A Query Facility for Common Intrusion Detection Framework

Peng Ning, X. Sean Wang, Sushil Jajodia

Center for Secure Information Systems
George Mason University, Fairfax, VA 22030, USA
Voice: (703)993-{1629, 1662, 1653}, Fax: (703)993-1638
{pning, xywang, jajodia}@gmu.edu

Abstract

It is essential for intrusion detection systems to share information in order to discover attacks involving multiple sites. Common Intrusion Detection Framework (CIDF) is an important step towards enabling different intrusion detection and response (IDR) components to interoperate with each other. Although CIDF provides an infrastructure and language support that allows an IDR component to understand the information sent by another component, it does not contain a facility for a component to request specific information from other components. The lack of such a facility may result in a waste of processing time, storage capacity and network bandwidth. This paper proposes an extension to the Common Intrusion Specification Language (CISL), the language adopted by CIDF, to model requests among CIDF components. The extension is simple and consistent with the original CISL. Each request for information is described as a pattern for relevant information and an optional format specification for the responding message. The use of pattern in modeling requests not only provides a way to represent queries, but also leads to a potential reuse of signature-based intrusion detection software.

1 Introduction

Sharing information among intrusion detection systems is important, especially for the purpose of detecting coordinated intrusions or intrusions distributed across a set of hosts and network elements [2, 5, 9]. Common Intrusion Detection Framework (CIDF) is the result of an on-going work that aims at enabling different intrusion detection and response (IDR) components to interoperate and share information [1, 4, 5, 11, 13]. The CIDF working group was formed as a collaboration among DARPA funded IDR projects.

Although CIDF provides an infrastructure and language support that allows an IDR component to understand the information that is sent by a remote IDR component, it does not contain a facility for an IDR component to request specific information from another component. The lack of such a facility may result in inefficient communication between IDR components. Indeed, if the sending IDR component sends all its information to the receiving component, the receiving component may be overwhelmed by information, most of which it does not want. Not only the sending and the receiving components may have to process unnecessary messages, but the network bandwidth may also be wasted. On the other hand, if the sending IDR component does not send all the information, the receiving components may miss some important messages. We believe that a more efficient approach is to have IDR components send information to communicating partners *only* when the particular information is requested.

For example, suppose the IDR system monitoring host A discovers a suspicious user who remotely logged in from host B, and it wants to trace the origin of the user. Instead of letting this IDR system get all the login events from host B and figure out from which host the suspicious user logged into B, a more efficient way is to let it request the information from the IDR system for host B, which then finds out and returns the result. (This process may continue when the IDR system for A learns that the suspicious user actually logged into B from another host, say C.)

In this paper, we propose an extension to the common intrusion specification language (CISL), the language adopted by CIDF. The extension allows IDR components to specify requests for particular information from other IDR components.

The extension describes requests for particular information in terms of patterns. A pattern specifies the characteristics of the information in which the requesting IDR component is interested.

Based on S-expressions, the language construct adopted by CISL, patterns are described using "wild-card terms" that stand for SIDs (semantic identifiers) or data values, and conditions that these wild-card terms must satisfy. Basically, each pattern gives a set of S-expressions that match the pattern. Together with the semantics assigned by CIDF to the "words" in CISL, patterns allow CIDF components to precisely express requests.

A component that receives a request is expected to find the information that matches the pattern, instantiate the wild-cards in the pattern according to the information, and return the resulting message composed from the instantiation. Optionally, the requesting component may specify the format for the responding message.

Note that the extension proposed in this paper is independent of the message transmission model, which can be either a push or a pull model. Indeed, the extended CISL not only enables an IDR component to send queries to other components, but also allows it to set up traps at remote systems.

A formal approach has been developed for general IDR systems to specify requests [10]. In this paper, we focus on CIDF and propose a language extension specifically for CISL. Many concepts introduced here are based on the formal model of [10]. An architecture for detecting coordinated attacks using predefined queries was also proposed in [14], where queries (which we also call *detection tasks*) are automatically generated from the attack descriptions and different IDR components cooperatively detect coordinated attacks by message passing.

The contribution of this paper is three-fold. First, we recognize that the querying capability is important to the performance of interoperating IDR systems. Second, we formalize a request for specific information as a pattern and an optional format specification for the response. Since patterns are intrinsic to IDR information (as evidenced by the existence of many pattern-based intrusion detection systems), our approach provides a natural way for IDR systems to express their interests. As a further benefit, using patterns to specify queries can potentially lead to a reuse of existing intrusion detection software. Finally, our work provides a simple and consistent extension to the CISL. Existing CIDF components can be easily adapted to take advantage of the language extension.

The rest of the paper is organized as follows. Section 2 briefly describes the background material about CIDF and CISL. Section 3 discusses our extension for specifying requests using S-patterns. Section 4 discusses some implementation issues involved in the deployment of the request facility. Section 5 concludes this paper and points out some future research directions.

