In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, California, May 9–12, 1999.

Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST)*

Ulf Lindqvist
Department of Computer Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
ulfl@ce.chalmers.se

Phillip A. Porras
Computer Science Laboratory
SRI International
333 Ravenswood Ave, Menlo Park, CA 94025
porras@csl.sri.com

Abstract

This paper describes an expert system development toolset called the Production-Based Expert System Toolset (P-BEST) and how it is employed in the development of a modern generic signature-analysis engine for computer and network misuse detection. For more than a decade, earlier versions of P-BEST have been used in intrusion detection research and in the development of some of the most well-known intrusion detection systems, but this is the first time the principles and language of P-BEST are described to a wide audience. We present rule sets for detecting subversion methods against which there are few defenses—specifically, SYN flooding and buffer overruns—and provide performance measurements. Together, these examples and performance measurements indicate that P-BEST-based expert systems are well suited for real-time misuse detection in contemporary computing environments. In addition, the simplicity of the P-BEST language and its close integration with the C programming language makes it easy to use while still being very powerful and flexible.

1. Introduction

Intrusion detection components analyze system and user operations in computer and network systems in search of activity considered undesirable from a security perspective. Data sources for intrusion detection may include audit trails produced by an operating system, or network traffic flowing between systems, or application logs, or data collected from system probes (e.g., file system alteration monitors). The collected data may be stored for batch-mode analysis or immediately analyzed in real-time.

For the most part, the various strategies for intrusion detection are not unique to the field, but are rather derived from applications established by other fields: knowledge-based expert systems, pattern recognition algorithms, statistical profiling techniques, neural networks, Bayesian statistics, information retrieval algorithms, state-transition models, Petri-net techniques, and so forth. Among the more widely used strategies proposed early within the intrusion detection community are signature-based analyses.

Intuitively, we describe a signature-based intrusion-detection component as an algorithm with which we specify the characteristics of malicious behavior and then monitor an event stream for activity that maps to the target behavior. Various signature-based systems have been developed, ranging from simple (but efficient) pattern-matching systems to more sophisticated algorithms that employ more general directed reasoning systems such as rule-based expert systems. In this paper, we describe in detail the principles and language of one forward-chaining rule-based expert system construction toolset called P-BEST (Production-Based Expert System Toolset), which has been continually applied to intrusion detection applications for more than a decade, but never before widely presented in this level of detail.

By using a general expert system, we can describe the behavior of our signature-based intrusion-detection component within an established theoretical framework. This choice also facilitates the evolution of the component, because new rules can be added without changing existing rules and without creating any undesired dependency. Traditional reasons for not choosing an expert system are related to low performance, difficult integration with other program components, and language complexity. However, in this paper we show that P-BEST is sufficiently

*The work presented in this paper is currently funded by the Information Technology Office, Defense Advanced Research Projects Agency (DARPA/ITO), under contract number F30602-96-C-0294. U. Lindqvist participated in this work while visiting SRI International as an international fellow and his work was partly funded by The Swedish National Board for Industrial and Technical Development (NUTEK), under project number P10435.

fast for real-time detection of currently widely used attack methods—SYN flooding and buffer overruns—against which systems usually have no defense mechanisms. We also show that P-BEST provides exceptional interoperability with native operating system libraries, and is easily integrated into a larger software framework for distributed anomaly and misuse detection. We also argue that while the production rule language is powerful, it remains easy to use for beginners.

2. Monitoring misuse through expert systems

Expert systems provide strategies and mechanisms for processing facts regarding the state of a given environment, and deriving logical inferences from these facts. With respect to intrusion detection, a fact maps to an event that is recorded and evaluated by the expert system. This process of fact evaluation leading to the assertion of a new derived fact or conclusion is referred to as *modus ponens*, which states that given $(p \Rightarrow q)$ and p we deduce q. Systems that iteratively apply modus ponens under a bottom-up reasoning strategy (from evidence evaluation to conclusion) are referred to as *forward-chaining* systems. Forward-chaining expert systems are well-suited for reasoning about activity within an event stream. A forward-chaining rule-based system is data-driven: each fact asserted may satisfy the conditions under which new facts or conclusions are derived. Alternatively, *backward-chaining* systems employ the reverse strategy; starting from a proposed hypothesis they proceed to collect supportive evidence. Backward-chaining systems are typically applied to problems of diagnosis, whereas forward-chaining strategies dominate systems involving prognosis, monitoring, and control applications.

Using a forward-chaining rule-based system, one may establish a chain of rules, or *rule set*, with which a series of asserted facts may lead the system to deduce that a targeted multistep scenario has occurred. Within an intrusion detection system, event records are asserted as facts and evaluated against penetration rule sets. As individual rules are evaluated against facts and satisfied, the individual event records provide a trail of reasoning that allows the user to analyze the evidence of malicious activity in isolation from the full event stream. In this section, we will discuss the basic elements of forward-chaining rule-based systems, and provide an overview of the P-BEST expert system and its language.

2.1. Components of forward-chaining systems

The underlying strategy of a forward-chaining reasoning system involves the atomic evaluation of each fact presented to the system against conditional expressions that, when satisfied by the arguments of a fact, establish new derived facts or conclusions. In this context, a *fact* is a statement that is asserted into the system and whose validity is accepted (for example, "smoke is present"). Facts are often implemented as attributes and values that represent the state of the environment to which the expert system is applied. A *rule* is an inference formula of the form $\phi_1, \ldots, \phi_n$ *infer* ψ. Inference formulae can be alternatively expressed as *production rules*, such as `IF ... THEN ...`. Production rules are the basic elements through which an expert system is programmed to interpret and discover meaning from environmental signals that it receives, as in

`IF` *smoke is present* `THEN` *fire is near*.

A production rule consists of two parts, the *antecedent* (or conditional part, left-hand side) and the *consequent* (or right-hand side) as shown in Figure 1. When the *conditions* (predicate expressions) in the antecedent are satisfied, the rule is *activated*. The logical component through which an expert system evaluates a fact against the production rules is referred to as the *inference engine*. As an antecedent is found to be satisfied by the attributes of a fact, the consequent of the rule is asserted to hold, and the rule is said to have *fired*. Expert systems might additionally allow the inference engine to initiate action within the consequent, for example:

`IF` *fire is near* `THEN` **initiate** *sprinkler*.

Abstractly, the assertion of action, such as the initiation of a response, based on a fact derived from an inference engine is placed within the purview of a *decision engine*, though in practice inference and response may be merged.

$$\begin{array}{l}
\texttt{IF} \\
\quad \left.\begin{array}{l} condition_1 \\ condition_2 \\ \quad\vdots \end{array}\right\} \text{antecedent} \\
\texttt{THEN} \\
\quad \left.\begin{array}{l} derived\_fact_1 \\ derived\_fact_2 \\ \quad\vdots \end{array}\right\} \text{consequent}
\end{array}$$

Figure 1. Production rule structure.

The collection of facts available to the system at any point in time is called the *factbase* (or working memory) of the system. The collection of rules is called the *knowledge base* (or production memory). Although separation of data (facts) from knowledge (rules) is an important abstraction within rule-based expert systems, some texts use the terms more loosely and consider the factbase to be part of the knowledge base. Another important abstraction is the separation of knowledge from the inference engine. In practice, an inference engine, also known as an expert system *shell*, provides several advantages over a one-of-a-kind system written in a procedural language. In particular, a knowledge-independent shell can be used to develop expert systems for many different knowledge domains. The knowledge in the expert system can also be incrementally extended by adding new rules, as opposed to implementing large portions of the decision process all at once. Next, we present the principles and language of P-BEST, a construction toolset for building customized inference engines, and discuss its applicability to intrusion detection.

