Attack Generation for NIDS Testing Using Natural Deduction

Shai Rubin, Somesh Jha and Barton P. Miller

January 23, 2004

Abstract

A common way to elude a signature-based NIDS is to transform an attack instance that the NIDS recognizes into another instance that it fails to recognize. For example, to avoid matching between the attack payload and the NIDS signature, attackers split the payload into several TCP packets, change it syntactically while preserving its semantics, or hide it between benign messages. We study attackers' ability to find attack instances that elude a NIDS and our ability to recognize such instances.

We observe that different instances of a given attack can be derived from each other using simple transformations that change either the attack transport mechanism or its payload. We model these transformations as inference rules in a formal natural deduction system. Starting from an exemplary attack instance, we use an inference engine to automatically generate all possible instances derived from a particular collection of rules. The result is a simple yet powerful tool capable of both generating attack instances for NIDS testing and determining whether a given sequence of packets is an attack.

During several testing phases using different sets of rules, our tool exposed serious vulnerabilities in Snort—a widely deployed NIDS. Attackers acquainted with these vulnerabilities would have been able to construct instances that elude Snort for any TCP-based attack, any Web-CGI attack, and any attack whose signature is a certain type of regular expression.

1 Introduction

The goal of a Network Intrusion Detection System (NIDS) is to alert a system administrator each time an intruder tries to penetrate the network. A *signature-based* NIDS defines penetration via a table of malicious signatures: if an ongoing activity matches a signature in the table, an alarm is raised [24, 32]. Such systems are widely used [39, 46] because they are simple to use and provide concrete information about the events that have occurred. The weakness of a signature-based NIDS is its inability to recognize an attack that is just slightly different from the attack signature it uses.

An attacker wishing to stealthily penetrate a network monitored by a signature-based NIDS can exploit this weakness in two ways. First, they can use an attack whose signature is not known to the NIDS. In an up-to-date system, such attacks are difficult to find. Second, they can use a known attack, but try to elude the NIDS by finding an instance of the attack that the NIDS does not detect. For example, to elude a NIDS that does not perform TCP reassembly, the attacker can fragment the attack signature into several TCP packets [12, 36, 42]. Or, to elude a NIDS that uses only printable characters in its signatures, an attacker can change the signature of an HTTP attack by substituting equivalent hexadecimal ASCII values for the characters in a URL [11]. If an attacker can find a single instance of the known attack that eludes the NIDS, then the NIDS is—simply put—useless.

We study the ability of attackers to find attack instances that elude a NIDS and the ability of a NIDS to detect such instances. To be more concrete, we translate these abilities into the following two problems.

1. The *black hat* problem: given an attack $\mathcal{A}$ and a specific NIDS, transform the attack into a variant that evades the NIDS.
2. The *white hat* problem: given an attack $\mathcal{A}$ and a sequence of network packets S, determine whether S is an instance of $\mathcal{A}$.

We propose a novel approach to rigorously tackle the black and white hat problems by formalizing them in terms of natural deduction [35]. We observe that variants of the same attack can be methodically derived from each other. To translate this observation into practice, we first formally express the attacker knowledge in a set of inference, or transformation, rules; each rule represents an atomic mutation the attacker can use to hide the attack signature. Then, starting from a known attack instance, we use an inference engine [43] to successively apply the rules and automatically compose all attack instances based on any combination of the rules. Finally, to solve the black hat problem, we feed the instances into the given NIDS until we find one that is undetected. To solve the white hat problem, we check whether the given instance S matches one of the instances generated.

Our approach has several advantages. First, it models a wide variety of the transformations that attackers use. Unlike previous work that focused on transport level mutations (e.g., TCP/IP) [12, 36], our work uses rules to model both transport and application level transformations (e.g., HTTP). Second, since rules represent simple independent transformations, our deduction system can (i) combine the transformations, (ii) incorporate other mutations not considered in this paper, and (iii) create inverse transformations; for example, TCP fragmentation vs. TCP reassembly, or HTTP decoding vs. HTTP encoding. These transformations enable us to start the derivation from any attack instance. First, we use them to go "backwards", until we derive an instance to which they cannot be applied; then, we use this instance as a root from which we generate all instances using the original ("forward") transformations.

Based upon these ideas, we used Prolog—a language particularly suitable for implementing natural deduction systems [43]—and implemented *AGENT*: an *attack generation for NIDS Testing* tool. AGENT's biggest advantage is its relative completeness: the Prolog engine can derive all possible instances from the given set of inference rules. With the complete set AGENT derives, we can find instances that elude a NIDS even when these instances are few and are unlikely to be found using random testing techniques. In practice, when we use many inference rules, generating all instances is infeasible. However, our results show that even though AGENT uses a small set of inference rules that derive a relatively small number of instances, it is still effective in finding instances that elude a widely-deployed NIDS.

To summarize, this paper makes three primary contributions:

- **A formal model for the black and white hat problems.** We formalize these problems as a natural-deduction system in which the inference rules capture the attacker's ability to transform attacks. Our model allows us to use automatic tools to derive mutants of known attacks.
  - The model is complete. For a given attack $\mathcal{A}$, the model concisely defines all instances of $\mathcal{A}$ derived from an exemplary instance of $\mathcal{A}$ by a given set of transformation rules.
  - The model is sound. For a given attack $\mathcal{A}$, when each inference rule is sound (i.e., never produces a sequence of packets that is not an instance of $\mathcal{A}$), our model is also sound.

  These properties enable us to generate sets of attack instances that can be used to detect the presence or the absence of vulnerabilities in a NIDS.

- **AGENT, a practical tool for testing NIDS.** We have used our formal model to build AGENT, a complete and sound tool for attack generation. Given a set of inference rules and a representative instance of an attack $\mathcal{A}$, AGENT generates all and only those attack instances that can be derived by the given rules.
  - When we connect AGENT to a specific NIDS, it can serve as a black hat tool. A failure of the NIDS to detect an instance indicates a vulnerability in the NIDS, and a successful detection of all instances demonstrates its correctness.
  - When we use AGENT without a NIDS, it can serve as a white hat tool. For any TCP sequence S, attack $\mathcal{A}$, and a set of transformation rules, AGENT determines whether S can be derived from an exemplary instance of $\mathcal{A}$.

  AGENT is not efficient enough to perform as a stand-alone on-line NIDS; it can be used as an aid for

NIDS developers. Either as a black or white hat tool, AGENT provides the derivation sequence for each instance it derives.

- **Improving a widely deployed NIDS.** Using AGENT, we found several serious vulnerabilities in Snort [24, 39]. We exposed vulnerabilities in the TCP engine of Snort, the way Snort handles HTTP requests, and its pattern-matching algorithm. An attacker acquainted with these vulnerabilities could have caused Snort to miss any TCP-based attack, any HTTP scripting attack, and many attacks that require wild characters in their signatures (a signature like "foo*bar"). These vulnerabilities were reported to the Snort development team. Some were immediately fixed in Snort version 2.0.2, others will be fixed by the time that this paper is published.

The rest of this paper is organized as follows. Section 2 presents related work on the black and white hat problems. Section 3 illustrates how attack variants of a real attack can be derived from each other. Section 4 formalizes the notion of derivation using natural deduction system in which attack variants can be automatically derived from each other. Section 5 starts with a general discussion about selection of transformation rules to use, and continues with a description of both the transport and application rules we used to find attack instances that elude Snort. Section 6 presents AGENT, how it was used to find vulnerabilities in Snort, and the specific vulnerabilities AGENT exposed. Section 7 discusses future work.

2 Related Work

The black hat problem. The work of Ptacek and Newsham [36] described methods for evasion of a signature-based NIDS. Their methods include transformations that modify the attack on both the link (IP) and transport (TCP) levels. They manually built a set of attack instances, and showed that these instances eluded every commercial NIDS they tested. Handley and Paxson discussed similar transformations exploiting inherent ambiguities of the TCP and IP protocols [12, 32].

There are three major differences between our work and the work of these researchers. First, while they focus on individual transformations, we provide a formal model to rigorously generate all possible combinations from a set of transformations. Second, while they provide examples of methods to elude a NIDS, we provide an automated tool that uses such methods to actually find the undetected instances. Third, although our transport level transformations are based on their methods, we explore payload level transformations as well, and our model can be extended to include their IP level transformations.

The black hat problem was also investigated in the context of other types of intrusion detection systems. Wagner and Soto showed a model, based on formal language theory, that attackers can use to evade a host-based IDS [49]. Tan et al. provide evidence that this theoretical model can be used in practice [17, 45]. Our approach is similar to Wagner's: we focus on a NIDS rather than on host-based IDS, and our formal model is based on natural deduction rather than on regular languages.

Hackers also have developed tools for attack obfuscation. Fragroute [42] splits an attack payload into several TCP and IP packets, but it neither always preserves the attack semantics nor enables automatic modifications of the attack payload. Tools to obfuscate binary code in shell exploits are well known [9], but we leave this type of transformation to future work.

The white hat problem and NIDS verification. While the white hat problem has attracted much more attention than the black hat problem (see survey papers, [4, 22, 25]), the particular problem of NIDS validation has not received much attention. Since network speed is increasing rapidly, some researchers have focused on the ability of NIDS to monitor large networks [16, 18]. Lippman et al. presented a comprehensive effort to evaluate IDS capabilities [20, 19] (with a seminal critique by McHugh [26]). However, they focused on comparing capabilities of several NIDS to detect a number of attacks, while our methods rigorously test a single NIDS for its ability to detect many instances of a single attack.

To the best of our knowledge, AGENT is the first tool that can be used to show that a NIDS correctly

identify all possible attack instances derived by a given set of transformation rules.

Resisting attacks on NIDS. Handley et al. [12] and Sommer et al. [41] present techniques that remove TCP and IP ambiguities from network connections. These techniques can be used to prevent at least one of the TCP vulnerabilities we found in Snort (Sections A.1.2). However, to the best of our knowledge, these methods are in a preliminary stage of research and are not yet widely deployed.

Security protocol verification. There is a vast body of work on verification of security protocols [5, 15, 21, 23, 28, 27, 29, 50]. Deductive systems are used to model the "knowledge" of the participants and the adversary in the security protocol. For example, the NRL protocol analyzer [28] uses Prolog to formalize the set of facts learned by a participant. A similar approach is taken by Paulson [31], who uses Isabell to prove the correctness of security protocols. Abstractly speaking, these techniques are related to the approach taken in this paper because we also use deductive systems to model the power of the adversary. In the future, we will explore techniques for state-space reduction available in the security protocol verification literature [40].

Deductive databases. Since we use deductive systems to model the transformations that an attacker can perform, the literature on efficient evaluation of logic programs from the deductive databases literature is relevant. There are several techniques and systems for efficient bottom-up [37] and top-down evaluation [10, 48] of logic programs. In our context, these evaluation techniques have the promise of providing efficient algorithms for the black and white hat problems. We will explore these connections with deductive databases in the future.