2 Background

CIDF is a framework that aims at interoperation and software reuse among IDR systems [1, 4, 5, 11, 13]. CIDF views IDR systems as consisting of discrete components that communicate via message passing. Four kinds of IDR components are envisaged: Event Generators (E-boxes), Event Analyzers (A-boxes), Event Databases (D-boxes) and Response Units (R-boxes). An event generator obtains events from the larger computational environment outside the IDR system; an event analyzer receives information from other components, analyzes them, and returns the analysis result; an event database stores information; and a response unit takes actions (e.g., killing a process) on behalf of other components.

Data exchanged among CIDF components is specified in the form of *generalized intrusion detection objects (gidos)* which are described in the Common Intrusion Specification Language (CISL). A gido encodes an occurrence of a particular event that happened at a particular time, a conclusion about a set of events, or an instruction to carry out an action.

CIDF components communicate over a three-layered architecture, which consists of the gido layer, the message layer and the negotiated transport layer. Data to be exchanged is first encoded in gidos. Then gidos are encapsulated in messages. Finally, messages are sent over a transport mechanism negotiated between the communicating components.

The purpose of the gido layer is to allow components to have a common understanding of the semantics of the data they send to each other. As the language to specify gidos, CISL provides a rich, extensible format with defined semantics so that the information being exchanged can be described. We will describe CISL in further detail in section 2.1.

The message layer is concerned with security, efficiency and reliability of the communication. Message format and message processing procedures are defined to ensure secure, reliable and efficient communication among CIDF components.

The transport mechanism is not part of the CIDF specification. However, a protocol based on the reliable UDP is selected as the default transport protocol. Other transport layer protocols may be used after a negotiation procedure.

CIDF also specifies how IDR components communicate with each other. A directory service based on the lightweight directory access protocol, called *matchmaking service* or *matchmaker*, provides a mechanism for CIDF components to advertise themselves and to locate communication partners with which they can share information.

2.1 Common Intrusion Specification Language

CISL is proposed to specify events, analysis results and responses among CIDF components [1, 5]. CISL is based on a Lisp-like general language construct, called S-expressions. S-expressions are syntactic constructs for nested groupings of tags and data, where the grouping is done with parentheses. For example, `(HostName 'ten.ada.net')` is a simple S-expression that groups two terms, `HostName` and `'ten.ada.net'`. The term immediately after the first left parenthesis of an S-expression (e.g., `HostName` in the example) is the *tag* of the S-expression, and the terms after the tag and before the matching right parenthesis (e.g., `'ten.ada.net'`) are the data values grouped with the tag. The data values grouped with a tag are called the argument of the tag. We also say an S-expression is headed by x if x is the tag of the S-expression. The argument grouped with a tag can be not only simple constants, but also S-expressions (i.e., nested S-expressions).

Tags are assigned with well-defined semantics in CIDF to help the interpretation of the data grouped with them. These tags are called Semantic IDentifiers, or SIDs. In the example above, `HostName` is an SID with the assigned semantics that the string `'ten.ada.net'` should be interpreted as a hostname.

SIDs are divided into several groups. The SIDs that denote actions, recommendations or state descriptions are called *verb SIDs*. An S-expression that has at least one verb SID appearing in it is called a *sentence*. A verb SID takes as its argument a sequence of S-expressions that tells the various aspects about the sentence. For example, in the following S-expression, `Delete` is a verb SID that denotes the action of *deletion*.

```
(Delete
    (Location
        (Time '15:30:34 10 June 1999')
    )
    (Initiator
        (UserName 'Joe')
    )
    (Source
        (Filename '/etc/passwd')
        (Owner
            (UserName 'root')
        )
    )
)
```

The SIDs that govern the information about the "players" associated with a verb are called *role SIDs*. A role SID takes as argument a sequence of S-expressions that identify or describe the entity playing the indicated role. An S-expression headed by a role SID is called a *role clause*. In the S-expression above, `Location`, `Initiator`, `Source` and `Owner` are all role SIDs.

There are two special kinds of role SIDs. A role SID that does not describe an object, but locates a verb SID or modifies its meaning is called an *adverb SID*. For example, in the S-expression above, `Location` is an adverb SID that governs the information about the the context of the deletion event. A role SID that can only occur directly under another role SID (i.e., the S-expression headed by the former role SID is an argument of the latter role SID) is called an *attribute SIDs*. For example, in the S-expression shown above, `Owner` is an attribute SID that governs the information about the owner of the deleted file. Attribute SIDs describe an entity that has a relation to another entity rather than to the whole sentence.

An SID that can only take a constant as its argument is called an *atom SIDs*. For example, in the S-expression shown above, `Time`, `UserName` and `Filename` are all atom SIDs. Atom SIDs can only appear inside role clauses. Atom SIDs give property values while verb and role SIDs organize the structure of the values. An S-expression headed by an atom SID is called an *atom clause*, or sometimes an *SID-data pair*.

