Towards Systematic Signature Testing

Sebastian Schmerl and Hartmut Koenig

Department of Computer Science
Brandenburg University of Technology Cottbus
PF 10 13 44, 03013 Cottbus, Germany
{sbs, koenig}@informatik.tu-cottbus.de

Abstract. The success and the acceptance of intrusion detection systems essentially depend on the accuracy of their analysis. Inaccurate signatures strongly trigger false alarms. In practice several thousand false alarms per month are reported which limit the successful deployment of intrusion detection systems. Most today deployed intrusion detection systems apply misuse detection as detection procedure. Misuse detection compares the recorded audit data with predefined patterns, the signatures. These are mostly empirically developed based on experience and knowledge of experts. Methods for a systematic development have been scarcely reported yet. A testing and correcting phase is required to improve the quality of the signatures. Signature testing is still a rather empirical process like signature development itself. There exists no test methodology so far. In this paper we present first approaches for a systematic test of signatures. We characterize the test objectives and present different test methods.

1 Motivation

The increasing dependence of human society on information technology (IT) systems requires appropriate measures to cope with their misuse. The enlarging technological complexity of IT systems increases the range of threats to endanger them. Besides preventive security measures reactive approaches are more and more applied to counter these threats. Reactive approaches allow responses and counter measures when security violations happened to prevent further damage. Complementary to preventive measures intrusion detection and prevention systems have proved as important means to protect IT resources. Meanwhile a wide range of commercial intrusion detection products is offered, especially for misuse detection. Nevertheless intrusion detection systems (IDSs) are not still deployed in a large scale. The reason is that the technology is considered not matured enough. Lacking reliability often resulting in high false alarm rates questions the practicability of intrusion detection systems [9].

The security function intrusion detection deals with the monitoring of IT systems to detect security violations. The decision which activities have to be considered as security violations in a given context is defined by the applied security policy. Two main complementary approaches are applied: anomaly and misuse detection. *Anomaly detection* aims at the exposure of abnormal user behavior. It requires a comprehensive

A. Petrenko et al. (Eds.): TestCom/FATES 2007, LNCS 4581, pp. 276–291, 2007.
© IFIP- International Federation for Information Processing 2007

set of data describing the normal user behavior. Although much research is done in this area it is difficult to achieve so that anomaly detection has currently still a limited practical importance. *Misuse detection* focuses on the (automated) detection of known attacks described by patterns, called *signatures*. These patterns are used to identify an attack in an audit data stream. This approach is applied by the majority of the systems used in practice. Their effectiveness, however, is also still limited. There are several reasons for this. On the one hand, many systems mainly confine themselves to detecting simply structured network based attacks, often still in a post-mortem mode. Multi-step or distributed attacks which are getting an increasing importance are not covered. On the other hand, the success and the acceptance of misuse detection systems essentially strongly depend on the conciseness and the topicality of the applied signatures. Imprecise signatures heavily confine the detection capability of the intrusion detection systems and lead to false alarms. The reasons of this detection inaccuracy can only in part imputed to qualitative restrictions of the audit functions of the monitored system or network. They must be rather sought in the signature derivation process itself. In particular, the derivation of signatures starting from given exploits often appears as weak point. An attack represents a sequence of actions that exploits a vulnerability in a program, operating system, or network. The derivation of a signature to detect the attack is mostly based on experience and expert knowledge. Methods for a systematic derivation have scarcely reported yet. Automated approaches to reusing design and modeling decisions of available signatures also do not exist. This results in relative long development times for signatures causing inappropriate vulnerability intervals [9].

In order to improve the accuracy of the derived signatures the signatures must be tested and corrected. The objective of a signature test is to prove, whether the derived signature is capable to exactly detect an attack in an audit trail. As the derivation process itself the testing of signatures is still rather empirical. There exist no test approaches and methods yet. This paper focuses on the testing of signatures. It present first approaches for a systematic test of signatures. The paper is structured as follows. In Section 2 we consider the signature derivation process and outline the reasons for the detection shakiness of current signatures. Section 3 backs up the need for a signature tests and outlines the two main issues signature tests have to cope with. In Section 4 we present four test strategies to testing signatures and describe their procedures. Section 5 sketches the application of one test strategy to a concrete signature. Some final remarks conclude the paper.

2 On the Derivation of Signatures

An Attack consists of a set of related security relevant actions or events in a system, e.g. a sequence of system calls or network packets. The task of an audit function is to capture information about the execution of each of these actions by generating audit data records that can be used for analysis. Misuse detection systems try to detect sequences that correspond to known signatures. Thereby it is assumed that security violations do manifest themselves in distinct audit data records, i.e. they are observable, and that they can be detected on the basis of these audit data, i.e. they are detectable.

Fig. 1 depicts these relations. To run an attack the attacker uses *exploits* which are usually known shortly after their appearance. These are programs or pieces of codes to execute an attack which exploit vulnerabilities (e.g. coding faults, configuration errors etc) of the target system. Exploits have various appearances, e.g. program code, protocol packets or scripts. They contain of a sequence of operations or actions (at least one) which cause an abnormal behavior of the attacked host or network.