3 Example: Derivation of Attack Variants

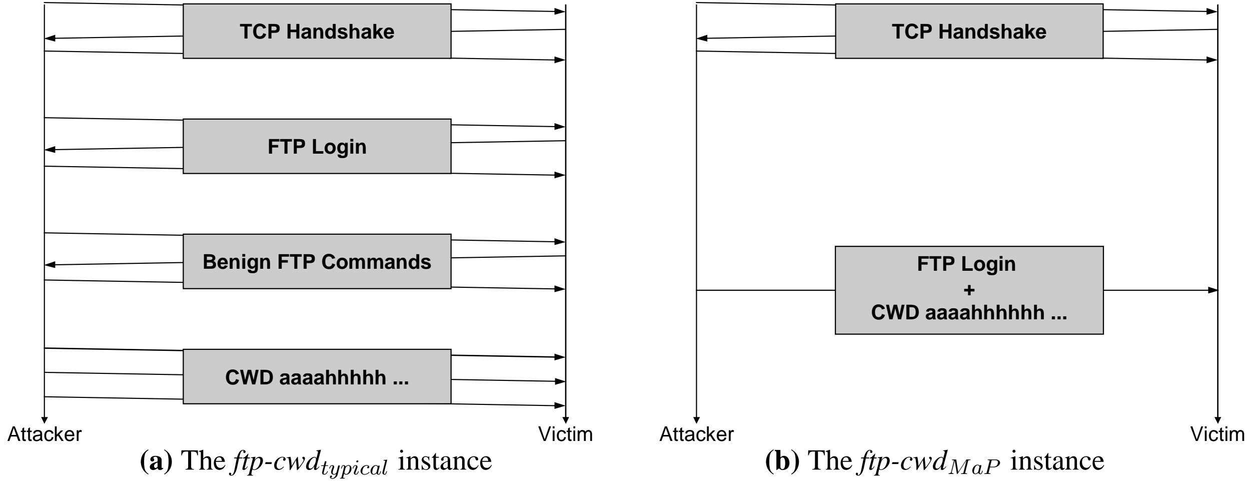
We illustrate the main idea behind our work: given an instance of an attack $\mathcal{A}$ and a set of transformations that preserve the semantics of $\mathcal{A}$, we can systematically transform this instance into another instance of $\mathcal{A}$. We start with examples of two attack instances of a known FTP vulnerability. Then, we describe semantics preserving *transformation rules*, which are single-step transformations that transform a known instance into a new one. Last, we illustrate that the two instances are variants of each other: one instance can be derived from the other by repeatedly applying the single-step transformations. While the example we present is simple, it is based on a real vulnerability found in Snort (Section A.2.3).

Our example vulnerability is a published buffer overflow in a commonly used FTP server (*BlackMoon* FTP server for Windows, CAN-2002-0126 in [30]); exploiting the overflow may crash the server or present root privileges. The exploit causes the overflow by providing an overly-long argument for the FTP CWD (change directory) command. We call this attack *ftp-cwd*.

The first instance of *ftp-cwd* we present is similar to instances that can be found on many hacker sites (e.g., [1]). Since this instance is so common, we call it *ftp-cwd*$_{typical}$ (Figure 1a). It contains four phases, each containing several TCP packets: (i) TCP handshake, (ii) FTP login, usually achieved by anonymous login, (iii) innocent phase in which the attacker browses the server using benign FTP commands, and (iv) attack phase in which the attacker launches the attack by sending a long CWD command. Since long FTP commands may look suspicious, attackers commonly fragment the long argument into several TCP packets.

To illustrate derivation of one *ftp-cwd* instance from another, we now present a much shorter instance of *ftp-cwd* (Figure 1b). We called it the *Meat and Potatoes* (*MaP*) version of *ftp-cwd*, denoted *ftp-cwd*$_{MaP}$, because, as we discuss in Section 4.1, it is the simplest instance possible with respect to our rules.

There are two main differences between *ftp-cwd*$_{MaP}$ and *ftp-cwd*$_{typical}$. First, *ftp-cwd*$_{MaP}$ contains a single attack packet (we do not count packets in the TCP handshake phase because they are part of any connection, benign or malicious.). Since FTP and TCP belong to two different levels of the protocol stack [52], the FTP server is (and should be) indifferent to the number of TCP packets used to deliver the FTP messages. Therefore, it is possible to send the three necessary FTP messages (USER, PASS, and CWD)



(a) The *ftp-cwd*$_{typical}$ instance **(b)** The *ftp-cwd*$_{MaP}$ instance

Figure 1: Two *ftp-cwd* variants.

in a single TCP packet. Second, *ftp-cwd*$_{MaP}$ contains only the data that is absolutely necessary for a successful *ftp-cwd* attack; it does not contain any victim response. Note that these differences do not reduce the effectiveness of the *ftp-cwd*$_{MaP}$ instance; from the attacker's point of view, if the victim responds to *ftp-cwd*$_{typical}$, it should also respond to *ftp-cwd*$_{MaP}$.

While the *ftp-cwd*$_{typical}$ and the *ftp-cwd*$_{MaP}$ might look different, both contain the necessary messages for a successful *ftp-cwd* attack. Hence, intuitively speaking, one can infer *ftp-cwd*$_{typical}$ from *ftp-cwd*$_{MaP}$, and vice versa. Next, we illustrate this inference.

Consider the following two *transformation* rules:

1. R_1 (*TCP-fragmentation*): if S_1 is an instance of an attack $\mathcal{A}$, and S_2 is obtained from S_1 by (i) fragmenting a single TCP packet $p_i \in S_1$ into two packets $\hat{p}_i, \hat{p}_{i+1} \in S_2$, and (ii) copying all packets other than p_i from S_1 to S_2 (shifting the indexes of all packets after p_i by one), then S_2 is an instance of $\mathcal{A}$.
2. R_2 (*FTP-padding*): if S_1 is an instance of an FTP attack $\mathcal{A}$ that consists of at least one malicious FTP command after login (e.g., like the CWD command in the *ftp-cwd* attack), and S_2 is obtained from S_1 by inserting a benign FTP command between the login and the malicious command (but not the "QUIT" command), then S_2 is an instance of $\mathcal{A}$.

We call these rules *semantics preserving*: they do not alter the semantics of S_1. According to the TCP specification [33], it is legal to fragment TCP packets as desired. To the best of our knowledge, every FTP attack can be inflated, or padded, using benign FTP commands[1].

If *ftp-cwd*$_{MaP}$ is an instance of the *ftp-cwd* attack, then by using R_1 and R_2 it is possible to derive the conclusion that the *ftp-cwd*$_{typical}$ (Figure 1a) is also an instance of *ftp-cwd*. We successively apply R_1 on *ftp-cwd*$_{MaP}$ to fragment the single attack packet into the attack packets of *ftp-cwd*$_{typical}$. On the resulting instance, we apply R_2 and pad the attack with benign FTP commands. Using natural deduction terminology, we say that the *ftp-cwd*$_{typical}$ is *derived* from *ftp-cwd*$_{MaP}$ using the rules R_1 and R_2. More formally we write: *ftp-cwd*$_{MaP} \vdash_{\{R_1,R_2\}}$ *ftp-cwd*$_{typical}$.

From the derivation process illustrated above, we can make three important observations:

1. R_1 and R_2 define a closure over a subset of *ftp-cwd* instances. R_1 and R_2 can be used to derive not only the *ftp-cwd*$_{typical}$ instance, but also other instances of *ftp-cwd*. Using these two rules we can derive every

[1]If there exists an FTP attack that cannot be padded by arbitrary FTP commands, then the rule is changed to only allow legal modifications.

ftp-cwd instance with several benign FTP commands and several TCP packets delivered in-order. This observation motivates us to automate the derivation process, because this enables (i) identification of every *ftp-cwd* instance that falls into the category mentioned above, and (ii) generation of finitely many instances to be used for a NIDS testing.

2. R_1 and R_2 are commutative. To derive *ftp-cwd*$_{typical}$ it is possible to first change the attack payload by padding the attack with benign FTP commands, and then to change way the attack is delivered by fragmenting it into the several packets. This observation greatly simplified the implementation of an automatic derivation tool as discussed in Section 5.2.2.
3. The inference process can be bi-directional. Consider the reverse rules: $\overleftarrow{R_1}$ as de-fragmentation and $\overleftarrow{R_2}$ as removal of padding. It is easy to see how *ftp-cwd*$_{MaP}$ can be derived from *ftp-cwd*$_{typical}$. This bi-directional property suggests that a derivation process can start from any attack instance, so finding instances that elude a NIDS may be less sensitive to the derivation starting point. We use this observation when we define the starting point for our automatic derivation tool in Section 4.1.

Next, we describe a model that formalizes our intuition of inferring attack instances.

4 A Natural Deduction Model for Attack Generation

We derive attack instances using natural deduction [35]. A natural deduction system uses a collection of predefined inference rules to derive conclusions from already known facts; the new conclusions can be used as facts to derive further conclusions, and so on. For an attack $\mathcal{A}$, we present an inference system to derive TCP sequences that have $\mathcal{A}$'s semantics. The derivation starts from a representative instance of $\mathcal{A}$, the meat-and-potatoes instance, and continues by successively applying syntactic transformations to derive new instances of $\mathcal{A}$.

Our goal is to define a natural deduction model for the black and white hat problems. To do so requires three steps. First, to precisely define attack instances, we need a way to represent the instances. Second, to start the derivation process, we need an exemplary attack instance. Third, to propel the derivation process, we need inference rules that show how to derive new instances from the others. Here, we discuss the attack representation, the selection of an exemplary instance, and the way we model the black and white hat problems in terms of natural deduction. Section 5 presents the inference rules that we used.

4.1 Attack Representation

Natural deduction uses syntactic transformations to derive conclusions from facts. Therefore, we need to represent an attack in a way that is easy to syntactically manipulate. To achieve this goal, we represent an attack as a sequence of TCP packets. For our purposes, each attack has two participants: the *attacker* and the *victim*. We call the packets the attacker sends *attack packets*, denoted a_i, and the packets the victim sends *response packets*, denoted r_i.

The choice of a sequence of TCP packets to represent attacks is not arbitrary and is convenient for several reasons. First, this method of representation is the most obvious choice because the majority of known attacks use TCP; for example, 88% of Snort rules target TCP communication. Second, since our focus is from the TCP level up, we use TCP to hide low level details of the network protocols. Last and most important, the TCP representation exposes both TCP parameters and application data. It enables modeling of the attacker's control over the application data, the attacker's control over TCP parameters and headers, and the attacker's ability to inject TCP packets at will.

The next step in the definition of a natural deduction system is defining the derivation starting point.