2.2. An overview of P-BEST

The Production-Based Expert System Toolset (P-BEST) was originally written by Alan Whitehurst, and employed in the Multics Intrusion Detection and Alerting System (MIDAS) [18], which performed misuse detection on the National Computer Security Center's Internet-connected mainframe, Dockmaster. P-BEST was later enhanced at SRI by Whitehurst, and later by Fred Gilham, and was employed in an early version of the Intrusion Detection Expert Systems (IDES) [14], and later Next-Generation IDES (NIDES) [1]. See Section 3 for details on the application of P-BEST on these systems.

The P-BEST toolset consists of a rule translator, a library of runtime routines, and a set of garbage collection routines. When using P-BEST, rules and facts are written in the P-BEST production rule specification language. The rule translator, *pbcc*, is then used to translate the specification into a C language expert system program. This expert system can then be compiled into either of two forms: a stand-alone self-contained executable program or a set of library routines that implement the core P-BEST inference engine, and which can be linked to a larger software framework. P-BEST has several features that make it well-suited for the type of application described in this paper:

- The P-BEST language is small and relatively intuitive to use and extend.

- It is easily applied to a variety of problem domains. P-BEST provides a general-purpose forward-chaining inference engine that can be targeted to a specific application domain. P-BEST does not inherently depend on the structure of the input data stream or the inference objectives of the application that employs it.

- By using translation instead of interpretation of rules, P-BEST can be used to build expert systems for performance-demanding applications. A pre-compiled expert system, rather than an expert system interpreter, provides a significant advantage in performing real-time event analysis.

- Pre-compilation also allows P-BEST components to be integrated well into larger program frameworks, and is easily called from, and can call out to, other C libraries. Arbitrary C functions can be called from the antecedent or consequent of any P-BEST rule. Thus, it is possible to write powerful rules without adding unnecessary complexity to the P-BEST language.

2.3. The P-BEST language

P-BEST provides a production rule language from which users may specify the inference formula for reasoning and acting upon facts asserted into its factbase from external sources or derived from the satisfaction of other production rules. This section provides a brief overview of the principle elements of this language, with common examples of its usage. The language overview provides the reader with a primer for understanding several examples of intrusion detection rules later in this paper.

In P-BEST, the structure of a fact is specified by the user through a template definition referred to as a pattern type or *ptype*. For example, to define a ptype named *event* that consists of the four fields *event_type* (an integer), *return_code* (an integer), *username* (a string), and *hostname* (a string), we define the fact template as in Figure 2. Facts from such a ptype definition could be constructed through the monitoring of audit records and asserted into the factbase for evaluation against the available production rules.

```
ptype[event event_type:int,
            return_code:int,
            username:string,
            hostname:string]
```

Figure 2. An example of a ptype declaration.

Fact evaluation is performed by the P-BEST inference engine, where the attributes of the fact are mapped against the predicate expression(s) of each rule antecedent. For example, we may want to determine whether the asserted fact represents an unsuccessful login attempt, which we shall refer to as e. To express this criterion using a mathematical notation style, we can form the statement in Equation 1.

Here, S represents the set of all facts known to the P-BEST factbase, and within which a production rule antecedent postulates the existence of a fact e that satisfies specific properties. In the P-BEST language, the statement in Equation 1 placed in the antecedent of a rule would be written as in Figure 3.

The term `e:event` allows one to assign an *alias* `e` to one fact (of possibly several) that satisfies the antecedent for the duration of the rule. The plus (`+`) sign after the opening bracket represents an existential quantifier that allows the rule to check for any fact that satisfies the conditions of the antecedent. Alternatively, a minus (`-`) sign searches for

$$\boxed{\begin{aligned}&\Big(\exists e\Big)\Big((e \in S) \wedge \text{event}(e) \wedge \\ &\qquad (e_{event\_type} = login) \wedge \\ &\qquad (e_{return\_code} = bad\_password)\Big) \quad (1)\end{aligned}}$$

```
[+e:event|event_type == login,
          return_code == BAD_PASSWORD]
```

Figure 3. An example of fact matching.

cases where no fact in the factbase satisfies the conditions of the antecedent. For example,

```
[-event|username == "GoodGuy"]
```

evaluates to true if there is no event in the factbase that has been asserted on behalf of "GoodGuy."

The plus and minus tests have corresponding assert and delete actions that can appear in the consequent of a rule. For example, to assert a new fact of ptype `bad_login` and give its fields initial values, we can write

```
[+bad_login|username = e.username,
            hostname = e.hostname]
```

To be deleted from the factbase, a fact must be matched and given an alias in the antecedent before it can be deleted in the consequent. This is illustrated in the example of a complete rule named `Bad_Login` in Figure 4.

```
1   rule[Bad_Login(#10;*):
2      [+e:event| event_type == login,
3                 return_code == BAD_PASSWORD]
4   ==>
5      [+bad_login| username = e.username,
6                   hostname = e.hostname]
7      [-|e]
8      [!|printf("Bad login for user %s from \
9         host %s\n", e.username, e.hostname)]
10  ]
```

Figure 4. An example of a rule declaration.

The `Bad_Login` rule in Figure 4 also demonstrates how the evaluation of an asserted fact can be used to derive subsequent facts that may themselves drive new inferences. That is, in the above rule, should a login event be encountered with a return code of `BAD_PASSWORD`, the rule creates a new fact of ptype `bad_login`, which saves the username and hostname of the event; the rule also destroys the event fact `e` from the factbase. Using a mathematical notation, we can represent this state transition in our factbase from S to a desired new state S' as in Equation 2 (this excludes lines 8 and 9 in Figure 4).

Within parentheses after the rule name (line 1), there is a semicolon-separated list of options. The option `#10` means that this rule is given a ranking (priority) of 10. Priorities allow one to specify well-defined orders in the sequences for rule evaluation, and are primarily used for rules required to be evaluated first for initialization purposes, or that must be evaluated last to perform garbage collection. The star option (`*`) indicates that the rule is repeatable, that is, the rule is allowed to fire repeatedly even if no other rule is fired in between. Thus, a key function of the consequent is to alter the state of the factbase such that the antecedent is not satisfied indefinitely (e.g., the consequent may mark or remove a fact). The arrow delimiter (`==>`) separates the antecedent and the consequent (line 4).

The `[!|...]` clause (line 8) within the consequent illustrates how the P-BEST inference engine may call out to native C functions should action be warranted when the antecedent is evaluated to true. Both inference and action can be taken directly within the P-BEST inference engine. P-BEST recognizes most of the standard library C functions, which may be invoked directly via the `[!|...]` clause, and which may refer to ptype attributes directly. User-defined C functions and auxiliary variables may also be invoked and referenced, respectively. To do this, we must declare our intentions to reference C variables and functions using the P-BEST external type declaration mechanism *xtype*. For example, the following external declarations will allow P-BEST to recognize a user-defined C function called *native_probe()* returning an integer and an integer variable *end_of_stream* as follows:

```
xtype [native_probe: intfunc]
xtype [end_of_stream: int]
```

We can then employ our native C routine and variable directly in a P-BEST production rule, as illustrated in Figure 5. The antecedent `[?|...]` clause (line 3) is a query clause used to evaluate conditional requirements. This rule will check to see whether the `end_of_stream` variable has been set to `1`, and if not, it will set the variable to the return code of the function `native_probe()` (line 5), which is invoked in the consequent. This *native_probe()* could, for example, provide an interface to the host operating system that allows the expert system to retrieve application records, which it may then assert as facts in the factbase. The rule also gives an example (line 6) of how a field in an existing fact can be modified; in this case, the field `rec_cnt` of the fact `counter`, aliased in the antecedent, is incremented by 1.

```
1   rule[get_native_record(-99;*):
2      [+c:counter]
3      [?|'end_of_stream != 1]
4   ==>
5      [!|'end_of_stream = native_probe()]
6      [/c|rec_cnt += 1]
7   ]
```

Figure 5. Example usage of external C types.

