An Evasive Attack on SNORT Flowbits

Tung Tran
University of Waterloo
Waterloo, Canada
t3tran@uwaterloo.ca

Issam Aib
University of Waterloo
Waterloo, Canada
issam.aib@gmail.com

Ehab Al-Shaer
Univ. North Carolina
Charlotte, NC
ealshaer@uncc.edu

Raouf Boutaba
University of Waterloo
Waterloo, Canada
rboutaba@uwaterloo.ca

***Abstract*—The support of stateful signatures is an important feature of signature-based Network Intrusion Detection Systems (NIDSs) which permits the detection of multi-stage attacks. However, due to the difficulty to completely simulate every application protocol, several NIDS evasion techniques exploit this Achilles' heel, making the NIDS and its protected system see and explain a packet sequence differently.**

In this paper, we propose an evasion technique to the Snort NIDS which exploits its *flowbits* feature. We specify the flowbit evasion attack and provide practical algorithms to solve it with controllable false positives and formally prove their correctness and completeness. We implemented a tool called SFET which can automatically parse a Snort rule set, generate all possible sequences that can evade it, as well as produce a patch to guard the rule set against those evasions. Although Snort was used for illustration, both the evasion attack and the solution to it are applicable to any stateful signature-based NIDS.

I. Introduction

Snort [12] is a popular open source and lightweight Network Intrusion Detection and Prevention System (NIDPS) [2]. It is mostly signature-based and famous for its intrusion detection capabilities that match packet content against a set of rules. Snort supports a flexible and rich rule language which allow users to inspect all fields of a packet. The *flowbits* option was first introduced in Snort 2.1.1 and allows the detection engine to track state across a single TCP session. The support of stateful signatures allows a signature-based IDS to detect multi-stage attacks. Because of its importance, this feature is sometimes separately implemented for specific services [21]. A *flowbit* is essentially a flag that can be set by some rule and then used by another one. The *flowbits* option works by using labels to set and change the session state. Formally, it has the format:

flowbits: [[un]set|toggle|[is|isnot|re]set,noalert][,<LABEL>];

Table I lists two flowbit rules taken from BleedingEdge [1], which tracks an FTP session. We will illustrate the essence of our evasion technique using this simple rule set. The first rule checks if a user tries to log into the FTP server, in which case it sets the flowbit `login` and no alert is raised (`noalert` flowbit). The second rule raises an alert (absence of a `noalert` flowbit) if the user uses the `LIST` command. However, this occurs only if the user has not logged in yet, *i.e.* the label `login` is not set. This rule set can detect someone using the `LIST` command without prior login into the server. However, a unauthorized attacker can always try to log in then use the `LIST` command even though the login was not granted. The rule set makes Snort treat any login attempt as successful and hence allows the `LIST` command. This is an example where a NIDS misjudges the application protocol session and the evasion succeeds (no alert raised).

TABLE I
Example of a Snort flowbits rule set

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;)

In this paper, *actual session* refers to the application protocol session being protected, and *actual session state* refers to the state of that application protocol session. *in-Snort session* pictures an internal representative session (of the application protocol) maintained by Snort, and *in-Snort session state* represents the collective value of each flowbit of the rule set (ref. Def.1 below). The *in-Snort session state* is supposed to reflect the *actual session state*. However, the large number and complexity of existing protocols and continuous appearance of new ones makes it generally impossible to completely simulate actual session states in Snort because of obvious performance and storage reasons. The evasion attack we identify in this paper exploits this practical inevitability. Although it is illustrated for Snort flowbit rule sets, our evasion can be applied to any NIDS supporting stateful signatures.

II. Related Work

Several attacks on IDSes have been identified in the literature. Ptacek and Newsham [11] 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. [6] describes a different solution to stateful IDS evasion that relies on an extended version of the IDS state transition diagram. Handley and Paxson [5] [10] discussed evasion techniques based on inherent ambiguities of the TCP/IP protocol which leads to a difference between a NIDS and its protected system in performing TCP segments and IP fragments reassembly. Traffic normalization suggested by Handley et al. [5] tries to remove these ambiguities by

978-1-4673-0269-2/12/$31.00 © 2012 IEEE

patching the packet stream. Another solution to this is Active Mapping, which was proposed by Shankar and Paxson [16] and eliminates TCP/IP-based ambiguity in a NIDS analysis with minimal runtime cost and is implemented in the Stream5 [9] preprocessor of Snort.