For a given attack $\mathcal{A}$ we define a special instance: a TCP sequence called the *Meat-and-Potatoes* sequence, denoted $\mathcal{A}_{MaP}$. The $\mathcal{A}_{MaP}$ is the single starting point for every derivation required to solve the black or white hat problems. For simplicity, we assume that a $\mathcal{A}_{MaP}$ contains only one attack packet that is

part of a legal TCP sequence, and does not contain any victim response (as in the *ftp-cwd*$_{MaP}$ sequence in Figure 1b). To the best of our knowledge, in the majority of network attacks, the exploit does not depend on data from the victim, so the attacker activity can be combined into a single TCP packet. In the future, if an attack must contain more than one packet, we will apply the rules for each packet separately.

For any attack $\mathcal{A}$, two questions about the $\mathcal{A}_{MaP}$ should be addressed.

1. How do we identify an $\mathcal{A}_{MaP}$? Identifying an $\mathcal{A}_{MaP}$ is driven by the common properties of the instances the rules derive. For example, the common property of instances derived by *TCP-fragmentation* alone is that they contain several TCP packets that are fragments of the $\mathcal{A}_{MaP}$ payload. To ensure that the natural deduction system generates all possible instances with certain properties, the $\mathcal{A}_{MaP}$ is defined as an instance that cannot be derived, using the inference rules, from any other instance. In other words, the $\mathcal{A}_{MaP}$ is the root of the derivation tree: it derives all instances and no instance derives it. For example, when considering only the *TCP-fragmentation* and *FTP-padding* rules (Section 3), there is no instance that can derive *ftp-cwd*$_{MaP}$ (Figure 1b).
2. How do we handle the case when the $\mathcal{A}_{MaP}$ is not unique? For example, since URLs in an HTTP attack can be expressed either by printable characters or their equivalent ASCII hexadecimal values [11], one might be tempted to use inference rules that substitute characters in both directions. However, since such bidirectional substitution rules enable circular derivation, for example from "CNN.COM" to "%43NN.%43OM" and back, they do not define a unique $\mathcal{A}_{MaP}$. In such a case, we artificially split the rules into two categories—*forward* and *backwards* rules—and define the $\mathcal{A}_{MaP}$ with respect to the forward rules only. For example, we force forward substitution from printable characters to their ASCII hexadecimal values and define the $\mathcal{A}_{MaP}$ to contain only printable characters. In all the rules used in this paper, forcing such an order enabled us to find a unique $\mathcal{A}_{MaP}$. This ordering did not reduce the number of attack instances that our natural deduction system generated; formally, it did not affect the completeness of our model.

The answers to these two questions suggest an automatic way to derive the $\mathcal{A}_{MaP}$. Since the $\mathcal{A}_{MaP}$ serves as the root for forward derivation, and since implementing backwards and forward versions of rules is simple, there is no limitation to start the derivation process from any instance, use the backwards rules to derive the $\mathcal{A}_{MaP}$, and then to generate all instances using the forward rules. The current implementation of AGENT does not include this capability; however, we plan to explore this opportunity in the near future.

4.2 A Natural Derivation System for Solving the Black and White Hat Problems

We have defined how to represent attack instances and the starting point for the derivation process. To complete the definition of the natural deduction system we need to define the inference rules and the black and white hat problems in terms of natural deduction. Here, we formally define the two problems and leave the inference rules for the next section. We start with the definition of attack closure.

Definition 1 (Attack Closure) *Let Φ be a collection of inference rules, and A_{MaP} be the MaP sequence of an attack $\mathcal{A}$. $\mathcal{A}$'s **closure** with respect to Φ, denoted $\mathcal{A}_\Phi$, is the collection of TCP sequences derivable from $\mathcal{A}_{MaP}$ using a finite number of applications of the inference rules. Formally, $\mathcal{A}_\Phi = \{s \mid \mathcal{A}_{MaP} \vdash_\Phi s\}$.*

Now we formalize the black and the white problems:

Definition 2 (Black Hat Problem) *Let $\mathcal{A}$ be an attack, $\mathcal{N}$ be a NIDS, and Φ be a collection of inference rules. Let $\mathcal{A}_\mathcal{N}$ be the collection containing each TCP sequence that $\mathcal{N}$ recognizes as $\mathcal{A}$. The black hat problem is to find a sequence S such that $S \in \mathcal{A}_\Phi \backslash \mathcal{A}_\mathcal{N}$.*

Definition 3 (White Hat Problem) *Let $\mathcal{A}$ be an attack, S be a TCP sequence, and Φ be a collection of inference rules. The white hat problem is to determine whether $S \in \mathcal{A}_\Phi$, or $\mathcal{A}_{MaP} \vdash_\Phi S$.*

The definitions above highlight the advantages and disadvantages of using inference to solve the black and white hat problems. Formally defining them as natural deduction problems enables the usage of formal

logic tools to automatically solve both problems. In the black hat case, we generate unrecognizable variants of $\mathcal{A}$ and in the white hat case we detect sequences that are variants of $\mathcal{A}$. However, the formal definition also exposes a limitation. We are able to find only instances that are derived from the $\mathcal{A}_{MaP}$ instance using the inference rules in Φ. Hence, our ability to find and detect attack instances greatly depends on the composition of the rules in Φ and the $\mathcal{A}_{MaP}$ instance. While the $\mathcal{A}_{MaP}$ may be easy to create, finding effective inference rules is a more delicate task. We address this task in the next section.

5 Transformation Rules

Our ability to find attack instances that elude a given NIDS or to detect whether a sequence S is a variant of an attack $\mathcal{A}$, depends on the composition of the inference rule set. We start by discussing the qualities that are desirable in a rule set. Then, we give an example of a practical rule set as used in our experiments. As the results in Section 6 show, working with these qualities in mind pays off: using this rule set exposes several serious vulnerabilities in Snort.

5.1 Building an Effective Rule Set

Selecting the transformation rules is similar to programming: it requires expertise and human thinking. We present the lessons we learned while building a rule set for AGENT. We believe that the guidelines provided here will be useful for others constructing their own rule set.

5.1.1 Desirable Properties of Transformation Rules

The most important property for a transformation rule is *soundness*. A rule is sound if it does not change the attack semantics: the rule can be applied to any instance of a given attack, and it derives a TCP sequence that is an instance of this attack. If every rule is sound, then the entire system is sound as well. Given an attack $\mathcal{A}$, a sound set of rules is important for solving both the black and white hat problems. For the black hat problem, soundness means never generating TCP sequences that do not have the semantics of $\mathcal{A}$. For the white hat case, soundness means detecting only those TCP sequences that do have $\mathcal{A}$'s semantics.

The second desirable property of a rule set is *completeness*. For a given attack $\mathcal{A}$, it means that the inference rules enable the derivation of any TCP sequence that has the semantics of $\mathcal{A}$. Like soundness, completeness also is important for solving the black and white hat problems. For the black hat problem, completeness means that if there exists an attack instance that eludes a NIDS, we will eventually find it. For the white hat case, completeness means the ability to detect any instance of A.

When a set of transformation rules Φ is both sound and complete, then for every TCP sequence S and for every attack $\mathcal{A}$, S is an instance of $\mathcal{A}$ if and only if S belongs to the closure of $\mathcal{A}$ (or $S \in \mathcal{A}_\Phi$). Essentially, a derivation tool that uses such a sound and complete Φ is a perfect NIDS.

5.1.2 The Structure of the Rule Set

Achieving soundness is not easy and requires expertise in the specifications of the protocol (e.g., TCP) and application (e.g., HTTP) that the attack exploits. What makes the situation even more difficult is that specifications can be ambiguous [12, 44] and not all implementations obey their specifications. An example of a disparity between implementation and specification is the BSD finger server (version 0.17). While the finger specification permits (but discourages) escape characters in a finger query [51], this server does not support queries with such characters, and a sound transformation rule for this server cannot dictate insertions of such characters. Achieving soundness requires the knowledge of both the specifications and implementations of the protocol and application that the attack exploits.

Theoretically, it is possible to build a complete rule set, but we found that such a rule set is impractical. Since the number of instances a rule set derives can be infinite, even an incomplete rule set with only a few rules derives a large number of instances (see Section 6).

From our experience with AGENT, we have developed two strategies that help to achieve soundness and

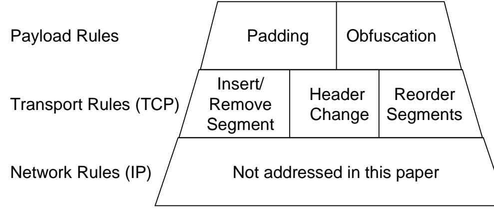


Figure 2: The hierarchy of inference rules.

deal with the practical limitations of completeness.

To address the difficulty in developing sound rules, we divide the rules into levels based upon the protocol stack model (Figure 2): *network* and *transport* level rules modify the way the attack is delivered but do not modify the attack payload[2], and *payload* level rules modify the attack payload itself. Within each level, rules are divided into different types according to the way they modify the attack. At the transport level, we have rules that add or remove packets from a TCP stream, change the packet header, and change the order of packets. At the payload level, we have rules that obfuscate the malicious subsequence in the attack payload in a way that the NIDS signature will not match it, and rules that pad the malicious subsequence with benign data. The advantage of this hierarchical structure is that it reduces the chance of writing an unsound rule; the person that develops a rule can focus on a single aspect of the attack.

To address the infinite number of instances that a complete rule set dictates, we adopt two strategies. First, we focus on rules that only derive a finite number of instances. For example, a rule that retransmits a packet many times is required for completeness, but it is not practical to use. Instead, we limit the number of retransmissions per packet to one; it is reasonable to assume that even few retransmissions will be enough to expose a bug in the way a NIDS handles retransmission. Second, we do not use all rules in every testing phase. While this hurts the overall completeness, it drastically reduces the number of instances we need to test. Our results show that these two strategies effectively expose vulnerabilities in Snort. Further, when a NIDS detects all instances derived from an incomplete set of rules, it increases our confidence that the NIDS behaves correctly with respect to the set of rules we considered.

5.2 Inference Rules Description

Each rule has the structure of $[RuleLevel][RuleName] \stackrel{\text{def}}{=} \frac{\mathcal{I}_{\mathcal{A}}(S_1), RulePredicate(\dots)}{\mathcal{I}_{\mathcal{A}}(S_2)}$. To specify the rule name we use the name of the protocol if the rule is a transport rule, or application if the rule is a payload rule. On the right hand side, we have the rule functional description that reads:

if ((S_1 is an instance of $\mathcal{A}$) && $RulePredicate(\dots)$) then S_2 is an instance of $\mathcal{A}$

RulePredicate specifies how the conclusion of the rule (S_2) relates to the fact the rule uses (S_1); this is a mechanism to enforce semantic preserving transformations. For example, *RulePredicate* may state that S_2 must be a permutation of S_1 to conclude that S_2 is an instance of $\mathcal{A}$. The predicate arguments can be either TCP sequences or packets, depending on the predicate.

5.2.1 Transport Level Inference Rules

We present the transport rules in Table 1. We use S_i to denote a TCP sequence, a_i to denote the ith packet of the attacker, and r_j to denote the jth response of the victim.