To further improve the performance of the expert system,

$$\boxed{\frac{\Big(\exists e\Big)\Big((e \in S) \wedge \text{event}(e) \wedge (e_{event\_type} = login) \wedge (e_{return\_code} = bad\_password)\Big)}{\vdash \Big(S' = S - \{e\} \ \bigcup \ \{ \text{ bad\_login}(b) \ \mid \ (b_{username} = e_{username}) \wedge (b_{hostname} = e_{hostname} \ ) \ \}\Big)} \qquad (2)}$$

rules can be disabled and enabled dynamically through actions in the consequents of rules. A rule can even disable itself, which means that it can fire once, at most, unless enabled again by another rule. To disable a rule, we can put the following action in a consequent:

```
[-#rulename]
```

To enable a rule, we can change the minus sign in the above statement to a plus sign. In addition, a rule can be declared as disabled from start by adding a single minus sign to the list of options after the rule name, for example:

```
rule[rulename(#10;*;-):
```

Using these features, we can build preconditional requirements that can enable or disable whole portions of the knowledge base, depending on the current state of the environment being monitored. For example, rules pertaining to the analysis of a service *A* can be dynamically added or removed from the knowledge base by the expert system itself, depending on whether service *A* is currently enabled or disabled within the analysis target. Another example is when the analysis is extended with previously disabled rules due to an increased level of suspicion reported by the basic rule sets.

Another powerful feature of P-BEST is the ability of rules to uniquely mark and unmark facts, and to test for these marks. This can be used when we want to give several groups of mutually exclusive rules the chance to examine a fact before it is deleted from the factbase. Each rule will evaluate the fact, and if the antecedent is satisfied, the consequent of the rule will mark the fact. This will allow the rule to avoid re-firing, while not having to remove the fact completely from the factbase. When all such rules have evaluated (and if necessary marked) the fact, the fact can then be removed by a lower-priority fact-removal rule that is run last. For example, to match an event that is not marked with `CHECKED`, we can put the following test in the antecedent of our rule:

```
[+e:event^CHECKED]
```

To mark a matched event fact `e` with `CHECKED`, we can add the following action to the consequent:

```
[$|e:CHECKED]
```

Alternatively, to unmark a fact we simply use a caret (^) instead of the dollar sign ($):

```
[^|e:CHECKED]
```

Finally, we can use the dollar sign to check for a marked fact, as follows:

```
[+e:event$CHECKED]
```

2.4. P-BEST language simplicity and usability tested in student experiment

Although the P-BEST language has proven itself suitable for intrusion detection systems, it is in fact also a general language for building rule-based expert systems in many different applications. The close integration with C makes it unnecessary to include more than the basic operations in the P-BEST language itself, because any needed operation can be designed as a C function and called from the antecedent or consequent of a P-BEST rule. Thus, the P-BEST language can be kept small and simple, resulting in a very low learning threshold for beginners.

In addition to its use in intrusion detection system development, P-BEST has recently for the first time been used for laboratory exercises in a university course in applied computer security at Chalmers. In addition to the educational goals of these exercises, we wanted to learn what amount of instruction is required for beginners when applying P-BEST to intrusion detection analysis and thereby see whether the experiment would support or contradict our hypothesis that the P-BEST is easy to use for beginners.

The assignment was to build a system that could be used to automatically detect attacks against a file transfer (FTP) server. For evaluation of their resulting system, the students were given a very large data file (3 megabytes of text) containing recorded network data representing actual FTP transactions. A small number of real and synthetic intrusions were mixed with a large number of normal transactions, and the students were to use their system to find those intrusions. It was supposed to be a pedagogic effect that the file was too large to be easily examined by hand, because this is the very reason for having automatic intrusion detection tools. It was also required by the students to include in their lab reports a discussion of their experiences of using the tool.

There were 87 students who participated in the assignment, and with a few exceptions they worked in pairs, making a total of 46 groups. The estimated maximum working time was two lab sessions of four hours each, plus another eight hours of homework to prepare the lab sessions and to complete the report. Out of the 46 groups, 25 had built a system that gave the completely correct answer. An additional 8 groups would most likely have got the correct result if they had not all misinterpreted a vaguely formulated part of the instructions. Only a handful of groups failed to hand in a report before the given deadline. Most students reported that they found the exercise interesting and some even took the time to give detailed suggestions of improvements to the tool. As we had expected, being used to writing programs in a procedural style, they had some initial difficulties in declarative programming. In summary, we claim that the student experiment shows that P-BEST has a low learning threshold for beginners and is thereby suitable both for building user-customizable intrusion detection systems as well as for student exercises in computer security courses.

3. Integration of P-BEST into IDS components

For more than 10 years, P-BEST has been successfully integrated into several intrusion detection systems (IDSs) that represent the state of the art for their time. The application of P-BEST to intrusion detection began in the mainframe world of Multics and lands in present time with the highly distributed, scalable, and network-oriented EMERALD (Event Monitoring Enabling Responses to Anomalous Live Disturbances) environment. It is not only the IDSs that have changed over time; P-BEST itself has been continuously improved as the requirements and its operational environment have changed. However, performance and language simplicity are issues that have had top priority from the beginning, and are no less important today.

3.1. P-BEST in MIDAS

P-BEST was developed at SRI International and first deployed as the core of MIDAS, which provided real-time intrusion and misuse detection for the National Computer Security Center's networked mainframe, Dockmaster, a Honeywell DPS-8/70 running Multics [18]. Audit data preprocessing and command monitoring was performed on the Dockmaster, and the data was sent to a separate Symbolics Lisp machine where the expert system and the user interface were running.

MIDAS used both static and dynamic knowledge for detecting intrusive user behavior. The static knowledge was represented in so-called immediate attack heuristics written as P-BEST rules that would trigger on events that were considered anomalous regardless of previous system activity. In terms of dynamic knowledge, MIDAS recorded user and system statistics in a database that would represent normal behavior. It is interesting to note that it was in fact another set of P-BEST rules—the user anomaly heuristics and the system state heuristics—that used threshold values derived from the statistics database to distinguish anomalous user and system behavior from normal activity. Thus, the P-BEST inference engine was the sole analysis component in MIDAS.

3.2. P-BEST in IDES and NIDES

In 1983, SRI International began research on statistical techniques for audit-trail reduction and analysis [6]. This research led to the development of a prototype IDES, capable of providing real-time detection of security violations on single-target host systems. Originally, IDES only used statistical anomaly detection [5, 12], but later a component for misuse detection based on static knowledge was added, using P-BEST [14]. The two components were fed the same audit records, but performed their inferences and reporting independently.

Next, SRI began a comprehensive effort to enhance, optimize, and re-engineer the earlier IDES prototype into a production-quality intrusion-detection system called Next-Generation Intrusion Detection Expert System (NIDES). Just like its predecessor, NIDES has both a statistical anomaly detection component and a rule-based misuse detection component [1]. Again, P-BEST was the expert system shell of choice for the rule-based component, but P-BEST was first extensively revised. Among other things, the revision gave P-BEST a new syntax and a very tight coupling to the C programming language. While the early version of P-BEST used in MIDAS and IDES compiled rules into Lisp object code, the new version produced C source code. NIDES collects host audit trail data from different host systems and converts it to the NIDES audit record format. The current version of NIDES has a default rulebase of 39 rule sets (69 total production rules) but also allows the user to write his or her own rules (that, for example, are specific to the user's environment or policy) and has a mechanism for dynamically adding new rules at runtime.