Besides NIDS evasion techniques, there are attacks on NIDSs as well. Wagner and Soto [23] revealed mimicry attacks on a Host-based IDS. Snot [18], Stick [4], IDSWakeup [15] and Mucus [8] are over-stimulation tools that cause a DOS attack on 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 [3] [17], especially the authors in [17] 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. [22] 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 NIDS. 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 [8] 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. [14] 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 can be used by our attack, however, these techniques only apply to injected packets which are not part of the actual session. Although our paper only deals with Snort rules, which are mainly manually written by users, automatically generated semantic-aware signatures [24] or session signatures [13] are also potentially vulnerable to our identified 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 actual session is following one of them.

III. Snort Flowbit Evasion

Let S = $\{R_1,R_2,\ldots,R_n\}$ be a Snort flowbit rule set. We define a rule that raises an alert when triggered (*i.e.* has no `noalert` flowbit) as a *Target rule*; and *Target rule set* the set of all target rules in a rule set. The evasion problem consists of finding all possible packet sequences that successfully attack the service protected by S yet manage not to trigger any target rule of S. It can also be defined on a *target rule group*, which is a subset of the target rule set containing rules having the same match options except the flowbits conditions.

Definition 1 (Session state): It represents the group of flowbits (labels) that are currently set (during a runtime session). If n is the number of flowbits used in the rule set, then there are potentially 2^n different in-Snort session states. It is a Target state if the flowbit set it represents correspond to the condition of a target rule, otherwise it is called a non-target state.

A *Target packet* is a packet that matches any target rule and presumably the last packet in the packet sequence of a real attack. A flowbit rule is *evadable* if it can be triggered by the attacker to change an in-Snort session state while preserving the actual session state.

An evadable rule can be triggered by two different kinds of packets: a packet (from the connection session) that triggers the rule 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 packets*, and P_i^* represent the second kind, called *evasion packets*. P_i^* packets cause a change in the in-Snort session state (by triggering R_i), but have no effect on the actual session state.

A packet sequence is considered a successful evasion attack 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 a target state and the in-Snort session in a non-target state.

IV. Language of all Flowbits Evasions

A Deterministic Finite State Automaton (DFA) representation of a flowbits rule set can be derived using a mapping from session states. Alg.1 constructs the in-Snort (D_s) and actual (D_a) session state DFA of a rule set S. As can be noticed, the two automata have the same alphabet, set of states, start state and the set of accept states. The only difference is in the transition function where D_s changes state for evasion packets (line 14) while D_a does not.