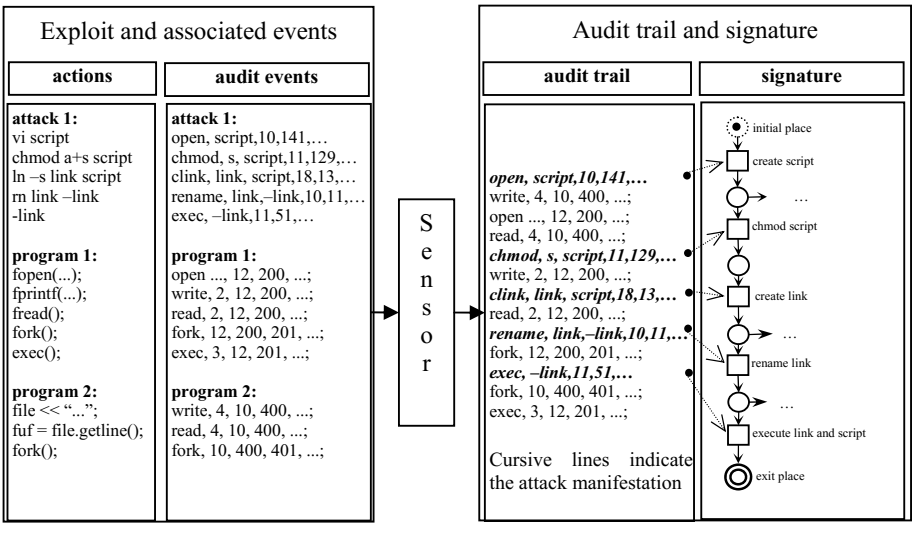

Fig. 1. Exploits, attack manifestations, and signatures

An attack represents a sequence of security relevant actions. They can be usually divided in three steps:

(1) to transfer the attacked system in a vulnerable state,
(2) to exploit the vulnerability to intrude the system, and
(3) to access to the compromised system and/or to change its system data. (This is the proper objective of the attack.)

A signature can only detect step (1) and (2) of the attack. They are predictable and describable based on the knowledge about the vulnerability. The proper concern of the attacker cannot be described because it is not predictable. Thus signatures comprise only the first two steps of an attack.

The execution of attacks leaves traces which can be audited by IDS sensors. These traces are called *manifestations of the attacks*. Fig. 1 shows the traces for the example exploits. The traces are not stated separately. They are hidden in the audit trail. The latter consists of a sequence of records which contain the traces of all actions executed by the system. In order to separate the attack manifestations the audit trail is searched for attack patterns. These patterns are defined by signatures. A *signature* of an attack describes the criteria (patterns) required to identify the manifestation of an attack in an audit trail. It is possible that several attacks of the same type are executed simultaneously and proceed independently. Therefore it is necessary to be able to

distinguish different instances of an attack. A *signature instance* identifies the manifestation of an attack instance in the audit data stream. Signatures are usually described by means of finite state automata, Petri nets, or special attack description languages [7]. Typically each intrusion detection system uses its own language which is customized to the applied analysis method. Fig. 2 shows an example of a signature modeling in a Petri net like languages described in [8].

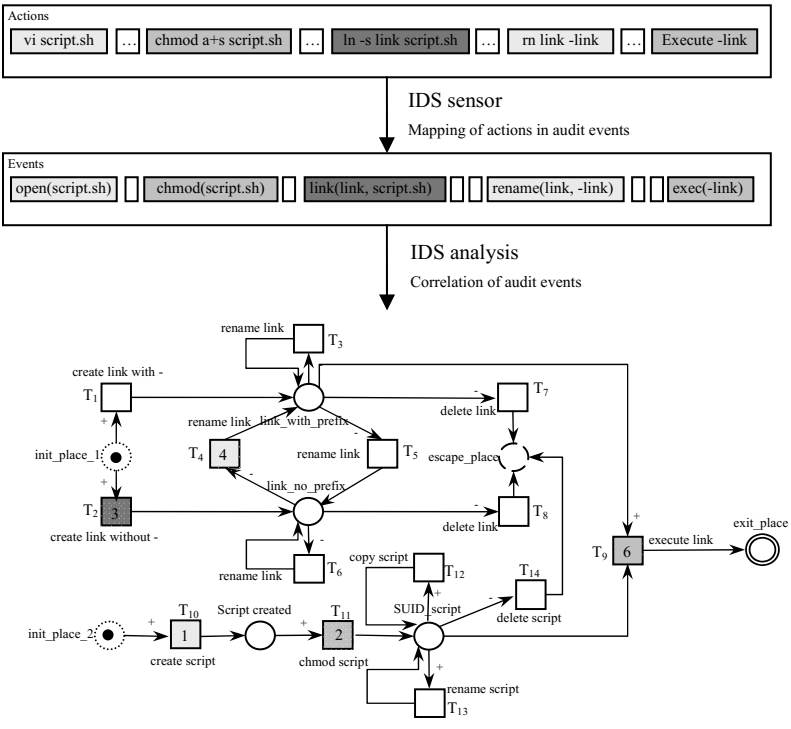

Fig. 2. Modeling of a signature in a Petri net like language [8]