3.3. P-BEST in the EMERALD eXpert

The EMERALD environment is a distributed scalable tool suite for tracking malicious activity through and across large networks [16]. EMERALD employs a building-block architectural strategy using independent distributed surveillance *monitors* that can analyze and respond to malicious activity on local targets, and can interoperate to form an analysis hierarchy. The generic EMERALD monitor architecture is designed to enable the flexible introduction and deletion of analysis engines from the monitor boundary as necessary. In its dual-analysis configuration, an EMERALD monitor instantiation combines signature analysis with statistical profiling to provide complementary forms of analysis over the operation of network services and infrastructure. In general, a monitor may include additional analysis engines that can implement other forms of event analysis, or a monitor may consist of only a single resolver implementing a response policy based on intrusion summaries produced by other EMERALD monitors. Monitors also incorporate a versatile application programmers' interface (API) that enhances their ability to interoperate with the analysis target, and with other third-party intrusion detection tools.

Underlying the deployment of an EMERALD monitor is the selection of a target-specific event stream. The event stream may be derived from a variety of sources, including audit data, network datagrams, SNMP traffic, application logs, and analysis results from other intrusion detection instrumentation. The event stream is parsed, filtered, and formatted by the target-specific event-collection methods provided by the monitor's pluggable configuration library, referred to as the *resource object*. Event records are then forwarded to the monitor's analysis engine(s) for processing.

The EMERALD *eXpert* (pronounced E-expert) is a generic signature-analysis engine based on the expert system shell P-BEST. The eXpert resource object has two parts, one of which consists of the configuration files for the EMERALD API that define the transports used for message passing (e.g., files or network connections), the message templates, and so forth, for the particular analysis target. The other part of the resource object is a P-BEST source file containing the fact type (ptype) declarations and rules. In the ptype declarations, the user must specify to what message field (if any) the ptype field corresponds.

Under EMERALD's eXpert architecture, special-purpose rule sets are encapsulated within resource objects that are then instantiated with an EMERALD monitor, and which can then be distributed to an appropriate observation point in the computing environment. This enables a spectrum of configurations from light weight distributed eXpert signature engines to heavy-duty centralized host-layer eXpert engines, such as those constructed for use in NIDES and MIDAS. In a given environment, P-BEST-based monitors may be independently distributed to analyze the activity of multiple network services (e.g., FTP, SMTP, HTTP) or network elements (e.g., a router or firewall). As each EMERALD eXpert is deployed to its target, it is instantiated with an appropriate resource object (e.g., an FTP resource object for FTP monitoring), while the eXpert code base remains independent of the analysis target.

EMERALD also introduces a target-independent code generation utility that allows one to automatically produce the library interfaces necessary to integrate a P-BEST expert system into the EMERALD monitor infrastructure. This utility effectively relieves the creator of a resource object from dealing with the internal operation of the eXpert codebase, even when redirecting the eXpert to a completely new event stream. This automated generation utility both enhances the rapid integration of eXpert to new analysis targets, and simplifies the process of augmenting the rule base with new heuristics. The basic operation of an eXpert analysis engine is as follows:

1. On startup, eXpert is initialized and its interface routine waits for messages on one or several transports, as specified in the configuration files of the resource object.

2. When an event record is received in the form of an EMERALD message, the message is matched against an interface data structure associated with the ptype definition in the eXpert's P-BEST fact base.

3. The message content is transferred to the interface data structure, which in turn is used to assert a fact into the expert system factbase.

4. The eXpert interface component hands over control to the expert system inference engine.

5. If a rule is fired, in which the consequent specifies that an alert shall be generated, the alert is propagated back to the analysis engine's interface component, which in turn composes and sends the alert on to the EMERALD resolver. The resolver operates as the monitor's decision engine, and can invoke local responses based on the alert or propagate the alert on to subscribers of the monitor's results (including administrative display interfaces).

6. When there are no more rules that can fire, the expert system returns control to the interface routine that again starts waiting for incoming messages.

In the following section, we discuss examples of how eXpert can be used to analyze very different types of event streams.

4. eXpert rule development examples

Throughout its usage, P-BEST inference engines have implemented a variety of intrusion detection rule sets for detecting and responding to numerous forms of malicious activity. Next, we describe the application of P-BEST in reasoning about attacks represented in two data streams: Solaris 2.5.+ audit trails, and TCP/IP packet streams. The examples illustrate the declarative style of the language, and how event streams can be represented and analyzed.

4.1. Examples of BSM audit trail analysis

The first example of an event stream to be analyzed is the audit trail produced by the Solaris Basic Security Module (BSM) from Sun Microsystems [19]. The audit records are normally saved in a file, but we have developed a BSM collection unit that receives audit records from the OS kernel in real time, formats and sends each record as an EMERALD message to the target monitor for analysis.

For all the rules that analyze BSM data, there is a ptype called `bsm_event` into which the relevant fields from incoming messages are mapped. There is also a rule that has highest priority and copies the time of every incoming `bsm_event` fact into a new `time` fact, and finally a rule with lowest priority that removes the `bsm_event` fact after all the other rules have had a chance to look at it. For the sake of brevity, these ptype definitions and administrative rules are omitted from the examples.

4.1.1. Failed authentication attempts. As an example of the declarative programming paradigm that P-BEST supports, we present a set of rules that are designed to detect a number of failed authentication attempts within a certain time window. The example illustrates how facts are created in rule consequents to keep state information between incoming events, and how the rule designer can make sure that facts are removed from the factbase when they are no longer needed.

Let us assume that we want to raise an alert if `x` user authentication failures occur within `y` seconds for a monitored target. A user authentication failure is defined as the case when either an invalid username or an invalid password is given to one of the programs *login, telnet, rlogin, rshd*, or *su*. To accomplish this, we may employ the rule set presented in Table 1, which is described as follows:

- `A1, A2`: For every incoming event that is a user authentication failure, save the event information in a `bad_login` fact and increment the counter for current bad logins (`current_bl_cntr`) by 1. The reason for having two rules is to separate the case where the username is invalid (`A1`) from the case where the username is valid but the password is invalid (`A2`). In the latter case, we want to include the username in the information we save and therefore need a rule consequent that is different from the former case where there is no username reported in the audit record.
- `A3`: When the `current_bl_cntr` counter has the value `x`, send an alert and create a `max_bl_reached` fact to indicate that the authentication failure threshold was reached.
- `A4`: If there exists a `max_bl_reached` fact, then loop through all saved `bad_login` facts. For every `bad_login` fact, print the information contained in the fact to a log file and delete the fact from the factbase.
- `A5`: If there exists a `max_bl_reached` fact but no `bad_login` facts (i.e., they were all printed and deleted by rule `A4`), then delete the `max_bl_reached` fact from the factbase.
- `A6`: If there exists a `bad_login` fact, but no `max_bl_reached` fact, and the difference between the `bad_login` timestamp and the current event timestamp is more than `y` seconds, then delete the `bad_login` fact from the factbase and decrement the `current_bl_cntr` by 1.