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 that successfully attack

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* $\mathbf{\Sigma} = \{P_i$: $R_i$ 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:         state B $\leftarrow$ A
9:         if $R_i$ can be triggered at A then
10:            B $\leftarrow$ output state of $R_i$ when triggered from $A$
11:        end if
12:        Add $(A, P_i) \rightarrow B$ to both $D_s$ and $D_a$
13:        if $R_i$ is evadable then
14:            Add $(A, P_i^*) \rightarrow B$ to $D_s$
15:            Add $(A, P_i^*) \rightarrow A$ to $D_a$
16:        end if
17:    end for
18: end for
```

the service protected by a rule set S using flowbits evasion equals:

$$L_e(S) = L_a(S) \cap \neg L_s(S) = L(D_e(S)) \quad (1)$$
$$\text{where } D_e(S) = D_a(S) \cap \neg D_s(S)$$

V. Triggering A Snort Rule Without Affecting The Actual Session State

A. Packet-based property of Snort

Snort is a packet-based NIDS and most packets it receives are checked by the detection engine. This feature allows the creation of evasion packets. One method is to construct a packet that matches a given rule, however, with an "out of order" TCP sequence number. This packet is not processed by the receiver' application layer but is still examined by the Snort detection engine (then triggers the rule). Snort, with the stream5 preprocessor enabled, knows that the packet does not have an expected sequence number for the session stream and is overlapping with a previous packet (assuming that this previous packet has some payload). In this case, Snort does exactly as the protected host: not reassemble this packet into the session stream. However, the packet is still passed down to the detection engine because the packet may match some TCP-based attacks where the sequence number is not important (e.g: Nmap [7] uses TCP packets with random sequence number to probe a host's OS). Hence, it is always possible to construct a packet (with an "out-of-order" TCP sequence number) to fake a request from the client and inject it into the connection session. Therefore, any Snort flowbit rule matching traffic from the client side can be triggered without causing the actual session state change. This was tested with Snort 2.9.1 [20], the newest version at this time.

B. Loose rules

We say that a flowbit rule is *loose* if it does not match packets using tight options like *dsize*, *depth*, or *offset*. 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 possible to create or trigger the sending of such a packet. The packet can even be created from the connection session itself.

Depending on the service protocol, an attacker may be able to make a request from the client-side that matches a loose rule while logically doing something else rather than what the rule expects. For example, consider a loose rule that checks if a user that is currently in an FTP session is trying to quit the session by examining client-side packets containing `QUIT\n` in their 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 to misjudge the session.

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 by the attacker. 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. Consider a rule which checks for any packet from the server in a telnet session containing the string "Granted" in the payload. The attacker can issue an invalid command containing "Granted". The server will send back an unknown command error message, which happens to contain the string "Granted", and hence triggers the rule. Another trick is to create a folder named "Granted" and then try to list all the folders.

