Edith Cowan University

Research Online

Australian Digital Forensics Conference

Conferences, Symposia and Campus Events

3-12-2007

Multi-step scenario matching based on unification

Soroto Panichprecha
Queensland University of Technology

George Mohay
Queensland University of Technology

Andrew Clark
Queensland University of Technology

Follow this and additional works at: https://ro.ecu.edu.au/adf

Part of the Computer Sciences Commons

Multi-Step Scenario Matching Based on Unification

Sorot Panichprecha, Jacob Zimmermann, George Mohay, Andrew Clark
Information Security Institute
Queensland University of Technology
{s.panichprecha, j.zimmerm}@isi.qut.edu.au, {g.mohay, a.clark}@qut.edu.au

Abstract

This paper presents an approach to multi-step scenario specification and matching, which aims to address some of the issues and problems inherent in to scenario specification and event correlation found in most previous work. Our approach builds upon the unification algorithm which we have adapted to provide a seamless, integrated mechanism and framework to handle event matching, filtering, and correlation. Scenario specifications using our framework need to contain only a definition of the misuse activity to be matched. This characteristic differentiates our work from most of the previous work which generally requires scenario specifications also to include additional information regarding how to detect the misuse activity. In this paper we present a prototype implementation which demonstrates the effectiveness of the unification-based approach and our scenario specification framework. Also, we evaluate the practical usability of the approach.

Keywords

Computer Forensics, Event Correlation, Multi-step Scenario, Signature Matching, Event Representation

INTRODUCTION

This paper presents our approach for detecting multi-step misuse scenarios using a unification algorithm as a signature matching mechanism. The unification algorithm has been mainly used in logic programming and in the artificial intelligence field due to its ability to confirm or refute solutions to problems using inference on a set of given rules in order to find an expression that matches the fact base. The unification algorithm, or some form of it, has been used by some misuse-based intrusion detection systems (Olivian and Goubault-Larrecq 2005, Mounji 1997). Rules in the unification algorithm, when applied to misuse-based intrusion detection systems, are considered to be descriptions of misuse activities. We have extended and refined previous approaches to such use of the unification algorithm and have adapted it to the special requirements of scenario detection in order to provide increased benefits of extensibility and flexibility. Previous work using built-in unification (as in Prolog based intrusion detection systems) has required scenario writers in addition to specify state information and context information. In addition, we incorporate an event correlation framework which utilises abstraction of events derived from heterogeneous sources, e.g., system audit data, application audit data, and captured network traffic.

Multi-step scenario matching is a complex process. There are several issues regarding the detection of multi-step scenarios as follows. The detection, in most cases, requires dealing with events from multiple sources e.g., system audit data, application logs, and captured network traffic. Therefore, multi-step scenario specification and detection is problematic due to the fact that event data from different sources may use different semantics and syntax. The specification of scenarios in most of the previous work (Kumar 1995, Lindqvist and Porras 1999, Yang et al. 2000, Meier et al. 2005, Illgun et al. 1995) requires an intimate understanding of the underlying matching mechanisms and how to match a given scenario. It is thus easily prone to human errors. In our work, we aim at minimising the scenario writers' focus on the how of the matching mechanism providing them with a framework which requires them only to express what to detect, as suggested by (Roger and Goubault-Larrecq 2001).

To address the lack of a uniform event representation, we employ an event abstraction model as proposed in (Panichprecha et al. 2006). The event abstraction model provides a uniform representation of events from multiple sources, e.g., system audit data, application logs, and network traffic. By incorporating the tools provided by the model, our work can access events across multiple platforms and sources.

We have developed a prototype of the unification-based multi-step scenario matching. The prototype is intended to be a tool which facilitates log analysis. We have also developed a Python-based scenario specification framework for specifying multi-step scenarios. A number of misuse scenarios have been used to demonstrate the framework and to evaluate the unification-based scenario matching framework.

RELATED WORK

There has been a considerable body of research into multi-step scenario detection focusing on intrusion detection (Lindqvist and Porras 1999, Mounji 1997, Ilgun et al. 1995, Kumar 1995), alert correlation (Morin et al. 2002, Carey et al. 2003), computer forensics (Abbott et al. 2006), and vulnerability assessment (Ou et al. 2005). Existing work in these fields share a common characteristic, in particular, they aim to compare two expressions (e.g., event data with scenario specifications or host information with threat information) and return the results of the comparison.

There are two multi-step scenario matching techniques of relevance to this work: the use of rule-based expert systems (Lindqvist and Porras 1999, Mounji 1997, Roger and Goubault-Larrecq 2001) and state transition models (Ilgun et al. 1995, Kumar 1995, Cuppens and Ortalo 2000, Meier et al. 2005). We now provide brief details of these techniques, their advantages, and limitations.