4.1.2. Buffer overrun attacks. Buffer overrun attacks are a common way for attackers to gain super-user privileges after first breaking into an unprivileged user account. Typically, a privileged (*setuid* to *root*) program is called with an extremely long and carefully crafted argument that overflows memory buffers and alters the program execution [3]. In principle, it would require a fair amount of programming skills and patience to exploit a buffer overrun vulnerability, but ready-to-use exploit programs that can be downloaded from Internet sites give immediate super-user access when executed. Here, we present an example of a simple heuristic P-BEST rule that detects the behavior of most of the exploit programs. For example, it has been tested against buffer overrun exploits that are based on subverting Solaris 2.5 *eject*, *fdformat*, *ffbconfig* and *ufsrestore*.[1]

The heuristic rule is based on the following observations of the audit trail characteristics of common buffer overrun exploits:

- We can detect the attack by analyzing a single *exec* system call audit record, as suggested in [2].
- To determine that the *exec* call concerns a *setuid* program (otherwise, it would not be a target for attack), we simply match only the audit records for which the *effective user id* and *real user id* fields are different.
- The argument passed to the *exec* call is relatively long (because it must overflow a buffer and contain executable code), making the length of the entire audit record significantly exceed the length of almost all normal *setuid exec* calls.
- By necessity of the applicable hardware (Sun and Intel), the *exec* argument contains binary opcodes in the range of ascii control characters. While such a property may not necessarily hold on all possible hardware platforms, this heuristic works exceptionally well for our purposes.

The P-BEST rule that uses the observations above to detect buffer overrun attacks is shown in Figure 6. This simple heuristic rule is not a fool-proof way to detect all possible buffer overrun attacks, but it is remarkably efficient in terms of coverage and correctness; it detects most common attacks and has not produced any false positives when tested on a collection of over 35 million audit records in which the location of buffer overflow attacks was known *a priori*.

[1]There are numerous additional buffer overrun attacks that employ the identical attack strategy as the four attacks discussed here. All should be subject to detection by this rule.

Table 1. Rule set for detection of failed authentication attempts.

```
rule[A1(*):
   [+e:bsm_event^A12]
   [?|e.header_event_type   == ’AUE_login  ||
      e.header_event_type   == ’AUE_telnet ||
      e.header_event_type   == ’AUE_rlogin ||
      e.header_event_type   == ’AUE_rshd   ||
      e.header_event_type   == ’AUE_su]
   [?|e.return_return_value == ’INVALID_USER]
   [+cc: current_bl_cntr]
   [-max_bl_reached]
==>
   [+bad_login |
      timestamp    = e.header_time,
      audit_seq_no = e.msequenceNumber,
      username     = "invalid username",
      command      = e.header_command,
      etype        = e.header_event_type,
      hostname     = e.subject_hostname,
      portID       = e.subject_port_id,
      processID    = e.subject_pid,
      textList     = e.textList]
   [/cc| value += 1]
   [$|e:A12]
]
```

```
rule[A2(*):
   [+e:bsm_event^A12]
   [?|e.header_event_type   == ’AUE_login  ||
      e.header_event_type   == ’AUE_telnet ||
      e.header_event_type   == ’AUE_rlogin ||
      e.header_event_type   == ’AUE_rshd   ||
      e.header_event_type   == ’AUE_su]
   [?|e.return_return_value == ’INVALID_PWD]
   [+cc: current_bl_cntr]
   [-max_bl_reached]
==>
   [+bad_login |
      timestamp    = e.header_time,
      audit_seq_no = e.msequenceNumber,
      username     = e.subject_runame,
      command      = e.header_command,
      etype        = e.header_event_type,
      hostname     = e.subject_hostname,
      portID       = e.subject_port_id,
      processID    = e.subject_pid,
      textList     = e.textList]
   [/cc| value += 1]
   [$|e:A12]
]
```

```
rule[A3(*):
   [-max_bl_reached]
   [+cc:current_bl_cntr | value == ’x]
   [+ts:time^A3]
==>
   [!|printf("ALERT: Max Bad Logins \n")]
   [+max_bl_reached | value = 1]
   [$|ts:A3]
   [!|EXpertReport(’eXpertMessagePointerString,
      1042, "description", ’pTypeString,
      "MAX LOGIN ALERT",
      "ruleName", ’pTypeString, "A3", "")]
]
```

```
rule[A4(*):
   [+max_bl_reached]
   [+bc:bad_login]
   [+cc:current_bl_cntr]
==>
   [!|printf("(%s): %s from %s on %s port %d, \
     PID = %d, time = %d, seq no = %d \n",
     bc.textlist, bc.command, bc.username,
     bc.hostname, bc.portID, bc.processID,
     bc.timestamp, bc.audit_seq_no)]
   [/cc|value -= 1]
   [-|bc]
]
```

```
rule[A5(*):
   [+mx:max_bl_reached]
   [-bad_login]
==>
   [-|mx]
]
```

```
rule[A6(*):
   [+ts:time^A6]
   [-max_bl_reached]
   [+bc:bad_login]
   [+cc:current_bl_cntr]
   [?|(ts.sec - bc.timestamp) > ’y]
==>
   [/cc|value -=1 ]
   [-|bc]
   [$|ts:A6]
]
```

```
rule[BSM_LONG_SUID_EXEC(*):
   [+e:bsm_event]
   [?|e.header_event_type == ’AUE_EXEC ||
      e.header_event_type == ’AUE_EXECVE]
   [?|e.subject_euid != e.subject_ruid ]
   [?|contains (e.exec_args, "ˆ\\") == 1]
   [?|e.header_size > ’NORMAL_LENGTH]
==>
   [!|printf("ALERT: Buffer overrun attack \
      on command %s\n", e.header_command)]
]
```

Figure 6. A heuristic rule for detecting common buffer overrun attacks.

To determine a suitable value for the `NORMAL_LENGTH` threshold parameter, we have analyzed in the order of 4 million audit records representing normal system usage (of which over 29 thousand were *exec* events) in addition to audit records representing common buffer overrun attacks. This analysis gave the following results:

- All the attacks we tested produce an *exec* audit record with a record length of at least 500 bytes.
- Only 0.15 per cent of the normal *exec* audit records were longer than 400 bytes.

Consequently, by setting the threshold to 400 and adding the conditions for *setuid* and control characters, false positives are effectively eliminated while exploits of the described type are detected.

4.2. Network-based traffic analysis

In addition to its extensive application to the area of audit trail analysis, P-BEST is now being applied to the analysis of network traffic streams. This work includes the analysis of TCP/IP packet streams for low-level TCP and IP layer attacks (i.e., attacks that target vulnerabilities at the transport layer and below) as well as higher-layer attacks involving vulnerabilities of application-layer (or network service-layer) protocols, such as FTP, SMTP, and HTTP.

4.2.1. Attack description: SYN flood attack. The SYN flood attack is a denial-of-service attack that prevents the target machine from accepting new connections to a given IP port [17]. Briefly, the attack exploits a resource exhaustion vulnerability in the way operating systems handle TCP/IP connections. A TCP/IP connection is established through a three-step handshake, in which the client sends a SYN packet, followed by the server responding with a SYN-ACK packet, which is then acknowledged by the client with an ACK packet. Of course, by no means is there an expectation that all TCP/IP handshakes run to completion. When the SYN packet is received, the server allocates an entry in a finite queue of pending connections. We refer to this stage as a *half-open* connection. The queue entry will either be released when the final ACK is received by the server, or the server will proceed to timeout the incomplete handshake and release the entry.

An attacker can exploit the TCP/IP connection logic by initiating a series of SYN packet connection requests to a server, but not completing the handshakes with an ACK packet. Internally, the server's queue of pending connections for the port will eventually be exhausted and will not be released until the timeout periods for the unfinished connections expire. As a result, subsequent connection requests to the server that occur while the connection queue is full will be dropped, effectively denying access to the server by other legitimate clients.