As can be noted, the rules not only change or add packets the attacker sends, but also add response packets from the victim. As mentioned in Section 4.1, the $\mathcal{A}_{MaP}$ sequence does not contain responses from the victim. However, responses or acknowledgments play a crucial part in intrusion detection. The ability of

[2] In this paper we do not address network rules, but our natural deduction model can support them.

Name	Description	Formal Description
TCP Fragmentation (R_1)	The ith attack packet, $a_i \in S_1$, is fragmented into two packets, $a'_i, a'_{i+1} \in S_2$. The *frag* predicate holds if and only if a'_i, a'_{i+1} is a legal TCP fragmentation of a_i [33]. When exists, the original response to a_i is deleted (r_i); two responses are added to each of the new attack packets (r'_i, r'_{i+1}).	$\dfrac{\mathcal{I}_\mathcal{A}([a_1,\ldots,a_i,r_j,\ldots,a_n,r_m]), frag(a_i,a'_i,a'_{i+1})}{\mathcal{I}_\mathcal{A}([a_1,\ldots,a'_i,r'_j,a'_{i+1},r'_{j+1},a_{i+1}\ldots a_n,r_m])}$
TCP Permutation (R_2)	S_2 is a restricted permutation of S_1. *restrictedPermute* holds if S_2 is a permutation of S_1 with the following two restrictions: (i) it preserves the original order between packets and their corresponding retransmission packets, and (ii) it preserves the original order between attack packets and their responses.	$\dfrac{\mathcal{I}_\mathcal{A}(S_1), restrictedPermute(S_1,S_2)}{\mathcal{I}_\mathcal{A}(S_2)}$
TCP Retransmission (R_3)	This rule specifies a family of rules in which the attack packet, $a_i \in S_1$, is retransmitted in S_2. $Retrans_k$ holds if and only if a_i is retransmitted in a way that preserves the semantics of S_1. Table 2 presents predicates that specify semantics preserving retransmissions.	$\dfrac{\mathcal{I}_\mathcal{A}([a_1,\ldots,a_i,r_j,\ldots,a_n,r_m]), retrans_k(S_1,a_i,a'_i)}{\mathcal{I}_\mathcal{A}([a_1,\ldots,a_i,a'_i,r_j,\ldots a_n,r_m])}$
TCP Header Change (R_4)	This rule specifies a family of rules in which the TCP header of the attack packet, $a_i \in S_1$, is changed in S_2. $hdChange_k$ holds if and only if the header change between a_i and a'_i does not alter the attack semantics. The investigation of these rules is left for future work.	$\dfrac{\mathcal{I}_\mathcal{A}([a_1,\ldots a_i,r_j,\ldots,a_n,r_m]), hdChange_k(a_i,a'_i)}{\mathcal{I}_\mathcal{A}([a_1,\ldots,a'_i,r_j,\ldots,a_n,r_m])}$

Table 1: Semantic preserving TCP inference rules.

Name	Holds is and only if	Scenario
$retrans_1(S,a_i,a'_i)$	$a'_i{=}a_i$ except that: (i) $a'_i.RST$ is set, and (ii) the sequence number in a'_i is smaller than the acknowledgment number in the last response in S before a_i. More formally, let $r_k{\in}S$ be the last response before a_i, then $a'_i.sequence{<}r_k.acknowledgment$.	The attacker retransmits a packet that was already sent and acknowledged. The attacker changes the packet into a RESET TCP packet.
$retrans_2(S,a_i,a'_i)$	$a'_i{=}a_i$ except that: (i) $a'_i.RST$ is set, and (ii) the sequence number in a'_i is too large to fit into the TCP window of the victim. More formally, let $r_j{\in}S$ be the last response before a_i, then $a'_i.sequence{>}r_j.acknowledgment{+}S.window_size$.	The attacker sends a TCP RESET packet that its sequence number is too large to fit into the victim's TCP window.
$retrans_3(S,a_i,a'_i)$	$a'_i{=}a_i$ except that: (i) $a'_i.RST$ is set, (ii) $a'_i.length{=}a_i.length{-}1$, and (iii) $a_i.sequence$ is the sequence number the victim expects to get next.	The attacker retransmits a packet that was already sent but not yet acknowledged. The attacker changes the packet into a RESET TCP packet[a].
$retrans_{4,5,6}(S,a_i,a'_i)$	The same as $retrans_{1,2,3}(S,a_i,a'_i)$ but instead of the RST flag, set the FIN flag.	Simulates FIN eluding attempt rather than a RESET eluding attempt

[a]Called the *ambiguous retransmission problem*, see p. 309 in reference [44].

Table 2: Semantics preserving predicates for TCP retransmission. We focused on a single retransmission of control packets, like RESET and FINISH, which occurs immediately after the original packet without adding a victim responses. Other possibilities are left for future work.

a NIDS to detect an attack depends not only on the attack packets it sees, but also on the interleaving of those packets with the victim's acknowledgments [12, 16, 24, 32, 41]. To accurately represent attack instances, a system that generates such instances must add victim acknowledgments. An easy way to do so is through TCP inference rules because they can be used to add acknowledgments to any TCP-based attack. However, this brings up the question of which acknowledgments to add. In general, a TCP implementation sends an acknowledgment for each packet it receives [44], so our TCP rules usually add an acknowledgment after each packet we add or modify. Other options to add acknowledgments or application specific responses are

possible, but we leave them for future work.

5.2.2 Payload Level Inference Rules

We present payload level rules and their integration into our natural deduction system. In general, two differences exist between the transport rules presented above and payload rules:

1. Payload rules operate only on the $\mathcal{A}_{MaP}$ instance rather than on any other variant of the attack. Since transport level protocols and application level protocols are independent, there should be no difference if we first change the attack payload and then change the way the payload is transmitted, or vice versa. So, it is possible to apply payload, or application, modifications on the $\mathcal{A}_{MaP}$ instance before any transport level modification. There are techniques to attack NIDS that are based on interleaving of transport and payload modifications [12, 36]; while these attacks can be modeled too, we do not address them here.

2. Payload rules are based upon the assumption that the attacker knows the signatures used by the NIDS. Since the attacker's goal is to elude a signature-based NIDS, they must change or hide the signature of the attack. Therefore, we assume that the attacker knows the signature used by the NIDS to detect the attack. We believe that it is a reasonable assumption for two reasons. First, NIDS are commodities, so it is easy to obtain the signatures provided with any NIDS. Second, developing signatures requires intimate knowledge of the network protocol and the attack itself. Since users of a NIDS usually do not have the time or the knowledge to customize the provided signatures, they use them "as is".

Regardless of the application the rules model, we divide them into two general categories. *Obfuscation* rules take the subsequence in the payload that matches the NIDS signature and change it. On the other hand, *padding* rules do not change the subsequence, but hide it among benign semantic-preserving sequences. HTTP can be used to illustrate the difference between the two types of rules. To encode a malicious URL (like "WWW.FOO.COM/SCRIPTS/CMD.EXE" in CVE-2001-0333), an attacker can obfuscate the URL by replacing characters with their equivalent hexadecimal values, or they can pad this request with benign HTTP requests in the same TCP packet. The main insight behind these two types of rules is that obfuscation rules elude a NIDS by exploiting the fact that its signature does not cover all attack instances, while padding rules attack the NIDS pattern matching algorithm rather than the signature it uses.

Type	Name	Description
Obfuscation	HTTP URL Encode (R_8)	Substitute printable characters in a URL with their equivalent ASCII values (was not investigated in this paper).
	HTTP space padding (R_9)	Insert spaces after an HTTP method: changes a signature from <HTTP Method>[SP]^n<URL> into <HTTP Method>[SP]$^{n+1}$<URL>.
Padding	finger padding (R_5)	Add spaces before the username. This is legal according to finger specification [51].
	FTP Padding (R_6)	Add benign semantics-preserving FTP commands before a malicious command. For Snort, one of the malicious commands is a CWD with an argument longer than 100 bytes (Snort Id (sid): 1919 [24]). Representative benign commands that preserve semantics are "CWD /tmp\n" and "LIST", while "QUIT" is benign but does not preserve semantics.
	HTTP Multiple Requests (R_7)	Add benign semantics-preserving HTTP requests before a malicious request. For Snort, an HTTP method followed by a URL that contains the string "perl.exe" is considered malicious (sid: 832 [24]). Benign semantics-preserving requests can be "index.html" without a "Connection: close" option which will turn them into requests that do not preserve semantics.

Table 3: Semantics preserving payload inference rules.

The distinction between the two types of payload rules is appealing for three reasons. First, it helps us to develop rules by dividing the task into two more focused subtasks. Second, it increases the variety of instances our model derives, because any combination of obfuscation and padding rules is possible. Last, it helps us improve the signatures a NIDS uses, as we illustrate below.

In a recent paper, Sommer and Paxson reduced the false positive rate of a NIDS by using a contextual signature that generates an alert only after matching several sub-signatures [41]. The padding rules can

be used to formalize their idea. For example, look at the *ftp-cwd* attack (Figure 1). For this attack, Snort generates an alert when observing a TCP packet containing the string "CWD ahhh...". As a result, Snort will generate an alert for a TCP packet that contains "QUIT\n CWD ahhh...". Unfortunately, this alert is a false positive because the FTP server first processes the "QUIT" terminating the connection. For that reason, no attacker will use "QUIT" to pad the *ftp-cwd* signature; they will only use a subset of the FTP commands that do not alter the attack semantics (e.g., "CWD"). Let us denote the language that the attackers will use for padding as $FTP_{padding}$. Now, if we extend Snort signature for *ftp-cwd* to $X\cdot$"CWD ahhh..." where $X \in FTP_{padding}$, then this false positive will not be generated. The real advantage of $FTP_{padding}$ is its applicability to other FTP attacks; we can use it for other buffer overflows occurring in FTP servers. Hence, it can be defined once and used in many signatures. The procedure we have illustrated can be formalized as a set-constraints problem as we show in Appendix B.

Table 3 presents the payload rules we consider in this paper. Since payload rules are application specific, we focus on three applications: Finger, FTP, and HTTP. For clarity, we provide only the informal description of the rules; the formal description presented next.

5.3 Implementation

We implemented the core of AGENT in Prolog [43]. Prolog is designed for natural deduction; using Prolog, it is easy to represent the $\mathcal{A}_{MaP}$ instance as a ground fact, the inference rules as Prolog rules, and to solve the black and white hat problems using queries. The implementation of AGENT in Prolog is compact enough to be included as part of this paper (Table 6, Appendix C). More importantly, the same Prolog program can be used to solve both the black and white hat problems as we illustrate below.

To solve the black hat problem, we used AGENT to generate $\mathcal{A}_\Phi$. First, we provide Φ—the set of inference rules we want to use—then, we issue the existential query:

derive($\mathcal{A}_{MaP}$*,X).*

which returns a list of all possible variants of $\mathcal{A}_{MaP}$ that the rules in Φ derive. Formally, this query returns $\mathcal{A}_\Phi$. In the next section we show how we connected AGENT to Snort to solve the black hat problem for a particular NIDS (Definition 2).