There are two special atom SIDs, *ReferAs* and *ReferTo*. They are collectively called *referent SIDs*. Referent SIDs allow one to link two or more sentences together (or two or more parts of the same sentence). The SID `ReferAs` labels a role clause or a sentence, and the SID `ReferTo` refers to the role clause or the sentence using the label.

The SIDs that can join sentences together are called *conjunction SIDs* (i.e., each S-expression in the argument of a conjunction SID must contain at least one verb SID). Conjunction SIDs state the relationship among sentences.

For detailed definition of SIDs, please refer to the CISL document [1].

3 Query

As mentioned in the introduction, for the purpose of efficiency, an IDR component needs to describe specifically what they would like to request from other components. In this section, we will model such requests by extending the CISL.

Our extension is to add new SIDs and language constructs so that a CIDF component may specify requests using the extended language. The specification of a query is separated into two parts: a condition that the information of interest must satisfy and an optional specification of the particular information that must be returned to the requesting component. We first use special S-expressions called *S-patterns* to capture the former part, then describe the latter part either implicitly with the patterns or explicitly with specifications.

3.1 S-Patterns

An S-pattern is an S-expression with "wild-card terms" that stand for certain kinds of SIDs or data values, and conditions that these wild-card terms must satisfy. Similar to other pattern representations (e.g., regular expressions), an S-pattern is a syntactic notation for describing a set of S-expressions. For example, the following S-expression is an S-pattern, in which the SID `AVerb` is a wild-card SID that stands for all verb SIDs.

```
(AVerb
    (Initiator
        (UserName 'Joe')
    )
)
```

The two SIDs, `Initiator` and `UserName`, are as defined in the CISL specification, and `'Joe'` is the data value grouped with `UserName`. This pattern represents the set of all S-expressions that have the above form but with `AVerb` replaced with a particular verb SID, e.g., the S-expression derived from above by replacing `AVerb` with `Delete`.

Given the semantics assigned to the SIDs, an S-pattern expresses the information that an IDR component may be interested in. For example, the aforementioned S-expression represents all the actions or responses that are initiated by the user whose username is `Joe`.

There are two kinds of wild-card terms, wild-card SIDs and wild-card data values. We introduce several wild-card SIDs, each of which represents a class of SIDs (e.g., `AVerb` is a wild-card SID representing verb SIDs). To allow conditions refer to the SID that takes the place of a wild-card SID, we also introduce naming SIDs for wild-card SIDs so that we can give a wild-card SID a name and later use the name to specify the conditions.

A different approach is used to represent wild-card data values due to the restriction imposed by the CISL (which will be discussed later). We introduce a couple of *escape SIDs* such that when they take atom clauses as arguments (recall that data values only appear in atom clauses), the data values will be reinterpreted as wild-card data values. To facilitate the use of wild-card data values in conditions, the escape SIDs also assign names to the wild-card data values.

We describe conditions with predicates and logical combinations of predicates. Accordingly, we introduce SIDs that stand for predicates and logical operations to specify the conditions. Predicates determine the relationship among their arguments, and logical operations allow representation of complex relationships by combinations of predicates.

In summary, we add the following kinds of SIDs to support the specification of patterns.

- wild-card SIDs: `AVerb`, `ARole`, etc.
- naming SIDs for wild-card SIDs: `SIDReferAs` and `SIDReferTo`
- escape SIDs: `ValueReferAs` and `ValueReferTo`
- predicate SIDs: `EqualTo`, `LessThan`, etc.
- logical operation SIDs: `LogicalAnd, LogicalOr` and `LogicalNot`
- other SIDs: `Condition`, `Query` and `SIDValue`

In the following, we explain these SIDs in further detail.

3.1.1 Wild-card SIDs

A wild-card SID is essentially a variable which stands for any SID that can be placed at that position. We have seen an example using the wild-card SID `AVerb` at the beginning of section 3.1.

The following wild-card SIDs are added to the CISL to support S-patterns: AVerb, ARole, AnAdverb, AnAttribute, AConjunction, and AnAtom. They are wild-cards for the corresponding kinds of SIDs. AVerb is a verb SID that stands for any verb SID other than AVerb. It can be placed in an S-expression as a normal verb SID and represents any verb SID other than AVerb. ARole is a role SID that stands for any role SID other than ARole. It can be placed in an S-expression as a normal role SID and represents any role SID other than ARole. The wild-card SIDs AnAdverb, AnAttribute, AConjunction and AnAtom are similarly defined.

