A Software Architecture to Support Misuse Intrusion Detection.*

Sandeep Kumar Eugene H. Spafford

The *COAST* Project
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398
{kumar,spaf}@cs.purdue.edu

Keywords: intrusion detection, misuse, anomaly.

June 16, 1995

Abstract

Misuse intrusion detection has traditionally been understood in the literature as the detection of specific, precisely representable techniques of computer system abuse. Pattern matching is well disposed to the representation and detection of such abuse. Each specific method of abuse can be represented as a pattern and several such patterns can be matched simultaneously against the audit logs generated by the operating system kernel. Using relatively high level patterns to specify computer system abuse relieves the pattern writer from having to understand and encode the intricacies of pattern matching into a misuse detector. Patterns represent a declarative way of specifying what needs to be detected, instead of specifying how it should be detected. We have devised a model of matching based on Colored Petri Nets specifically targeted for misuse intrusion detection. In this paper we present a software architecture for structuring a pattern matching solution to misuse intrusion detection. In the context of an object oriented language used for the prototype implementation we describe the abstract classes encapsulating generic functionality and the interrelationships between the classes.

1 Introduction

Intrusion detection is an important monitoring technique in computer security aimed at the detection of security breaches that cannot be easily prevented by access and information flow control techniques. These breaches can be a result of software bugs, failure of the authentication module, improper computer system administration, etc. Intrusion detection has historically been studied as two sub-topics: *anomaly detection* and *misuse detection*. Anomaly detection is based on the premise that many intrusions appear as anomalies on ordinary or specially devised computer system performance metrics such as I/O activity, CPU usage, etc. By maintaining profiles of these metrics for different subject classes, for example individual users, groups of users, or programs and monitoring for large variations on them, many intrusions can be detected. Misuse intrusion detection has traditionally been understood in the literature as the detection of specific, precisely representable techniques of computer system abuse. For example, the detection of the Internet worm attack by monitoring for its exploitation of the `fingerd` and `sendmail` bugs [Spa89] would fall under misuse detection.

Several approaches to misuse detection have been tried in the past. They include language based approaches to represent and detect intrusions, such as ASAX [HCMM92]; developing an

*This work was funded by the Division of INFOSEC Computer Science, Department of Defense.

API[1] for the same purpose, such as in Stalker [Sma95]; using expert systems to encode intrusions such as in MIDAS [SSHW88], Haystack [Sma88], and NIDX [BK88]; and high level state machines to encode and match signatures[2] such as STAT [PK92] and USTAT [Ilg92]. We proposed using a pattern matching approach to the representation and detection of intrusion signatures [KS94b]. This approach resulted from a study of a large number of common intrusions with the aim of representing them as patterns to be matched against the audit trail [KS94a]. The signatures were also classified into categories based on their theoretical tractability of detection [Kum95]. We consider the following to be advantages unique to our model of pattern represention and matching.

- Sequencing and other ordering constraints on events can be represented in a direct manner. Systems that use expert system rules to encode misuse activity specify ordering constraints by directly specifying temporal relationships between facts in rule antecedents. This makes the Rete match procedure [For82] of determining the eligible production rules for firing, inefficient. STAT [PK92] and USTAT [Ilg92] permit the specification of state transition diagrams to represent misuse activity but their transition events may be high level actions that need not correspond directly to system generated events. ASAX [HCMM92] is the closest to our approach but it is less declarative. In specifying patterns in their rule based language RUSSELL, one must explicitly encode the order of rules that are triggered at every step. While ASAX tends to be a mechanism for general purpose audit trail analysis, our effort is a combination of mechanism and policy. The features provided in our work are closely tied to the intrusion characteristics we are trying to detect.
- Our model provides for a fine grained specification of a successful match. The use of pattern invariants (to be explained later) allows the pattern writer to encode patterns that do not need to rely on primitives built into the matching procedure to manage the matching, for example to clean up partial matches once it is determined that they will never match. This frees the matching subsystem from having to provide a complete set of such primitives and, in the process, couple the semantics of pattern matching with the semantics of the primitives.

Our method also has the following benefits but these are not necessarily a consequence of our approach.

Portability. Intrusion signatures can be moved across sites without rewriting them to accommodate fine differences in each vendor's implementation of the audit trail. Because pattern specifications are declarative, a standardized representation of patterns enables them to be exchanged between users running variants of the same flavor of operating system, with syntactically differing audit trail formats.