Rule-based expert system techniques define mechanisms to compare rules (scenarios or signatures) against audit records. Examples of such systems are EMERALD (Lindqvist and Porras 1999), ASAX (Mounji 1997), and ORCHIDS (Olivain and Goubault-Larrecq 2005). EMERALD uses a forward chaining rule-based expert system where the system establishes a chain of rules which links facts (audit records) to goals (signatures). Similarly, ASAX uses a rule-based expert system, however, it specifies signatures as pairs of conditions and actions. When a condition is met, the corresponding action is triggered, which can either be activating another set of chain condition pairs or reporting an alert. ORCHIDS is an intrusion detection system based on the technique proposed in (Roger and Goubault-Larrecq 2001) whose idea is derived from ASAX. The detection is performed by comparing event streams against application-specific temporal logic expressions. The advantages of the rule-based expert system approach and its variants are the simplicity and straightforwardness of the signature matching mechanism. However, this technique generally suffers from the problem of being inefficient and if the set of rules is large, which is common to intrusion detection systems, this technique will not perform well.

Similar techniques have been employed in vulnerability assessment and attack graph generation fields, i.e., adopting a logic programming approach to detect vulnerabilities on computer hosts. MulVAL (Multihost, multistage Vulnerability Analysis) proposed by Ou et. al. (Ou et al. 2005) is an example of a vulnerability assessment tool. It uses a unification-based model for the analysis of a system's exposure to various threats. The MulVAL system comprises a Datalog implementation[1], a library of predefined predicates that model common threat effects (such as the ability to execute code, to transmit data over the network, to modify access control or users' privileges, etc.), a vulnerability scanner that generates a base of Datalog clauses which list the vulnerabilities present on the analysed host(s), and a base of clauses which describe the effects of known vulnerabilities. The MulVAL system utilises the Datalog built-in unification algorithm to carry out various types of analysis, namely: threat assessment, security policy assessment, and speculative analysis. One advantage of the MulVAL system is its ability to represent threats that result from a combination of vulnerabilities. This is a direct consequence of the use of Prolog-style unification: each vulnerability effect is represented as a Datalog clause, which can in turn be reused as a condition for another clause. Our present work is motivated by this benefit, however its approach and purpose are different. Our goal is the detection of actual occurrences of multi-step attack scenarios in analysed logs. Instead of relying on Prolog-style variables and clauses only, we use a higher-level data model, i.e., the event abstraction model proposed in (Panichprecha et al. 2006), which provides a uniform and generic representation of events on the system and network being monitored. We also represent attack scenarios as clauses, however we introduce specific features designed to take full advantage of the underlying framework's expressive power. We have implemented a dedicated unification engine, adapted to the framework and scenario model. Finally, for prototyping purposes, we define a simple concrete syntax for attack scenario specification, based on the Python programming language (Python Software Foundation 2007).

State transition techniques model attacks against a system in terms of system states and state transitions. An occurrence of an attack is identified by reaching the terminal state of a signature. Examples of such systems are IDIOT (Kumar 1995), EDL (Meier et al. 2005), LAMBDA (Cuppens and Ortalo 2000), and STAT (Ilgun et al. 1995). IDIOT and EDL use a Petri-net based modelling approach to signature specification and matching. The state transition techniques have been proved to perform well in real-time detection. However, they suffer from the complexity of the state and transition instantiation mechanism.

Scenario and signature specification languages have been typically designed to match their underlying matching techniques. In this paper, we are interested in the languages which allow the expression of multi-step misuse activities (Lindqvist and Porras 1999, Mounji 1997, Eckmann et al. 2000, Cuppens and Ortalo 2000, Michel and Mé 2001, Meier et al. 2002, 2005, Lin et al. 1998, Ning et al. 2002, Pouzol and Ducassé 2001). To name a few, P-BEST (Production-Based Expert System Toolset) (Lindqvist and Porras 1999) and STATL (State Transition

---

[1] Data log is a subset of the Prolog programming language.

Analysis Technique Language) (Eckmann et al. 2000) are widely recognised and are good examples of systems which implement the rule-based expert system techniques and the state transition techniques respectively.

P-BEST is used in several rule-based expert system namely EMERALD (Porras and Neumann 1997) and several other systems (Sebring et al. 1988, Lunt et al. 1989, Anderson et al. 1995). The P-BEST language provides syntax for expressing inference rules and responses to derived facts. It provides operators and data structures for modelling low-level operating system and network activities. Also, the language provides an interface to the C programming language. Hence, it allows scenario developers to incorporate functions written in the C programming language.

STATL is used in the STAT framework (Eckmann et al. 2000). The STATL provides syntax for modelling system states and activities which affect the states. The modelling of activities is expressed in terms of STATL's internal structures which are provided by STAT's providers and extensions. The providers and extensions provide a single-level event abstraction where audit data and network traffic are converted into STATL's built-in structure, which is in fact a struct data type in the C++ programming language.