3.1.2 Naming SIDs and SIDValue

Two naming SIDs, SIDReferAs and SIDReferTo, are introduced to facilitate the use of wild-card SIDs in conditions. Following the convention of CISL, they are atom SIDs that take unsigned long integers as arguments. SIDReferAs is usually placed under a wild-card SID such that the S-expression headed by the SIDReferAs is in the argument of the wild-card SID. SIDReferAs gives names to wild-card SIDs, while SIDReferTo uses the SID by making a reference to the name.

For convenience of presentation, however, we will use "symbolic names" for the data values grouped with the naming SIDs hereafter. They can be easily transformed to unsigned integers.

Another SID, SIDValue, is introduced to facilitate the comparisons involving wild-card SIDs and "constant" SIDs. SIDValue is an atom SID that takes a valid SID as its argument. It instructs that its argument be interpreted as an SID. For example, the S-expression (SIDValue Delete) means that Delete is an SID.

The following S-expression shows an example of the naming SIDs for wild-card SIDs.

```
(AVerb
    (Initiator
        (UserName 'Joe')
    )
    (SIDReferAs var_verb)
    (Condition
        (NotEqual
            (SIDReferTo var_verb)
            (SIDValue Delete)
        )
    )
)
```

The clause (SIDReferAs var_verb) gives the name var_verb to the wild-card SID AVerb, and (SIDReferTo var_verb) later uses the wild-card SID by making reference to var_verb. Here NotEqual is a predicate SID which determines whether the verb SID represented by var_verb is SID Delete or not, and Condition is the SID that governs the conditions for the pattern. Thus, together with the semantics assigned to the SIDs, this S-expression represents the S-pattern for all the actions or responses initiated by user 'Joe' except for deletion. We will explain predicate SIDs and the SID Condition in more detail later.

Note that SIDReferAs/SIDReferTo SIDs are different from ReferAs/ReferTo. ReferAs/ReferTo link roles or sentences represented by the corresponding S-expressions, while SIDReferAs/SIDReferTo refer to wild-card SIDs.

3.1.3 Escape SIDs

Because of the restriction imposed by the CISL, we cannot use the same naming mechanism as we used for wild-card SIDs. Since sentences specified in CISL are intended to be encoded in binary forms and some types of data values are assigned a fixed-length field (e.g., IPV4Checksum is a 16-bit integer), we cannot use a special data value as a wild-card term (otherwise, we won't be able to represent some data value, for example, a checksum of an IP packet). Thus, we introduce a couple of escape SIDs such that when they take atom clauses as their arguments, the data values will be reinterpreted as wild-card data values.

Two escape SIDs, ValueReferAs and ValueReferTo, are introduced to represent wild-card data values and assign names to the data values.

An escape SID takes atom clauses as its argument, and causes the data values grouped with the atom SID to be interpreted as "names" for wild-card data values. Also, within an S-expression headed by an escape SID, an atom SID takes an unsigned long integer as argument (for convenience of presentation again, we shall use symbolic names instead of integers). For example, the S-expression

```
(ValueReferAs
    (UserName var_uname)
)
```

makes var_uname being interpreted as a reference to a possible username. An S-expression headed by an escape SID is called an *escape clause*, and can be used as an atom clause.

As SIDReferAs and SIDReferTo, ValueReferAs gives a name to a data value that is grouped with an atom SID, and `ValueReferTo` catches this value by making a reference to the name. For example, in the following S-expression that represents an S-pattern of all the delete actions initiated by users other than Joe, the possible value of `UserName` under the `Initiator` role clause is given a name `var_uname`, and later `ValueReferTo` makes a reference to this value under the `NotEqual` SID.

```
(Delete
    (Initiator
        (ValueReferAs
            (UserName var_uname)
        )
    )
    (Condition
        (NotEqual
            (ValueReferTo
                (UserName var_uname)
            )
            (UserName 'Joe')
        )
    )
)
```

3.1.4 Predicate SIDs

Predicate SIDs are introduced to help state conditions for patterns in S-expressions. Predicate SIDs are role SIDs. A predicate SID takes as its argument role clauses, atom clauses and (or) escape clauses, representing the relationship among them.

An S-expression headed by a predicate SID is called a *predicate clause*, or more specifically a *simple predicate clause*. A predicate clause is expected to return True or False according to whether the relationship holds or not.

The following S-expression shows an example of predicate clause that represents whether the host denoted by `var_host` is in the domain `'ada.net'`. Here we assume that `var_host` is defined by `ValueReferAs` somewhere else.

```
(HostInDomain
    (ValueReferTo
        (FQHostName var_host)
    )
    (DomainName 'ada.net')
)
```

Predicate SIDs are important for the expressiveness of the extended language. Since the original language was not developed for specifying requests, no SIDs pertaining to request are included in the language specification. To fully support CIDF components to ask queries, extensive work is needed to determine what predicate SIDs should be provided. We will not give a complete list for predicate SIDs in this paper, but consider it as future work. The predicate SIDs used in the examples of this paper are `HostInDomain`, `NotEqual` and `LessThan`, whose semantics are explained along with the examples.

