_{es}) = F_s(T_{n-1}, Y_{n-1}X_{n-1}){=}T_n$ or $F_s(T_n, Y_{n-1}X_{n-1}){=}\ T_n$. $\rightarrow F_s(T_1, X_{es}) = T_{n.}$

- So X_{es} is accepted by $D_S(S_Y)$ and hence detected by S_Y.

- Since S_Y is independent with the rest in the new rule set (because S_Y uses a different label set), a packet sequence detected by S_Y is detected by the new rule set (by Lemma 1)

B. The new rule set is sound (proof)

Lemma 3: Each added rule set corresponding to a simple path is not vulnerable to the proposed attack.

Proof:

Assume the rule set S is added for the simple path K. We need to show that $L_e(S) = \varnothing$.

Assume $S=\{R_1,R_2,\dots,R_n\}$ and uses label set $\{A_1, A_2,\dots,A_n\}$.Let T_i be the state where $A_1,\dots,A_i$ are set and $A_{i+1},\dots,A_n$ are not set.

Let F_s , F_a, and F_e be transition functions for $D_s(S)$, $D_a(S)$, and $D_e(S)$ respectively.

- Based on the construction of S from K, we have

$F_s (T_i, P_J) = T_{i+1}$ if i=j and T_i if i!=j (i<n)

$F_s (T_i, P_J*) = T_{i+1}$ if i=j and T_i if i!=j (If R_J is evadable) (i<n)

$F_s (T_n, x) = T_n$

$F_a (T_i, P_J) = T_{i+1}$ if i=j and T_i if i!=j (i<n)

$F_a (T_i, P_J*) = T_i$ (If R_J is evadable) (i<n)

$F_a (T_n, x) = T_n$

- Based on the construction of $D_e(S)$, we have:

  + If i=j:

    $F_e([T_i,T_J], x) = F_e([T_i,T_i], x) = [F_s(T_i, x), F_a(T_i, x)] = [T_i, T_i]$ or $[T_{i+1}, T_{i+1}]$ or $[T_{i+1},T_i]$

  + If j<i:

    $F_e([T_i,T_J], x) = [F_s(T_i, x), F_a(T_J, x)] = [T_i, T_J]$ or $[T_i, T_{J+1}]$ or $[T_{i+1},T_J]$

Initially, the start state is $[T_1, T_1]$, so based on the transition functions above, it is clear that $[T_i , T_J]$ (j>i) is not reachable in the $D_e(S)$

However, accept states of $D_e(S)$ are $[T_i,T_n]$ where 1<= i < n , so $L_e(S) = \emptyset$.

Therefore, each added rule set corresponding to a simple path is not vulnerable to the proposed attack.

The new rule set itself is not vulnerable to the proposed attack

Proof :

Assume $S_1, S_2 \ldots ,S_n$ are added rule sets for n simple paths in ***SP***

We have $S_{new} = S \cup S_1 \cup S_2 \ldots \cup S_n$ where $S, S_1, S_2 \ldots ,S_n$ use non-overlapping label sets.

We need to prove that the new rule set S_{new} is safe against the proposed attack. In other words, we need to prove that $L_e(S_{new}) = \emptyset$.

From what we have proved in Appendix A.2, we have: $L_e(S) = L_s(S_1) \cup L_s(S_2) \cup \ldots \cup L_s(S_n)$ (*)

Now, we have $L_e(S_{new}) = L_e(S \cup S_1 \cup S_2 \ldots \cup S_n) = \neg L_s (S \cup S_1 \cup S_2 \ldots \cup S_n) \cap L_a (S \cup S_1 \cup S_2 \ldots \cup S_n)$

$= (\neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n)) \cap (L_a (S) \cup L_a (S_1) \cup L_a (S_2) \ldots \cup L_a (S_n))$ (by Lemma 1 and Lemma 2)

$= (\neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n) \cap L_a (S)) \cup (\neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n) \cap L_a (S_1)) \cup \ldots . \cup (\neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n) \cap L_a (S_n))$

