Improving Signature Testing Through Dynamic Data Flow Analysis

Christopher Kruegel
Technical University Vienna
chris@auto.tuwien.ac.at

Davide Balzarotti, William Robertson, Giovanni Vigna
University of California, Santa Barbara
balzarot,wkr,vigna@cs.ucsb.edu

Abstract

The effectiveness and precision of network-based intrusion detection signatures can be evaluated either by direct analysis of the signatures (if they are available) or by using black-box testing (if the system is closed-source). Recently, several techniques have been proposed to generate test cases by automatically deriving variations (or mutations) of attacks. Even though these techniques have been useful in identifying "blind spots" in the signatures of closed-source, network-based intrusion detection systems, the generation of test cases is performed in a random, unguided fashion. The reason is that there is no information available about the signatures to be tested. As a result, identifying a test case that is able to evade detection is difficult.

In this paper, we propose a novel approach to drive the generation of test cases by using the information gathered by analyzing the dynamic behavior of the intrusion detection system. Our approach applies dynamic data flow analysis techniques to the intrusion detection system to identify which parts of a network stream are used to detect an attack and how these parts are matched by a signature. The result of our analysis is a set of constraints that is used to guide the black-box testing process, so that the mutations are applied to only those parts of the attack that are relevant for detection. By doing this, we are able to perform a more focused generation of the test cases and improve the process of identifying an attack variation that evades detection.

1. Introduction

Intrusion detection systems (IDSs) can be broadly divided into two classes: those that rely on models of normal behavior and detect deviations from these models (i.e., anomaly-based systems), and those that contain descriptions of malicious behavior and detect events (or sequences of events) that match these descriptions (i.e., signature-based systems). While both classes of intrusion detection systems have complementary strengths, they are both vulnerable to evasion attacks.

In the case of anomaly-based systems, evasion techniques are used to craft an exploit so that it resembles normal behavior. The application of these techniques is usually called a *mimicry attack* [30]. In the case of signature-based systems, evasion techniques are used to modify an exploit so that it does not match any of the signatures used by the intrusion detection system, while retaining the ability to compromise the security of the target system [20].

Recently, a number of approaches [4, 13, 16, 18, 22, 23, 29] have been proposed to test the effectiveness and precision of network-based intrusion detection systems. In particular, approaches based on the generation of test cases by automatically deriving variations (or mutations) of known exploits have been shown to be able to identify problems in the detection mechanisms used by both open-source and commercial, state-of-the-art systems [13, 29]. These approaches leverage a number of transformations, called "mutant operators", that are applied to an exploit template. The goal of applying these mutation operators is to obtain a modified version that has a different network manifestation with respect to the original attack, but it is still able to compromise a vulnerable target.

Mutant operators can work at different levels of abstraction (e.g., at the network level or at the application level), and they can be composed and/or applied multiple times. For example, consider a first mutant operator that adds effect-free commands to an FTP session (e.g., adds a "`CWD .`" or a "`NOOP`" command) and a second one that applies fragmentation to the IP traffic. The first operator can be applied multiple times to an FTP-based exploit template without invalidating the attack (unless, of course, the length of the session affects the success of the exploit), while the second one can be applied in different ways (e.g., by specifying different fragment sizes). Thus, the number of possible variations of the original exploit that can be used as test cases quickly grows very large.

In current approaches, the generation of test cases is either manually guided or a random process. In the former case, a human expert selects which operators to apply to the exploit template and which parameters to use for each operator. The results obtained by running the selected test cases might provide hints on how to select the operators and their parameters in the next round of tests. In the latter case, the operators (and the values of their parameters) are selected randomly. Both these approaches are less than optimal because they either require extensive expert knowledge or represent "shots in the dark." Therefore, there is the need for a new technique for testing network-based signatures that is both automated and more focused than a purely random approach. In theory, some guidance about how to generate the relevant test cases can be derived from the signatures themselves. For example, by looking at which features of the network traffic are analyzed by a signature, it is possible to focus the test case generation by using only the mutant operators that affect those features. Unfortunately, most intrusion detection system vendors do not make their signatures available because they consider them to be their intellectual property and an advantage with respect to their competitors. Thus, in general, one cannot rely on the availability of the signatures to guide the generation of the test cases.

To address this problem, we propose a novel approach to drive the generation of test cases based on the analysis of the dynamic behavior of a network-based intrusion detection system. As a first step, we apply dynamic data flow analysis techniques to the NIDS binary to determine which parts of the attack trace are checked by the NIDS. We then leverage this information to restrict the test case generation process to only use the mutant operators that modify the relevant parts of the attack.

Based on the knowledge of *which* parts of a network trace are considered by the detection process, we further refine our analysis to also take into account *how* these parts are used. For simple checks (e.g., the comparison of a source port number with an integer constant), the constant value specified by the signature is extracted from the dynamic trace. Most of the signatures also specify strings or regular expression to be matched against the packet payload. To address these cases, we developed a technique that aims at reconstructing a finite state machine that captures the behavior of the pattern matching process. That is, the state machine derived from the analysis should accept an input string if and only if this string matches a pattern specified by the signature. While it might not always be possible to precisely reconstruct this state machine (particularly in the case of regular expressions), patterns can be reconstructed by observing the execution of popular string matching algorithms such as Boyer-Moore [5] or Aho-Corasick [1].