The detection power of signature analysis depends on the accuracy of the signatures applied. To estimate the detection quality of intrusion detection systems usually two measures are applied: the number of security violations not detected (*false negatives*) and the frequency of false alarms (*false positives*). Not detected security violations are caused by over specification of the signatures, whilst false alarms are triggered by inaccurate specifications. The experience shows that not detected security violations have a more grave impact on the systems behaviour than false alarms. Nevertheless, a high false positives rate reveals as a severe problem for running intrusion detection systems in practice. Since misuse detection systems apply deterministic methods, the search for signature patterns, strictly speaking, excludes false positives per definition (assuming an effective audit function). The reality, however, is different, e.g. [3] reports about 10.000 false alarms per month for the use of commercial intrusion detection systems. Other evaluations [2], [5], confirm this

experience. Small false positive rates are an important presumption for the acceptance of misuse detection systems in practice. Inaccurate signatures, therefore, strongly confine the detection power and acceptance of misuse detection systems.

The reasons for the detection shakiness of signatures are only in part caused by qualitative shortages of the used audit functions. They lie in the signature derivation process itself. The derivation of signatures from exploits is the actual weak point. Signatures are mostly empirically derived based on long-term experience of the security administrators. There are scarcely heuristics and methods for a systematic derivation. This often results in inaccurate signatures which have to be step-by-step refined during practical deployment. Therefore, relative long periods for the derivation of good, practically valuable signatures are needed. This means, on the other hand, long vulnerability intervals of the respective systems which cannot be accepted in practice (see [6]). Even if accurate signatures are found further adaptations may be required. This is due to the diversity of today IT environments which force further adaptations to the given deployment environment and the security policies applied. Additional adaptations and enhancements of the signatures are needed when new vulnerabilities or attack mutations become known. The derivation and the maintenance of signatures, therefore, represent one of the most complex tasks for the development and deployment of misuse detection systems.

Only a few approaches have been reported up to now on the systematic derivation of signatures from exploits. Cheung et al. try to simplify the signature design by applying attack models [1]. This approach corresponds to the design patterns of software engineering. It allows the reuse of architectural design decisions. The reuse of concrete modeled signatures or signature fragments is, however, not possible. Rubin et al. describe how mutants can be generated for a given attack [10]. Attack mutants exploit the same vulnerabilities as basis attack without, however, performing the same security relevant actions. If a signature for an attack mutant is supposed to be developed the signature of the basis attack could be reused, if available. Rubin et al. further describe in [11] a refinement of signatures based on formal languages. This approach can help the signature developer to remove triggers for false positives caused by imprecise signatures. The procedure, however, assumes an almost error-free reference signature. In [13] an approach is proposed to use diversity to modeling an implicit complete attack model. This has the advantage of an improved model, however multiple specifications are needed. Larson et al. [4] present a tool for extracting the significant events of an attack from the audit trail. It executes the attack and records the respective audit data. Then the differences between these audit data and an attack free audit trail are derived. The problem of deriving a signature from this difference, however, remains unsolved. In [12] the authors presented an approach to reusing patterns of existing signatures for the development of new signatures. It exploits the fact that similar attacks produces similar traces so that existing signatures may provide an informative basis for the development of new signatures. The approach is based on an iterative abstraction of signatures. Based on a weighted abstraction tree it selects those signatures or signature fragments, respectively, which possess similarities with the novel attack. The reuse of proved structures may not only reduce the efforts of the signature derivation process but it can also considerably shorten the costly test and correction phase.

3 On the Test of Signatures

Inaccurate signatures strongly limit the detection power of misuse detection systems as well as their economic profitability. As discussed the signature development process is complicated and tedious. Systematic derivation procedures are scarcely available. A certain inaccuracy is, therefore, inherent to the derived signatures. A testing and correcting phase is indispensable to improve the quality of the signatures. This phase is an essential part of the signature development process independently of the fact, whether the signatures are derived systematically or by experience.

The objective of a signature test is to prove the accuracy of the given signature by applying it to an audit trail which contains traces of the respective attack. If the signature does not completely detect all traces it must be corrected to approximate the *ideal signature*, i.e. the signature which describes all manifestations M_I of the attack. Normally the signature derivation process does not induce ideal signatures. The derived signatures are either under or over specified.

Under specified signatures describe beside action sequences which are required for a successful attack also actions which either correspond to legitimate behavior or which do not exploit the vulnerability. That means they describe a manifestation set M_U which represents a superset of the manifestations of the ideal signature M_I, i.e. $M_I \subset M_U$. Under specified signatures thus increase the false positives rate. A test strategy to detect under specified signatures has to investigate, whether the actions recorded in the audit trail really exploit the vulnerability. To derive test data actions have complementarily to be assigned to the audit events. These action sequences are then tested on a dedicated system concerning the exploitation of the vulnerability. If the vulnerability is not exploited a specification error exists and the signature must be corrected. The difficulties and limits of this approach lie beside the derivation of the action sequence in the assessment, whether the vulnerability is really exploited.