VI. Example of a Flowbits Evasion

The rule set in Table II follows an FTP session and raises an alert if a non-admin user tries to do anything related to an important file whose access is restricted to administrators only. For simplicity, we show only the *flowbits* and the *msg* options in a rule. The *msg* option denotes the purpose of a rule.

Rule R_1 and R_2 determine if a non-admin user is logging in. Rule R_3 indicates that the user is denied to login. R_4 checks if the user has successfully logged in. R_5 indicates that the user has logged out of the FTP session, and R_6 checks if the logged-in user tries to do anything with a restricted file and raises an alert. Only R_6 is a target rule. Theoretically, any rule is evadable, but in this example, for simplicity, we assume that only R_5 is. This means that the attacker can come up with a way to trigger R_5 without affecting the actual session state. He can accomplish this in many different ways which are discussed in Section V.

Three labels are used in this rule set, leading to $2^3 = 8$ possible states. However, only four of these are reachable, including the start state (no label set): A ={}, B={nalu}, C={nalu, nalp} and D={nalu, nalp, nauld}. The in-Snort DFA D_s and actual session state DFA D_a are depicted in Fig.1 and Fig.2. The intersection of D_s and D_a, which is D_e is constructed in Fig.3, where (A,A) is the start state and (D,A),

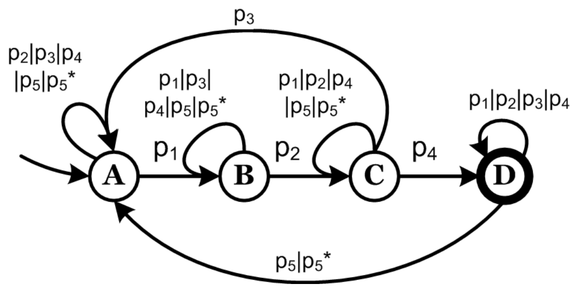

Fig. 1. D_s of the FTP rule set

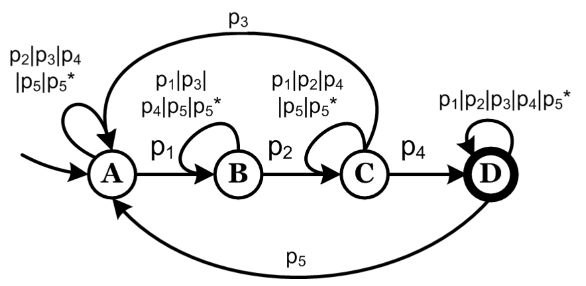

Fig. 2. D_a of the FTP rule set

TABLE II
A FLOWBITS RULE SET TO DETECT A NON-ADMIN USER ACCESSING A RESTRICTED FILE FROM AN FTP SESSION

R_1	msg: "FTP Non-admin User Login Attempt - Send username"; *flowbits*: set, nalu; *flowbits*: noalert;
R_2	msg: "FTP Non-admin User Login Attempt - Send password"; *flowbits*: isset, nalu; *flowbits*: set, nalp; *flowbits*: noalert;
R_3	msg: "FTP login denied"; *flowbits*: isnotset, nauld; *flowbits*: isset, nalu; *flowbits*: isset, nalp; *flowbits*: unset, nalu; *flowbits*: unset, nalp; *flowbits*: noalert;
R_4	msg: "FTP login granted"; *flowbits*: isset, nalp; *flowbits*: set, nauld; *flowbits*: noalert;
R_5	msg: "FTP user exits"; content: "QUIT|0D0A|"; nocase; *flowbits*: isset, nauld; *flowbits*: unset, nauld; *flowbits*: unset, nalu; *flowbits*: unset, nalp; *flowbits*: noalert;
R_6	msg: "Non-admin User accesses restricted file"; *flowbits*: isset, nauld;

(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, $p_1p_2p_4p_5^*$ is a successful evasive packet sequence accepted by this D_e. The attacker can apply this packet sequence to perform a real attack as follows: First, the attacker logs in as a normal user (non admin) with a 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 two options to create p_5^*. The first is to manually construct and inject into the connection a packet that matches R_5 but has an out-of-order sequence number. The second is to send a packet that matches R_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 download or access the restricted file at the server. This last action does not triggers the target rule and the evasion succeeds.

Algorithm 2 flowbitsRectify_small (TargetRuleGroup T)