The contributions of this paper are the following:

- We present a novel, practical technique to effectively drive the generation of test cases for the evaluation of network-based signatures. Our technique analyzes the dynamic behavior of a NIDS program to determine which parts of an attack are used by the detection process.
- In addition to locating the parts of the attack traffic that are used in the detection process, we also determine the nature of the checks that the NIDS performs. In particular, our analysis can automatically extract both specific numerical values and strings that the NIDS is searching for.
- We have developed a prototype tool to evaluate our technique on both open-source and closed-source commercial NIDSs. The results demonstrate that our approach allows for effective generation of exploit mutants that are able to avoid detection, even when no signature information is available.

The remainder of the paper is structured as follows. Section 2 presents the dynamic analysis technique utilized to analyze the behavior of the IDS being tested. Section 3 introduces our mechanisms to extract signature constraints from the observed behavior. Section 4 explains how the analysis results can be used to generate the test cases for a network-based signature. Section 5 evaluates the effectiveness of our approach. Section 6 discusses related work. Finally, Section 7 draws conclusions and outlines future work.

2. Dynamic Data Flow Analysis

The goal of our dynamic data flow analysis is to determine which parts of a network stream are used by the intrusion detection system to identify the presence of an attack. More precisely, we are interested in the positions of all values, or bytes, that are analyzed by the IDS during the detection process.

To determine the input bytes that affect detection, we dynamically monitor the intrusion detection sensor while it is processing the network data. In particular, we tag each input byte that is introduced into the address space of the IDS process with a unique label. This label establishes a relationship between a particular input byte and a location in memory. Then, we keep track of each labeled value as the sensor's execution progresses. To this end, the output of every instruction that uses a labeled value as input is tagged with the same label as well. For example, consider the case of a data transfer operation that loads a value with the label "`123`" from memory into a register. After the instruction is executed, the contents of the target register is also labeled with "`123`". Clearly, it is possible that the result of an operation depends on more than one input byte. For example, consider an operation that adds together two values, each of which is tagged with a different label. In this case, the result is tagged with a set that holds both labels (called a *label set*).

Machine instructions typically read one or more data values from registers or memory locations that are specified by their source operands. These values are then processed and a result is written to the location specified by the destination operand. For example, move operations (e.g., `mov`), arithmetic instructions (e.g., `add`), logic operations (e.g., `and`), and stack manipulation operations (e.g., `push`, `pop`) all belong to this class. For these instructions, the label set that is assigned to the result of the operation is the union of the label sets of all the operation's operands. Propagating label information by tracking the use of input bytes as source (and destination) operands results in an analysis that is very similar to the propagation of taint values in Perl or as implemented by TaintCheck [19] and related approaches [6, 7, 8]. That is, for every instruction that is executed, we can determine whether there exists a *direct dependency* of the value of one or more of its operands on certain input bytes.

In addition to directly influencing operand values by labeled input, input bytes can also have an indirect effect on an instruction's operands. More precisely, the value of a memory operand can also depend on the value of an input byte if this byte is used during the operand's address calculation. That is, when a labeled value is used to determine the location from which a certain value is loaded, the outcome of the load operation depends not only on the loaded value itself (a direct dependency) but also on the memory address where this value is taken from. This is called an *indirect* or *address dependency*. Therefore, when a value is loaded from memory location L, we perform the union of the label set of the value at location L with the label sets of all values that are used to determine the address L.

A typical example for an indirect dependency is the use of labeled data as an index into a table. In this case, the result of a table lookup does not directly depend on the input value, but it is indirectly influenced by the selection of the respective table element. It is important that indirect dependencies are tracked as well. For example, the simple transformation of a string contained in the payload of a network packet into its uppercase representation (e.g., using the `toupper()` function) would break the dependencies between the resulting string and the original labeled input if only direct dependencies were taken into account. The reason is that `toupper()` relies on a table that stores the mappings of all 255 possible input characters to their corresponding uppercase representations.

Our data labeling mechanism is used as a basis to identify all input bytes that can influence the detection process. A byte of the network stream is considered to be involved in the detection process if it has an influence on the IDS' control flow. More precisely, the control flow is influenced by an input byte whenever the outcome of a conditional branch or the target of an indirect control transfer instruction (i.e., an indirect `call` or `jmp` instruction) depends on that byte. The influence of input bytes on control flow decisions can be determined in a straightforward fashion using the propagation of label sets during program execution. To this end, whenever a labeled operand is used in a branch or indirect control flow operation, its label set can be inspected and the appropriate labels extracted. An interesting technical detail is related to the fact that the Intel x86 instruction set does not contain conditional branch instructions that use register or memory operands. Instead, these branch instructions evaluate a number of flag bits, which are usually set by preceding compare or test instructions. As a consequence, our dynamic analysis has to retain the label sets of operands of *compare* and *test* operations until a subsequent conditional branch operation is encountered.

The dynamic monitoring of the IDS is realized with the help of *iTrace*, an instruction-tracing tool developed by our group. The iTrace tool leverages the single-step functionality of ptrace to execute the process under analysis one instruction at the time. Before each instruction is run, iTrace propagates the label information appropriately to keep track of both direct and indirect dependencies. This allows us to keep track of which bytes of the network traffic are involved in the detection process.

3. Constraint Generation

In addition to the knowledge of *which* bytes of the traffic generated by the exploit are used to identify an attack, our dynamic data flow analysis was extended to also provide information about *how* these bytes are used by the intrusion detection sensor. In particular, in this section we explore the approach we use to extract simple byte comparisons (called *basic constraints*) as well as the approach we use to reverse-engineer the automata used by the string matching process.

3.1. Basic Constraints

A basic constraint is a relationship between a value from the attack stream and a constant value that was observed to hold for the execution trace. The value in the attack stream can be a byte, a 16-bit short integer, or a 32-bit long integer. For example, a constraint could specify that the 16-bit short value in the `UDP` header that represents the destination port was used in an equality comparison with the constant '53'.