In summaries, there are two main limitations with existing scenario specification languages. Firstly, most languages require scenario developers to specify scenario details at a low-level, e.g., system calls and network protocol-level details. Secondly, many of these languages expose their internal scenario matching mechanisms to scenario developers. For instance, the STATL requires scenario developers to specify transition types which need an intimate understanding about the transitions. Thus, the scenarios are often complex and easily prone to human error.

The multi-step scenario matching framework proposed in this work is rule-based and employs a unification algorithm. In fact, existing systems implicitly use some form of unification. EMERALD and ASAX use a forward-chaining (bottom-up) unification approach to detect misuse activities which consider event data to be facts and conclude new facts from the event data and assertion rules but do so by employing a built-in or implicit unification engine which is correspondingly inflexible and non-extensible.

Our work differs from previous systems which make use of the unification algorithm in several points. Firstly, we use the unification algorithm explicitly and thus derive the benefits of flexibility and extensibility which we demonstrate later in this paper. We have implemented the unification algorithm and adapted it to our needs. The advantage of implementing the unification algorithm is the flexibility to extend and optimise the matching engine thus addressing one of the limitations of previous work on scenario specification and matching. Secondly, we utilise the event abstraction model proposed in (Panichprecha et al. 2006) which allows our misuse scenario specification to specify scenarios using event abstraction rather than low-level events as in the previous work. Finally, our misuse-scenario specification approach does not require scenario developers to compile scenarios into binary executable format unlike EMERALD, ASAX, and ORCHIDS. Scenarios in our framework can be used by the engine right away which reduces the complexity and makes the system more human operator friendly.

OUR APPROACH: UNIFICATION-BASED SCENARIO MATCHING

The unification algorithm was first proposed by Robinson (Robinson 1965) as a mechanism for comparing two expressions and substituting variables in one expression with variables or sub-expressions from the other so that the two expressions can be tested for syntactic equivalence (Baader and Snyder 2001). This section discusses our approach to scenario matching using the unification algorithm. We first describe how events are represented in our framework, followed by a description of the operators that can be applied to scenario specifications. We then present our overall architecture and the Python-based scenario specification framework, and describe how composite scenarios can be implemented using the framework.

Event Representations

The event representation proposed in the event abstraction model (Panichprecha et al. 2006) provides a range of event representations from operating system activities, to network activities, and to application events. The event representation is built based on an object-oriented approach which comprises two object hierarchies: the Sensor Event Tree (SET) and the Abstract Event Tree (AET). The SET provides an abstraction of sensor event types derived from heterogeneous sources, e.g., audit data, application data, and network traffic. The entries from the SET are then used by the AET to represent abstract system and network activities occurring on the system or network being monitored.

In order to use this event abstraction model with the unification algorithm, we need to define additional rules to the unification algorithm. The rules are as follows:

1. Free variables can represent either objects or values of object attributes.

2. If a variable is constrained by the class of the objects it can represent, it can be instantiated with any object which belongs to that class, or one of its subclasses.

3. A free variable that represents the value of an attribute can be instantiated with either an atomic value, e.g., string, integer, or date, or with an object when appropriate. In the latter case, it can also possibly be constrained by the class of the object.

We use the modified unification algorithm with these rules to match attack signatures against events.

The Operators

We have defined the following operators to enable scenario writers to specify patterns of scenarios in several aspects, i.e., term equivalence, string patterns, and chronological order of events:

- *"=="* defines equality of two terms based on their type. For example, if the two terms are strings, the operator means the two strings must be an exact match. If one of the two terms is a variable and the other is a class, the operator produces a class constraint on the variable;

- *"!="* is a shorthand notation of the complement of ==. Note that this operator does not support event objects, due to the fact that the complement of an event object refers to all other types of event objects which can cause the number of unifiers to grow exponentially. As far as we concerned, there is no scenario that requires the application of event object complement;

- *containsPattern* defines string pattern matching using regular expression;

- *sizeGreaterThan* and *sizeLessThan* compare the lengths of two strings;

- *before* and *after* compare the timestamps of two events. Optionally a time threshold (*timeout*) can be specified.

The last two operators (before and after) are significant for multi-step scenario specifications since they allow scenario writers to specify chronological relationships between two events, which is common to multi-step scenarios. However, these two operators rely on the assumption that the timestamps are derived from a trustworthy time source and clocks of the two event sources are synchronised.

Architecture

The architecture of our system is depicted in Figure 1. The left-hand side of the figure shows components of the event abstraction model framework proposed in (Panichprecha et al. 2006). The framework comprises a set of sensor event generators, a SET persistent object store, a set of abstract event generators, and an AET persistent object store. When the framework is executed, the sensor event generators read data from heterogeneous sources and generate corresponding sensor event objects which are recorded in the SET persistent object store. Then, the abstract event generators read sensor event objects from the SET persistent object store and generate abstract event objects which are recorded in the AET persistent object store. The right-hand side of the figure shows our unification-based scenario matching system proposed in this paper. The prototype comprises three components: a set of scenarios, the unification-based scenario matching engine, and a reporting module. When our system is executed, the unification-based scenario matching engine reads the scenarios written in our Python-based scenario specification framework and unifies them with event objects from the AET persistent object store. If the unification succeeds, the unifiers are sent to the reporting module which, at this stage, prints out all values stored in the event objects.