```
1: Let $D_e$ be the flowbits evasion DFA for target rule group $T$
2: for all accept 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 an accept state.
6: $k \leftarrow 0$;
7: for all simple path $P \in SP$ do
8:     $k \leftarrow k + 1$;
9:     Let $P = q_1q_2 \ldots q_m$, 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 a flowbits label that is set by default
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, $A_{i-1}^k$; flowbits: set $A_i^k$; flowbits: noalert;
17:        all options in $R(q_i)$ in the original rule set (header and
           body) except the flowbits options
18:        end
19:    end for
20:    // target rule
21:    Create flowbits label $A_{m+1}^k$
22:    Let $R^T$ be a target rule in $T$
23:    Add to $S_k$ the rule consisting of:
24:    begin
25:    flowbits: isset, $A_m^k$; flowbits: set $A_{m+1}^k$;
26:    all options in $R^T$ in the original rule set (header and body)
       except the flowbits options
27:    end
28: end for
```

VII. FLOWBITS EVASION RECTIFICATION

A. Solution for small rule sets

We can consider D_e as a directed graph. Theoretically, we need to add a rule set to detect each path from the start state to an accept state, which we call an evasion path. Fortunately, it is enough to consider only simple evasion paths, where a simple path is a path with no cycles. This is because rules added to detect all simple evasion paths can actually detect all evasion paths. Moreover, it is sufficient to consider only subset paths over all simple paths. Alg.2 details this procedure. Simple path translation into a rule set is given in lines 12-19 with the target rule in lines 21-27. For example, the set of

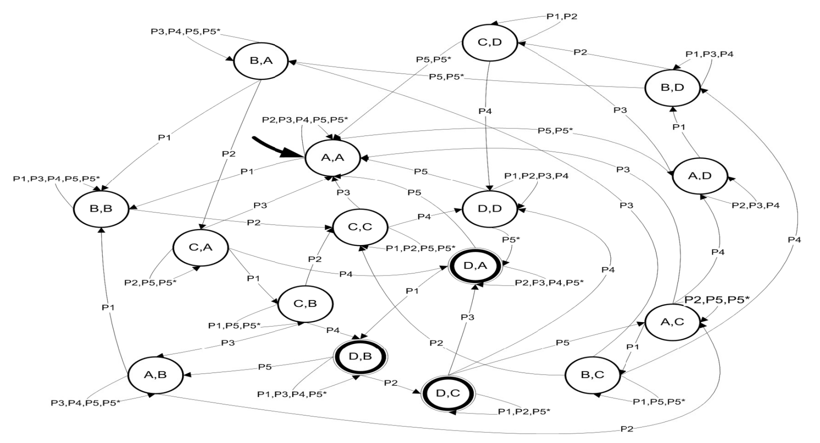

Fig. 3. D_e of the FTP rule set

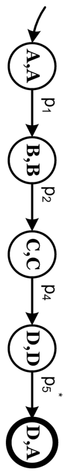

Fig. 4. Simple D_e evasion path

simple paths SP collected from D_e of Fig.3 has one simple path (after removing subset paths) as shown in Fig.4. There are five rules added for this simple path as shown in Table III.

The fact that Alg.2 searches for all simple paths to target states (line 5) attributes to it an exponential worst time complexity. Hence, Alg.2 is only suitable for rule sets of a small size. In the evaluation, the computational limit is quickly achieved for rule sets of size ≥ 9. In the following, we present a different solution that is feasible for both small and large rule sets.

B. Solution for large rule sets

For large rule sets, we will use Thm.1 below (proof in appendix).

Definition 2 (Vulnerable Rule): It is an evadable rule that renders the rule set vulnerable ($L_e(S) \neq \emptyset$) even if all other rules are not evadable.

Theorem 1: If a set Q of evadable rules makes the rule set vulnerable, then at least one of these evadable rules is vulnerable. (This theorem can be proven using contradiction) It follows that any evasion sequence needs to exploit at least one evadable rule, i.e., it contains at least one evasion packet. Hence, the idea is to behave in a *pessimistic* way and set a flag whenever an evadable rule is triggered. If Snort sees a target packet while the flag is set, it raises an alert. Interestingly, Thm.1 tells us that we only need to do this with vulnerable rules.

TABLE III
RULES ADDED FOR THE SIMPLE PATH IN FIG.4

$R_1(1)$	flowbits: set, A_1^1; flowbits: noalert;
$R_1(2)$	flowbits: isset, A_1^1; flowbits: set, A_2^1; flowbits: noalert;
$R_1(3)$	flowbits: isset, A_2^1; flowbits: set,A_3^1; flowbits: noalert;
$R_1(4)$	flowbits: isset, A_3^1; flowbits: set, A_4^1; flowbits: noalert;
$R_1(5)$	flowbits: isset, A_4^1;

Algorithm 3 flowbitsRectify_Large (TargetRuleGroup T_m)