3.1.5 Logical Operation SIDs

Logical operation SIDs are used to represent complex conditions by logically combining simple predicate clauses. Logical operation SIDs are role SIDs representing logical operations. Corresponding to the three logical operations AND, OR and NOT, three logical operation SIDs, `LogicalAnd`, `LogicalOr` and `LogicalNot`, are used.

A logical operation SID takes as its argument a sequence of predicate clauses (in the case of `LogicalNot`, only one S-expression is allowed as its argument), representing the result of applying the corresponding logical operation to the argument. An S-expression headed by a logical operation SID is also called a predicate clause, or more specifically a *complex predicate clause*. Therefore, logical operation SIDs along with predicate SIDs can recursively express complex conditions. For example, the following S-expression shows a condition that `var_verb` is not `Delete` and `var_host` is within the `'ada.net'` domain. Here we assume that `var_verb` and `var_host` have been defined.

```
(LogicalAnd
    (NotEqual
        (SIDReferTo var_verb)
        (SIDValue Delete)
    )
    (HostInDomain
        (ValueReferTo
            (FQHostName var_host)
        )
        (DomainName 'ada.net')
    )
)
```

3.1.6 SIDs `Condition` and `Query`

The SIDs `Condition` and `Query` are used to organize S-expressions for conditions and patterns, respectively. The SID `Condition` is introduced to govern the S-expressions that describe the conditions that must be satisfied by the wild-card terms. The `Condition` SID takes as argument an

S-expressions headed by predicate SIDs or logical operation SIDs (i.e., a predicate clause). An S-expression headed by the `Condition` SID is called a *condition clause*. To make conditions specific, we require that each condition clause be placed directly under a verb or `AVerb` SID, meaning that the condition must be satisfied when the action, description of analysis result or response corresponding to the verb SID occurs.

The SID `Query` is introduced to govern the S-expressions that describe a pattern. `Query` SID is a conjunction SID that takes as its argument a sequence of sentences. An S-expression headed by `Query` SID is called a *query sentence*.

With the SIDs introduced earlier, CIDF components are provided with support for specification of S-patterns. The following example shows a complete S-pattern.

Example 1 Suppose a component wants to know from what IP address and port a user with username `'Joe'` telnets to the host having IP address `'10.0.0.3'`. The component can specify this request using the following pattern.

```
(Query
   (Login
      (Initiator
         (UserName 'Joe')
      )
      (Session
         (ValueReferAs
            (SourceIPV4Address s_IP)
            (TCPSourcePort s_port)
         )
         (DestinationIPV4Address
          '10.0.0.3')
         (TCPDestinationPort 23)
      )
      (Condition
         (NotEqual
            (ValueReferTo
               (SourceIPV4Address s_IP)
            )
            (DestinationIPV4Address
             '10.0.0.3')
         )
      )
   )
)
```

3.2 Format of Returning Message

With the extension introduced above, a CIDF component is able to describe its interest as S-patterns. However, S-patterns are not specific enough to express the exact request of a component. In other words, it is not clear what should be included in the reply to a query and how the information should be arranged. As a result, it is still possible that a requesting component gets unnecessary messages or misses important information. Thus, additional mechanism is needed in order to provide a sufficient solution.

There are alternative ways to solve this problem. The content and the arrangement of the reply (i.e., the format of the returning message) can be specified either implicitly or explicitly in the request. In the following, we will discuss them respectively.

3.2.1 Implicit Request for Returning Message

The information that must be returned can be specified implicitly. When a component finds the information that matches a request, it is required to instantiate all the wild-card terms in the request using the information and return the resulting S-expression. This approach basically assigns a "returning all" semantic to a query sentence.

In order to have the requested information, a component has to list the particular aspects of information as wild-card SIDs or wild-card data values.

For example, with the assigned "returning all" semantic, the query sentence shown in example 1 specifies a request for all the session information about user Joe's telnet sessions. The requesting component can directly send the query sentence to the event analyzer that monitors the host. Suppose the event analyzer that receives the query does find a telnet session to `'10.0.0.3'` initiated by user `'Joe'` and the source IP address and the source port number are `'129.174.40.15'` and 6543, respectively. It will instantiate the wild-card data values `s_IP` and `s_port` in the above query and return the following S-expression.

```
(Login
    (Initiator
        (UserName 'Joe')
    )
    (Session
        (SourceIPV4Address
         '129.174.40.15')
        (TCPSourcePort 6543)
        (DestinationIPV4Address
         '10.0.0.3')
        (TCPDestinationPort 23)
    )
    (Condition
        (NotEqual
            (SourceIPV4Address
             '129.174.40.15')
            (DestinationIPV4Address
             '10.0.0.3')
        )
    )
)
```

The advantage of this approach is its simplicity. By assigning the "returning all" semantic to a query sentence, it doesn't need any additional mechanisms. In addition, the relationships represented by the conditions in queries are also kept in the returning message. Thus, the returning messages are complete in the sense that all the related constraints are contained in the message.