Declarative Specification. Patterns representing intrusion signatures can be specified by defining what needs to be matched, not how it is matched. That is, the pattern is not encoded by the signature writer as code that explicity performs the matching. This cleanly separates the matching from the specification of what needs to be matched.

In this paper we describe our implementation of the model that was presented in [KS94b]. We have used C++ [Str91] as the programming language for the implementation of the prototype. The prototype runs under the Solaris 2.3 operating system and uses the Sun BSM [Sun93] audit trail as its input to detect intrusions. The programming techniques and language features we have used for the implementation are applicable to other programming languages as well. Our implementation is directed at providing a set of integrated classes that can be used in an application program to implement a generic misuse intrusion detector. The implementation also suggests a possible way of structuring classes encapsulating generic functionality and the interrelationships between the classes to design any misuse detector. The paper also describes that structure.

[1] Application Programming Interface, i.e., a set of library function calls employed for representing and detecting intrusions.

[2] We use the terms intrusion signature and intrusion pattern synonymously.

Our choice of the language was dictated by the free availability of quality implementations of C++, our familiarity with it and the linguistic support provided in it to write modular programs. The set of integrated classes we have developed can be programmed in many other object oriented languages as well because no properties specific to C++ have been assumed or used. We only exploit the language's encapsulation and data abstraction properties. We use the word *class* in a generic sense and the corresponding notion from many other languages can be substituted here.

2 Our Approach

The model of pattern representation and detection on which the implementation is based was described in [KS94b]. Briefly, each intrusion signature is represented as a specialized graph in this model. These graphs are an adaptation of Colored Petri Nets described by Jensen [Jen92] with guards defining the context in which signatures are considered matched. Vertices in the graph represent system states. The pattern represents the relationship among events and their context that forms the crux of a successful intrusion or its attempt. Patterns may have pre-conditions and post-actions associated with them. A pattern pre-condition is a logical expression that is evaluated at the time the pattern springs into existence. It can also be used to set up state that may be used later by the pattern. Post-actions are performed whenever the pattern is matched successfully. For example, it might be desirable to raise the audit level of a user if he fails a certain number of login attempts within a specified time duration. This can be expressed as a post-action. Patterns may also include invariants to specify that another pattern cannot appear in the input stream while it is being matched. If a pattern is regarded as a set of event sequences P that it matches, and an invariant is regarded as another set of event sequences I that it matches, then a pattern with an invariant specification corresponds to the set $P \wedge \overline{I}$. A pattern can have more than one invariant. That corresponds to $P \wedge \overline{I_1} \wedge \cdots \wedge \overline{I_n}$. Invariants are needed to specify cases when it is no longer useful to continue a pattern match. For example, a pattern that matches process startups and records all file accesses by the process may require an invariant that specifies that matching be discontinued once the process has exited. From the practical viewpoint of specifying intrusion patterns, invariants usually result in more efficient matching rather than adding functionality to the pattern specification.

As a concrete example of a pattern, consider the monitoring of Clarke-Wilson [CW89] integrity triples in a computer system using the system generated audit trail. Clarke-Wilson triples are devised to ensure the integrity of important data and specify that only authorized programs running as specific user ids are permitted to write to files whose integrity must be preserved. This is similar to the maintenance of the integrity of the password file on UNIX systems by allowing only some programs, like `chfn`[3] to alter it.

One pattern that might be used for this purpose is formed by a sequence of two sub-signatures: (1) that matches the creation of a process and (2) that matches any process writing to a file. By appropriately specifying that the created process is the same as the one that writes, and retrieving the user id, the program name, and the file name from the context of the match, Clarke-Wilson integrity triples can be monitored. See figure 1 for a pictorial representation of the signature.

The implementation of this model can be broken down into the following sub-problems:

1. The external representation of signatures. That is, how does the signature writer encode signatures for use in matching.
2. The interface to the event source. In our example it would be the interface to the C2 audit trail.
3. Dispatching the events (audit records) to the signatures and the matching algorithms used for matching.

[3] `chfn` is used to change information about users which is stored in a well-known file, `/etc/passwd`.

