Intrusion Detection Inter-component Adaptive Negotiation[1]

Richard Feiertag, Lee Benzinger, Sue Rho, Stephen Wu
{feiertag, benzinger, rho, wu}@tislabs.com
NAI Labs at Network Associates, Inc.
3965 Freedom Circle
Santa Clara, CA 95054

Karl Levitt, Dave Peticolas, Mark Heckman
{levitt, peticola, heckman}@cs.ucdavis.edu
Computer Science Department
University of California, Davis
Davis, CA 95616

Stuart-Staniford-Chen
stuart@silicondefense.com
Silicon Defense
791 Shirley Blvd
Arcata, CA 95521

Cui Zhang
zhangc@ecs.csus.edu
Computer Science
Department
California State University
Sacramento, CA 95819-6021

September 29, 1999

[1]This work was supported by DARPA under contract F30602 - 97 - C - 0187

Abstract

The Intrusion Detection System (IDS) community is developing better techniques for collecting and analyzing data in order to handle intrusions in large, distributed environments [1, 5, 6]. To take advantage of this ongoing work, IDSs should be able to dynamically adapt to new and improved components and to changes in the environment. The Intrusion Detection Inter-component Adaptive Negotiation (IDIAN) project has developed a negotiation protocol to allow a distributed collection of heterogeneous ID components to inter-operate and reach agreement on each other's capabilities and needs – i.e., the information that can be generated and processed. Moreover, the negotiation is dynamic so the information generated and processed can evolve as the IDS evolves or the environment changes.

The IDIAN project leverages the Common Intrusion Detection Framework (CIDF), an effort by DARPA to develop a common language, protocols, and APIs that allow intrusion detection components to interoperate and share information. The IDIAN project has extended the CIDF language CISL (Common Intrusion Specification Language) with constructs for dynamic negotiation. One such construct is the notion of a *filter* to specify sets of IDS messages. Filters are useful in negotiating, for example, what data will be transmitted. The IDIAN project also adopts the CIDF framework architecture that classifies ID components according to their function.

The negotiation protocol uses the notion of a *contract* – an association between two ID components, a producer and a consumer – which contains one or more possible *agreements*. An agreement commits the producer to provide the consumer with a set of services. For example, a detection component (producer) might have a contract with an analysis component (consumer) to provide a specific set of audit data. At any give time, at most one of the agreements in a contract is in effect, although the ID components may switch to one of the alternatives dynamically. Furthermore, two components may have multiple contracts operating at the same time.

The primary function of the protocol is to allow ID components to dynamically negotiate new contracts/agreements and to change existing ones. An extension of the protocol is designed to handle hierarchical negotiations in which top-level components obtain the services of low-level components through intermediaries.

To facilitate choosing among several options, the protocol uses the notion of *cost* to capture the relative cost to a producer (resp., consumer) to provide (process) a specific set of services. Services provided by producers require a variety of system resources. A consumer may decide to use a particular service only if the cost is below a certain threshold. The absolute and relative amount of resources required to supply a particular service may vary over time, and the protocol allows producers and consumers to renegotiate when necessary.

In addition to the protocol, the IDIAN project has developed several scenarios that demonstrate situations where an IDS must adapt to a changing environment. The scenarios can be divided into two general classes:

1. The acquisition of a new capability by the IDS. For example, an ID component may acquire a new attack signature or a new ID component may come on-line. The IDS must adapt by incorporating the new signature or ID component into the overall system.

2. An overload of the IDS caused, for example, by faults in the IDS itself or by a flooding attack. The IDS could adapt by reducing the amount of information being gathered, or by cutting off the flow of data from the flooding source.

The IDIAN project has developed a demonstration of the protocol in which a new producer comes on-line and negotiates with an existing consumer. The new producer can report buffer overflow attacks detected by the StackGuard [2] system.

Finally, the IDIAN project is using the formal description language Estelle [3] to specify the negotiation protocol.

1 Introduction

The Intrusion Detection System (IDS) community is developing better techniques for collecting and analyzing data in order to handle intrusions in large, distributed environments [1, 5, 6]. To take advantage of this on-going work, IDSs should be able to dynamically adapt to new and improved components and to changes in the environment. The Intrusion Detection Inter-component Adaptive Negotiation (IDIAN) project has developed a negotiation protocol to allow a distributed collection of heterogeneous ID components to inter-operate and reach agreement on each other's capabilities and needs – i.e., the information that can be generated and processed.