However, some information may be redundant in the returning message. Some part of the query may be presented to specify the conditions that the requested information should satisfy, and may not be of interest to the requesting component. If we use this approach, these parts will have to be returned by the requested component.

3.2.2 Explicit Request for Returning Message

Alternatively, we can specify explicitly what should be returned and how the information is arranged. We introduce an additional component into a query, which arranges the format of returning messages.

A new SID, `Format`, is introduced to govern S-expressions that specify the format. `Format` is a conjunction SID that takes sentences as argument. It is always placed directly under the SID `Query`, representing the request by the query for particular messages. S-expressions under the SID `Format` are described using constants as well as references to wild-cards and data values that appear in the pattern of the same query. An S-expression headed by the SID `Format` is called a *format sentence*.

A format sentence describes the requested aspect of the information. When there is information that matches the S-pattern, related aspects are extracted and described in S-expression according to the format sentence.

For example, with the explicit approach, the query about the telnet session shown in example 1 can be specified as follows.

```
(Query
   (Login
      (Initiator
         (UserName 'Joe')
      )
      (Session
         (ValueReferAs
            (SourceIPV4Address s_IP)
            (TCPSourcePort s_port)
         )
         (DestinationIPV4Address
          '10.0.0.3')
         (TCPDestinationPort 23)
      )
      (Condition
         (NotEqual
            (ValueReferTo
               (SourceIPV4Address s_IP)
            )
            (DestinationIPV4Address
             '10.0.0.3')
         )
      )
   )
   (Format
      (Login
         (Session
            (ValueReferAs
               (SourceIPV4Address s_IP)
               (TCPSourcePort s_port)
            )
         )
      )
   )
)
```

When the event analyzer finds the corresponding telnet session information, it will arrange the session information according to the S-expression under the SID `Format` and return the following responding message.

```
(Login
   (Session
      (SourceIPV4Address
       '129.174.40.15')
      (TCPSourcePort 6543)
   )
)
```

Using the explicit approach, the responding message can be shorter than when the implicit approach is used, since at least those parts that specify conditions in the pattern can be omitted from the responding message. This will save network bandwidth and processing time for the reply.

However, the requesting messages usually become larger because of the explicit specification of the general form of responding messages. In addition, the requesting components should be able to link responding messages to requesting ones. In other words, the requesting components must know which responding message replies to which requesting message. Here we assume that there exist other mechanisms that link the corresponding requesting and responding messages together. A simple solution could be embedding in the reply the identifier of the requesting message.

3.3 An Example - Tracing Suspicious Users

We conclude this section with an example of tracing suspicious users. Tracing techniques have been studied by various research groups and some solutions specialized for this problem have been proposed (e.g., thumbprinting [12]). Here we show how we can achieve the same purpose through the cooperation of CIDF components that comunicate using the extended CISL. The language extension is certainly not limited to this problem.

Suppose the Event Analyzer monitoring host A detects a suspicious user who remotely logged in from host B and wants to trace the origin of this user. The Event Analyzer discovers that the user was connected from host B to host A through a telnet session beginning at 14:45:36 on May 17 1999, and the telnet session is carried over a TCP connection from IP address '10.0.0.2' port 4321 to IP address '10.0.0.1' port 23 ('10.0.0.1' and '10.0.0.2' are IP addresses of host A and B, respectively). Then the Event Analyzer for host A can start tracing by posing the following query to the Event Analyzer for host B instead of getting all login-related events from the corresponding Event Generator.

```
(Query
   (Login
      (Location
         (ValueReferAs
            (Time login_time)
         )
      )
      (Initiator
         (ValueReferAs
            (HostName src_host)
         )
      )
      (Session
         (ValueReferAs
            (SourceIPV4Address s_IP)
            (TCPSourcePort s_Port)
            (DestinationIPV4Address d_IP)
            (TCPDestinationPort d_Port)
         )
         (ReferAs first_session)
      )
      (Account
         (HostName B)
      )
      (Condition
         (NotEqual
            (ValueReferTo
               (HostName src_host)
            )
            (HostName B)
         )
      )
      (ReferAs first_login)
   )
   (Login
      (Location
         (Time '14:45:36 17 May 1999')
      )
      (Session
         (SourceIPV4Address '10.0.0.2')
         (TCPSourcePort 4321)
         (DesinationIPV4Address '10.0.0.1')
         (TCPDestinationPort 23)
         (Ancestor
            (ReferTo first_session)
         )
      )
      (Condition
         (LessThan
            (ValueReferTo
               (Time login_time)
            )
            (Time '14:45:36 17 May 1999')
         )
      )
   )
   (Format
      (ReferTo first_login)
   )
)
```

Informally, this query asks: From where and when did the suspicious user log into host B, given the clue that he (or she) telneted to host A through a TCP connection from IP address `'10.0.0.2'` port

4321 to IP '10.0.0.1' port 23 at time '14:45:36 on May 17 1999?