```
1: construct $D_s$ and $D_a$ corresponding to $T_m$
2: // Find vulnerable rules of $D_s$
3: set $V_m = \{\}$; // set of vulnerable rules
4: **for all** non target rule $R_i$ in rule set S **do**
5:     construct $D_s^i$ and $D_a^i$, where $R_i$ is the only evadable rule.
6:     $D_e^i \leftarrow D_a^i \cap \neg D_s^i$
7:     **if** $D_e^i$ has a reachable accept state **then**
8:         $V_m \leftarrow V_m + \{R_i\}$
9:     **end if**
10: **end for**
11: **if** $V_m \neq \phi$ **then**
12:     create new label $F_m$
13:     Add new rule $R_m^T$ consisting of:
14:     begin
15:      flowbits: isset, $F_m$; //the flowbits condition
16:      all header and body options in a target rule of $T_m$ except
         flowbits.
17:     end
18:     **for all** $R_i \in V_m$ **do**
19:         Add to $R_i$ flowbits:set, $F_m$
20:     **end for**
21: **end if**
```

Alg.3 starts by determining the set V_m of vulnerable rules (lines 4–10). Next, it patches in D_s every destination state of a vulnerable rule by setting the flag F_m (line 19). Once a vulnerable rule has been triggered, an alert is raised at the encounter of a signature belonging to a target rule regardless of the flowbits state (lines 15–16). This pessimistic approach comes at a performance cost. However, it brings the benefit of having a polynomial complexity, which is an important scalability enhancement over Alg.2.

Considering the rule set of Table II and assuming all rules are evadable, the first step of the algorithm (lines 1–9) indicates that only rules R_3 and R_5 are vulnerable (note that Fig.3 is the D_e created assuming only R_5 is vulnerable).

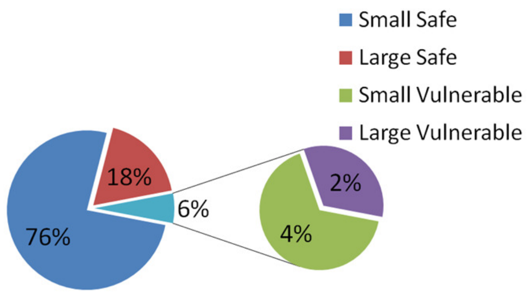

Fig. 5. Vulnerable and safe rule sets percentage when SFET is run in the cautious mode

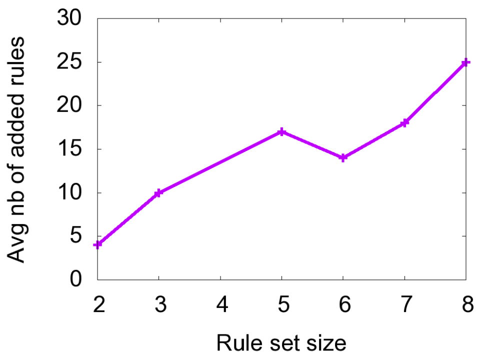

Fig. 6. Average Overhead to patch small rule sets

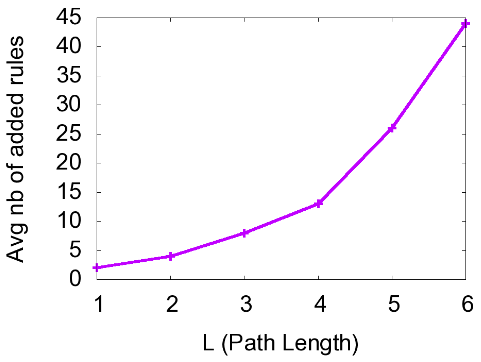

Fig. 7. Average Overhead from false positives control patch (for small and large rule sets)

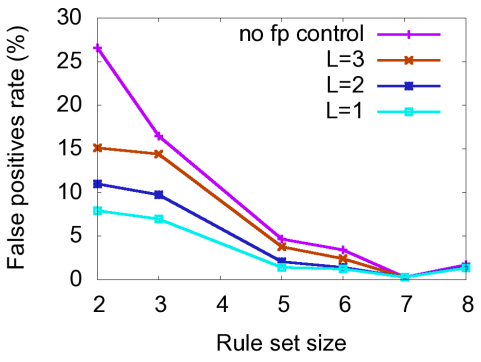

Fig. 8. Average false positives rate caused by the small rule set solution

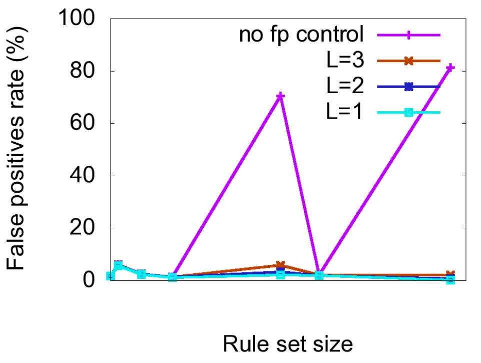

Fig. 9. Average false positives rate caused by the large rule set solution

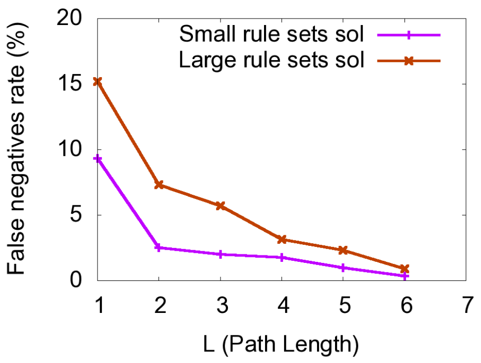

Fig. 10. Average false negatives rate for the small and large rule set solutions

So R_3 and R_5 are modified by inserting the flowbits option `flowbits:set` F_6. R_6^T is the added rule and all other rules are the same. Table IV shows the modified and added rules to patch the rule set.

We can formally prove that both Alg.2 and Alg.3 are complete (rule set's semantics preserved) and sound (fully eliminate the flowbits evasion).

C. False positives control

Even though the approach used for large rule sets also works for small rule sets, it potentially causes more false positives. While the latter only raises an alert if a complete evasion sequence is seen, the former does so only for important packets in an evasion sequence. However, the overhead caused by the latter is larger. In order to avoid false positives, Snort needs to consider session packets beyond a target state (in the patched rule set). These can give clues about who is running the session. If a target packet is encountered right

TABLE IV
MODIFIED AND ADDED RULES USING THE LARGE RULE SETS APPROACH

R_3	flowbits: isnotset, nauld; flowbits: isset, nalu; flowbits: isset, nalp; flowbits: unset, nalu; flowbits: unset, nalp; flowbits: set, F_6; flowbits: noalert;
R_5	flowbits: isset, nauld; flowbits: unset, nauld; flowbits: unset, nalu; flowbits: unset, nalp; flowbits: set, F_6; flowbits: noalert;
R_6^T	msg: "Normal User accesses important file"; flowbits: isset, F_6;

away it is most likely that it is an attack. Otherwise, if the following packets continue triggering normal transitions in D_s as normal users do, it becomes more and more probable that it is a benign session. Hence, the more session packets are considered afterward, the more accurate the decision becomes.

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 has been identified (for small rule set solution) or after a vulnerable rule is triggered (for large rule set solutions). Then all possible actions of a normal user are equivalent to all paths starting from any of these states. As a result, we can create rules corresponding to these paths to control the false positives rate caused by the patched rule set.