Python-based Scenario Specification Framework

As a proof-of-concept, we have defined a Python-based scenario specification framework and have incorporated the event representations and the operators as discussed above into the framework. It is not intended that our framework introduce novel features or define new syntactic constructs. Rather, it is intended to demonstrate the expressiveness of unification for scenario matching. We avoid introducing new syntax and ``yet another scenario specification language'', but instead use the popular Python programming language (Python Software Foundation 2007) as a foundation for scenario expressions.

We have developed a set of Python APIs which provide the means to write multi-step scenarios (i.e., expressions) for our unification algorithm. Since they are built on the Python programming language, the APIs have full access to the Python language's features and other APIs if needed. In addition, our framework incorporates the APIs provided by the event abstraction framework which enable access to uniform event representations regardless of platforms or applications (Panichprecha et al. 2006).
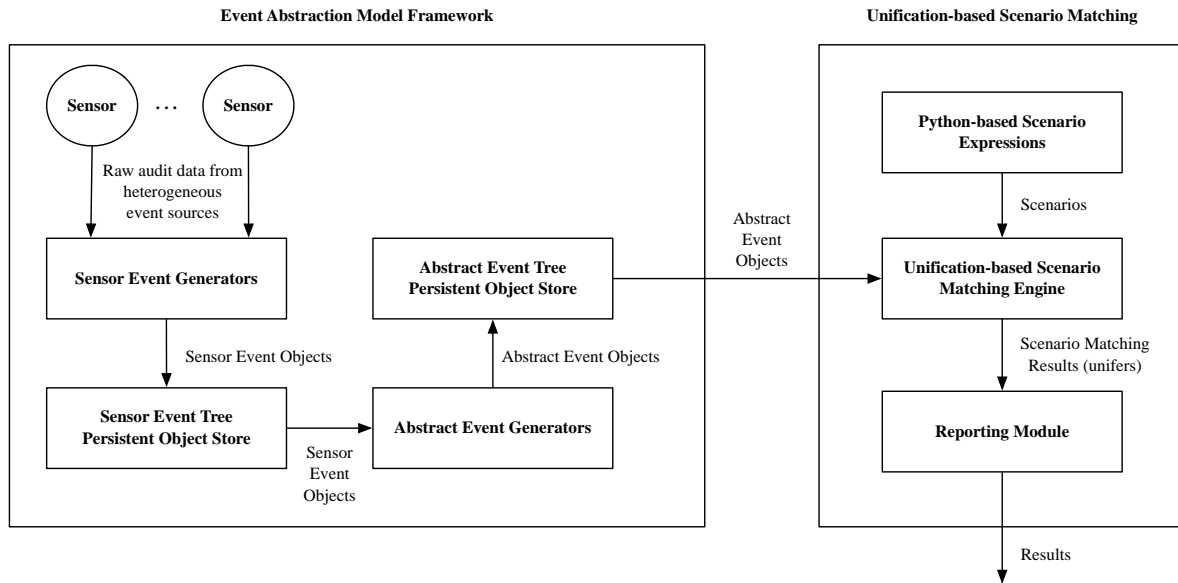
Event Abstraction Model Framework   **Unification-based Scenario Matching**

```
┌──────────────────────────────────────────────┐   ┌──────────────────────────────┐
│  ⭘ Sensor  · · ·  ⭘ Sensor                     │   │  ┌────────────────────────┐  │
│                                                │   │  │ Python-based Scenario  │  │
│         Raw audit data from                    │   │  │     Expressions        │  │
│         heterogeneous                          │   │  └────────────────────────┘  │
│         event sources                          │   │           │ Scenarios        │
│                           ┌──────────────────┐ │   │           ▼                  │
│  ┌──────────────────┐     │ Abstract Event   │ │Abstract  ┌────────────────────┐  │
│  │ Sensor Event     │     │ Tree Persistent  │ │Event  │ Unification-based   │  │
│  │ Generators       │     │ Object Store     │─┼─Objects▶│ Scenario Matching  │  │
│  └──────────────────┘     └──────────────────┘ │   │  │     Engine         │  │
│         │ Sensor Event Objects    ▲            │   │  └────────────────────┘  │
│         ▼              Abstract Event Objects  │   │    Scenario Matching │     │
│  ┌──────────────┐  ┌──────────────────┐        │   │    Results (unifers) ▼     │
│  │ Sensor Event │  │ Abstract Event   │        │   │  ┌────────────────────┐  │
│  │ Tree         │─▶│ Generators       │        │   │  │ Reporting Module   │  │
│  │ Persistent   │  └──────────────────┘        │   │  └────────────────────┘  │
│  │ Object Store │  Sensor Event Objects        │   │           │ Results        │
│  └──────────────┘                              │   │           ▼                  │
└──────────────────────────────────────────────┘   └──────────────────────────────┘
```