In the query sentence, the `Query` SID takes as argument three sentences. The first sentence is headed by a `Login` SID, representing the login event that the suspicious user logged into host *B* remotely. The references to login time and the parameters for the TCP session expresses the interest of the requesting component, and the condition explains that the initiating host should be one different from host *B*. The second sentence is also headed by a `Login` SID, representing the login event that the suspicious user remotely logged into host *A* from host *B*. The `Location` clause and the `Session` clause specify the login time and TCP connection that carried the login event, respectively. The attribute clause headed by `Ancestor`, which is under the `Session` SID, also requires that this session must be started within the session of the first login event (i.e., the session referred by `first_session` is an ancestor session of the second session). The condition clause of the second sentence explains that the first login event should be before the second one. The third sentence states that the returning message should be in the same form as the first login sentence, which is denoted by the reference `first_login`.

Suppose the Event Analyzer for host *B* discovers that the suspicious user logged into *B* from host 'another.hop.com' at time '14:40:04' on May 17 1999, and the TCP connection that carried this remote login was from '129.174.142.177' port 1234 to '10.0.0.2' port 23 (i.e., a telnet session). Then the Event Analyzer will return a responding message as follows (according to the format sentence in the requesting message).

```
(Login
   (Location
      (Time '14:40:04 17 May 1999')
   )
   (Initiator
      (HostName 'another.hop.com')
   )
   (Session
      (SourceIPV4Address '129.174.40.15')
      (TCPSourcePort 1234)
      (DestinationIPV4Address '10.0.0.2')
      (TCPDestinationPort 23)
   )
   (Account
      (HostName B)
   )
   (Condition
      (NotEqual
         (HostName 'another.hop.com')
         (HostName B)
      )
   )
)
```

After receiving the responding message, the Event Analyzer for host *A* may send the Event Analyzer for host 'another.hop.com' a similar query sentence with the new information to determine the origin of the user. This process may continue until the origin of the user is found.

4 Discussion

4.1 Impact on CIDF

By extending CISL, we add a new facility into CIDF, namely specification of request for particular information. CIDF components are given a mechanism to specify requests for selected information so that message processing effort, storage capacity and network bandwidth can be saved. However, it also imposes new requirements on CIDF components.

In order to take advantage of the new functionality, a CIDF component has to decide what to request according to its needs, and describes them in correct forms. This requires that the component not only understand the language used to describe requests (i.e., CISL), but also send right requesting messages to the right partners when necessary. This seems to be a strong requirement. However, this requirement can be satisfied by classifying typical situations and arranging possible requesting messages ahead of time. Rule-based expert systems may help to generate requests automatically. Of course, requests may also be improvised by system administrators or site security officers to handle exceptional situations.

When cooperating with components from which requesting messages have been received, a CIDF component should understand the requesting messages correctly, find the necessary information, and send back the replying messages in the CISL. This requires that the component have some mechanisms to find the requested information. This requirement is outside the scope of the original CIDF. One possible way to generate reply for a query is to take advantage of the signature-based intrusion detection techniques, which are well studied and widely adopted [3, 6, 7, 8]. Since both queries in our language extension and the signature-based intrusion detection techniques are based on patterns, it is possible to translate a query in the extended CISL into a description in a certain signature-based intrusion detection model, find the answer using existing intrusion detection software modules (which is possibly modified), and translate the result back to description in the CISL.

More SIDs than those described in this paper may be needed for CIDF components to specify requests. Indeed, the SIDs introduced in the previous section are the minimum set of SIDs that provide the language support for specifying requests. Since the original language specification is developed for CIDF components to make statements about events, analysis results and responses, the SIDs may not be enough for making queries. For example, there is no SIDs that direct a CIDF component to collect statistics for certain events.

Determining additional SIDs that are needed for queries will involve extensive exploration of CIDF components' requirements. We don't discuss this issue in this paper but consider it as future work.

4.2 Alternative Approaches

There are alternative approaches in addition to extending CISL. The simplest solution is to classify the information into several classes, each of which represents one kind of information that may be requested by an IDR component. The original CIDF proposal adopts a similar approach, where information is sent to an IDR component according to the gido classes being requested [4]. The major drawback of this approach is the lack of flexibility. An IDR component will not be able to express a request that is not predefined. One can certainly try to make up by listing all possible requests. However, even a huge number of predefined requests cannot ensure that there are no exceptions. On the contrary, our extension to CISL provides some basic language constructs that allow flexible specification of requests.