4.2.2. Event stream format. The requirements for detecting the occurrence of a SYN flooding attack against a host are rather minimal. From the perspective of TCP/IP traffic monitoring, the analysis engine need only monitor SYN-ACK and ACK packet exchanges to identify incomplete TCP/IP handshakes. In this example, the traffic monitor is placed on a segment of the network capable of observing traffic to and from the analysis target (the host being monitored). All SYN-ACK packets sent from—and ACK packets sent to—the analysis target are recorded, and the following event record is derived:

```
Connection Event Format:  <Event_Type>
    <Timestamp> <Seq_ID> <Client_ID>
```

The Event_Type field is simply a binary flag, which indicates whether the packet has its SYN and ACK flags enabled (which we can denote with 0), or only the ACK flag enabled (denoted by 1). The timestamp is a numeric encoding of the time at which the packet is observed from the monitor. The sequence ID represents the TCP Sequence ID field, which is used to associate client requests with server replies. Last, the Client_ID can be used to identify the client who initiated the connection. The Client_ID is not critical for detection, and in all likelihood will not be reliable (i.e., attackers will manufacture IP packets with bogus IP source addresses). Nevertheless, we may choose to capture such information as the IP address and port number of the client packet for reporting purposes only.

4.2.3. P-BEST fact type definitions. Table 2 illustrates the ptype definitions of three example facts that are specified for use in performing the TCP SYN flooding analysis. The first ptype, `conn_event`, is used to assert the connection event described in the connection event record format discussed above. As connection events are captured by the network monitor, their fields can be mapped (one to one) to the fields of the `conn_event` ptype, and the `conn_event` ptype is then asserted into the factbase of the SYN flood eXpert. The `open_conn` ptype is used to construct facts regarding half-open connections that are pending completion of the TCP/IP handshake. Note, although we use the shorthand name `open_conn`, the fact actually represents the assertion that a TCP *half-opened* connection has been observed. The fields of the `open_conn` contain the TCP sequence ID of the pending connection, a `client_ID` string (as discussed above), the timestamp as copied from the connection event, and an expired flag used for garbage collection by the production rules. Last, the bad connection fact, `bad_conn`, maintains a running count of the number of bad connection requests detected through the observations of SYN-ACK and ACK packages between the analysis target and external clients.

4.2.4. Example P-BEST rules for SYN flood detection. The following illustrates one inference strategy that P-BEST can employ for deducing a TCP SYN flooding attack, using the fact definitions defined above. In addition, a few constants are referenced from the rule set, and are defined as follows:

• `max_bad_conns:` Number of bad connections tolerated before SYN flood alert.

• `expire_time:` Amount of time to wait on ACK before a connection is declared a bad connection.

• `bad_conn_life:` Number of seconds that a bad connection fact will live before being released.

Abstractly, the rules attempt to identify half-open TCP connections that expire beyond a user-defined waiting period. As we assert half-open connection facts into our factbase, we must include logic to recognize both when the connections are successfully completed and when half-open connection expire beyond the user-defined waiting period, from which we deduce the occurrence of a bad connection. SYN flood attacks will result in excessive bursts of bad connections, which we monitor with rules that maintain a running count of bad connections over a sliding window of time. When the number of bad connections exceeds our maximum tolerance for bad connections within our sliding time window, we raise an alert to denote the burst of noncompleted connection requests. The following is a brief summary of the rule set shown in Table 3.

• `create_open_conn`: determines whether the event connection represents a SYN-ACK packet (from the monitor target). If so, the rule asserts a new fact into the factbase called `open_conn`, which records the TCP sequence number, the timestamp at which this half-opened connection was first observed, an expired flag to indicate when the half-open connection exceeds a time threshold, and the `client_ID`.

• `destroy_open_conn`: removes an open connection fact when the corresponding ACK packet is received from the client.

• `ignore_spurious_acks`: removes events involving ACK packets that are not associated with a specific SYN-ACK pending connection. In practice, such packets are normal.

• `first_bad_conn`: This and the following rule manage a running count of the set of bad connections observed by the inference engine. They are driven by time facts (line 24) which are used to monitor whether there exists a half-open connection that has exceeded the `expire_time` limit. This rule is applied once, to the first `open_conn` fact encountered that is older than `expire_time`. Its consequent creates the `bad_conn` fact, which initializes the bad connection counter upon the first encountered expired connection. Note that the antecedent line 25 evaluates to false once the `bad_conn` fact has been initialized. In addition, the rule marks the `open_conn` fact as expired (line 30), which is consulted by `free_bad_open_cons` when performing garbage collection.

• `add_to_bad_cons`: is applied while the total number of `bad_conn` facts is less than the maximum tolerated. If an `open_conn` fact timestamp exceeds the expiration time and the fact has not been counted earlier, then the `bad_conn` count is incremented, and the expired flag for the `open_conn` fact is set.

• `max_open_cons`: is applied when the maximum number of `bad_conn` facts is encountered during a burst of `bad_conn_life` time units. If a `bad_conn` count reaches the maximum tolerated `bad_conn` facts, the consequent initiates a SYN flood alert, and resets the bad connection count.

• `free_bad_open_cons`: limits the amount of time that a bad open connection is counted against the system. The `bad_conn_life` variable provides a user-defined length of time with which a bad connection is considered relevant to the bad connection count. This variable effectively represents the burst duration for accumulating bad connections. Once an open connection exceeds the `bad_conn_life`, then it is removed and the bad connection count is reduced.

5. Performance

There are a variety of factors that influence the amount of time required to process records through a P-BEST-based signature analysis engine. In this section, we briefly discuss some of these factors and summarize several performance measurements in analyzing both Solaris audit records and TCP packets through an EMERALD eXpert P-BEST engine. These measurements are intended to reflect the pure processing time required by the eXpert in receiving events, translating and asserting the events into the eXpert fact base, processing the events through the inference engine, and

Table 2. Facts for TCP SYN flood detection.

```
ptype[conn_event              ptype[open_conn               ptype[bad_conn
      e_type:integer,               expired:integer,              count:integer]
      sec:integer,                  sec:integer,
      seq_id:integer,               seq_id:integer,
      client_ID:string]             client_ID:string]
```

Table 3. Rule set for detection of TCP SYN flood attacks.

```
rule[create_open_conn(*):
   [+ev:conn_event|e_type == 0]
==>
   [+open_conn |seq_id = ev.seq_id,
                sec = ev.sec,
                expired = 0,
                client_ID = ev.client_ID]
   [-|ev]
]
```

```
rule[add_to_bad_cons(*):
   [+ts:time]
   [+oc:open_conn|expired == 0]
   [?|(ts.sec - oc.sec) > 'expire_time]
   [+bc:bad_conn|count < 'max_bad_conns]
==>
   [/bc|count += 1]
   [/oc|expired = 1]
]
```

```
rule[destroy_open_conn(*):
   [+ev:conn_event|e_type == 1]
   [+oc:open_conn|seq_id == (ev.seq_id - 1)]
==>
   [-|oc]
   [-|ev]
]
```

```
rule[ignore_spurious_acks(*):
   [+ev:conn_event|e_type == 1]
   [-oc:open_conn|seq_id == (ev.seq_id - 1)]
==>
   [-|ev]
]
```

```
rule[max_open_cons(*):
   [+ts:time]
   [+oc:open_conn|expired == 0]
   [?|(ts.sec - oc.sec) > 'expire_time]
   [+bc:bad_conn|count == 'max_bad_conns]
==>
   [!|syn_alert("SYN Attack: Last Host %s.\
      SeqID = %d. Time = %d",
       oc.client_ID, oc.seq_id, oc.sec)]
   [/bc|count = 1]
   [/oc|expired = 1]
]
```

```
 rule[first_bad_conn(*):
   [+ts:time]
   [-bad_conn]
   [+oc:open_conn|expired == 0]
   [?|(ts.sec - oc.sec) > 'expire_time]
==>
   [+bad_conn|count = 1]
   [/oc| expired = 1]
]
```

```
rule[free_bad_open_cons(*):
   [+ts:time]
   [+bc:bad_conn]
   [+oc:open_conn|expired == 1]
   [?|(ts.sec - oc.sec) > 'bad_conn_life]
==>
   [-|oc]
   [/bc|count -= 1]
]
```

handling alert reporting.