O*ver specified signatures* do not detect all variants of the attack, i.e. there exist action sequences which successfully exploit the vulnerability but are not captured by the signature. The set of detected manifestations M_O is a subset of the manifestations of the ideal signature M_I, i.e. $M_O \subset M_I \subset M_U$. Over specified signatures induce not detected security violations, i.e. they increase the *false negatives* rate. The objective of a test and correction strategy for detecting over specified signatures is to enhance the signature to approximate the ideal signature. This can be achieved by extending the signature, by substituting actions, and by changing the order of the actions. The test strategy has to ensure that the detection of the attack remains guaranteed, if some actions are replaced by semantically equivalent actions, and that the vulnerability is further exploited.

4 Methods for the Test of Signatures

In this section we present four methods for testing signatures. The main approaches deal with the tests for under and over specified signatures. Furthermore, we present a preliminary test and a test of escape events.

4.1 Preliminary Test

The objective of preliminary tests is to ensure that the derived signatures do not contain grave errors.

Test method: The test consists of two steps. *Test step* (1): Assuming a newly derived signature *S* of attack *A*. This attack is first executed on a dedicated system. The resulting audit trail *T* is recorded and analyzed to determine the events representing traces of *A*. We call these events characterizing events $CE \subset T$ here. In test step (1) *S* is tested against *CE*, i.e. it is proved, whether a misuse detection systems containing signature *S* detects *A*. If the test fails a grave error in the signature specification can be assumed.

Test step (2): Now *S* is tested against the whole audit trail *T*, i.e. it is proved, whether the misuse detection system triggers an alarm when *A* is executed. If the test passes the test procedure can be continued. A negative test outcome usually indicates that the newly derived signature does not correctly correlate the characterizing events *CE* in *T*. Reasons for this are not exactly or too weakly specified signature conditions.

The test method can be mostly automated depending on the applied attack description language.

4.2 Tests for Under Specified Signatures

Under specified signatures contain specifications of action or event sequences which correspond to legitimate behavior or which do not exploit system vulnerabilities. They cause false positives.

The test methods presented in the following to detect under specified signatures is based on the mapping relation δ of the IDS sensor which maps the various security relevant actions into audit events. This relation is usually bijective realized in intrusion detection systems, i.e. there is a δ^{-1}. This means that the corresponding action sequences of the attack can be derived from the audit events demanded by the signature. The objective of the tests is to validate whether these action sequences corresponds to a successful attack. If not, the signature contains a specification error which triggers a false alarm.

Test method: The proposed test method comprises 3 steps. In ***step* (1)** appropriate test cases are derived. Since there is a wide range of conditions which have to be fulfilled between the correlating audit events it is not possible like in many other tests to exhaustively test all possible action sequences. Therefore an appropriate subset of test cases has to be selected depending of the coverage aimed at. Here either path or test coverage criteria can be applied as in software testing.

***Step* (2)** derives for the selected test cases the corresponding action sequences from the signatures by means of δ^{-1}. Signatures specify the properties of the audit events, e.g. type, parameters etc during the attack. They also define conditions regarding the appearance and the context of the correlated audit events. Furthermore, the temporal order of the audit events can be demanded. If these conditions are taken into account the corresponding attack can be re-established.

In ***step* (3)** each derived action sequence is executed on a vulnerable system to prove whether they correspond to attacks. This first requires that correct attack

conditions are established, especially temporal constraints have to be preserved if necessary. This is a decisive precondition because attacks are only successful, when certain conditions are fulfilled, e.g. a load situation. However, not all attacks depend on additional conditions. We distinguish *deterministic successful attacks*, *attacks with preconditions* and *brute force attacks*. Former attacks are independent of special system or application circumstances, therefore correct execution of a deterministic attack is always successful. Consequently this class of attacks is unrestricted testable with this test strategy. In the case of attacks with preconditions, the vulnerability is only exploitable if specific system parameters are fulfilled, e.g. special system load situations. Therefore the necessary preconditions must be synthesized before the test. The class of brute force attacks summarizes all attacks which do not exploiting a concrete vulnerability, e.g. brute force attacks of single authentication systems. Signatures for this kind of attacks are not testable with this test strategy.

The detection of a successful attack execution depends on the attack strategy and the exploited vulnerability, respectively. For disclosing this, four approaches may be generally applied. (a) *Instrumentation of the exploit*, i.e. the derived action sequence has to be changed so that a significant system change is observable. This can be done, for instance, by appending additional actions to the derived attack actions, e.g. setting up a root file or starting/terminating a privileged system process. (b) *Instrumentation of the vulnerability*, i.e. the vulnerable system or code is changed so that the exploitation of the vulnerability becomes observable. This is, for instance, useful, if the program code is available but the vulnerable system (e.g. a technical facility) cannot be patched. (c) *Instrumentation of the whole system*: This approach compares normal system behaviour recorded by an appropriate audit component with the system behaviour after executing an attack. Unlike the other approaches a detailed observation of the system is required. On the other hand, no interference of the attack and the system is needed. (d) *Using a test oracle* which passively examines the system behaviour. This approach though is not able to detect attacks which do not destroy systems functions, e.g. backdoors.

The test method can be automated if certain constraints are fulfilled which is often given. Table 1 contains these constraints. However, there are also shortages which limit the practicality of the method. One problem is that not every action generates an audit event so that δ^{-1} does not always re-establish the complete action sequence. It is not always required to correlate all events of an attack to detect the attack. The crucial issue, however, is to re-establish the attack preconditions. This requires a detailed knowledge of the system behaviour and the attack strategy by the test engineer.