Another alternative approach is to use mobile code. One IDR component may send another component a piece of mobile code (e.g., Java script), which collects the desired information and sends it back using CISL. This approach is more expressive and flexible than ours. However, the disadvantages are also serious. First, the component executing the mobile code is under more security risk, since the mobile code, which is on behalf of a remote system, may bypass the security control and perform some malicious actions. Second, since it is working in a CIDF environment, the mobile code has to be able to "speak" in CISL. This is a strong requirement because a piece of mobile code has to bring with itself a parser (for S-expression) and have access to the dictionary of the SIDs. On the other hand, our approach extends CISL by adding query facility. A request specified in the common language can be sent to a remote IDR component as a query or set up as a trap, and the requested information can then be collected and returned by the remote component if its security policy permits.

Furthermore, some query languages (e.g., SQL) seem to be good choices because of their expressiveness and flexibility. However, query languages usually assume particular data models that they work on and these data models are usually quite different from what is considered by CISL. For example, a SQL query statement takes one or more relations as input and produce a relation as output, while there is no relation at all in CISL. In addition, an IDR component may not understand a query specified in a query language like SQL.

4.3 Query Templates

Query templates may be used to reduce the work involved in writing queries. A query template specifies a request for information using some parameters. When interoperating with other CIDF components, a requesting component can replace the parameters with constants and send out the instantiated query. Libraries of query templates may be published and shared among many systems.

One way to take advantage of query templates (and the query facility) is to use them along with some triggering mechanism such as rule-base expert system. For example, we may associate appropriate query templates with the rules in a rule-based expert system. When these rules are fired due to certain events, the associated templates can be instantiated using the event attributes and sent to other IDR components. The automatic generation and processing of IDR queries is an interesting topic; however, we do not cover it in this paper but consider it as future work.

5 Conclusion and Future Work

This paper described the result of an ongoing research effort. Extensions to the Common Intrusion Specification Language were designed on the basis of a formal approach for general IDR systems [10]. The goal of this work was to provide language support for components in the Common Intrusion Detection Framework so that they can request specific information from interoperating partners. Based on the original language, several kinds of semantic identifiers were added and requests can be specified as patterns in the extended language.

Several issues are worth further research. The first is to decide additional SIDs that should be included to fully support CIDF components' requests. The second is to define libraries of query templates for frequently used requests. In addition, the automatic generation and processing of IDR queries is also an interesting work that will facilitate the deployment of the query facility. Finally, approaches to efficiently obtain query results are also important.

References

[1] R. Feiertag, C. Kahn, P. Porras, D. Schnackenberg, S. Staniford-Chen, and B. Tung. A common intrusion specification language (CISL). http://seclab.cs.ucdavis.edu/cidf/cisl_current.txt, 1998.

[2] IETF Intrusion Detection Working Group. Intrusion detection exchange format. http://www.ietf.org/html.charters/idwg-charters.html.

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

[4] C. Kahn, D. Bolinger, and D. Schnackenberg. Communication in the common intrusion detection framework. http://seclab.cs.ucdavis.edu/cidf/cidfcomm.txt, 1998.

[5] C. Kahn, P. A. Porras, S. Staniford-Chen, and B. Tung. A common intrusion detection framework. Submitted to Journal of Computer Security, July 1998.

[6] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, August 1995.

[7] S. Kumar and E. H. Spafford. A pattern matching model for misuse intrusion detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, October 1994.

[8] J. Lin, X. S. Wang, and S. Jajodia. Abstraction-based misuse detection: High-level specifications and adaptable strategies. In *Proceedings of the 11th Computer Security Foundations Workshop*, pages 190–201, Rockport, MA, June 1998.

[9] T. Lunt and C. McCollum. Intrusion detection and response research at DARPA. Technical report, The MITRE Corp., 1999.

[10] P. Ning, X. S. Wang, and S. Jajodia. Modeling requests among cooperating intrusion detection systems. To appear in the special issue "Advances in Research and Application of Network Security" of the Computer Communications Journal, 2000.

[11] P. Porras, D. Schnackenberg, S. Staniford-Chen, M. Stillman, and F. Wu. The common intrusion detection framework architecture. http://seclab.cs.ucdavis.edu/cidf/draft.txt, 1998.

[12] S. Staniford-Chen and L. Heberlein. Holding intruders accountable on the internet. In *Proceedings of 1995 IEEE Symposium on Security and Privacy*, pages 39–49, Oakland, May 1995.

[13] B. Tung. CIDF APIs: Their care and feeding. http://seclab.cs.ucdavis.edu/cidf/apis.txt, 1998.

[14] J. Yang, P. Ning, X. S. Wang, and S. Jajodia. CARDS: A distributed system for detecting coordinated attacks. To appear in the 15th International Conference on Information Security (SEC 2000), August 2000.