There is always a tradeoff between vulnerability and false positives. A patched rule set (whether for the small or large rule set algorithms) has zero false negatives yet potentially a lot of false positives. On the other hand, a non patched vulnerable rule set has no false positives (with regards to flowbit evasion). The 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 false positives control patch is useful when missing few evasions is better than having too many false positives.

Let L be the length of actions (or path length) Snort considers after it is put into a target state to decide if the session is run by a normal user or not. The tradeoff we have is that the shorter L we use, the less false positives we obtain, however, the more false negatives we might cause.

VIII. IMPLEMENTATION AND EVALUATION

We developed 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. A rule set is considered vulnerable if there exists an evasion sequence for any chosen target rule.

We collected publicly available rule sets (mostly from BleedingEdge [1] and SourceFire [19]). About 60% of the rules use flowbits matching 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 as evadable. In addition, there are about 6% and 4% 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.

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 ≥ 9) make up only 20% 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.5.

When applying the proposed solution to small vulnerable rule sets, the number of added rules in average is triple that of rules in the rule set (for both automatic and cautious modes). Fig.6 shows the average number of added rules for each rule set size (note: we do not find any vulnerable rule set of size 4).

For large vulnerable rule sets, the number of modified rules is the same as that 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 at 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 applied the false positives control patch for different values of L. On average, the number of rules added to control false positives increases exponentially as L increases (as expected) and this is shown in Fig.7.

To measure false positives caused by our patches, we run Snort with vulnerable rule sets and generated traffic according to their DFAs. To be more accurate, we generated traffic with both normal and evasion packets. However, we set the rate of evasion packets to be small (anywhere from 1% to 5%). A false positive occurs when the patched rule set raises an alert on a benign packet sequence (not an evasion sequence and the original rule set does not raise an alert on).

Fig.8 shows the average false positives rate for the small rule set solution. We see that patched rule sets of smaller size tend to have more false positives than those of bigger size. Besides, rule sets of size 2 or 3 have high average false positives rate because their DFAs are small and there is no significant difference between the number of evasion sequences and the number of normal sequences. However, when the false positives rate is high, the false positives control can be effective in decreasing the rate (e.g. the average false positives rate for rule sets of size 3 goes down from 17% to 7% when false positive control L=1 is used) .

Fig.9 shows the average false positives rate for the large rule set solution. In general, the false positives rate is small even when the false positives control is not applied. There are two exceptions where the false positives rates are very high ($\approx$75%). These occurred when we dealt with cases having many vulnerable rules in a rule set. Fortunately, the false positives control significantly decreases the false positives rates in these cases (to $\leq$5%).

Since the false positives control puts a patched rule set back to vulnerability, it is important to measure the false negatives rate it might cause. Specifically, we measure the number of evasion attacks not detected by the patched rule set (with the false positive control enabled) over the number of all evasion attacks. Fig.10 shows the average false negatives rate for different L. As we expected, the smaller L we use, the higher false negatives rate we have. In fact, when L goes to infinity, the false negatives rate goes to 0. This reflects the situation when we do not use the false positives control for a patched rule set and we can detect all evasion attacks. Besides, Fig.10 also shows that the large rule set solution causes more false negatives than the small rule set solution when the false positives control is applied. Finally, it seems that L=2 is a good value to use in order to balance between the false positives rate and the false negatives rate.