A PROGRAM STARTS UP	A PROCESS WRITES TO A FILE
PR = this program's name PID = this process's pid	F = this file's name PID' = this process's pid

Context: $PID = PID' \wedge$ Clarke-Wilson access triples do not permit PR running as user id PID to write to file F.

Figure 1: Monitoring Clarke-Wilson triples as a pattern match.

These issues are discussed in the next section. In addition to solving these requirements, our implementation is designed to simplify the incorporation of the following:

- The ability to create signatures and to destroy them dynamically, as matching proceeds.
- The ability to partition and distribute signatures across different machines for improving performance.
- The ability to prioritize matching of some patterns over others.
- The ability to handle multiple event streams within the same detector without the need to coalesce the event streams into a single event stream.

We describe our design in the next section and show how the library classes implement the design.

3 Overall Architecture

The library consists of several classes, each encapsulating a logically different functionality. An application program that uses the library includes appropriate header files and links in the library.

The external representation of signatures (sub-problem 1) is done using a straightforward representation syntax that directly reflects the structure of their graph. These specifications can be stored in a file or maintained as program strings. When a signature is instantiated in an application, a library provided routine (a Server class member function) is called that compiles the signature description to generate code that realizes the signature. This code is then dynamically linked to the application program and pattern matching for that signature is initiated. The application also instantiates a server for each type of event stream used for matching. Events are totally encapsulated inside the server object (sub-problem 2) and are only used inside signature descriptions. As signature descriptions are compiled they are added to the server queue. The server accesses and dispatches events to the patterns on its queue in some policy specifiable order (sub-problem 3).

The application structure is explained below which gives an overall view of the application. Section 3.2 looks at the structure of events. Section 3.3 explains the structure of the server itself in detail and its relationship to the patterns that are instantiated by the application.

3.1 Application Structure

As an example application structure, consider matching the pattern described in figure 1. This may look as shown below.

```
//file application.C
1    #include "C2_Server.h"
2
3    int main()
4    {
5       C2_Server S;
6       C2_Pattern *p1 = S.parse_file("CW"); //read signature from "CW"
7
```

```
8       /* duplicate a thread of control if necessary. run() doesn't return */
9       S.run();
10
11      return(1);
12  }
```

The application program makes use of a C2_Server object. The server object understands the layout of events and the event types that can be legally used in a signature definition. C2_Server also knows how to access events, in this case from the audit trail, and how to dispatch them to the signatures that are registered with it. The server is also responsible for parsing signature descriptions and can check it for correctness because it understands the data format of the events. The call to the server member function parse_file reads, compiles, and registers a new pattern with the server object. When the server object member function S.run() is called, it starts reading events and dispatching them. This consumes one thread of control as S.run() never returns. The server is responsible for implementing concurrency control among its member functions to ensure that concurrent calls to its public member functions do not corrupt its internal state. Our implementation uses the idea of monitors [Hoa74] to ensure this. The pattern description contained in file *CW* looks as shown in listing 1 below. The pattern is written to match against the Sun BSM [Sun93] audit trail.

```
//file patterns-ip
1   pattern CW "Clarke Wilson Monitoring Triples" priority 10
2     int PID, EUID; /* token local variables. may be initialized. */
3     str PROG, FILE;
```

PROG is a token local variable that stores the program name corresponding to the process id PID, FILE stores the file name that PROG opens for writing. EUID stores the effective user id of PROG.

```
4     state start, after_exec, violation;
5     post_action {
6      printf("CW violated for file %s, PID %d, EUID %d\n", FILE, PID, EUID);
7     }
```

The post action is code that is executed when the pattern is successfully matched.

```
8     neg invariant first_inv
9       state start_inv, final;
10
11      trans exit(EXIT)
12        <- start_inv;
13        -> final;
14        |_ { PID = this[PID]; }
15      end exit;
16    end first_inv;
```

The invariant specifies the removal of partial matches once a process has exited. What follows is the pattern description. The pattern matches all EXECVE records to monitor the creation of all processes in the system. Once a process creation is matched, the pattern further attempts to match all possible ways in which the process could modify a file. These could be:

- Open a file to read and create it if it doesn't exist. Or, open a file to read and truncate it if it exists.....and so on for all the other valid audit record types involving an open that might change the file. These are handled in transition mod1.
- Delete a file. This is handled in transition mod12.

```
17    trans exec(EXECVE) /* EXECVE is the event type of the transition */
```

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.