Table 1. Automation degree of test steps

Step		Objective	Can be automated?
1		Test case selection	yes, using typical approaches of software testing
2		Derivation of action sequences	yes, if bijective IDS sensors are deployed
3	I	Establishing the correct attack conditions	yes, for deterministic attacks yes, if constraints can be automated re-established
	II	Execution of action sequences	yes
	III	Test of success	yes, if detection of a successful attack can be automated

4.3 Tests for over Specified Signatures

O*ver specified signatures* do not capture all action sequences which successfully exploit a given vulnerability. Attackers often replace one or more actions of the attack by semantically equal actions. The aim of this transformation is to change the traces of the attack so significantly that the attack is not detected by the intrusion detection system. The proper attack strategy, however, remains preserved, i.e. the given vulnerability is further exploited for running the attack. If the signature does not recognize these attacks it produces false negatives. Test strategies for over specified signatures aim at detecting this detection weakness. Before describing the test strategy we first have to introduce the different types of transformations to change the attack.

There are three types of transforming an attack: No-Op insertion, permutation of actions, and action substitution.

No-Op insertion: In this transformation redundant actions are added which do not change the attack but its traces in the audit trail. This transformation tries to exploit deficiencies of the intrusion detection system to correlate audit events. The evasion of intrusion detection by means of No-Op insertion due to a signature error can be excluded as long as the signature does not strictly demand a direct timed sequence of certain audit events. This demand is only useful in very seldom, specific scenarios. Therefore a signature test can be waived for this transformation.

Permutation of actions: This transformation changes the order of certain actions of the attack and thus the sequence of their related audit events. Two actions can be changed if their execution and their influence on the attack do not depend on each other. These transformations allow bypassing signatures with over specified event sequences. The reason for this kind of over specification is mostly a not correct understanding of the semantics of the attack actions.

Action substitution: This transformation replaces single actions or action sequences by semantically equal action sequences. Thus the IDS sensor registers different actions and events, respectively, although the result of the actions remains the same. A simple example is the substitution of the file renaming operation *mv file1 file2* by a copy and an erase operation: *cp file1 file2*; *rm file1*.

Permutations and substitutions of attack actions produce isomorphic action sequences without enlarging the attack/exploit by redundant actions. Now we present a strategy to test a signature on the detection of isomorphic attack sequences. First we introduce some needed basic notions.

An action a of an attack changes, erases, creates, or uses a certain type of system resource, an *object*. The *object type* O characterizes the type of the system resource, e.g. a process, a socket, or a file. It is represented by the tuple (P, A, r, f) where P defines the set of *object properties*. The object type *file*, for instance, possesses among others the properties file name/path, creation date and access rights. A describes the set of actions of the object type O. Some of them can change the properties of O. The relation r: $a \rightarrow p$ with $a \in A$, $p \subseteq P$ describes for each action $a \in A$ the subset $p_a \subseteq P$ of the object properties which are changed by a. Finally f defines a relation $f: a,p \rightarrow A_a$, with $a \in A$, $p \subseteq P$, $A_a \subseteq A$ which defines for a given action a the set of semantically equivalent actions A_a preserving the properties of the object type

unchanged. An object $o \in O$ is an instance of O which is characterized by the concrete property values. The object types together with the associated relations can be defined for a concrete system by an expert with average effort. These definitions have to be performed mostly once per system (e.g. with host based intrusion detection systems once per operating system or monitored application) and can be used for all signature tests concerning the system.

We now describe rules for transforming action sequences into semantically equal sequences using object types. Each action belongs to a certain object type. The information which action can be executed on an object is sufficient. This information can be easily derived from the signature by the test engineer. If the object types are given each action or action sequence, respectively, can be assigned an object type. Thus an action sequence $a_1a_2a_3....a_n$ with $a_i \in A$, can be mapped on objects $o_1o_2o_3...o_n$ (with $o_i=o_j$, if a_i and a_j relate on the same object). An action a_i can be replaced by a semantically equal action $\acute{a}_i$ if (1) $\acute{a}_i$ compared to a_i does not change additional properties of the respective object, or (2) if the additionally changed object properties by $\acute{a}_i$ are not changed by former or later actions in the action sequence. Thus all substitutable action sequences $\acute{a}_1\acute{a}_2\acute{a}_3...\acute{a}_n$ with $\acute{a}_i \in f(a_i, X/Y)$ and $X = \left(\bigcup_{l=1}^{i-1} r(a_l)\right), Y = \left(\bigcup_{k=i}^{n} r(a_k)\right)$ can be generated from the original sequence $a_1a_2a_3....a_n$ by means of the respective object types and the relations r and f. The approach can be extended without loss of generality to replace single actions by action sequences and vice versa.

Permutations of actions in an action sequence require additional specifications by the test engineer to indicate dependencies between objects and between the actions of an object. In many cases though the following semantics preserving permutation can be performed: An action a_i is almost always exchangeable with action a_j ($i<j$), if a_i and a_j relate to the same object and a_i influences other properties than a_j ($r(a_i) \cap r(a_j) = \emptyset$). Further there exists no action a_l with $i<l<j$ which uses the object associated with a_i and a_j. If such an action exists and a_i is exchanged with a_j then action a_l is executed under different conditions. The order of actions which create and erase objects remains unchanged due to the above mentioned rule that they change all properties. If the test engineer further specifies which objects are independent of each other so the associated actions can be exchanged as long as the before mentioned condition is fulfilled. This specification requires though certain knowledge about the system behaviour and the attack strategy. It is required once per signature.