$= (\neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n) \cap L_a (S)) \cup (\neg L_s(S) \cap \ldots \cap \neg L_s(S_n) \cap L_e (S_1)) \cup \ldots . \cup (\neg L_s(S) \cap \ldots \cap \neg L_s(S_{n-1}) \cap L_e (S_n))$

$= (\neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n) \cap L_a (S)) \cup \emptyset \ldots \cup \emptyset$ (by lemma 3)

$= \neg L_s(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n) \cap L_a (S)$

$= (\neg L_s(S) \cap L_a (S)) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n)$

$= L_e(S) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n)$

$= (L_s(S_1) \cup L_s(S_2) \cup \ldots \cup L_s(S_n)) \cap \neg L_s(S_1) \cap \ldots \cap \neg L_s(S_n)$ (by (*))

$= \emptyset$ (use induction on n and use $L_s(S_i) \cap \neg L_s(S_i)= \emptyset$)

So we have $L_e(S_{new}) = \emptyset$, this means S_{new} is not vulnerable to the proposed attack

C. Theorem

Theorem: If a set Q of evadable rules makes the rule set vulnerable, then at least one of these evadable rules is vulnerable.

Proof:

We'll prove this theorem by contradiction.

Assume there is no rule in the set is vulnerable. Let K be an evasion sequence that exploits these evadable rules in Q. K contains evasion packets correponding to evadable rules in Q and normal packets.

So $K \notin L_s(S)$ and $K \in L_a(S)$.

Let K^a be the packet sequence from K without evasion packets (removing evasion packets from K), so $K^a \in L_a(S)$.

Let K^f be the packet sequence from K with P_i*'s replaced by P_i's for all $R_i \in Q$. We have $K^f \notin L_s(S)$ because from Snort's perspective, K and K^f are the same, and we also have $K \notin L_s(S)$

Now we consider this procedure:

Let $K^s = K^a$

For each $R_i \in Q$:

1- Set $K^a = K^s$

2- Add P_i*'s to K^a at the positions where they are removed from K

3- Set K^s = packet sequence from K^a with P_i*'s replaced by P_i's.

Before the first iteration: $K^a \in L_a(S)$

After the first iteration: We have $K^a \in L_a(S)$ because we just add evasion packets to K^a. We also have K^a contains evasion packets corrensponding to only R_i for some i. So K^a can not be an evasion sequence, otherwise R_i is vulnerable. Together we must have $K^a \in L_s(S)$. But from Snort's perspective, K^a and K^s are the same, so $K^s \in L_s(S)$. Since K^s does not contain any evasion packet, then if $K^s \in L_s(S) \rightarrow K^s \in L_a(S)$.

With the same arguments, we always have $K^s \in L_s(S)$ after each iteration. But when the loop is done, we have $K^s = K^f$. This means $K^f \in L_s(S)$: a contradiction.

So the assumption is wrong. Therefore, there is at least one rule in the set is vulnerable.

D. Small rule set solution complexity proof

In D_e , a simple path from the start node to a target node can be of length M^2. There is one choice for the start node in the path, |Σ| choices for the second node, |Σ|^2 for the third, … and so on until node M^2-|T|, entailing an upper bound complexity of ~ |Σ|^(M^2-|T|-1) for each target state in |T|.

For each simple path in SP of size |p|, a set of |p|-1 flowbits and rules is created. |p| has an upper bound length of M^2-|T|.

In summary, an upper bound on the complexity of Alg.2 is |T| x (M^2-|T|-1) x |Σ|^(M^2-|T|-1) new rules!

E. Large rule set solution complexity proof

Let M be the number of states of the DFA

Let |Σ| be the size of the signature alphabet

Let |R| be the size of the flowbit rule set.

We always have |Σ| <= |R|.

Number of edges in the DFA= M x |Σ|.

The evasion DFAconstruction has a worst case cost of M^2 x |Σ|.

For each constructed evasion D, there is a need to run an algorithm to find if a final state is reachable from the start state. This can be avoided by doing this test while constructing the D, so the this operation can be assumed to have no cost.

Lines 12-22 add rule R_m and change all rules in Vm. This has an upper bound of |R| operations.

In total, the worst case complexity is <= |R| x (M^2 x |Σ| + 1), which is polynomial.