It is important to note that a comparison operation is used to generate a basic constraint only when the labeled operand directly depends on the input. In addition, it is required that each byte of the operand value depends on only a single input byte (i.e., the label set associated with each byte contains only a single label). These restrictions ensure that we only generate a constraint when a change in the input value gets directly reflected in the operand of the corresponding comparison. Otherwise, it is not possible to predict the effect of a change in the input, and the mutant generation process only receives the information that a certain input byte had some effect on the IDS' execution (without any indication of the exact check performed).

Basic constraints provide a valuable guidance to the mutant generation process. In particular, the test generation engine attempts to modify the attack so that the constraints that were collected when the IDS successfully detected the attack are violated. Of course, it is not always possible to modify input values that are part of constraints. For example, consider a constraint that relates the destination port of a TCP packet to the value 25; since this constraint determines that the (mail) service is attacked, the mutant engine cannot change the destination port value without rendering the exploit ineffective as a consequence.

3.2. String Constraints

To improve the detection accuracy and reduce the false positive rate, most intrusion detection systems use signatures that, in addition to checking for constant numeric values, also specify strings or regular expressions that are matched against the packet payload or parts thereof. In such cases, the strings that are searched for cannot be easily determined by only considering basic constraints. More precisely, the basic constraints generated as a byproduct of the pattern-matching process usually provide no indication of which strings the sensor is actually searching for. The situation is exacerbated by the fact that most pattern-matching algorithms do not directly compare input bytes with expected character values but use state machines or shift tables to find relevant matching strings. In these cases, the input bytes are not directly used in comparison operations, but, instead, they are used indirectly by indexing a state-transition matrix or a shift table. Thus, a different approach is required to extract the strings and the regular expressions that the pattern-matching component of an IDS is searching for.

Our technique to extract strings and regular expressions is based on the observation that most pattern-matching algorithms use finite state machines, either explicitly or implicitly, to perform the matching task. That is, at every point of its analysis, the pattern matcher is in a certain state. Whenever a new input character is checked (or consumed), a transition is performed and the pattern matcher follows the appropriate outgoing edge from the current state to the next state (which, of course, can be the same state again). The basic idea of our technique to extract string constraints is to map out the finite state machine of the pattern matcher by analyzing the execution traces associated with the matching process. More precisely, we gradually explore all the states and transitions of the pattern-matching automaton.

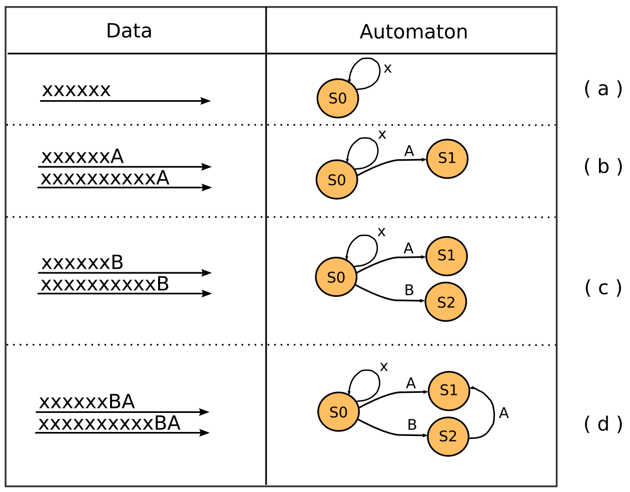

Figure 1. Example of the automaton reconstruction process.

Dynamic Reconstruction of Finite State Automata

The process of mapping out the finite state automaton used by the NIDS is performed by sending a series of carefully crafted packets with slightly different content. We start this process by sending a packet with a payload that contains an initial string composed of a sequence of identical padding characters. Optimally, the padding character is not part of any string that the pattern matcher searches for. However, this is not strictly required and any character can be selected, provided that repetitions of this character do not result in a matched pattern. This can easily be checked by inspecting the detection result reported by the IDS.

The execution trace that is obtained when the pattern-matcher processes the initial string provides the starting point for our subsequent analysis. In particular, after the pattern matcher has consumed a number of identical characters, an additional instance of this character should not cause a transition to another state. That is, the pattern matcher remains in a certain state as more padding characters are consumed. If this behavior can be observed in the initial trace, we consider this state the initial point for our analysis. Otherwise, a different padding character is chosen.

Based on the initial state, we can start the reconstruction of the finite state machine of the pattern matcher. This is done by injecting a single character of the input alphabet into the initial string and observing the change in the execution trace. In particular, we record the target state after the pattern matcher has processed the injected character. This target state is included into our reconstruction of the pattern-matcher automaton, and we insert an edge from the initial state to this target state, labeled with the input character. The process is then repeated by iterating over the remaining characters of the input alphabet, each time recording the target state of the transition that is based on the novel character. When a target state has not been seen before, it is included into our state machine reconstruction. In any case, an appropriate edge is added that connects the current state with the target state. Whenever a state is added to the automaton, we associate with it the string that was sent to the pattern matcher. This string is subsequently used to explore the outgoing transitions of the new state. Note that although, in theory, the alphabet should contain all the possible 256 single-byte characters, it can often be reduced to contain only a small subset of them (e.g., the alphanumeric characters only).

After all possible outgoing transitions of the initial state have been identified, the next state is examined. This process is repeated until all states have been analyzed, and no new states are discovered. At that point, the complete pattern-matching automaton has been reconstructed. Final states are identified by observing that, when reaching one of those states, the IDS produces an alert messages.