Figure 1: The Prototype Unification-based Multi-Step Scenario Matching

Composite Scenarios

In our Python-based scenario specification framework, a scenario can incorporate other scenarios (i.e., a composite scenario) and use them to describe attack steps in a multi-step scenario. Composite scenarios are used for two reasons: to represent abstract events and to facilitate signature development. Firstly, for abstract event representations, a scenario can represent a set of concrete events as one abstract event which can then be included by referencing it in other higher order scenarios. Secondly, for facilitating scenario development and management, the scenario composition provides re-usability of existing scenarios. This feature helps decrease time for scenario development and eases scenario management.

THE PROTOTYPE

We have developed a proof-of-concept implementation of our unification-based multi-step scenario matching framework. The scenario matching framework has been implemented in the Python programming language due to a number of benefits of the language. For instance, the Python language's built-in data structures (dictionaries and lists) facilitate the development of the unification algorithm. Also, some implementations of the Python language allow a Python program to incorporate APIs from other programming languages, e.g., the Jython interpreter allows a Python program to incorporate Java libraries (Bock et al. 2007) and the IronPython project which enables a Python program to call Microsoft .Net libraries (Microsoft Corporation 2007). The Jython interpreter provides two major benefits. Firstly, it allows a Python program to run on any platform that supports the Java Virtual Machine. Secondly, it allows our unification-based scenario matching framework to incorporate the APIs provided by the event abstraction framework (Panichprecha et al. 2006). Therefore, we have chosen the Jython interpreter for implementing our prototype. We have implemented the Python-based scenario specification framework which provides the necessary components for developing scenarios, i.e., variables, operators, and access to abstract events.

Note that it is possible to use the Prolog language to implement the scenario matching framework, since the matching mechanism used in the Prolog language is a unification algorithm. However, it will be difficult and time consuming to develop a set of Prolog programs to interface with the event abstraction framework. Also, we have little-to-no control of the unification algorithm in Prolog. In particular, we need to use the persistent object stores (object-oriented database) provided by the event abstraction model framework but the Prolog interpreter uses its own internal database, to which we have no direct access. All in all, the Python programming language allows quick development of the unification matching framework and more importantly can connect to the AET persistent object store.

CASE STUDIES AND EVALUATION

This section demonstrates the proof-of-concept implementation of our system. The prototype has been run in an experimental environment, which comprises four machines, i.e., a victim machine (running as a PXE server[2] and a mail server), a PXE-enabled client machine, a Microsoft Windows 2000 machine, and an adversary machine. All machines, except the machine running Microsoft Windows 2000, use the Linux operating system. All machines are connected to the same physical network and allocated in the same subnet.

Due to space limitations, in this paper, we demonstrate two multi-step attacks. All attacks related to the signatures are run in the experimental network, alongside random harmless traffic such as SSH sessions, HTTP traffic, etc. Related logs and network traffic were collected and parsed using SET and AET generators. The unification algorithm was then applied to the generated AET database to identify the attacks post-hoc. Although, these attacks are tested in a controlled environment, they are real attacks and they can be run in real environments.

During the course of the experiment, audit data was collected from corresponding systems and applications i.e., Apache web server logs, Unix system logs, Unix system call logs, and Microsoft Windows security logs. Also, network traffic from the experimental network was captured with tcpdump. All audit data and network traffic data was converted to event objects using the parsers and generators provided by the event abstraction model framework. From the collected data, the generators produce *24,981* sensor event objects and *25,325* abstract event objects. The objects are stored in the SET and AET persistent object store respectively. The abstract event objects from the AET persistent object store are considered to be facts and are used by our unification-based scenario matching framework.

Masqueraded Preboot Execution Environment Server Scenario

Due to the lack of a host authentication mechanism, the PXE operation is prone to at least two attacks: denial of service attacks and an adversary host masquerading as a PXE server. In the first attack, an adversary launches a number of successful DHCP handshakes, where a newly generated MAC address is used for each request, until the DHCP server's table of allocated IP addresses is full. This attack causes the PXE server to enter a state where it cannot provide IP addresses (denial of service). In the second attack, the adversary can run DHCP and TFTP services on his/her host which serves as a PXE server. By combining these two attacks, the adversary can create a masqueraded PXE server and use it to serve malicious operating system images (for example, images containing backdoors or spyware) to clients. The scenario comprises steps listed below:

1. DHCP no lease event: The error message reported by a DHCP server when the pre-allocated block of IP addresses is exhausted;

2. DHCP offer: The network event which indicates that a DHCP server is offering an IP address to a client. If the IP address of the DHCP server is not the same as the address in step 1, it signifies an anomalous event;

3. TFTP session: This event identifies TFTP communications between a TFTP server and a TFTP client. In this scenario, if the IP address of the TFTP server is not the same as in Step 1, it indicates that a PXE client downloads a bootstrap file namely "boot.msg" from the adversary machine. The download leads to retrieving an operating system image which may be preloaded with backdoors.