This paper presents several different components of this research. Section 2 presents background information on the Common Intrusion Detection Framework, a body of research which the IDIAN project has leveraged. Section 3 presents a set of scenarios which demonstrate the need for adaptive negotiation. Section 4 presents research in constructs for expressing component capabilities, and Section 5 presents the protocol itself. Section 6 describes the testbed used to demonstrate the protocol in operation and Section 7 concludes the paper and explores areas of future research.

2 Common Intrusion Detection Framework

The Common Intrusion Detection Framework (CIDF) [4] is an effort by DARPA to develop a common language, protocol, and API that allow ID components to inter-operate and share information. Since a thorough exposition of CIDF would be too lengthy, we will only give enough information to understand the rest of the document.

2.1 CIDF Architecture

The CIDF architectural model divides an IDS into components, all of which have a persistent identity. New components may be introduced and other components may be removed. However, the model assumes that the lifetime of a component is long compared to both the time required to deploy the component and to the duration of intrusion incidents. Components consist mainly of software code with some configuration information (with the exception of database components which store extensive amounts of data).

CIDF components interact in a real-time dataflow model and exchange data using Generalized Intrusion Detection Objects (GIDOs). GIDOs consist solely of data (not code) and carry information about possible intrusions and responses to intrusions. GIDOs are discussed in greater detail in Section 2.2.

CIDF defines four types of components. First, event generators passively monitor sources of information and transmit that information using GIDOs. For example, an event generator might monitor an IP network and turn packet level information into GIDOs that are sent to other components. Another event generator might monitor host audit trails. Event generators send GIDOs to analyzers, the second type of component. Analyzers examine the incoming GIDOs, draw conclusions about what intrusive activity might be occurring, and create new GIDOs that encapsulate these conclusions and, possibly, prescribe responses.

Third, response components accept GIDOs that order a particular response (for example, killing a connection), and carry out the response. Finally, database components store GIDOs and provide answers to queries.

2.2 Generalized Intrusion Detection Objects

A GIDO consists of two components: a fixed format header and a variable length body. (There may also be GIDO addenda and signatures, but these are beyond the scope of this discussion.) The header contains information such as the version of CIDF in use, a timestamp, and the length of the body.

We explain the structure of the body using a sample body (Figure 1) that might be generated by the StackGuard-based event generator described in Section 6. This GIDO body expresses the fact that an attack occurred on the host `somehost.someplace.net`, was detected by the `StackGuard` process, and was directed at the `fingerd` program.

The structure of GIDO bodies is that of Lisp S-expressions; the parse tree of the expression is explicitly delineated with parentheses. Two kinds of constructs occur within the GIDO body. The first is actual data (all of which is shown in single quotes in Figure 1). The second type of construct is a Semantic Identifier (SID). There is a SID associated with each piece of data in the body. For example, the SID associated with the data `'fingerd'` is `ProcessName`, indicating that the string `'fingerd'` refers to a process name. SIDS which are directly associated with data are called Atom SIDS.

The Atom SIDS alone, however, are not sufficient. Above the Atom SIDS in the parse tree are additional SIDS which describe how the atoms of data

Figure 1: A sample GIDO body

```
(ByMeansOf
   (Attack
      (Observer (ProcessName ‘StackGuard’) )
      (Target (HostName ‘somehost.someplace.net’) )
      (AttackSpecifics
         (Certainty ‘100’)
         (Severity ‘100’)
         (AttackID ‘1’ ‘0x4f’) )
      (Outcome (CIDFReturnCode ‘2’) )
      (When
         (BeginTime ‘14:57:36 24 Feb 1999’)
         (EndTime ‘14:57:36 24 Feb 1999’) ) )
   (ByMeansOf
      (Execute
         (Process (ProcessName ‘fingerd’) )
         (When
            (BeginTime ‘14:57:36 24 Feb 1999’)
            (EndTime ‘14:57:36 24 Feb 1999’) ) ) ) )
```

combine to form a more complex meaning. For example, the `Attack` SID in the example is a Verb SID. Verbs describe events, in this case an attack.