IX. Conclusion And Future Work

In this paper, we proposed an evasion technique to Snort flowbits rule sets which exploits, among others, the packet-based nature of Snort and/or the possibility of having loose rules. We suggested two main algorithms to resolve it. The first algorithm generates low false positives but can potentially run in exponential time and generate a large set of patch rules. Hence, it practically yields results for small rule sets of about less than dozen rules. The second algorithm runs in polytime and has a linear output with respect to the size of the rule set DFA. In this regard, it works for both small as well as large rule sets. It however generates high false positives. A tunable false positive control method was then suggested to balance the tradeoff between soundness and efficiency.

We implemented the SFET tool, which automatically calculates the evadability of a rule based on the its content and generates all possible evasion sequences to a given rule set. Besides, SFET can augment a vulnerable rule set with additional rules that thwart the flowbit evasion using the suggested algorithms. Evaluations showed that a good number of available rule sets are vulnerable to the proposed evasion. The practical nature of our solutions in generating little overhead for both small and large vulnerable rule sets was also demonstrated.

Our solution, in addition to curing existing rule sets, also provides a characterization of evadable rule sets, which helps administrators in the design of future flowbit evasion free rule sets and/or redesign existing ones. Finally, we note that, although our proposed flowbit evasion was studied for Snort, the underlying concept extends to any NIDS with stateful signatures. Hence, we consider its application to different NIDSs in future work.

References

[1] BleedingEdge Inc. http://www.bleedingthreats.net.

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

[3] S.A. Crosby and D.S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security Symposium*, Aug 2003.

[4] C. Giovanni. Fun with Packets: Designing a Stick. Technical report, Draft White Paper on Stick, Mar. 2001. http://www.eurocompton.net/stick.

[5] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2001. USENIX Association.

[6] Aib Issam, Tung Tran, and Raouf Boutaba. Characterization and solution to a stateful IDS evasion. In *IEEE ICDCS*, Canada, 2009.

[7] Gordon "Fyodor" Lyon. *Nmap Network Scanning*. Insecure.org, Sept. 2008.

[8] D. Mutz, G. Vigna, and R. Kemmerer. An experience developing an IDS stimulator for the black-box testing of network intrusion detection systems. In *ACSAC*, 2003.

[9] J. Novak, S. Sturges, and I. Sourcefire. Target-Based TCP Stream Reassembly. Sourcefire, Incorporated, Aug 2007.

[10] V. Paxson. Bro: a system for detecting network intruders in real-time. In *USENIX Security Symposium*, pages 3–3, 1998.

[11] T.H. Ptacek and T.N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection, 1998.

[12] M. Roesch. Snort–Lightweight Intrusion Detection for Networks. In *USENIX LISA*, 1999.

[13] S. Rubin, S. Jha, and B.P. Miller. Language-based generation and evaluation of NIDS signatures. In *IEEE S&P*, pages 3–17, May 2005.

[14] Shai Rubin, Somesh Jha, and Barton P. Miller. Automatic generation and analysis of NIDS attacks. In *IEEE ACSAC*, pages 28–38, 2004.

[15] S.Aubert. http://www.hsc.fr/ressources/outils/idswakeup/index.html.en.

[16] U. Shankar and V. Paxson. Active mapping: resisting NIDS evasion without altering traffic. In *IEEE S&P*, pages 44–61, May 2003.

[17] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking algorithmic complexity attacks against a NIDS. In *IEEE ACSAC*, 2006.

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

[19] SourceFire, Inc. http://www.sourcefire.com.

[20] Sourcefire, Inc. *Snort 2.9.1*. The Snort Project, 23 Aug 2011. http://www.snort.org/downloads/1107.

[21] G. Vigna, W. Robertson, Vishal Kher, and R.A. Kemmerer. A stateful intrusion detection system for world-wide web servers. In *IEEE ACSAC*, 2003.

[22] Giovanni Vigna, William Robertson, and Davide Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *ACM CCS*, 2004.

[23] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM CCS*, 2002.

[24] Vinod Yegneswaran, Jonathon T. Giffin, Paul Barford, and Somesh Jha. An architecture for generating semantics-aware signatures. In *USENIX Security Symposium*, 2005.