The described substitution and permutation rules do not cover the whole range of possible action sequence transformations. There are certain types of attacks which can be transformed into action sequences which do correspond to a valid attack. These exceptional cases must be handled by the test engineer.

Test method: For the test, all action sequences are derived which distinguish concerning action sequence. This can be done analogously to step (1) and (2) of test method 2 whereby path coverage is applied in step (1). Next all possible combinations are generated for each action sequences according to the above given rules. Thereafter it is proved using an intrusion detection system, whether one of the derived action sequences is not detected by the signature. In this case the intrusion detection system

does not trigger an alarm for this signature and action sequence. If the signature does not detect a transformed action sequence it has to be checked, whether this action sequence corresponds to a valid attack sequence. This test can be performed analogously to test method 2. If the test outcome is positive the signature has to be completed so that this sequence is also detected.

4.4 Test of Escape Events

Signatures only describe action sequences which represent successful attacks. When during analysis events are recognized which make the successful completion of an attack impossible the analysis has to be stopped to avoid false negatives and, of course, for performance reasons. Actions which prevent the attack to be completed are called *escape events*. They transfer the signature into the initial state. Many escape events are implicitly given by contrary events. For example, in Solaris OS the system calls *fork* and *exit* for creating and terminating processes are complementary events. If the creation of a new process is an indispensable condition for the success of an attack *fork* is a significant part of the signature. The corresponding escape event is *exit*. Escape events are, therefore, an indispensable part of the signature to stop or to re-initialize attack tracking. Consequently, their handling has to be tested.

For this, all events specified in the signature are again converted into actions according to step (2) test strategy 2 (comp. Section 4.2). Next the contrary events are assigned to each signature event using lists of actions with their corresponding contrary actions. The resulting sequence of contrary actions is then again converted into the corresponding events. This can be done using an IDS sensor. In the last step it is proved, whether the signature handles each contrary event. If this is not the case, the escape event is generally not modeled in the signature.

5 Example: Test for Under Specified Signatures

Signatures are specified using various description languages. Therefore the test scenarios has to be adapted to the given signature description language or semantic model, respectively. We now demonstrate this for the test method for under specified signatures with a concrete signature description language. We use EDL (*Event Description Language*) [8] which is based on a Petri-net like modeling approach. It supports the specification of complex multi-step attacks and possesses a high expressiveness and nevertheless allows for efficient analysis. Before describing the test procedure we first outline some essential features of EDL. More details can be found in [8].

5.1 Modeling Signatures in EDL

The descriptions of signatures in EDL consist of places and transitions which are connected by directed edges. Places represent states of the system which are traversed by the related attack. Transitions represent the state changes. They describe the specific events which cause the state change, e.g. security relevant actions. These events are contained in the audit data stream recorded during the attack. The signature execution

is represented by tokens which flow from state to state. *Tokens* represent concrete signature instances. They can be labeled with values as in colored Petri-nets.

Places describe the relevant system states of an attack. They are characterized by a set of features and a place type. Features specify the properties of the tokens which are located in a place. The information contained in a token can change from place to place. EDL distinguishes four place types: *initial*, *interior*, *escape*, and *exit places*. *Initial places* are the starting places of a signature. They are marked with an initial token at the start of analysis. Each signature has exactly one *exit place* which describes the final place of signature. If a token reaches this place, then the signature has identified a manifestation of an attack in the audit data stream. *Escape places* indicate an analysis stop of an attack instance. They are reached if events occur which make the completion of the attack instance impossible. Tokens which reach these places are discarded. All other places are *interior places*. Fig. 3 shows a simple signature with places P_1 to P_4 for illustration.

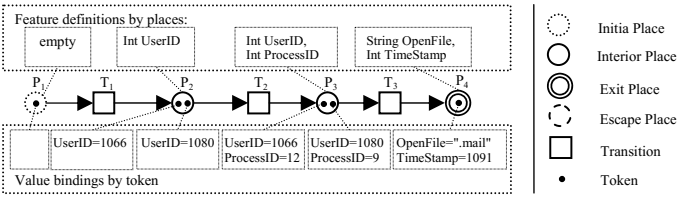

Fig. 3. Features and places

Transitions represent events which trigger state changes of signature instances. A transition is characterized by input places, output places, event type, conditions, feature mappings, consumption mode, and actions. *Input places* of transition *t* are places with an edge leading to the transition *t*. They describe the required state of the system before the transition can fire. *Output places* of transition *t* are places with an incoming edge from the transition *t*. They characterize the system state after the transition has fired. A change between system states requires a security relevant event. Therefore each transition is associated with an event type. Further, a system change can require additional conditions which specify that certain features of the event (e.g. user name) are assigned with particular values (e.g. root). Conditions can require distinct relationships between event and token features on input places (e.g. same values).