The scenario definition to detect this attack is shown in Figure 2. The scenario comprises four methods. Note that this is purely a code readability choice and it illustrates the seamless use of all Python language features in the scenario specification. The details of the four methods are described as follows:

- `detect`: This is the main method of the scenario. This method is executed by our unification-based scenario matching engine;

- `dhcp_no_lease`: This method detects the DHCP no lease event. Two variables *dhcpnolease* and *realDHCPServer* are instantiated with the *DhcpNoLeases* events and the address of DHCP server respectively.;

- `dhcp_offer`: This method detects a potential masqueraded DHCP server on line 14 which specifies that the address of the DHCP server is different from *realDHCPServer*. On line 13, we demonstrate the application of our after operator which specify a 2 second timeout between the DHCP offer and DHCP no leases events.;

---

[2] PXE (Preboot Execution Environment) is a hybrid of DHCP and TFTP (Intel Corporation 1999).

- `tftp_session`: This method detects a TFTP download session which corresponds to a client downloading an operating system image from the masqueraded PXE server.

```
1   class pxeattackScenario(Scenario):
2           def detect(self):
3                   self.dhcp_no_lease()
4                   self.dhcp_offer()
5                   self.tftp_session()
6
7           def dhcp_no_lease(self):
8                   Variable('dhcpnolease') == AET(DhcpNoLeases)
9                   Variable('realDHCPServer') == Variable('dhcpnolease').dhcpServerIPAddress
10
11          def def_dhcp_offer(self):
12                  Variable('dhcpoffer') == AET(DhcpOffer)
13                  Variable('dhcpoffer').eventTime == after(dhcpnolease.eventTime, 2000)
14                  Variable('dhcpoffer').serverIPAddress != Variable('realDHCPServer')
15                  Variable('fakeServer') == Variable('dhcpoffer').serverIPAddress
16
17          def tftp_session(self):
18                  Variable('tftpsession') == AET(TFTPSession)
19                  Variable('tftpsession').eventTime == after(Variable('dhcpoffer').eventTime, 2000)
20                  Variable('tftpsession').destinationAddress == Variable('fakeServer')
21                  Variable('tftprequest') == AET(TFTPReadRequest)
22                  Variable('tftprequest').destinationAddress == Varible('fakeServer')
23                  Variable('tftprequest').fileName == containsPattern("boot.msg")
```

Figure 2: Masqueraded Preboot Execution Environment Server Scenario

This scenario demonstrates an application of our Python-based scenario specification framework. The scenario also shows how to specify a sequence of events and timeout using the *after* operator. Corresponding events and their attribute values are stored in six variables. The attack scenario was successfully detected. However, two sets of events have been generated, while there is only one instance of the attack. This is caused by the fact that there are two instances of DHCP no leases events recorded by syslog with the same timestamp due to the well known 1 second resolution of syslog. This problem can be solved by either implementing a simple 'tidy-up' in the AET generators which detects exact duplicate objects or adding an expression which checks for duplicates.

Sendmail Executing a Shell Scenario

Several versions of Sendmail are vulnerable to buffer overflow attacks. In this example, we demonstrate a scenario specification which detects a buffer overflow attack in Sendmail version 8.11.6 (SecurityFocus and Zalewski 2003). The attack exploits vulnerability in the *prescan* function, where it fails to check the size of e-mail addresses in SMTP headers. We have analysed the exploit code, *sormail.c* from (SecurityFocus and Zalewski 2003), and found that the code exploits the vulnerability and executes a shell with the privilege of the user who runs Sendmail which, in most cases, is run with system administrator privileges.

The signature for this attack is shown in Figure 3. It comprises two scenarios: the scenario which detects that the Sendmail process executes a shell (Figure 3a) and the scenario which detects that Sendmail is executed (Figure 3b). When executed, Scenario 3a, on line 3, invokes Scenario 3b, where the variable `execSendmail` is instantiated with an Execute event. Note that the Execute event is an abstract event which represents program execution independent from the platform. Also, the `sendmailPID` is instantiated with the process ID of *sendmail*. Then, the expression on line 4 of Scenario 3a calls the `start_root_shell` method which instantiates the variable `startingRootShell`. Also, the `start_root_shell` method specifies two constraints on the variable where one of them specifies that the process ID of the *sendmail* must be equal to the variable `processID` which is instantiated in Scenario 3b.