To understand the process of mapping the states and transitions of a finite state pattern matcher in detail, a number of questions need to be answered. In particular, we have to introduce our approach to define the states of the pattern matcher and describe the mechanism to recognize transitions.

State Recognition

The *state* of a pattern matcher is defined as the content (values) of all memory addresses and registers that are relevant for the matching process. In this definition, the term "relevant memory addresses and registers" refers to those locations in the virtual address space of the IDS process that are read or written between two state transitions. Of course, it is possible that a certain location is both read and written (or even overwritten multiple times) between two transitions. In these cases, only the *last* read or write operation is taken into account. More precisely, the content of all relevant locations is taken as a snapshot directly before the state transition. The rationale behind our state definition is the fact that *if the relevant memory content between two execution traces is identical at the point before a state transition, the outcome of the matching process is only determined by the characters that are consumed afterwards*. In other words, the previously-consumed characters, even if different, have lead the pattern matcher into exactly the same state.

Unfortunately, simply including all memory addresses and registers that are accessed into the state can be problematic. The reason is that the IDS process might also update variables that are not related to, or relevant for, the internal state of the pattern matching process. For example, consider a variable that counts the number of input bytes that have been processed so far or pointers into the input stream that are increased every time a new character is processed. If these values were included into the description of a state, identical states would be recognized as different, thereby preventing the extraction of the desired state machine.

To prevent irrelevant variables (i.e., variables that are not directly related to the internal state of the pattern matcher) from being incorrectly included into a state, two execution traces are performed. Recall that whenever a transition of a certain state must be analyzed, a character from the input alphabet is appended to the string associated with this state. The resulting string is then embedded into the packet payload (using padding characters) and sent to the IDS. Finally, the execution trace is examined. To exclude unrelated variables, this process is extended by sending the resulting string twice instead of only once. The second time, however, the string is shifted by a few bytes. The two execution traces are then independently used to determine the respective target states. Finally, the states are compared and all locations (memory addresses and registers) that are different are removed. The idea behind this procedure is that, since the same string is sent twice, all variables that are directly related to the pattern matching process, should be identical. Locations that store values related to the position inside the payload, on the other hand, differ and can be safely removed.

It is also possible that locations that are completely unrelated to the pattern-matching process are occasionally touched (read or written). Including these locations into the state is not problematic, provided that they are always the same for a particular internal state of the finite state machine. However, we have not observed this problem in our experiments, probably because pattern matching is usually performance-critical, and thus implemented as succinctly as possible in terms of memory and code.

Transition Recognition

The correct recognition of *state transitions* constitutes a central part of our automaton-reconstruction process. A transition from one state to another occurs every time a new input character is processed (or consumed). This event is recognized by checking for points in the execution trace where a labeled input byte is used in a control-flow decision *for the first time* (i.e., as an operand of a branch instruction or as the target of an indirect jump/call). When this happens, we know that a transition has occurred. In other words, the pattern matcher has processed the labeled byte, and moved into a new state.

By checking the execution trace for control flow instructions that process a label for the first time, we can locate those points where the pattern-matcher transitions into new states. Based on this information, we then extract the memory reads and writes that the IDS performs between each pair of transitions. This information provides us with the relevant memory locations needed to determine the states of the pattern matcher. Note that it is possible that more than one new label is used by a certain control flow instruction. This situation implies that the pattern matcher has consumed more than a single input character before transitioning to a new state. However, no special treatment is required. It is only necessary to record this fact by tagging the edge in the reconstructed automaton appropriately.

Consider the example shown in Figure 1. After processing the first characters of the input stream (which are denoted by 'x'), the automaton reaches a certain state (which is shown as the start state S0 in the example). Then, the next character of the input stream is consumed. This is shown in Figure 1 (b), where the character is an 'A'. To recognize the fact that the automaton processes 'A', we look for the first time a control-flow decision is based on this character. When this decision occurs, we assume that a transition was taken, leading to a new state S1. The next test uses the character 'B', as shown in Figure 1 (c). The processing of this character leads the automaton to a new state S2. The exploration will continue further from S2. For example, in Figure 1 (d), it is shown how the character 'A' might lead the automaton from S2 to S1.

Assumptions and Limitations

The string constraint extraction process described previously relies on one important underlying assumption, which states that each input byte is only considered once for each state transition. That is, we assume that there is a deterministic, finite state machine underlying the pattern-matching process that checks each input byte at most once. In other words, it is not necessary to backtrack and "undo" previous state transitions. This assumption holds for many important algorithms that search for single strings (such as Boyer-Moore [5]) or multiple strings in parallel (such as Aho-Corasick [1]). However, this assumption is not generally valid for all the algorithms used to match regular expressions.

In particular, there are two main techniques that are used to match regular expressions. One relies on a deterministic, finite automaton, which is extracted from a non-deterministic representation of the regular expression. For pattern matchers that use this technique (e.g., Henry Spencer's regular expression library for C [28], which was later utilized in Perl), our approach is capable of correctly reconstructing the automaton. The second technique for regular expressions matching relies on backtracking. Backtracking is required in cases where the regular expression language provides an expressive power that exceeds regular languages. For example, the ability to group a sub-expression with brackets and recall it in the same expression is not present in a regular language and hence, cannot be realized with a finite state machine. As a result, for pattern matchers that use backtracking (such as the Perl-compatible regular expression libraries - PCRE [10]), our automaton reconstruction process will not produce correct results. In such cases, a reconstructed automaton will typically accept a superset of the actual regular expression, because it cannot model the "secondary checks" performed via backtracking.