To solve the white hat problem we used AGENT alone using all rules we have. To determine whether s is an instance of an attack $\mathcal{A}$, we issue the ground query:

derive($\mathcal{A}_{MaP}$*,s).*

Prolog will return yes if and only if $s \in \mathcal{A}_\Phi$, as required by the white hat definition (Definition 3).

6 Finding Attack Instances that Elude Snort

Our goal was to use AGENT to test a real NIDS. Our testing strategy was to use AGENT to generate instances of known attacks and to feed them into Snort—a publicly-available widely-used signature-based NIDS [39]. When Snort missed an instance we stopped and investigated Snort code to find out the reason. We generated instances of three known attacks: (i) *finger-root*, used to gain root sensitive information from a victim (CVE-1999-0612, [30]), (ii) *perl-in-cgi*, used to execute arbitrary commands on a Web server (CAN-1999-0509), and (iii) *ftp-cwd*, a buffer overflow used to gain root access to an FTP server (CAN-2002-0126). To generate the instances of each attack, we used the transformation rules discussed in Section 5.2.

We chose Snort as a target NIDS for several reasons. First, Snort comes with more than 1500 signatures, so it was easy to find the signatures of our chosen attacks. Second, Snort is considered a state-of-the-art NIDS. Snort performance is comparable to commercial NIDS [8, 46], and it seems to be aware of many evasion techniques that were reported in the past [12, 36] and therefore uses techniques such as IP and TCP

Level	Name	Description	**Implications:** enables attackers to find an attack instance that eludes Snort for
Transport	Evasive RST.	A bug in Snort's TCP state tracking. Snort accepts an illegal TCP RESET packet; as a result, Snort stops tracking a live TCP connection (Section A.1.2).	Any TCP-based attack.
Payload	Flushing	Exploits a vulnerability in Snort's TCP reassembly mechanism. Snort misses a signature that is fragmented over several TCP packets (Section A.2.1).	Any attack whose signature can be inflated by a context-based payload rule.
	HTTP space padding	Exploits Snort's default configuration together with its nature to report only a single alert per TCP packet. Snort misses the attack or generates a general alert instead of the *perl-in-cgi* alert (Section A.2.2).	Any Web-CGI attack. With a default configuration, Snort completely misses the attack; with a user-defined configuration, Snort generates a general HTTP alert rather than the specific alert for the attack.
	HTTP multiple requests	Exploits a bug in Snort's HTTP decoding mechanism. Snort does not analyze more than a single HTTP request per TCP packet (Section A.2.2).	Any Web-CGI attack.
	Pattern matching	Exploits a bug in Snort's pattern matching algorithm (Section A.2.3).	Any attack that uses a signature of the form "foo*bar".

Table 4: Summary of Snort bugs found by AGENT.

reassembly, HTTP encoding, and TTL checks. As far as we can tell, Snort uses balanced data structures, so it is not sensitive to algorithmic complexity attack as was shown for another NIDS [7]. Third, since it is maintained regularly and bugs are fixed periodically, we assumed that it would be non-trivial to find instances that elude it.

For each attack we tested, AGENT found instances that eluded Snort. These instances exposed vulnerabilities in different portions of Snort's code: the TCP engine, the HTTP decoder, and the pattern matching mechanism. We reported these vulnerabilities to the Snort development team. Some of the vulnerabilities have been fixed (Snort version 2.0.2) and others will be fixed in the upcoming releases of Snort. Table 4 presents a summary of vulnerabilities our testing effort exposed. For each vulnerability, the table specifies the type of the transformation rules that exposed it, a short vulnerability description, and the vulnerability implications.

Next, we describe the testing environment we built around AGENT and then we present a summary of our testing efforts. The description of the individual instances that eluded Snort and the vulnerabilities they exposed appears in Appendix A.

6.1 NIDS Testing Using AGENT

To test Snort, we used AGENT as a black hat tool: for a given attack $\mathcal{A}$, we tried to find instances that Snort does not detect. In particular, for a given attack $\mathcal{A}$, this testing process contains the following three stages (see Figure 3):

1. **Closure Generation**. This stage generates all instances in $\mathcal{A}_\Phi$. We provided two inputs to AGENT: the attack MaP instance, $\mathcal{A}_{MaP}$, and a collection of transformation rules, Φ. The output of this stage is a text file containing $\mathcal{A}_\Phi$. Each instance in $\mathcal{A}_\Phi$ is represented as a list of TCP packets; as mentioned in Section 5.2.1, instances contain both attack and response packets.

2. **Eluding-Instance Search**. This stage finds an attack instance that eludes Snort. To perform this search, we implement a *instance simulator* that plays the instances in $\mathcal{A}_\Phi$; the simulator writes both the attacker and the victim packets to the network. On the simulator's machine we also installed Snort, which reads from the network. Snort raises an alert each time it identifies $\mathcal{A}$ in a TCP sequence. The search stops when an undetected instance is found, or when all instances have been checked.

   We implemented the simulator using C libraries that enable creation of raw TCP packets [47, 38]. The

simulator plays complete TCP sessions, including TCP handshake and termination procedures and it simulates an average of 350 instances per second on a Pentium III, 850MHz.

3. **Instance Feasibility Check**. This stage illustrates that the instance found by the search stage can be used by attackers over the network. In the previous stage, we simulated both the attacker's packets and the victim's responses. We used two machines connected by a LAN to separate the attacker from the victim. In this stage, we used the instance simulator to send the attacker's packets only, the victim responses were generated by a real application.

   Strictly speaking, this stage is unnecessary. As long as we use sound rules, any attack instance generated by AGENT can be used by any hacker. We included this stage to validate our own methodology and to illustrate that AGENT can find attack instances that can exist in the wild.

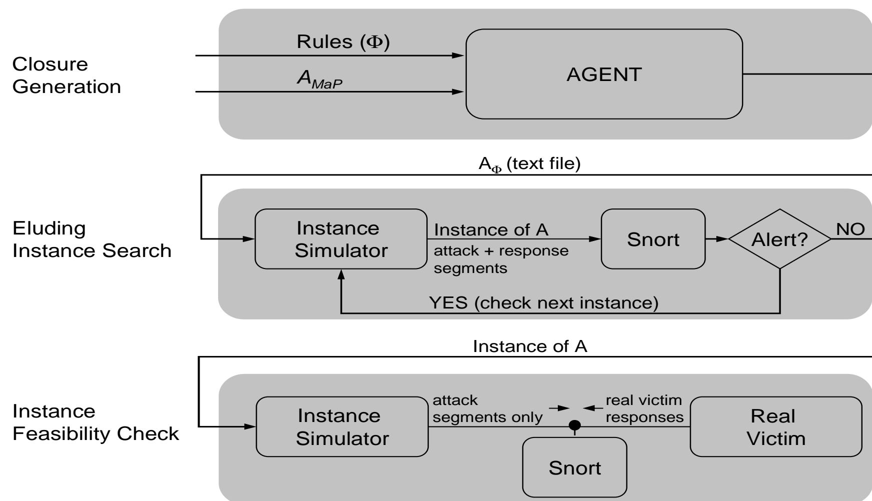


Figure 3: NIDS Testing Using AGENT.

6.2 Testing Effort Summary

Software testing is usually an incremental process. One starts with simple test cases, and gradually adds cases to increase coverage. The ideal goal is to test every possible case. Since this is infeasible, one usually splits the testing into more manageable *phases*; in each phase the goal is to test a particular type of test cases.

Here, we describe this process in the context of AGENT and Snort. We performed a total of seven phases that yielded five vulnerabilities. We started with a simple attack and with a rule set that derived a small number of instances. To increase coverage, in each phase we either added rules to AGENT or changed the attack.

Table 5 presents a summary of our seven test phases. In the first two phases we used *finger-root* with transport rules alone. In the second phase AGENT exposed the Evasive-RST vulnerability. We continued to use the *finger-root* attack, but added the *finger-padding* rule. Using this rule alone did not yield new vulnerabilities (Phase 3), but combining it with transport rules exposed the Flushing vulnerability (Phase 4). We continued with *perl-in-cgi* and each HTTP rule we used exposed a vulnerability in Snort's HTTP decode engine (Phases 5,6). Last, we tested Snort with instances of the *ftp-cwd* attack and discovered the Double-Signature vulnerabilities (Phase 7).

Here is a summary of the lessons we learned from working with AGENT:

1. **Selection of rules**. In our current settings (Figure 3), we can test 350 attack instances per second, or 10^7 instances in about 8 hours. This limitation, together with the desire to increase coverage, propelled the selection of rules in each phase. We composed rule sets that do not derive more than a few millions

	Testing Phases						
	1	**2**	**3**	**4**	**5**	**6**	**7**
Vulnerability name	Frag and Permute	Evasive RST	Finger Padding	Flushing	HTTP Space Padding	HTTP Multiple Request	Double Signature
Tested attack	*finger-root*	*finger-root*	*finger-root*	*finger-root*	*perl-in-cgi*	*perl-in-cgi*	*ftp-cwd*
Described in Section	A.1.1	A.1.1	A.2.1	A.2.1	A.2.2	A.2.2	A.2.3
Rules in Φ[a]	$\{R_1, R_2\}$	$\{R_1, R_2, R_3\}$	$\{R_5\}$	$\{R_1, R_2, R_5\}$	$\{R_9\}$	$\{R_7\}$	$\{R_1, R_6\}$
Instances in $\mathcal{A}_\Phi$	1631	$3,628,960$	25	$6,820,346$	$677,960$	100	$178,585$
$\mathcal{A}_\Phi$ generation time (sec)	0.1	70	< 0.1	180	5	< 0.1	4
% of eluding instances	None	33	None	0.15	> 99	99	23
First eluding instance	None	14	None	$1,037,096$	6	1	2280

[a]See rules description in Tables 1 and 3.

Table 5: Testing effort summary.

of instances (in some cases we slightly changed the rule specifications, see details in Appendix A). Our goal is to improve AGENT, so testing more instances will become more practical.

2. **The advantages of soundness**. Unlike tools that modify an attack in a way that may not preserve its semantics (e.g., [42]), every instance generated by AGENT implements the attack under consideration. This greatly helps in finding attack instances that elude Snort, because no time was wasted on understanding whether a given sequence of TCP packets really implements the attack. This illustrates the usefulness of sound testing tools, and AGENT in particular.

3. **The advantages and disadvantages of completeness**. In Phases 1 and 3 AGENT did not expose any vulnerability. Since AGENT generates all instances with certain properties, in these phases it serves like a verification tool. For example, after Phase 1, we can say that Snort correctly reassembles TCP streams with six characters or less. Similarly, after Phase 3 we can say that the Snort pattern matching algorithm correctly ignores spaces before the attack signature. While these are simple claims, they do provide important information about Snort reliability. To the best of our knowledge, such verification capabilities were not reported in the past in the context of NIDS. We hope that after improving AGENT performance, we will be able to verify more complex properties.