This scenario demonstrates the ability of our Python-based scenario specification framework to invoke a scenario from another scenario (scenario composition). The composition is possible because the unifiers are implemented as a global Python variable. Thus, a variable instantiated in one scenario is accessible from all other scenarios. In this scenario, the variable `sendmailPID` is instantiated in Scenario 3b. Thus, the value is accessible by Scenario 3a. The attack was successfully detected with no false alarms. Since abstract events are used in the scenario, this scenario can detect other attacks which have similar behaviour, i.e., the sendmail process executing a shell.

```
1     class sendmailExecutingShell(Scenario):
2           def detect(self):
3                     executingSendmail()
4                     self.start_root_shell()
5
6           def start_root_shell(self):
7                     Variable('startingRootShell') == AET(ProcessOperationEvent)
8                     Variable('startingRootShell').processID == Varible('sendmailPID')
9                     Variable('startingRootShell').processName == comtainsPattern("sh")
```

(a) Sendmail Executing a Shell

```
1     class executingSendmail (Scenario):
2           def detect(self):
3                     Variable('execSendmail') == AET(Execute)
4                     Variable('execSendmail').newProcessName == containsPattern("sendmail")
5                     Variable('sendmailPID') == Variable('execSendmail').processID
```

(b) Executing Sendmail

Figure 3: Sendmail Executing a Shell Scenario

CONCLUSION AND FUTURE WORK

We have developed a proof-of-concept unification-based multi-step scenario matching framework. The framework uses abstract events to address the heterogeneity of event sources and aims to address the complexity of scenario specification by using a unification algorithm and a Python-based scenario specification framework. The unification algorithm and our scenario specification framework enable scenario writers to specify descriptions in terms of *what* to detect rather than *how* to detect it. By employing the unification algorithm, scenarios are easier to read, write, and understand. In our Python-based scenario specification framework, we have implemented operators and data types which are sufficient for specifying multi-step scenarios as well as single-step scenarios.

Our proof-of-concept framework has been tested with several scenarios which involve events from multiple sources. Due to space limitations, only two scenarios have been demonstrated in this paper. We have demonstrated our Python-based scenario specification framework in specifying multi-step attacks and constructing composite scenarios. The system has successfully detected all instances of the misuse activities in those scenarios.

Our future work plans are to focus on addressing time-related issues. Time related issues are very important for correlating events from heterogeneous sources and multi-step scenario specifications. In the current stage, we correlate events from heterogeneous sources based on the assumption that clock synchronisation of all the event sources is properly implemented. However, this is not always the case. Although clock synchronisation mechanisms are widely available, they are often not properly implemented. In order to address these issues, we are planning to look into employing *time tolerance*. The principle of time tolerance is to use a time range instead of a single point of time. By applying time tolerance to an event, the timestamp is converted into a time range. In addition to time tolerance, our future work will incorporate a standard reporting format, i.e., Intrusion Detection Message Exchange Format (IDMEF) (Debar et al. 2007), into the reporting module.

REFERENCES

Jonathon Abbott, Jim Bell, Andrew Clark, Olivier De Vel, and George Mohay. (2006) Automated Recognition of Event Scenarios for Digital Forensics. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing,* Dijon, France.

Debra Anderson, Thane Frivold, and Alfonso Valdes. (1995) Next-generation Intrusion Detection Expert System (NIDES): A summary. Technical Report SRI-CSL-95-07, SRI International.

F. Baader and W. Snyder. (2001) Unification theory. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, pages 447–533. Elsevier Science Publishers.

Finn Bock, Barry Warsaw, Jim Hugunin, and The Jython Development Team. The jython project., Accessed April 2007.

Nathan Carey, George Mohay, and Andrew Clark. (2003) Attack Signature Matching and Discovery in Systems Employing Heterogeneous IDS. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)*, pages 245–254, Las Vegas, Nevada.

Frédéric Cuppens and Rodolphe Ortalo. (2000) LAMBDA: A Language to Model a Database for Detection of Attacks. In *Proceedings of Recent Advances in Intrusion Detection, 3rd International Symposium, RAID 2000*, volume 1907 of *Lecture Notes in Computer Science*, pages 197–216, Toulouse, France, Springer. ISBN 3-540-41085-6.

H. Debar, D. Curry, and B. Feinstein. (2007) The Intrusion Detection Message Exchange Format (IDMEF). Request for Comments (RFC): 4765.

S.T. Eckmann, G. Vigna, and R.A. Kemmerer. (2000) STATL: An Attack Language for State-based Intrusion Detection. In *Proceedings of the ACM Workshop on Intrusion Detection Systems*, Athens, Greece.

K. Ilgun, R.A. Kemmerer, and P.A. Porras. (1995) State Transition Analysis: A rule-based intrusion detection system. *IEEE Transactions on Software Engineering*, 21(3):181–199.

Intel Corporation. (1999) Preboot execution environment (PXE) specification version 2.1, September 1999.

Sandeep Kumar (1995). *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University.