4. Efficient Test Case Generation

Once the analysis process has determined which parts of the traffic generated by an exploit are used to detect a certain attack, we leverage this information to generate test cases in a more focused way. Consider, for example, a signature that detects a certain attack by searching for a particular string in the URL field of every HTTP request. All attack instances that are different only in their HTTP header values (but not in the URL) can be considered equivalent for this test, as these differences do not influence the IDS detection process. Unfortunately, without any knowledge of the signature, the test generation process can only try to blindly generate all the possible variations, leading to a very large number of equivalent test cases. However, by using the set of constraints extracted by our dynamic data flow analysis, it is possible to drive the test case generation process in order to generate only the "interesting" test cases. In fact, the knowledge of how a signature is matched allows the test case generator to focus only on those attack variations that, while remaining inside the space of valid attacks, lie outside the signature constraints of the IDS. The result is a more focused approach that takes into account the actual signature constraints that are relevant for detection.

We implemented this technique as an extension to Sploit [29]. Sploit is a mutant exploit generation tool that takes an attack template and a set of transformations as input, and generates a set of different, semantically equivalent versions of the attack. These variations, called mutants, can then be executed against a target system in order to test a NIDS's ability to detect different variations of the same attack. For this project, Sploit was modified to incorporate the constraint information generated by iTrace into its mutant generation engine, as described in the following sections.

4.1. Mapping Constraints to Exploit Features

To be able to process the information provided by our dynamic data flow analysis of the execution of an attack, Sploit must first map the constraints generated by iTrace into the corresponding features of the attack. All the dynamic information discussed in the previous sections is based on the absolute positions (or labels) of bytes in the packets that comprise the attack stream. However, Sploit does not reason in terms of bytes, but rather in terms of protocols, commands, and command fields. Thus, a mapping between the two representations is needed. This mapping leverages an *execution table*, which stores the associations between the positions of the bytes in the network stream and the object that was responsible for their generation.

In Sploit, whenever an exploit sends data using a certain protocol, the corresponding protocol manager (i.e., the object in charge of managing the protocol communication and the list of mutant operators working at that layer) adds to the execution table a new row with the details of the data that is added and its location in the network stream. As a result, by knowing the location of a byte in the network stream, Sploit can identify the appropriate set of mutant operators that can operate on that byte.

Note that a given byte position can be associated with more than one mutant operator. For example, consider the case of an attack that injects shellcode inside one of the header fields of an HTTP request. When the HTTP request is sent, each byte of the shellcode will be associated with both the `ShellCode` and the `HTTPRequest` sets of mutant operators.

4.2. Focusing the Test Case Generation

During signature analysis, Sploit first executes the base exploit (i.e., the attack with no transformations applied), and collects the constraints generated by iTrace. Then, it relies on two subsequent refinement phases in order to determine which mutant operators should be applied to generate the relevant test cases.

The first phase consists of taking into account only the positions of the bytes that the IDS used to detect the attack. As mentioned previously, based on the execution table, Sploit maps each byte back to the corresponding portion of the attack that was executed. Once all the byte positions identified as relevant for the detection process are translated into commands and field locations within the attack, Sploit can use this information to refine the mutant generation process. In particular, Sploit disables every mutant operator that does not affect any relevant part of the attack. More general transformations that do not apply to particular parts of the attack (e.g., IP fragmentation) are temporarily deactivated as well, even though they can be reconsidered again when the tool is not able to evade detection using other techniques.

At the end of the first phase, all transformations that cannot affect the parts of the attack checked by the IDS have been removed. Even though this can considerably improve the relevance of the test cases that are generated, a better result can be obtained analyzing the type of constraints extracted by iTrace. In order to take them into account, Sploit implements a second refinement phase, based on a local simulation of the effects of each mutant operator.

In Sploit, a mutant operator can have a set of parameters that describe in which way the transformation is applied to the original exploit. For example, an operator that obfuscates the shellcode's NOP sled can have a parameter that contains the list of bytes that can be used to compose the sled. This means that a single mutant operator can generate multiple (but finite) different attack mutations. Consider an example in which Sploit is able to pinpoint that the IDS only checks the part of the network stream where the exploit shellcode is stored. By analyzing the dynamic constraints, Sploit can now incorporate information on the exact checks that were performed by the IDS on the shellcode section. For example, suppose that the IDS was searching for a particular string, such as a sequence of $0x90$ bytes, which is often used as a NOP sled. Then, the simulation routine cycles through the set of mutant operators selected by the first phase and uses each of them to mutate the shellcode for every possible value of the operator parameters. The result of each iteration is checked against the signature constraints (in this example, represented by the string automaton that encodes a series of $0x90$ bytes). If the constraint is not violated, Sploit removes the parameter value from the list of possible alternatives. Furthermore, if all possible parameters values have been eliminated for a mutant operator, the operator itself is deactivate, because its transformation cannot evade the detection process.

It is important to note that this simulation phase does not require generating all the possible mutants locally. For example, suppose that for a certain attack, ten mutant operators are available, each with ten possible parameter values. Combining them, the total number of possible test cases that can be generated is 10^{10}. However, the number of instances that must be locally simulated is only 100: one for each parameter value of each operator.

By applying the previous refining phases, Sploit can focus the test case generation process to produce only those test cases that can evade at least one of the derived constraints. If the resulting test suite is empty, it might be the case that some constraints could not be evaded by the available mutant operators, and more general mutation techniques can be applied to try to evade the IDS.