   Completeness has a disadvantage too. In Phase 4, for example, AGENT found the first instance that eluded Snort only after generating more than a million instances. If we compare AGENT to a tool that randomly samples instances out of a set of sound attack instances, the random tool would have found an instance after checking 666 instances (on average). This observation suggests that AGENT and a random tool could complement each other. We leave this investigation for future work.

7 Future Work

There are several directions for future work. We are working to expand our knowledge-base of rules. We are exploring other link, transport, and payload level rules, to model attackers' knowledge. In particular, we intend to model code obfuscation rules that enable attackers to change binary code of network exploits. We also intend to explore ways showing that the rules cover all possible ways to modify an attack.

We envision integrating AGENT into a NIDS development cycle. While AGENT is a powerful testing tool, it can help NIDS developers in other tasks as well. To understand why a stream of packets implements a given attack, AGENT can provide a derivation sequence that shows all transformations used by attackers.

Moreover, developers can use AGENT transformation rules, particularly the payload level rules, to construct better signatures as we illustrated in Section 5.2.2 and in Appendix B.

Last, to improve AGENT capabilities as a NIDS validation tool, we intend to improve AGENT performance using techniques described by the deductive databases community.

References

[1] Digital information society. www.phreak.org.

[2] AIKEN, A. Introduction to set constraint-based program analysis. *Science of Computer Programming* **35**, 1 (1999).

[3] AIKEN, A., AND WIMMERS, E. Solving systems of set constraints. In *The 7th Annual IEEE Symposium on Logic in Computer Science (LICS)* (Santa Cruz, CA, June 1992).

[4] ALLEN, J., CHRISTIE, A., FITHEN, W., MCHUGH, J., PICKEL, J., AND STONER, E. State of the practice of intrusion detection technologies. Tech. Rep. CMU/SEI-99-TR-028, Software Engineering Institute, Carnegie Mellon, Jan. 2000.

[5] BURROWS, M., ABADI, M., AND NEEDHAM, R. A logic of authentication. Tech. Rep. 39, DEC Systems Research Center, Feb. 1989.

[6] CASWELL, B., BEALEAND, J., FOSTER, J. C., AND FAIRCLOTH, J. **Snort 2.0 Intrusion Detection**. Syngress, Feb. 2003.

[7] CROSBY, S., AND WALLACH, D. Denial of service via algorithmic complexity attacks. In *USENIX Security Symposium* (Washington, D.C., Aug. 2003).

[8] DEBAR, H., AND MORIN, B. Evaluation of the diagnostic capabilities of commercial intrusion detection systems. In *International Symp. on Recent Advances in Intrusion Detection (RAID)* (Zurich, Switzerland, Oct. 2002).

[9] DETRISTAN, T., ULENSPIEGEL, T., MALCOM, Y., AND UNDERDUK, M. S. Polymorphic shellcode engine using spectrum analysis. *Phrack Online Magazine* **61** (Aug. 2003).

[10] DIETRICH, S. W. Extension tables: Memo relations in logic programming. In *The Fifth International Conference and Symposium on Logic Programming* (San Francisco, Aug. - Sept. 1987).

[11] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. *RFC 2616 - Hypertext Transfer Protocol*. The Internet Engineering Task Force, June 1999.

[12] HANDLEY, M., AND PAXSON, V. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *USENIX Security Symposium* (Washington, D.C., Aug. 2001).

[13] HEINTZE, N., AND JAFFAR, J. A decision procedure for class of set constraints. In *The 5th Annual IEEE Symposium on Logic in Computer Science (LICS)* (Philadelphia, PA, June 1990).

[14] HOPCROFT, J., MOTWANI, R., AND ULLMAN, J. **Introduction to Automata Theory, Languages, and Computation**. Addison-Wesley, 2001.

[15] KINDRED, D., AND WING, J. M. Fast, automatic checking of security protocols. In *USENIX 2nd Workshop on Electronic Commerce* (Pittsburgh, PA, Nov. 1996).

[16] KRUEGEL, C., VALEUR, F., VIGNA, G., AND KEMMERER, R. A. Stateful intrusion detection for high-speed networks. In *IEEE Symp. on Security and Privacy* (Berkeley, CA, May 2002).

[17] KYMIE M.C. TAN, K. S. K., AND MAXION, R. A. Undermining an anomaly-based intrusion detection system using common exploits. In *International Symp. on Recent Advances in Intrusion Detection (RAID)* (Zurich, Switzerland, Oct. 2002).

[18] LEE, W., CABRERA, J. B. D., THOMAS, A., BALWALLI, N., SALUJA, S., AND ZHANG, Y. Performance adaptation in real-time intrusion detection systems. In *International Symp. on Recent Advances in Intrusion Detection (RAID)* (Zurich, Switzerland, Oct. 2002).

[19] LIPPMANN, R., HAINES, J. W., FRIED, D. J., KORBA, J., AND DAS, K. Analysis and results of the 1999 DARPA off-line intrusion detection evaluation. In *International Symp. on Recent Advances in Intrusion Detection (RAID)* (Toulouse, France, Oct. 2000).

[20] LIPPMANN, R. P., FRIED, D. J., GRAF, I., HAINES, J. W., KENDALL, K. R., MCCLUNG, D., WEBER, D., WEBSTER, S. E., WYSCHOGROD, D., CUNNINGHAM, R. K., AND ZISSMAN, M. A. Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In *DARPA Information Survivability Conference and Exposition* (Hilton Head, SC, Jan. 2000).

[21] LOWE, G. Casper: A compiler for the analysis of security protocols. In *IEEE Symp. on Security and Privacy* (Oakland, CA, May 1997).

[22] LUNT, T. Automated audit trail analysis and intrusion detection: A survey. In *The 11th National Computer Security Conference* (Baltimore, MD, Oct. 1988).

[23] MARRERO, W., CLARKE, E. M., AND JHA, S. A model checker for authentication protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols* (Piscataway, NJ, Sept. 1997).

[24] MARTIN ROESCH. Snort: the Open Source Network Intrusion Detection System. http://www.snort.org.

[25] MCAULIFFE, N., SCHAEFER, L., WOLCOTT, D., HALEY, T., KALEM, N., AND HUBBARD, B. Is your computer being misused? In *The Sixth Computer Security Applications Conference* (Tucson, AZ, Dec. 1990).

[26] MCHUGH, J. Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory. *ACM Transactions on Information and System Security* **3**, 4 (Nov. 2000).

[27] MEADOWS, C. Formal verification of cryptographic protocols: A survey. In *4th International Conference on the Theory and Applications of Cryptology (Asiacrypt)* (Wollongong, Australia, Nov. 1994).

[28] MEADOWS, C. A model of computation for the NRL protocol analyzer. In *IEEE Computer Security Foundations Workshop* (June 1994).

[29] MITCHELL, J. C., MITCHELL, M., AND STERN, U. Automated analysis of cryptographic protocols using murϕ. In *IEEE Symp. on Security and Privacy* (Oakland, CA, May 1997).

[30] MITRE CORPORATION. CVE: Common Vulnerabilities and Exposures. http://www.cve.mitre.org.

[31] PAULSON, L. Mechanized proofs of security protocols: Needham-schroeder with public keys. Tech. Rep. 413, University of Cambridge Computer Laboratory, 1997.

[32] PAXSON, V. Bro: a system for detecting network intruders in real-time. *Computer Networks* **31**, 23–24 (Dec. 1999).

[33] POSTEL, J. *RFC 793 - Transmission Control Protocol*. The Internet Engineering Task Force, Sept. 1981.

[34] POSTEL, J., AND REYNOLDS, J. *RFC 959 - File Transfer Protocol*. The Internet Engineering Task Force, Oct. 1985.

[35] PRAWITZ, D. **Natural Deduction: a Proof-Theoretical Study**. Almquist and Wiskell, 1965.

[36] PTACEK, T. H., AND NEWSHAM, T. N. Insertion, evasion, and denial of service: Eluding network intrusion detection. Tech. Rep. T2R-0Y6, Secure Networks, Inc., Calgary, Alberta, Canada, 1998.

[37] RAMAKRISHNAN, R., SRIVASTAVA, D., AND SUDARSHAN, S. Efficient bottom-up evaluation of logic programs. In **Computer Systems and Software Engineering: State-Of-The-Art**. Kluwer Academic Publishers, June 1992.

[38] SCHIFFMAN, M. D. Libnet: A C library for portable packet creation and injection. http://www.packetfactory.net/libnet.

[39] SECURITY ADMINISTRATOR NEWSTELLER. Instant poll, October 2002. www.windowsitsecurity.com.

[40] SHMATIKOV, V., AND STERN, U. Efficient finite-state analysis for large security protocols. In *IEEE Computer Security Foundations Workshop* (June 1998).

[41] SOMMER, R., AND PAXSON, V. Enhancing byte-level network intrusion detection signatures with context. In *ACM conference on Computer and Communications Security* (Washington, DC, Oct. 2003).

[42] SONG, D. Fragroute: a TCP/IP fragmenter. http://www.monkey.org/ dugsong/fragroute/.

[43] STERLING, L., AND SHAPIRO, E. **The Art of Prolog**. The MIT Press, Cambridge, MA, USA, 1994.

[44] STEVENS, W. R. **TCP/IP Illustrated**, vol. 1. Addison Wesley, 1994.

[45] TAN, K., MCHUGH, J., AND KILLOURHY, K. Hiding intrusions: From the abnormal to the normal and beyond. In *The 5th International Workshop on Information Hiding* (Noordwijkerhout, Netherlands, Oct. 2002).

[46] THE NSS GROUP. Intrusion detection systems (IDS) group test (Edition 4), 2003. www.nss.co.uk/ids/edition4/index.htm.

[47] THE TCPDUMP GROUP. Libpcap: Packet capture library. www.tcpdump.org.

[48] VIELLE, L. Recursive axioms in deductive databases. In *First International Conference on Expert Database Systems* (Oct. 1986).

[49] WAGNER, D., AND SOTO, P. Mimicry attacks on host-based intrusion detection systems. In *ACM conference on Computer and Communications Security* (Washington, DC, Nov. 2002).

[50] WOO, T. Y. C., AND LAM, S. S. A semantic model for authentication protocols. In *IEEE Symp. on Security and Privacy* (Oakland, CA, May 1993).

[51] ZIMMERMAN, D. *RFC 1288 - The Finger User Information Protocol*. The Internet Engineering Task Force, Dec. 1991.

[52] ZIMMERMANN, H. OSI reference model - the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communication* **28**, 4 (Apr. 1980).