The measurements exclude the processing time added to the system for event generation; that is, it excludes the impact to system resources in audit record generation or the capturing and filtering of TCP packets. It is difficult to estimate the daily expected volumes of audit and network traffic across a computing environment, in that such statistics are directly dependent on the structure of the computing environment, network topology, and behavior and size of the user community. Furthermore, the EMERALD architectural model lends itself well to the separation of the event generation and collection components from the analytical engines, which could in fact operate in parallel on separate hosts.

The performance measurements were collected on a FreeBSD 2.2.6 host computer system using a Pentium II 333 Mhz processor with 128 MB RAM. In addition to the processing capabilities of the host platform, there are several factors that significantly influence the overall performance of the analysis engine. For example, the average record size and total event stream size dictate the amount of I/O overhead required. As each event is asserted by the rule base, the antecedent evaluation also impacts performance: the sheer number of rules to evaluate, as well as the complexity of each antecedent evaluation, significantly influence event processing throughput. Consequent activation is also a consideration, as is the management of derived facts that are asserted during the analysis.

Table 4 presents a summary of three analyses performed on 1 and 5 day collections of Solaris 2.5.1 audit records and

Table 4. Performance of sample BSM and TCP analysis engines.

	24 hrs BSM 43 users 365 MB total 1.1 million recs	120 hrs BSM 44 users 1.41 GB total 4.2 million recs	24 hrs IP 496 connects 331 MB total 83,002 recs	120 hrs IP 1,343 connects 1.3 GB total 352,445 recs
1 rule set *2 rules* *buffer overrun*	4:10 min:sec	15:41 min:sec	—	—
16 rule sets *28 rules* *various intrusions*	8:09 min:sec	30:53 min:sec	—	—
1 rule set *12 rules* *TCP SYN flood*	—	—	1:33 min:sec	3:02 min:sec

TCP packet streams. The audit and TCP data sets were collected by MIT Lincoln Laboratories, and made available for the DARPA Intrusion Detection Evaluation Program. The BSM audit logs analyzed here represent the simulated usage of a server with 43 users over one 24 hour period and 44 users over a 5 day work week, with minimal filtering. While it is difficult to generalize what such loads imply for other computing environments, the data set is representative of the volume and type of audit activity observed during a prolonged study of several Air Force local area networks.

The first row in Table 4 summarizes the performance of an EMERALD eXpert implementing the buffer overflow rule presented in Section 4.1, which is roughly able to apply this rule to 24 hours of audit data (over one million audit records) in 4 minutes, and 120 hours of audit data (4.2 million audit records) in under 16 minutes. In the second row, we present an eXpert with a more extensive collection of 28 rules. These rules implement 16 sets of Solaris BSM intrusion detection heuristics, including threshold analyses, immediate attack recognition, process subversion detection, and illegal file access recognition. While the knowledge base of this second eXpert represent an increase of fourteen fold over the 2-rule eXpert system in row one, it introduces only a two fold increase in the overall processing time of the 1 and 5 day data sets. In this computing environment, the 16 rule sets can process the full five day data set in just over 30 minutes; this represents a small fraction of the overall audit generation time.

The third and fourth columns of Table 4 present an analysis of TCP/IP traffic through a gateway that provides service between an internal domain of 4 servers and 20 workstations, and an external untrusted network. Row three of Table 4 summarizes the performance of the TCP SYN flood detection rules presented in Section 4.2 (with a few additional administrative rules). Here, a server was selected for analysis, and all TCP packets sent to and from it were monitored for 24 and 120 hours, during which 496 and 1,343 connections were observed over 24 and 120 hours, respectively. The SYN Flood eXpert monitored only those TCP packets targeted for the host of interest in which the SYN or ACK flags were enabled. The filtering out of unnecessary packets is critical to managing the performance of a real-time signature analysis engine, and in the SYN flooding case, the criteria for analysis excludes all packets that are not directly involved in the TCP handshake. In our simulated analysis, the SYN Flood eXpert is capable of performing the 24 hour packet analysis in 1.5 minutes, and the 120 hour analysis in 3 minutes.

6. Related work

P-BEST has evolved over a substantial lineage of intrusion detection projects, which include MIDAS, IDES, NIDES, and now the EMERALD eXpert. It represents a very early example of the application of a forward-chaining rule-based expert system to the problem of misuse detection in computer system activity logs. However, P-BEST is by no means the only system to have applied rule-based expert system techniques to detecting misuse in computing environments.

Several other systems have been developed that also center around the use of forward-chaining inference logic, and have applied a variety of techniques for representing the underlying heuristics used to represent misuse. The ASAX (Advanced Security and Audit Trail Analysis on UniX) project [9], produced a highly specialized rule-based programming language called RUSSEL (Rule Based Sequence Evaluation Language), which provides a combination of procedural and rule-based programming constructs to reason about activity in Unix audit trails.

The University of California at Santa Barbara proposed the use of state transition diagrams to model the sequence of operations and state changes that occur during the execution of a penetration [15]. This technique was prototyped for SunOS 4.1.3+ and Solaris audit trails in a tool called the Unix State Transition Analysis Tool (USTAT) [10]. While it did not represent its knowledge base using production rules, USTAT was architected as a classic expert system, with an inference engine, knowledge base, fact base, and separate decision engine. Another system, called IDIOT (Intrusion Detection In Our Time), took a similar graphical approach to the analysis of signature operations, but used Colored Petri-nets to model its analysis of the patterns of execution represented in an event stream [13].

Wisdom and Sense [20] and NADIR [11], both from Los Alamos National Laboratory, are further examples of intrusion detection systems that employed rule-based analyses to identify known malicious activity. In the case of W&S, the anomaly detection component was also implemented as a rule-base. The signature analysis component was combined into the same rule-base to represent site-specific policies, expert penetration rules and other administrative data. NADIR's expert rule-base consists of penetration rules that are developed by interviewing and working with security personnel.

Last, it is important to recognize a continuing growth in the number of commercial products that provide forms of signature analysis for various computing environments. Given the proprietary nature of these systems, it is difficult to understand which have chosen hard-coded narrow solutions to their problem sets, and which have chosen more broad techniques that may be portable beyond their current customers' needs.

7. Limitations

In Section 2 we attempted to summarize how and why forward reasoning systems provide a good foundation for modeling known abusive activity represented in an event stream. There are, of course, limitations that are fair to point out with respect to this general method. In our own system, antecedent evaluation is absolute, and less capable in environments where uncertainty, incompleteness, or inaccuracies exist within the event stream content. Other reasoning systems can provide some options for handling belief and uncertainty within the analysis framework [8]. In the presence of incomplete data, backward reasoning systems can operate in a diagnosis mode to seek out collaborative evidence of problems, and furthermore provide quantitative probabilities based on "evidence to date" that a certain problem is the culprit responsible for the presence of given symptoms. Such reasoning capabilities could be valuable if applied well to the intrusion detection domain.