5. Evaluation

To demonstrate the effectiveness of our technique, an evaluation was conducted using prototype implementations of Sploit and iTrace. In particular, we developed two dynamic analysis components: one for the extraction of basic constraints and one for the extraction of string automata. As we previously explained in Section 3.2, the string extraction algorithm requires a traffic generator that can be controlled to inject different payloads. We implemented two different modules to create both UDP packets and TCP sessions. Finally, we extended the Sploit tool to process the constraints' feedback and use them to focus the test case generation.

A limit of the current implementation is that complex signatures cannot be analyzed in a fully-automated fashion. In fact, most IDS engines are optimized to check the string constraints only after all the other simple constraints contained in the signature have matched the traffic. This implies that the IDS string analysis routine is not invoked when simpler constraints do not match. As a result, in order to reverse-engineer a complex signature, the user must first set up iTrace to extract the basic constraints, then modify the traffic generator to match these constraints, and finally run the string extraction tool. Once the constraints have been retrieved, the mutant generation process is completely automatic and does not require any human intervention.

We tested our technique against two intrusion detection systems: Snort and Symantec's NIDS. The evaluation testbed was composed of a RedHat Linux 9 system running several vulnerable services, a RedHat Linux 9 machine running the Sploit prototype, a Gentoo Linux 2005.0 machine running iTrace and Snort 2.4.3, and the Symantec Network Security 7120 appliance (also running iTrace, and patched to version 4.0.0.11).

5.1. Dynamic Analysis of Snort Signatures

One may naturally question the use of Snort in this evaluation, as its standard signatures set is freely available for analysis. This would seem to negate the motivation behind our approach, since it would be easier to manually analyze the signatures rather than infer the signature constraints from the execution of the IDS. However, we felt it necessary to demonstrate our technique against known signatures, in effect establishing a "ground truth" with respect to its effectiveness.

For the first experiment, we tested the ability of our approach to generate and process basic constraints derived from direct numeric comparisons observed by iTrace. The signature we examined was the Snort signature for the Samba trans2open buffer overflow [27], the relevant portions of which are shown in Figure 2.

This signature contains four basic constraints, based both on single bytes (specified through the `content` keyword) and on a 16-bit short comparison (specified through the `byte_test` keyword). Our tool was able to extract a set of constraints that correctly reflects all these checks. When Sploit analyzed these constraints, it determined that the tests for `0x00`, `0xff` and `SMB2` were performed on the SMB header of the packet; because Sploit possesses no available mutant operators that operate on the SMB header, the mutation engine could not violate these constraints. The checks for `0x00` and `0x14`, however, were performed on a portion of the exploit that was equivalent to padding, and, therefore, Sploit's shellcode generator was able to generate a semantically-equivalent padding byte to replace the `0x00` byte. As a result, the attack based on this transformation was able to evade Snort, while successfully exploiting the target application.

In order to test the ability of our approach to infer string-matching automata, we run two sets of experiments. In the first set, we run our component to analyze a simplified version of the complete Snort's FTP-based signature set (with 76 signatures). Each rule was modified in order to eliminate any constraint that would prevent the string matching mechanism to be executed, by including only the `content` rules that were used to define the strings. In this case, our tool was able to correctly extract all the 76 corresponding automata.

For the second set of experiments, we examined the signature shown in Figure 3, associated with a remote command execution vulnerability in the Avenger's News System [25].

The automaton extracted with our tool matched exactly the string constraint specified by the signature (i.e., the constraint that requires that the URI contains the string "`/ans.pl?p=../../`"). The automaton was then loaded in Sploit and used to verify if any of the applied mutant operators would affect the string. The result was a test suite containing only the mutants with a modified URL. Also in this case, the first test case executed, which had a "`/./`" inserted into the path passed to the argument `p`, was able to successfully evade Snort, while being successful at compromising the target application.

5.2. Dynamic Analysis of Closed-source Signatures

Having validated our approach against an open-source IDS with known signatures, we wanted to demonstrate its effectiveness against a closed-source intrusion detection system. This required two different experiments. The first experiment aimed at demonstrating that the constraints derived in the case of a closed-source system correspond to those specified by the signatures. Unfortunately, the details of the signatures are not known for the Symantec IDS, and, therefore, we had to use a user-defined signature to test the precision of our constraint-derivation process with respect to the closed-source detection engine. The goal of the second experiment was, instead, to prove that our technique could be used to automatically evade a signature whose implementation was not known *a priori*.

For the first experiment, we loaded into the Symantec NIDS a signature to detect the IIS chunked encoding attack [26]. The signature contained some basic constraints that tested for the presence of a series of constant bytes in an HTTP chunk (`PADP\r\n`), used in a particular version of the chunked encoding exploit. Then, we executed the attack, which was correctly detected, and we collected the sensor's execution traces using iTrace. The analysis of the traces allowed for the derivation of basic constraints that

```
netbios.rules:alert tcp $EXTERNAL_NET any -> $HOME_NET 139 (
  msg:"NETBIOS SMB trans2open buffer overflow attempt";
  content:"|00|"; depth:1;
  content:"|FF|SMB2"; depth:5; offset:4;
  content:"|00 14|"; depth:2; offset:60;
  byte_test:2,>,256,0,relative,little;
  reference:bugtraq,7294; reference:cve,2003-0201;
  reference:url,www.digitaldefense.net/labs/advisories/DDI-1013.txt;
  classtype:attempted-admin; sid:2103; rev:9;)
```

Figure 2. Snort SMB trans2open overflow signature.

```
web-misc.rules:alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (
  msg:"WEB-MISC ans.pl attempt"; flow:to_server,established;
  uricontent:"/ans.pl?p=../../";
  reference:bugtraq,4147; reference:bugtraq,4149; reference:cve,2002-0306;
  reference:cve,2002-0307; reference:nessus,10875;
  classtype:web-application-attack; sid:1522; rev:10;)
```

Figure 3. Snort Avenger's News System remote command execution signature.

were identical to the ones contained in our signature. The constraints were then passed to the Sploit engine, which generated a variation of the exploit that successfully evaded detection.