Every Verb SID has a set of Role SIDS that may appear beneath it. Role SIDS provide additional information about the event (beyond the mere fact that, e.g., an attack occurred). For example, the `Observer` Role provides information about the observer of an event (in this case the `StackGuard` process). Roles are completed by supplying the Atoms beneath them (such as `ProcessName`); the Atoms tie the actual data into the structure.

Multiple events, each with their own verb-headed S-expression, can be joined together with conjunction SIDS, such as `And`. In the example, the attack is linked to the `fingerd` process with the `ByMeansOf` conjunction.

GIDOs are flexible in that a GIDO producer can leave out any unavailable information and may choose what information to include.

We note that the ASCII representation of the GIDO body shown above is not the format used in transmission. The CISL specification defines a compact GIDO representation for transmission and storage.

Finally, because the GIDO body is the main component of a GIDO, hereafter GIDO bodies are referred to as GIDOs.

3 Negotiation/Adaptation Scenarios

This section describes some of the scenarios which the IDIAN project has developed to illustrate the need for adaptive negotiation.

3.1 New ID Component or Capability

ID systems must function continuously; any gaps in operation leave the protected system vulnerable to attack. However, the configuration of an IDS may be dynamic. Specifically, an IDS should incorporate new resources and technology, in the form of new ID components or new capabilities of existing components, while maintaining continuous operation.

In order to incorporate a new ID component, the other components in the IDS must be aware of the new component's capabilities, i.e., the data it can provide or consume. Additionally, the other components may only want to use a subset of the new capabilities in order to conserve resources and to utilize resources efficiently.

For example, a network sniffer may be able to provide data on every packet traversing the network to which it is connected. However, a specific analyzer may only be able to detect intrusions based on the file transfer protocol (FTP). Therefore, the analyzer should be able to negotiate with the sniffer to obtain only FTP packets. This would allow both components to operate more efficiently in that the sniffer only needs to format and transmit FTP packets and the analyzer will only receive packets it can analyze. Thus, the negotiation protocol must provide:

1. The ability to publish the capabilities of a new component. This requires the ability to describe capabilities and to disseminate descriptions to other components.

2. The ability for collections of components to determine a specific set of capabilities that they will use. Thus, the protocol must be able to describe the data to be exchanged and to provide a quantitative measure of the resources required to provide that data. Other information may

also be required, such as the latency for providing the data or a negotiation progress metric to ensure that the negotiation will terminate.

These abilities are also useful when new capabilities are added to an existing component. For example, new response capabilities may be added to an operating system, such as the ability to maintain a log of a user's actions. The OS would announce these capabilities and possibly negotiate with an analyzer to accept requests for these new responses.

3.2 Overloading and Flooding

The second class of scenarios where adaptation and negotiation are important are those in which the IDS becomes overloaded. Overloading can occur due to faults on the network, on hosts, in applications, or within the IDS itself. Alternatively, overloading might be due to deliberate attempts to flood the IDS by an attacker.

For example, an event generator which is monitoring an on-going attack could generate so much audit data that the analyzer is unable to handle the load. In such a situation, an analyzer may wish to:

- Request the event generator to switch to a pre-negotiated "fallback" setting in which only critical audit data is sent.
- Request that other event generators reduce their output so the analyzer can concentrate on the attack.

Thus, the negotiation protocol must support both the negotiation of "fallback" or alternative positions and dynamic switching among those positions.

4 GIDO Filters

Negotiating over services requires the ability to express what services are offered or desired. For example, an event generator must be able to express the particular set of events that it can detect. Conversely, an analyzer must be able to express the set of audit data it can analyze.

Since, in the CIDF framework, all communication is in the form of GIDOs, the services which, for example, an event generator can provide can be represented by the set of all GIDOs it can generate. Consider again the GIDO in Figure 1. That GIDO is generated in response to a StackGuard detection of a stack overflow. The structure of the generated GIDO is always the same – only the data fields change. Thus, the service the StackGuard detector provides is represented by the set of all such GIDOs.

Similarly, the audit data which an analyzer requires might be represented by the set of all GIDOs it can process. Finally, the services of a response component could be represented as the set of response GIDOs it understands and can carry out.

The IDIAN project has developed the notion of a GIDO *filter*, a construct used to describe a set of GIDOs. GIDO filters are themselves GIDOs (or, perhaps, meta-GIDOs) that ID components can use to specify the services they provide or require.