A Description of Eluding Instances

A.1 Eluding Snort Using Transport Level Rules

In the first experiments, we focused on transport level transformations. Since these transformations affect any TCP-based attack, we choose for testing the *finger-root* attack which is the simplest TCP-based attack we could find.

finger-root is an information-leak attack where the attacker uses the finger service [34] to find the last time root logged into the host. Since applying security patches typically requires root login, hosts that root has not logged into for a long period of time are more likely to be vulnerable. The finger-root$_{MaP}$ sequence contains a single packet with six characters: the string "root" and two characters (carriage return and line feed) used as an end-of-message marker for the finger server.

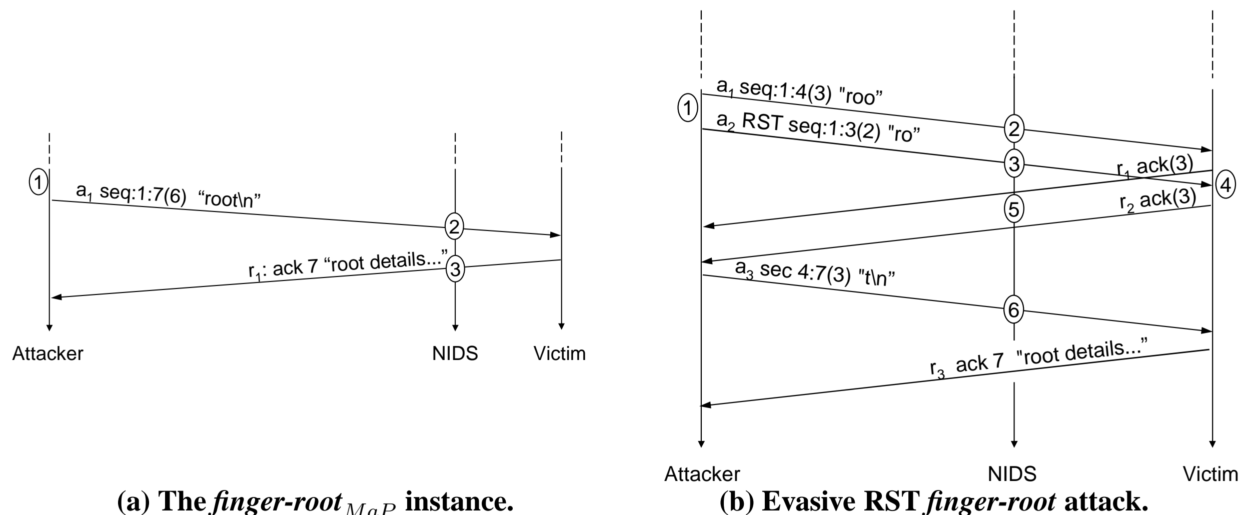
A.1.1 Frag-and-Permute

One of the earliest transport transformations documented in the literature targets the NIDS TCP reassembly mechanism [36]. In the fragment and permute transformation, the attacker first fragments the attack and then permutes the fragments. Since the NIDS observes the permuted attack, if the NIDS TCP reassembly engine is malfunctioning, then the NIDS may miss some of the attack instances. Since Snort performs TCP reassembly, it should be robust against such attacks.

Snort correctly identifies all instances of the frag-and-permute *finger-root* attack. We conclude that, at least for short attacks, Snort reassembly mechanism works correctly.

A.1.2 Evasive RST Injection

We added the rules for semantically preserving retransmissions of RESET packets (Table 2). Since legitimate RST (reset) packets cause a TCP connection to terminate, the purpose of an evasive RST packet is to convince the NIDS that the attacker terminates the connection while the connection was not truly terminated. When this happens, the NIDS stops tracking the connection while the attacker and victim continue to accept and respond to messages.



(a) The *finger-root*$_{MaP}$ instance. **(b) Evasive RST *finger-root* attack.**

Figure 4: TCP evasive RST attack.

Evasive RST Description. To better understand the evasive RST bug, we first demonstrate Snort behavior on the *finger-root*$_{MaP}$ instance that does not contain an evasive RST (Figure 4a): (1) The attacker sends a_1 containing the string "root\n". (2) To avoid accepting evasive RST packets, Snort first verifies that a_1 fits into its image of the victim's TCP window, then it applies the pattern matching algorithm on a_1, and

finally it generates a *finger-root* alert because a_1 matches Snort *finger-root* signature. (3) Snort observes the victim's response (r_1) and updates its image of the victim's TCP window: it changes the boundaries of the window so any packet that ends with a sequence number smaller than 7 will be (justifiably) ignored.

Figure 4b describes an instance of a *finger-root* attack that include an evasive RST. (1) The attacker sends a packet containing the string "roo" (a_1). The attacker immediately retransmit a_1, but in the retransmitted packet (a_2) the payload size is smaller, and the RST flag is set. (2) Snort verifies that a_1 fits into (its image of) the victim's TCP window and applies the pattern matching algorithm. Since there is no match, no alert is generated. (3) Since the victim's window was not updated yet (it is updated only after victim responses), Snort considers a_2 valid and acts according to RST semantics: it stops tracking this TCP session and deletes all data associated with this connection. (4) The victim receives a_1 before a_2, so the victim rejects a_2 because it is out of its TCP window (due to a_1). As a result, the victim does not terminate the connection. (5) Snort observes r_1 and r_2. Since the connection was terminated, Snort initializes a new connection and start following it. (6) Snort validates and accepts a_3. Since a_3 does not contains "root", and since a_1 was already deleted from memory, Snort misses the signature "root".

What went wrong? At point (3) in Figure 4b, there is no way to know whether the RST packet will be accepted or rejected (*ambiguous retransmission problem*, see p. 309 in [44]). Hence, Snort concludes that the connection was terminated was done too early and wrongly. Interestingly, when we analyzed Snort code, it was clear that the developers made an effort to validate RST packets, but missed this corner case.

How did AGENT find it? AGENT used set of transport rules to generate all possible evasive RST cases (Table 2). In this case, the size *finger-root*$_\Phi$ is large, more than 3.6 million instances. Still, AGENT generates all attack variants in this case. Since 2% of the variants exposed the RST bug in Snort, and, more importantly, since even short sequences with a single RST exposes it, the bug was exposed in the first 1000 attack instances checked.

Remedy. One way to solve this problem is to defer handling a RST packet until it is clear whether the packet has been accepted. This solution was proposed by Handley and Paxson [12], but it complicates the TCP state tracking in Snort. Instead, after we reported the bug, Snort developers issued a fix (in version 2.0.2) in which Snort does not terminate a connection when it observes a RST packet. Snort waits until the connection is idle for a certain amount of time and then flushes the connection out of memory.

A.2 Eluding Snort Using Payload Level Rules

Since other variants of transport transformations had already been investigated [12, 36], the rest of our experiments focus on payload level transformations. We do not claim that Snort is robust against other transport level modifications, but we leave investigating those for future work.

A.2.1 Flushing: A General Payload Attack

We first focus on the simple *finger-root* attack. To find payload rules for the finger service we reviewed the finger specification [51]. We observed that there are two options to change the payload of a finger query: to add spaces before the username, and to include escape characters, such as backspace, in the username. Since the finger daemon in our experiments does not support escape characters (BSD-finger version 0.17), we only used the first option, the *finger-padding* rule (Table 3).

When using *finger-padding* alone, AGENT did not find any undetected instance. However, adding the *TCP-Fragmentation* rules (Table 1) exposed the flushing vulnerability.

Flushing Description. As previously mentioned, Snort performs TCP reassembly to avoid simple frag-and-permute attacks. Since attack signatures can be fragmented over several TCP packets that can arrive out-of-order, Snort buffers the packets data. Once in a while, using a pseudo-random method, Snort flushes the data buffer: it reassembles the data, checks the data for matching signatures, and deletes the data from

memory.

AGENT found an attack instance the eluded Snort by exploiting this mechanism (Figure 5): (1) the attacker sends a long packet (a_1) that contains 258 spaces followed by the string "ro". Snort observes a_1 and buffers it. (2) When Snort observes that the victim received a_1, it flushes its TCP data buffer because the buffer size is larger than a given threshold. Since "root" is not part of the buffer, Snort does not generate any alert. (3) When Snort observes a_2, its reassembly mechanism cannot reassemble the full "root" because a_1 was already deleted from memory.

What went wrong? The fundamental problem is that Snort reapplied the pattern matching algorithm separately on each data packet and independently from previous packets. Buffering of data, checking the data for matching signatures at random points in time, and deleting the data after checking, reduced the probability of a signature being split across checks but did not completely prevent such a case. Furthermore, AGENT found that the implementation of random flushing in Snort was not effective. Even though flushing points were randomly selected, they were never larger than 260 bytes, so the reassembly buffer was always flushed when its size exceeded 260 bytes. If attackers could inflate the attack payload by 260 bytes, they would always be able to split the signature between two independent applications of the pattern matching algorithm[3].

How did AGENT find it? AGENT generated all possible instances from the *finger-padding* and *TCP-Fragmentation* rules that are shorter than 500 bytes and contain up to three TCP packets. In this case, *finger-root*$_\Phi$ contains 6.8×10^6 instances and the first eluding instance was found after less than an hour.

Remedy. The pattern matching module should continuously monitor the stream of data, and should not back off to its initial state after each packet it receives. Bro, which is a NIDS developed for research purposes, adopts this approach [32].

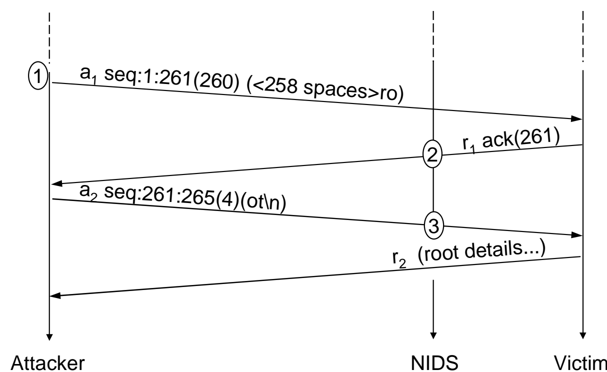


Figure 5: *finger-root* flushing attack.

A.2.2 Eluding Snort using HTTP transformations

Our next goal was to find instances that elude Snort for an attack more serious than *finger-root*. We choose the *perl-in-cgi* attack in which the attacker tries to force a Web server to execute a PERL script on their behalf. When the Web server is mis-configured, the attacker script run under the privileges of the Web server, usually root, so the attacker can execute arbitrary commands on the server.

This attack uses HTTP, which is very common among attackers; 39% of Snort rules target HTTP communication. To identify *perl-in-cgi*, Snort uses the signature "GET*/perl.exe" (sid #832 [24])[4]. We have

[3]Just before the submission of this paper, this vulnerability was reported by Sommer and Paxson [41]. However, they did not show that it can always be exploited.

[4]The "GET" is not part of the signature, but Snort checks for its existence as part of its HTTP decoding

developed three payload rules for HTTP: *HTTP multiple requests*, *HTTP space padding*, and *HTTP URL encoding* (Table 3). We have investigated only the first two; the third is left for future work.

We applied the above rules and found two type of instances of *perl-in-cgi* that Snort did not detect.