For the following experiments, we loaded a user-defined signature into the Symantec IDS that employed a string match to detect an attack. More precisely, we used the same string-based signature that we used in the analysis of Snort (see Figure 3). Our string-derivation technique was able to correctly identify that the string "`/ans.pl?p=../../`" was used as part of the detection process. This information was used to drive the Sploit tool, which, as in the Snort case, focused its mutant operators on the URL content, and successfully evaded the signature.

Finally, once we determined that our technique was able to correctly derive both the basic and the string constraints, we ran a second set of experiments to demonstrate that our technique can successfully guide the evasion of closed-source signatures for which we have no *a priori* knowledge. To this end, we ran the unmodified Samba trans2open exploit over the link monitored by the Symantec IDS and observed that the sensor indeed included a signature for this attack. The dynamic taint analysis revealed a set of constraints that included the equality constraints on a 16-bit word `0xd007` contained in the shellcode portion of the attack. Using this information, Sploit was able to successfully generate a mutant that violated that particular constraint of the Symantec's signature. Thus, the resulting mutant exploit was able to successfully compromise the target system while evading detection by Symantec's NIDS.

5.3. Discussion

From these experiments, we can conclude that our dynamic taint analysis methodology is accurate enough to properly reconstruct basic and string constraints used by both open-source and closed-source intrusion detection engines. The knowledge of what attack data is used by the signatures and how it is used allowed Sploit to focus the generation of test cases, thus increasing the chances of spotting flaws in the corresponding signatures in a reasonable amount of time.

For example, let us consider the experiment with the IIS chunked encoding attack, which involves the HTTP protocol. In this case, the large number of available mutant operators for HTTP would make the test suite far too large for an exhaustive analysis. In fact, without any information on the way in which the IDS detects the attacks, Sploit generated a test suite containing tens of millions of attack mutations. Because of that, the information about the position of the attack data in the network stream was extremely important. By determining that the signature matched only a few bytes inside one of the chunks, our mutation engine could disable all the transformations that did not operate on the data content of an HTTP request. This reduced the suite to just a few hundreds of test cases. Finally, taking into account also the iTrace basic constraints, Sploit could remove additional mutant operators, leaving just the ones that operate on the end-of-line characters of the chunk data.

A similar reduction in the number of test cases was observed in the case of the `ans.pl` experiment. In this case, however, the knowledge that the IDS is looking for something inside the URL field did not help much, because most of the HTTP operators operate on the URL content. Fortunately, the string-based constraints derived by the analysis were very effective, reducing the number of test cases to only five mutants.

It is important to note that, in each experiment, the first mutant generated from the reduced test suite was successful in both compromising the target and evading the NIDS under test. As a result, instead of blindly testing a potentially large set of attack variations hoping to find a combination of mutant operators that can evade the signature, our approach was able to eliminate a large portion of useless test cases, evading the signature under analysis on the first attempt.

6. Related Work

The use of variations of attacks to test intrusion detection systems and other security mechanisms has recently received considerable attention.

The idea of performing desynchronization attacks was initially introduced by Ptacek and Newsham [20] and implemented in evasion tools such as nidsbench [3] and congestant [11]. Recently, a number of other techniques to perform desynchronization at the application level [21] and at the attack payload level [15, 17] have been proposed. However, these techniques are mostly used as a way to evade detection and not as comprehensive tools to test and evaluate intrusion detection systems.

One of the earliest works that systematically considered attack variations as a way to test intrusion detection systems was Raffael Marty's Thor [12]. Thor's design included the possibility to generate variations at both the network and the application layer. However, Thor's implementation is limited to network-level evasion techniques, which are orthogonal with respect to malicious payload being delivered with an attack. In addition, the only result mentioned is the application of an evasion technique based on IP fragmentation to an HTTP-based attack.

The first complete framework for the generation of mutant attacks was Sploit [29], which was used as a basis for the work described in this paper. Sploit defines a number of mutant operators and provides a mutation engine that applies the operators to an exploit template to automatically generate variations of attacks. The first Sploit prototype successfully evaded both open-source and commercial intrusion detection systems. Note that Sploit does not claim to completely cover the space of possible variations of an attack, nor states that it guarantees that the variations of attacks are successful. Nevertheless, it provides an effective framework for the composition of evasion techniques to test the quality of intrusion detection signatures.

An approach similar to Sploit was introduced by Rubin et al. in [22]. In this case, a tool called AGENT uses inference rules to produce attack variations. The advantage (and novelty) of AGENT is its formal characterization of the type of transformations applied to an exploit. This allows one to better characterize the mutation process and the mutation space. However, its formal approach does not allow one to easily model very complex transformations and, even though the mutation space can be formally described, the approach provides no guidance as to how to explore this space.

In [24], the same authors proposed a model to assess the coverage of their mutant generation approach. In particular, given a set of transformation rules Φ, they defined a mutation algorithm as Φ-complete if it can generate all the possible attack instances derivable from the original exploit with respect to Φ. To address the fact that this algorithm could potentially generate an infinite number of test cases, the authors limit the size (i.e., the number of bytes) of the mutants that can be generated.