4.1 Filter Requirements

The IDIAN project has identified a number of requirements for GIDO filters. We list the main requirements below:

1. Filters should be expressive enough that components can specify all sets of GIDOs that are useful to them. In particular, it must be possible to specify only part of the GIDO required to match the filter, allowing the GIDO to contain additional information that is not of interest. It must be possible to filter on any data field in the GIDO, and on any combination of data fields.

2. Filters should allow the possibility to specify sets of hosts, users or other categorical variables in a convenient way.

3. Filters should allow the extraction of particular data values from matching GIDOs, so that only the data of interest, rather than the whole GIDO, is sent.

4. The filter language should allow for efficient implementations.

5. Given two filters, F_1 and F_2, which match sets of GIDOs G_1 and G_2 respectively, it should be possible to define unambiguously a filter matching the union of G_1 and G_2, and a filter matching the intersection of G_1 and G_2. This requirement facilitates the construction of more complex filters when, for example, an event generator can produce many different kinds of event GIDOs.

6. Filters should have conceptual clarity. People who understand GIDOs should be able to easily write and understand GIDO filters.

4.2 Filter Design

In general, the format of a filter is the same as the format of a GIDO as explained in Section 2.2, with certain extensions. The main difference between an ordinary GIDO and a filter is in the body. The basic filter body starts with the `Filter` SID as shown in Figure 2.

Figure 2: A sample filter

```
(Filter
   (Fragment
      (Attack
         (Observer (ProcessName ‘observer:exp₁’) )
         (Target (HostName ‘target:exp₂’) ) ) )
   (Permit ‘ByMeansOf’)
   (Variables ‘observer’ ‘target’) )
```

A `Filter` can contain two SIDS: `Fragment` and `Permit`. A fragment specifies a “piece” of a GIDO. A GIDO matches a fragment if the piece specified by the fragment occurs anywhere in the GIDO, although the GIDO may contain extra information not in the fragment.

A fragment contains variable names and data expressions which appear in the fragment in place of actual data. A variable/expression pair will match any actual data in a GIDO which matches the expression. Thus, the GIDO in Figure 1 matches the fragment in Figure 2, with the variables `observer` and `target` instantiating to `‘StackGuard’` and `‘somehost.someplace.net’` respectively (assuming expressions exp_1 and exp_2 match those data items). The use of variables in filters allows only the relevant information (i.e., observer and hostname) to be transmitted rather than the whole GIDO.

The `Permit` SID specifies any number of SIDS which must appear at the top of a GIDO's parse tree. A GIDO matches a filter if it matches both the fragment and the permit clauses. The example filter specifies that matching GIDOs must have `ByMeansOf` as their first SID, and thus the GIDO of Figure 1 matches the filter of Figure 2.

Basic filters may be joined using `AndFilter` and `OrFilter`. `AndFilter` expressions must appear below the `OrFilter`, if present. Example:

```
(OrFilter (AndFilter (Filter ...) (Filter ...) ...) (Filter ...))
```

A GIDO matches an `OrFilter` (`AndFilter`) if it matches any (all) of the child clauses.

Please note the design of filters is an active area of research. This section presented the current state of GIDO filters as of this writing.

5 The Negotiation Protocol

This section describes the components of the IDIAN negotiation protocol, including the state model, message types, and the finite state machine description of protocol interactions.

5.1 State Model

Under the protocol, each component has a state composed of the elements below.

Agreement An agreement is a relationship between a producer and a consumer. An agreement specifies a set of services which the producer must provide to the consumer. For example, an event generator may agree to provide a particular set of audit data to an analyzer. At a minimum, an agreement must specify the producer, consumer, and the set of services to be provided.

Contract A contract is a set of agreements, each of which involve the same producer and consumer (the *partners* to the contract). At all times, exactly one agreement in a contract is in effect. Thus, the agreements in a contract constitute a set of alternatives. Every contract has an implicit "null" alternative in which no service is provided.

Contract Database A contract database is a set of contracts. Every component has a contract database containing all the contracts to which it is a partner.

Capability Database A capability database associates services (e.g., provide IP audit data, filter packets, etc.) with the components which can provide those services. Each component has a database containing its own capabilities and, possibly, those of other components.