In addition to event stream inadequacies, heuristics presuppose the existence of detailed insight into that which constitutes abusive system activity. The problem of recognizing and responding to *unknown* malicious phenomena is extremely difficult, and not directly addressed under signature analysis. Only in the cases where it is possible to look for certain *results*—rather than explicit action sequences leading to those results—does signature analysis have a chance to detect new attack methods. For example, if an anonymous user causes the deletion of a file from our FTP server, we can detect this result without knowing exactly how the attack was carried out.

Other techniques that attempt to understand *normal* system operation and to provide quick recognition of anomalous activity have been proposed; statistical profiling [5], neural networks [4], and sequence analysis [7]. The intent of these systems is to maximize the points at which anomalous activity corresponds to malicious activity, which as a general property does not always hold. In addition, attempting to maximize such systems' sensitivity to malicious activity also tends to increase their sensitivity to inane anomalies.

8. Future work

In parallel with our current academic experiments with P-BEST, we are developing an Internet-accessible P-BEST translation service, which will allow users to develop and compile rule sets into self-contained expert systems. Linkage modules will be provided to allow users to feed the expert system Solaris 2.5.+ audit records and TCP/IP packets in batch and real time. Users will be provided an HTML interface from which ptype definitions and production rules can be submitted to the P-BEST translation service. The translation service will attempt to compile an expert system based on the ptypes and rules; if successful, the user will receive a URL link from which the expert system can be downloaded and tested in the user's own environment. If errors are identified in the rules or ptype declarations, a summary of the errors will be returned to the user for revision. In addition, a simple reporting utility will be provided to convert alerts generated by the expert system to HTML or email notifications. We will make the following components available to other universities interested in conducting classroom experiments involving signature-based intrusion detection:

- P-BEST expert system generation service available via HTML-based interface
- Solaris audit and TCP/IP batch and real-time event collection interface modules
- HTML and email alert reporting interface module

- Language manuals and supporting documentation (including exploit detection exercises) developed in support of our current university classroom experiment

For more information on the Internet-accessible P-BEST translation service for academic experimentation, the reader may refer to the following URL:
`http://www.csl.sri.com/emerald/`

9. Conclusion

We have presented the operation of a production-based expert system toolset, and its application to the problem of computer and network signature-based intrusion detection. P-BEST has had considerable exposure to the intrusion detection problem domain over the past decade, under the MIDAS, IDES, and NIDES projects, and now within the EMERALD eXpert. P-BEST has been employed on a Symbolics processor for handling Multics audit records, SunOS 4.1.+, Solaris 2.5.+, FreeBSD, and Linux for real-time audit trail analysis, accounting log analysis, and TCP/IP packet analysis.

We presented details of the P-BEST production rule specification language, and illustrated its use with example rule sets for detecting misuse in Solaris 2.5.+ audit trails and TCP/IP packet streams. We also discussed the performance of P-BEST inference engines in analyzing millions of events, which illustrates that P-BEST has been—and continues to be—useful in live monitoring of computer and network operations.

In addition, work is in progress to move P-BEST into academic environments, where it will be made openly available as an instructional tool for illustrating signature-based intrusion detection. P-BEST is currently being used for laboratory exercises in one university course on applied computer security, where students are guided through its usage and assigned rule development tasks for analyzing given intrusions. We have demonstrated that the P-BEST language is not too complex for beginners to employ, and is efficient for supporting the iterative development of increasingly complex inference logic for automated reasoning about misuse in computer and network operations.

Acknowledgments

The authors are grateful to Alan Whitehurst, the original author of P-BEST, and to Fred Gilham who has made later enhancements and documentation and has answered many questions from us. Several people have participated in discussions and contributed with ideas and suggestions; we thank Peter Neumann, Keith Skinner, and Martin Fong at SRI and Erland Jonsson and Stefan Axelsson at Chalmers. We also thank the Chalmers students who participated in the P-BEST lab exercise.

References

[1] D. Anderson, T. Frivold, and A. Valdes. Next-generation intrusion-detection expert system (NIDES). Technical Report SRI-CSL-95-07, Computer Science Laboratory, SRI International, Menlo Park, CA 94025-3493, USA, May 1995.

[2] S. Axelsson, U. Lindqvist, U. Gustafson, and E. Jonsson. An approach to UNIX security logging. In *Proceedings of the 21st National Information Systems Security Conference*, pages 62–75, Arlington, Virginia, Oct. 5–8, 1998. National Institute of Standards and Technology/National Computer Security Center.

[3] D. Bruschi, E. Rosti, and R. Banfi. A tool for pro-active defense against the buffer overrun attack. In J.-J. Quisquater et al., editors, *Computer Security – Proceedings of ESORICS 98*, volume 1485 of *LNCS*, pages 17–31, Louvain-la-Neuve, Belgium, Sept. 16–18, 1998. Springer-Verlag.

[4] H. Debar, M. Becker, and D. Siboni. A neural network component for an intrusion detection system. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, pages 240–250, Oakland, California, May 4–6, 1992. IEEE Computer Society Press.

[5] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, Feb. 1987.

[6] D. E. Denning and P. G. Neumann. Requirements and model for IDES—a real-time intrusion detection expert system. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025-3493, USA, 1985.

[7] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, California, May 6–8, 1996. IEEE Computer Society Press.

[8] T. D. Garvey and T. F. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, pages 372–385, Washington, D.C., Oct. 1–4, 1991. National Institute of Standards and Technology/National Computer Security Center.

[9] J. Habra, B. Le Charlier, A. Mounji, and I. Mathieu. ASAX: Software architecture and rule-based language for universal audit trail analysis. In Y. Deswarte et al., editors, *Computer Security – Proceedings of ESORICS 92*, volume 648 of *LNCS*, pages 435–450, Toulouse, France, Nov. 23–25, 1992. Springer-Verlag.

[10] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, Mar. 1995.

[11] K. A. Jackson, D. H. DuBois, and C. A. Stallings. An expert system application for network intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, pages 215–225, Washington, D.C., Oct. 1–4, 1991. National Institute of Standards and Technology/National Computer Security Center.

[12] H. S. Javitz and A. Valdes. The SRI IDES statistical anomaly detector. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 316–326, Oakland, California, May 20–22, 1991. IEEE Computer Society Press.

[13] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, West Lafayette, Indiana, Aug. 1995.

[14] T. F. Lunt, R. Jagannathan, R. Lee, A. Whitehurst, and S. Listgarten. Knowledge-based intrusion detection. In *Proceedings of the Annual AI Systems in Government Conference*, pages 102–107, Washington, D.C., Mar. 27–31, 1989. IEEE Computer Society Press.

[15] P. A. Porras and R. A. Kemmerer. Penetration state transition analysis: A rule-based intrusion detection approach. In *Proceedings of the Eighth Annual Computer Security Applications Conference*, pages 220–229, San Antonio, Texas, Nov. 30–Dec. 4, 1992. IEEE Computer Society Press.

[16] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365, Baltimore, Maryland, Oct. 7–10 1997. National Institute of Standards and Technology/National Computer Security Center.

[17] C. L. Schuba, I. V. Krsul, M. G. Kuhn, E. H. Spafford, A. Sundaram, and D. Zamboni. Analysis of a denial of service attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 208–223, Oakland, California, May 4–7, 1997. IEEE Computer Society Press.

[18] M. M. Sebring, E. Shellhouse, 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, Baltimore, Maryland, Oct. 17–20, 1988. National Institute of Standards and Technology/National Computer Security Center.

[19] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043-1100, USA. *SunSHIELD Basic Security Module Guide*, Nov. 1995.

[20] H. S. Vaccaro and G. E. Liepins. Detection of anomalous computer session activity. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 280–289, Oakland, California, May 1–3, 1989. IEEE Computer Society Press.