A first step towards a guided form of mutation exploration was presented in [14], where the authors reverse-engineered a commercial, closed-source intrusion detection system to determine the inner workings of the signature matching process. This work introduced the use of dynamic analysis to identify which portions of an attack were actually used in the signature matching process. The results of this analysis were used to guide a manual evasion attacks. The work presented in this paper extends the idea of using dynamic analysis, but relies on more sophisticated techniques to *automatically* drive an exploit mutation engine.

An area that is related to the extraction of finite state automata (FSA) is automata induction or grammar inference. With grammar inference, the task is to identify an automaton, given only examples of positive (and possibly negative) instances of the language that this automaton accepts. While the task is very hard in theory [9], there have been numerous approaches to find acceptable solutions in practice [2]. The most significant difference to our work is that we do not only have information about the input and output behavior of the automaton that we aim to infer, but we can also observe its internal state while processing input. Thus, we can potentially produce much more accurate results.

7. Conclusions and Future Work

Mutant exploits are an effective way to test intrusion detection signatures and identify "blind spots" in intrusion detection systems. Even though there exist several systems that support the generation of mutant exploits by applying mutant operators to an exploit template, the selection of the test cases to execute is performed either manually by a human expert or randomly.

This paper presents a novel approach to reverse engineering NIDS's signatures in order to automatically extract a set of constraints that are used to guide the test case generation process. The approach is based on the dynamic analysis of the network intrusion detection binary to identify which parts of a network stream are checked to identify an attack and how the data is used in the decision process. The results of the analysis are then used to automatically drive a mutation engine so that it applies the most relevant mutant operators to the detection-critical portions of the exploit. The proposed approach was used to evade both an open-source and a commercial, closed-source intrusion detection systems.

Future work will focus on extending the set of constraints that we can extract from the signatures, and on providing a theoretical model for the automaton learning algorithm that we use to reverse engineering the string constraints.

Acknowledgments

This research was partially supported by the National Science Foundation, under grants CCR-0238492, CCR-0524853, and CCR-0716095.

References

[1] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the Association for Computing Machinery*, 18(6), 1975.

[2] D. Angluin and C. Smith. Inductive Inference: Theory and Methods. *ACM Computing Surveys*, 15(3), 1983.

[3] Anzen. nidsbench:a network intrusion detection system test suite. `http://packetstorm.widexs.nl/UNIX/IDS/nidsbench/`, 1999.

[4] D. Balzarotti. *Testing Intrusion Detection Systems*. PhD thesis, Politecnico di Milano, 2006.

[5] R. Boyer and J. Moore. A Fast String Searching Algorithm. *Communications of the Association for Computing Machinery*, 20(10), 1977.

[6] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security Symposium*, 2004.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worms. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2005.

[8] J. Crandall and F. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *37th International Symposium on Microarchitecture*, 2004.

[9] E. Gold. Language Identification in the Limit. *Information and Control*, 5(1967), 10.

[10] P. Hazel. PCRE: Perl Compatible Regular Expressions. `http://www.pcre.org/`, 2005.

[11] horizon. Defeating Sniffers and Intrusion Detection Systems. *Phrack Magazine*, 8(54), December 1998.

[12] IBM Zurich Research Laboratory. Thor. `http://www.zurich.ibm.com/csc/infosec/gsal/past-projects/thor/`, 2004.

[13] S. Jha, S. Rubin, and B. Miller. Using Attack Mutation to Test a High-End NIDS. Information Security Bulletin, vol. 10, April 2005.

[14] C. Kruegel, D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer. Reverse Engineering of Network Signatures. In *Proceedings of the AusCERT Asia Pacific Information Technology Security Conference*, Gold Coast, Australia, May 2005.

[15] S. Macaulay. ADMmutate: Polymorphic Shellcode Engine. `http://www.ktwo.ca/security.html`.

[16] R. Marty. Thor: A Tool to Test Intrusion Detection Systems by Variations of Attacks. Master's thesis, ETH Zurich, March 2002.

[17] Metasploit Project. Metasploit. `http://www.metasploit.com/`, 2005.

[18] D. Mutz, G. Vigna, and R. Kemmerer. An Experience Developing an IDS Stimulator for the Black-Box Testing of Network Intrusion Detection Systems. In *Proceedings of the 2003 Annual Computer Security Applications Conference (ACSAC '03)*, pages 374–383, Las Vegas, Nevada, December 2003.

[19] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[20] T. Ptacek and T. Newsham. Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, January 1998.

[21] R. Graham. SideStep. `http://www.robertgraham.com/tmp/sidestep.html`, 2005.

[22] S. Rubin, S. Jha, and B. Miller. Automatic generation and analysis of NIDS attacks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2004.

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

[24] S. Rubin, S. Jha, and B. Miller. On the Completeness of Attack Mutation Algorithms. *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 43–56, 2006.

[25] SecurityFocus. Avenger's News System Remote Command Execution Vulnerability. `http://securityfocus.com/bid/4149`, 2002.

[26] SecurityFocus. Microsoft IIS Chunked Encoding Transfer Heap Overflow Vulnerability. `http://www.securityfocus.com/bid/4485`, 2002.

[27] SecurityFocus. Samba 'call_trans2open' Remote Buffer Overflow Vulnerability. `http://securityfocus.com/bid/7294`, 2005.

[28] H. Spencer. regex: Regular Expression Library. `http://arglist.com/regex/`, 2005.

[29] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In *Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS)*, pages 21–30, Washington, DC, October 2004.

[30] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the 9^{th} ACM Conference on Computer and Communications Security*, pages 255–264, Washington DC, USA, November 2002.