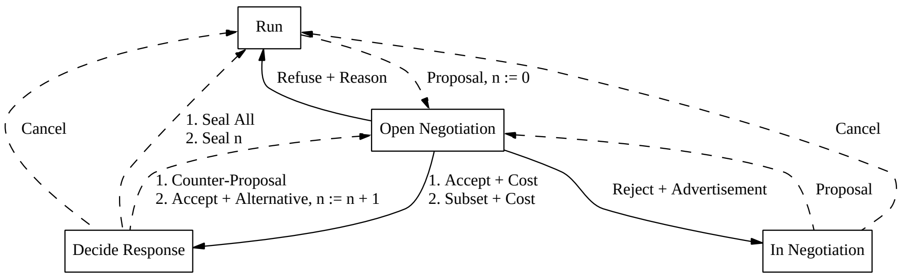
The state of a component consists of its contract and capability databases. The IDIAN protocol defines the steps by which a component's state may be changed, i.e., how contracts and agreements are negotiated, altered and removed, and how capability databases are updated.

5.2 Protocol Description

A portion of the IDIAN negotiation protocol is graphically depicted in Figure 3 in the form of a state machine. The machine shows the allowed behavior of a consumer component during a negotiation. Transitions are labeled with the message types which cause the transition. Dotted (resp. solid) lines are transitions caused by the consumer sending (receiving) a message to (from) the producer. The diagram for a producer component is mostly identical to that for a consumer, but with the dotted and solid lines switched.

The diagram only depicts the portion of the protocol specifically related to negotiation (the creation and modification of contracts). Other states and transitions related to, e.g., removing contracts or dynamically switching among existing agreements have been omitted for clarity.

Figure 3: The negotiation protocol – consumer



The basic consumer protocol consists of four states: Run, Open Negotiation, In Negotiation, and Decide Response. In the Run state, a consumer is performing normal operation, i.e., accepting and processing incoming GIDOs.

While in the Run state, a consumer can receive Advertisement messages from producers or initiate a negotiation by sending a Proposal message.

Advertisement A component sends an Advertisement to announce its capabilities. An Advertisement contains a filter to denote the set of GIDOs which it can produce or consume. A component receiving an Advertisement could use the information to update its capability database, or decide to initiate a negotiation to obtain the advertised services. Note that the reception of an Advertisement in the Run state is not shown in Figure 3.

Proposal A Proposal is used to initiate a negotiation. The component sending the proposal is the consumer, while the recipient is the producer. A Proposal contains a filter specifying the service the consumer is requesting.

The object of a negotiation is always a specific agreement in a specific contract. The contract may already exist, in which case the consumer is proposing to change an existing contract, otherwise the consumer is proposing to create a new contract. The same applies to the agreement.

Once the consumer has sent a Proposal, it enters the Open Negotiation state to await the producer's response. Negotiations in the IDIAN protocol are atomic – any given component can be negotiating with at most one other component at any time.

During a single negotiation session (the time between leaving and then returning to the Run state) a consumer may negotiate several agreements, all of which must be in the same contract. The variable `n` in the diagram records the number of agreements which have been negotiated so far.

A producer that has received a proposal can respond with one of several types of messages:

Accept + Cost The producer accepts the proposal and provides an estimate of the cost to supply the requested service.

Subset + Cost The producer rejects the proposal because it does not have the resources to provide all the requested service. This message includes a filter specifying a subset of the services which the producer could provide and the cost of providing it.

Reject + Advertisement The producer rejects the proposal because it is unable to supply the requested service. In addition, the producer provides an advertisement of its current capabilities.

Refuse + Reason The producer refuses to negotiate with with consumer and provides a reason for the refusal. Possible reasons include the fact that the producer is negotiating with another component (all negotiations are atomic) and the fact that the consumer does not have authority to negotiate with the producer. Refusals terminate the negotiation immediately, so the consumer returns to the Run state.

Upon receiving a Reject message, the consumer enters the In Negotiation state, from which it can send another proposal, or Cancel the negotiation altogether. A consumer that has received an Accept or Subset message enters the Decide Response state, where it must choose between accepting or rejecting the producer's offer. Even if the producer simply Accepted the proposal, the consumer may still reject if it deems the cost too high.

Cancel The consumer rejects all the agreements negotiated so far (`n` may be greater than zero) and terminates the negotiation.