If a transition fires, then tokens are created on the transition's output places. These tokens describe the new system state. To bind values to the features of the new tokens the transitions contain *feature mappings*. These are bindings which can be parameterized with constants, references to event features, or references to input place features. The *consumption mode* (cf. [8]) of a transition controls whether tokens that activate the transition remain on the input places after the transition fired. This mode can be individually defined for each input place. The consumption mode can be considered as a property of a connecting edge between input place and transition. Only in the consuming case the tokens which activate the transition are deleted on the input places.

Fig. 4 illustrates the properties of a transition. The transition T_1 contains two conditions. The first condition requires that feature *Type* of event *E* contains the value *FileCreate*. The second condition compares feature *UserID* of input place P_1, referenced by *"P1.UserID"*, and feature *EUserID* of event type *E,* referenced by *"EUserID"*. This condition demands that the value of feature *UserID* of tokens on input place P_1 is equal to the value of event feature *EUserID*. Transition T_1 contains two feature mappings. The first one binds the feature *UserID* of the new token on the output place P_2 with the value of the homonymous feature of the transition activating token on place P_1. The second one maps the feature *Name* from the new token on place P_2 to event feature *EName* of the transition triggering event of type *E*.

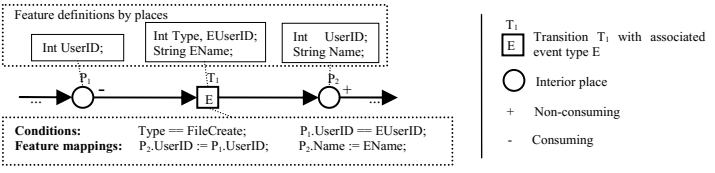

Fig. 4. Transition properties

5.2 Test Steps

We now explain the test for under specified signatures according Section 2.2 for a shell-link-attack which is described in EDL. A **shell-link-attack** exploits a special shell feature and the SUID (*Set-User-ID*) mechanism. If a link to a shell script is created and the link name starts with "-", then it is possible to create an interactive shell by calling the link. In old shell versions regular users could create an appropriate link which points to a SUID-shell-script and produce an interactive shell. This shell runs with the privileges of the shell-script owner (maybe root).

Fig. 5 depicts the respective EDL-signature.

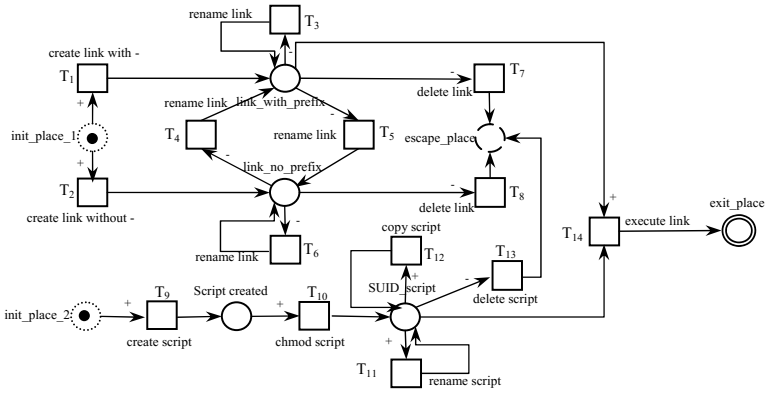

Fig. 5. Simplified EDL-signature of the shell-link-attack

Applying the test method of Section 4.2 to the test of shell-link-attack signature the following test steps have to be executed:

Step1: For test case selection, we use the path coverage criteria (C4). Thus every possible path from the initial to the exit place will be selected. In our example this results in 19 different paths (e.g. T_1,T_5,T_4,T_9,T_{10},T_{12},T_{14}), if each loop is passed maximum once.

Step2: Based on the selected paths action sequences are assigned to the events required by the transition by means of the inverse relation δ^{-1}. In this case timed independencies between the action sequences of transitions T_1 to T_8 and T_9 to T_{13} can be neglected. This restriction is possible, since the actions of the two transition paths from places *init_place_1* and i*nit_place_2*, respectively, to the e*xit_place* are concurrent. Only transition T_{14} synchronizes the two concurrent action sequences. Since the shell-link-attack represents an attack which is executed within a shell, the inverse relation δ^{-1} assigns shell commands to the associated events. The transition conditions are used to implement the parameters of the shell commands.

We show this as example for transition T_3: T_3 fires, when a *rename_link-event* occurs and there is a token on place *link_with_prefix* and the event fulfils the two transition conditions (*link_with_prefix.link_name == new_link_name*) and (*new_link_name == RegExp*("-.*")). The associated *rename_link-event* is mapped by δ^{-1} onto the shell command *mv*. This command has two parameters: the old (*old_name*) and the future name (*new_name*) of the link or the file, respectively. The first transition condition determines the *old_name* parameter of *mv* command with the value of the *link_name* feature from the token of the place *link_with_pefix.* The second parameter (*new_name*) of the *mv* command is arbitrary, but due the second transition condition it is restricted to a name which begins with "–".

Step3a (*Establishing the correct attack conditions*): The step is dropped, since the shell-link-attack is a deterministic successful attack.

Step3b: Because the derived action sequence are shell commands they are simply executed by means of a scripts in a shell.