1. **HTTP space padding.** More than 8 spaces after the "GET" cause Snort to miss the attack when Snort uses its default configuration, and report a "Large HTTP method" alert instead of the *perl-in-cgi* alert after we have modified the configuration as we describe below.

   What went wrong? The fundamental problem is that Snort reports only a single alert per TCP packet. In this case, Snort identifies an abnormally large HTTP method ("GET" + eight spaces), so it generates a "Large HTTP Method" and does not continue to check for other alerts. However, the "Large HTTP Method" is generated only if Snort's *internal_alerts* flag is set. Unfortunately, the *internal_alerts* flag is unset by default, so under the default configuration no alert is generated.

   To the best of our knowledge, there is no description of this flag anywhere in Snort distribution, though a recently published book describes it [6]; one learns about the existence of this flag only by browsing the code.

   How did AGENT find it? AGENT uses the *HTTP-space-padding* to add spaces between the "GET" and "/perl.exe"; it added 5 spaces at a time, up to 250 spaces. We also used TCP-fragmentation rule limited to three packets in each instance (TCP-fragmentation was not necessary for this attack). $perl\text{-}in\text{-}cgi_{\Phi}$ contains 2.7×10^3 instances and more than 99% of them eluded Snort.

   Remedy. The fix for the default configuration is easy. It would be helpful if all options supported by Snort would be documented (although undocumented options are common among publicly available tools). The situation where one alert hides a more meaningful one is common in NIDS [8], but we are not aware of any systematic solution for this problem.

2. **Multiple HTTP Requests.** Snort does not detect a malicious HTTP request that is placed after a benign request in the same TCP packet.

   What went wrong? Snort's HTTP decode engine, which decodes hexadecimal values in a URL into printable characters, decodes only the first HTTP request in each TCP packet. Interestingly, the software interface to the decoding engine permits more than a single HTTP request. This indicates that Snort developers were aware of the possibility to have several HTTP requests in a single TCP packet, but this functionality was not implemented yet.

   How AGENT we find it? AGENT uses the *HTTP Multiple Requests* rule. The second instance AGENT generated eluded Snort.

   Remedy. Enables Snort to handle more than one HTTP request in a single TCP packet.

A.2.3 Double Signature: Another General Payload Attack

Our last goal was to find an instance of the *ftp-cwd* attack that eludes Snort. This goal is challenging because Snort has two rules that it uses to detect the attack.

1. **CWD rule (sid: #1919 [24])**. According to Snort documentation [6], this rule should trigger an alert if an end-of-line character, ("\n"), is not found within 100 characters after a CWD command. However, in the current Snort implementation, this rule triggers an alert if an end-of-line character is not found within 100 characters after a CWD command, and before the end of the TCP packet. In other words, a TCP packet that contains "CWD" must also contain a "\n" somewhere after the "CWD". One might consider such behavior overly strict because it increase Snort detection sensitivity: the attacker must include both "CWD" and "\n" in the same packet, so it limits the attacker's ability to find *ftp-cwd* instances that elude Snort.

2. **Large packet rule (sid:#1748 [24])**. Since FTP commands are usually short, the packets that flow into the FTP server are also short. This rule triggers an alert if a TCP packet that flows into the FTP server is larger than 100 bytes.

Given this situation, we had no choice but to let AGENT to perform an exhaustive search for a new *ftp-cwd* variant that eludes Snort. AGENT exposed the regular expression attack described below.

Double signature description. Figure 6 presents an instance of the *ftp-cwd* that eluded Snort. (1) After login, the attacker sends a packet containing two FTP commands: an innocent CWD command ("CWD /tmp\n") and the malicious CWD command. (2) Snort applies pattern matching. Due to a bug in the pattern matching algorithm, it identifies the innocent "CWD /tmp\n" but misses the beginning of the buffer overflow ("CWD aa...") even though the second CWD command violates the CWD rule mentioned above. (3) Even after reassembly, due to the same bug, Snort misses the malicious CWD command.

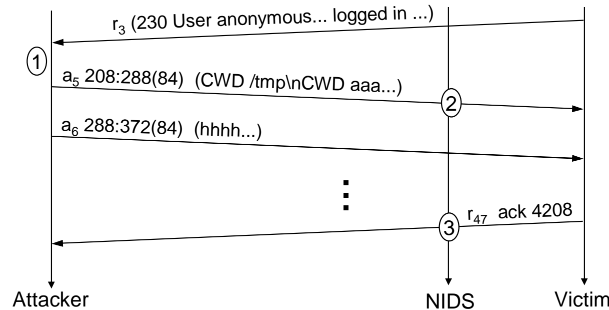


Figure 6: FTP-CWD Double Signature Attack.

What went wrong? The pattern matching algorithm does not correctly handle signatures from the type "foo*bar". The algorithm fails to recognize this pattern in a string like "foo_JUNK_rab_foo_JUNK_bar". After analyzing the prefix "foo_JUNK_rab" the algorithm incorrectly concludes that the string does not contain the pattern "foo*bar".

Similarly, in the case of Figure 6, the algorithm failed to identify the pattern "CWD"$\cdot$(¬'\n')^{100} when it appears after the pattern "CWD*\n".

How did AGENT find it? AGENT uses *FTP-padding* and *TCP-Fragmentation* rules. *ftp-cwd*$_\Phi$ contains 179×10^3 instances and 23% of them eluded Snort. The first eluding instance was found immediately.

Remedy. Use a good library for regular expression matching.

B Using Set Constraints to Generate Signatures

Recall that an attacker can transform a malicious HTTP request by encoding the URL, adding spaces between the HTTP method and the URL, and adding a sequence of innocent HTTP requests before the malicious one. These transformations can be modeled as set constraints [2, 3, 13] and solved using standard techniques from the literature. The solution to these set constraints can be used as signatures to detect malicious payloads in NIDS. We plan to explore this avenue in the future. However, we explain this idea using the HTTP example. Let $\{b_1, \cdots, b_k\}$ be the set of malicious URLs and h be the substitution that corresponds to encoding printable characters in a URL with their equivalent ASCII values. Recall that regular languages are preserved under substitutions [14]. Let *badURL* and *goodURL* be the set variables corresponding to malicious and innocent URLs, respectively. The transformations that an attacker can make to a URL can be formulated as follows:

$$\{b_1, \cdots, b_k\} \subseteq \mathit{badURL} \quad (1)$$

$$h(\mathit{badURL}) \subseteq \mathit{badURL} \quad (2)$$

$$L_{\mathit{URL}} \cap (\neg \mathit{badURL}) \subseteq \mathit{goodURL} \quad (3)$$

The language L_{URL} is the regular expression for all valid URL names according to the standard [11]. The set constraints given above can be solved using standard techniques from the literature. However, in this special case we can obtain the following solution:

$$h(\{b_1, \cdots, b_k\}) = \mathit{badURL}$$

$$L_{\mathit{URL}} \cap \neg(h(\{b_1, \cdots, b_k\})) = \mathit{goodURL}$$

Using the regular expressions for *badURL* and *goodURL*, we can derive regular expressions for malicious HTTP requests and use them as signatures in a NIDS.

C Prolog Implementation

Description	Prolog Implementation
AGENT's main predicate. $\mathcal{A}'$ is a variant of $\mathcal{A}_{MaP}$. The main predicate is based on the observation that payload rules can be applied before transport rules (Section 5.2.2).	$derive(\mathcal{A}_{MaP}, \mathcal{A}') \leftarrow$ $apply_payload(\mathcal{A}_{MaP}, \mathcal{A}'_{MaP})$, $apply_transport(\mathcal{A}'_{MaP}, \mathcal{A}')$.
Payload rules application. Here, the predicate contains only a single finger-padding rule which inserts between 0 to 250 spaces before the username in a finger query. To support other attacks, the finger-padding rule should be replaced with the specific payload rules.	$apply_payload([P, S, F], [P', S, F]) \leftarrow$ $finger_space_pad(P, 250, P')$.
Transport rules application. The predicate fragments the input, then permutes it, and last it adds retransmission packets. For ease of presentation, the implementation here is slightly different then the specification given in Table 1. In Table 1, permutation is done on a stream that contains retransmitted packets, but here it is done without them. However, all vulnerabilities reported in Section 6, can be found using this implementation.	$apply_transport(IN, OUT) \leftarrow$ $tcp_frag(IN, TMP1)$, $tcp_permute(TMP1, TMP2)$, $tcp_retrans(TMP2, OUT)$,
Insert between 0 and L spaces before the payload P.	$finger_space_pad(P, L, P') \leftarrow$ $between(0, L, X)$, $spacelist(X, SP)$, $append(SP, P, P')$.
A recursive predicate that performs TCP fragmentation. The first packet ($[P, S, F]$) is fragmented into two parts, $P1$ and $P2$ ($frag_packet([P, S, F], [P1, P2])$). For each possible fragmentation of P, $P2$ is pushed in front of the stream tail ($push_front(P2, TL, NTL)$). Then the new tail is fragmented recursively ($tcp_frag(NTL, S1)$). Last, for each possible fragmentation of the new tail, $P1$ is pushed in the front ($push_front(P1, S1, L)$).	$tcp_frag([], []) \leftarrow true$. $tcp_frag([[P, S, F]\|TL], L) \leftarrow$ $frag_packet([P, S, F], [P1, P2])$, $push_front(P2, TL, NTL)$, $tcp_frag(NTL, S1)$, $push_front(P1, S1, L)$.
Permutation of a TCP stream. Uses built-in Prolog permutation predicate.	$tcp_permute(IN, OUT) \leftarrow$ $permutation(IN, OUT)$.
A recursive predicate that performs TCP retransmission. The $retrans_packet$ predicate returns a stream ($S1$) containing the first packet ($[P, S, F]$) and its retransmitted version. Then the predicate is applied recursively on the tail of the original stream, For each new tail, $S1$ is push in the front.	$tcp_retrans([], []) \leftarrow true$. $tcp_retrans([[P, S, F]\|TL], OUT) \leftarrow$ $retrans_packet([P, S, F], S1)$, $tcp_retrans(TL, S2)$, $append(S1, S2, OUT)$.
Fragmentation of a single packet. The output is a stream of two packets, $[P1, S, F]$ and $[P2, S2, F]$, which are the fragmentation of the input packet, $[P, S, F]$. First, the payload is fragmented into two parts ($append(P1, P2, P)$).$P2$ may be empty, so $P1$ holds the original payload. Then, the sequence number of $P2$ is fixed ($plus(L, S, S2)$). For ease of presentation, the current implementation does not show the adding of an acknowledgment after P1 as specified in Table 1.	$frag_packet([P, S, F],$ $[[P1, S, F], [P2, S2, F]]) \leftarrow$ $append(P1, P2, P)$, $not_empty(P1)$, $length(P1, L)$, $plus(S, L, S2)$.

Table 6: The core of AGENT implementation. Due to space constraints we do not show the implementation for all rules discussed in this paper.