Jia-Ling Lin, X. Sean Wang, and Sushil Jajodia. (1998) Abstraction-based Misuse Detection: High-level Specifications and Adaptable Strategies. In *The Eleventh Computer Security Foundations Workshop*, pages 190–201, Rockport, MA.

Ulf Lindqvist and Phillip A Porras. (1999) Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California. IEEE Computer Society Press, Los Alamitos, California.

T. F. Lunt, R. Jagannathan, R. Lee, A. Whitehurst, and S. Listgarten. (1989) Knowledge-based Intrusion Detection. In *Proceedings of the Annual AI Systems in Government Conference*, pages 102–107, Washington, D.C., IEEE Computer Society Press.

Michael Meier, Niels Bischof, and Thomas Holz. (2002) SHEDEL-A Simple Hierarchical Event Description Language for Specifying Attack Signatures. In *Proceedings of the Security in the Information Society: Visions and Perspectives, IFIP TC11 17th International Conference on Information Security (SEC2002)*, pages 559–572.

Michael Meier, Sebastian Schmerl, and Hartmut Koenig. (2005) Improving the efficiency of misuse detection. In *Proceedings of the Second Conference on Detection of Intrusion and Malware and Vulnerability Assessment (DIMVA2005)*. Springer Verlag.

Cédric Michel and Ludovic Mé. (2001) ADELE: An Attack Description Language for Knowledge-based Intrusion Detection. In *Proceedings of the 16th International Conference on Information Security (IFIP/SEC 2001)*, pages 353–365.

Microsoft Corporation (2007) IronPython, URL http://www.ironpython.com.

Benjamin Morin, Ludovic Mé, Hervé Debar, and Mireille Ducassé. (2002) M2D2: A Formal Data Model for IDS Alert Correlation. In *Proceedings of Recent Advances in Intrusion Detection, 5 t h International Symposium, RAID 2002*, volume 2516 of *Lecture Notes in Computer Science*, pages 115–137, Zurich, Switzerland, Springer. ISBN 3-540-00020-8.

Abdelaziz Mounji.(1997) *Languages and Tools for Rule-Based Distributed Intrusion Detection*. PhD thesis, University of Namur, Belgium.

Peng Ning, Sushil Jajodia, and X. Sean Wang. (2002) Design and Implementation of a Decentralized Prototype System for Detecting Distributed Attacks. *Computer Communications, Special Issue on Intrusion Detection Systems*, 25(15): 1374–1391.

Julien Olivain and Jean Goubault-Larrecq. (2005) The ORCHIDS Intrusion Detection Tool. In Kousha Etessami and Sriram Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 286–290, Edinburgh, Scotland, UK, Springer.

Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. (2005) MulVAL: A Logic-based Network Security Analyzer. In *Proceedings the 14th USENIX Security Symposium*, Baltimore, Maryland.

Sorot Panichprecha, Jacob Zimmermann, George Mohay, and Andrew Clark. (2006) An Event Abstraction Model for Signature-based Intrusion Detection Systems. In *Proceedings of the 1st Information Security and Computer Forensics (ISCF-2006)*, pages 151–162, Chennai, India. Allied Publishers Pvt. Ltd.

Phillip A. Porras and Peter G. Neumann. (1997) EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings of the 10th National Information Systems Security Conference*, pages 353–365, Baltimore, Maryland. National Institute of Standards and Technology/National Computer Security Center.

Jean-Philippe Pouzol and Mireille Ducassé. (2001) From Declarative Signatures to Misuse IDS. In *Proceedings of Recent Advances in Intrusion Detection, 4th International Symposium, RAID 2001*, volume 2212 of *Lecture Notes in Computer Science*, pages 1–21, Davis, CA, USA. Springer. ISBN 3-540-42702-3.

Python Software Foundation. (2007) Python programming language, URL http://www.python.org, Accessed April 2007.

J. A. Robinson. (1965) A Machine-oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12(1):23–41. ISSN 0004-5411.

Muriel Roger and Jean Goubault-Larrecq. (2001) Log Auditing Through Model Checking. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 220–236, Cape Breton, Nova Scotia, Canada. IEEE Computer Society Press.

M. M. Sebring, E. Shellhouse, (1988) M. E. Hanna, and R. A. Whitehurst. Expert Systems in Intrusion Detection: A case study. In *Proceedings of the 11th national Computer Security Conference*, pages 74–81. National Institute of Standards and Technology/National computer Security Center.

SecurityFocus and Michal Zalewski. (2003) Sendmail address prescan memory corruption vulnerability, Bugtraq id: 7230. URL http://www.securityfocus.com/bid/7230.

Jiahai Yang, Peng Ning, X. Sean Wang, and Sushil Jajodia. (2000) CARDS: A Distributed System for Detecting Coordinated Attacks. In *Proceedings of IFIP TC11 the Sixteenth Annual Working Conference on Information Security*, pages 171–180.

COPYRIGHT