Step3c: The successful exploitation of the vulnerability by the action sequence can be proved by means of the shell command *id* which outputs the effective UserID. In case of a successful attack it should correspond to the UserID of the script owner. Therefore each action sequence has to be completed by appending an *id*-statement for comparing the UserID. Thus a successful attack execution can be determined automatically.

Test outcome: The execution of the 19 action sequences showed that all sequences which containing copy statements for triggering transition T_{12} don't leading to a successful attack. Accordingly the transition T_{12} must be incorrect. The analysis of this transition and the associated copy command *cp* revealed in a short time that *cp* during copying removes the SUID bit set before with *chmod* (T_{10}). Accordingly the signature must be corrected so that the outgoing edge of transition T_{12} leads to *script_created* and not as up to now to place *SUID_scrip*.

6 Final Remarks

The derivation of signatures from new exploits is still a tedious process which requires much experience. Systematic approaches are still rare. Newly derived signatures often possess therefore significant detection shakiness. Inaccurate signatures strongly limit the detection power of misuse detection systems as well as their acceptance in practice. A longer test and correction phase is needed until qualitative and accurate signatures can be applied which implicates an unacceptable vulnerability window. Systematic test methods can help to accelerate the signature development process and to reduce the vulnerability period of affected systems. In this paper we presented first approaches for a systematic signature test.

The detection shakiness of newly derived signatures is the result of heuristic derivation procedures. Even the rare systematic approaches scarcely induce ideal signatures due to the broad range of system details to be taken into account. Normally derived signatures are either under or over specified. We presented two test methods to detect these both kinds of variances as well as preliminary tests and a test on escape events. Test methods for signature tests require a strong involvement of the test engineer in details of the considered system. Unlike other tests signature tests do not require specific test architecture. All tests are executed on the vulnerable system and the monitoring intrusion detection system. A central issue of signature testing is to re-establish the conditions needed for successfully running an attack. This requires a lot of experience and limits the practicability of the tests. Beside this or in the case of deterministic successful attacks the test engineer needs only sparsely knowledge about the concrete attack and signature to accomplish the tests. We are currently investigating the proposed test methods with concrete signatures. Further we look for other test strategies.

References

1. Cheung, S., Lindqvist, U., Fong, M.: Modeling Multistep Cyber Attacks for Scenario Recognition. In: Proc. of the 3rd DARPA Information Survivability Conf. and Exposition, pp. 284–292. IEEE Computer Society Press, Washington (2003)
2. Debar, H., Morin, B.: Evaluation of the Diagnostic Capabilities of Commercial Intrusion Detection Systems. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, pp. 177–198. Springer, Heidelberg (2002)
3. Julisch, K.: Dealing with False Positives in Intrusion Detection. In: Debar, H., Mé, L., Wu, S.F. (eds.) RAID 2000. LNCS, vol. 1907, Springer, Heidelberg (2000)
4. Larson, U., Lundin Barse, E., Jonsson, E.: METAL - A Tool for Extracting Attack Manifestations. In: Julisch, K., Krügel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 85–102. Springer, Heidelberg (2005)
5. Lippmann, R., Haines, J.W., Fried, D.J., Korba, J., Das, K.: The 1999 DARAP Off-Line Intrusion Detection System Evaluation. In: Debar, H., Mé, L., Wu, S.F. (eds.) RAID 2000. LNCS, vol. 1907, pp. 162–182. Springer, Heidelberg (2000)
6. Lippmann, R., Webster, S., Stetson, D.: The Effect of Identifying Vulnerabilities and Patching Software on the Utility of Network Intrusion Detection. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, pp. 307–326. Springer, Heidelberg (2002)

7. Meier, M., Bischof, N., Holz, T.: SHEDEL - A Simple Hierarchical Event Description Language for Specifying Attack Signatures. In: Proc. of the 17th IFIP International Conference on Information Security, pp. 559–571. Kluwer, Dordrecht (2002)
8. Meier, M., Schmerl, S., Koenig, H.: Improving the Efficiency of Misuse Detection. In: Julisch, K., Krügel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 188–205. Springer, Heidelberg (2005)
9. Ranum, M.J.: Challenges for the Future of Intrusion Detection. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, Springer, Heidelberg (2002)
10. Rubin, S., Jha, S., Miller, B.: Automatic Generation and Analysis of NIDS Attacks. In: Proc. of. 20th Annual Computer Security Applications Conference, Tucson, AZ, USA, pp. 28–38. IEEE Computer Society Press, Los Alamitos (2004)
11. Rubin, S., Jha, S., Miller, P.B.: Language-based generation and evaluation of NIDS signatures. In: Proc. of the IEEE Symposium on Security and Privacy, Oakland, CA, USA, pp. 3–17. IEEE Computer Society Press, Los Alamitos (2005)
12. Schmerl, S., König, H., Flegel, U., Meier, M.: Simplifying Signature Engineering by Reuse. In: Müller, G. (ed.) ETRICS 2006. LNCS, vol. 3995, pp. 436–450. Springer, Heidelberg (2006)
13. Totel, E., Majorczyk, F., Mé, L.: COTS Diversity Based Intrusion Detection and Application to Web Servers. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 43–62. Springer, Heidelberg (2006)