Counter-Proposal A Counter-Proposal implicitly rejects the producer's offer and offers a new proposal instead.

Accept + Alternative The consumer temporarily accepts (pending a Seal message) the previous negotiation and offers a new one representing a different agreement in the same contract. Negotiation now begins on this new proposal.

Seal All The consumer accepts all of the agreements negotiated so far and ends the negotiation. The contract databases of both components are updated to reflect the changes.

Seal `n` The consumer accepts all of the agreements except the one currently under negotiation, which is rejected. The negotiation ends and the contract databases are updated.

Seal All and Seal `n` are the only messages which cause the components' contract databases to be changed, after which the new agreements immediately go into effect.

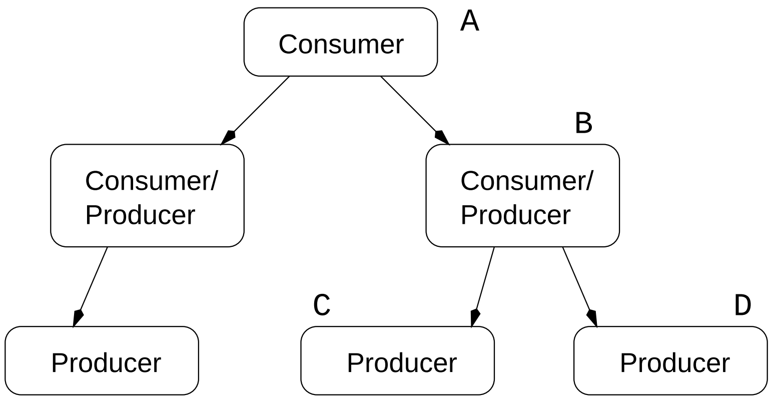
5.3 Additional Protocol Features

The previous section presented that subset of the protocol directly related to negotiation. However, the protocol contains additional features. Specifically, the protocol supports messages which accomplish the following:

- Termination of agreements and contracts.
- Querying of component capabilities and contract status.
- Dynamic switching among agreements.

Furthermore, an extended version of the protocol has been developed which supports *hierarchical* negotiation. Under hierarchical negotiation, the ID components are arranged in a hierarchy, such as in Figure 4.

Figure 4: Hierarchical negotiation



In hierarchical negotiation, consumers may only negotiate with producers directly below them in the hierarchy. Components in the middle layers of the hierarchy behave as producers with respect to the components above, and like consumers with respect to the components below.

For example, when high-level consumer `A` sends a proposal to component `B`, component `B` may, in turn, negotiate with low-level producers `C` and `D` to obtain the requested service for `A`.

Hierarchical negotiation could be used, for example, to help manage very large IDS systems.

5.4 Protocol Specification

The IDIAN project has developed a formal specification of the protocol in the Estelle [3] description language. Estelle is a Pascal-like language with constructs for specifying processing, communication channels, and state transitions.

Figure 5 shows a fragment of the Estelle specification of a consumer. The fragment specifies component behavior during the state transition from Run to Open Negotiation, in which a suitable producer is selected and a proposal is transmitted. The function `SelectOneComponent` is left unspecified, a useful feature of Estelle.

Figure 5: A fragment of the protocol specification

```
FROM Run TO OpenNegotiation
  BEGIN
    InitProposal(MyProposal);
    ProducerNegotiator :=
      IdToNum(SelectOneComponent(MyID, RelationshipDataDB,
                                 CapabilityReq));
    OUTPUT C[ProducerNegotiator].proposal(MyProposal);
  END;
```

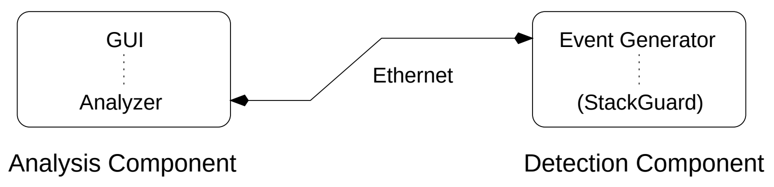
Both the basic and hierarchical versions of the protocol have been specified. In the future, the project plans to use the Estelle specifications together with the XEC Estelle compiler [7] to aid in protocol implementation and analysis.

6 IDIAN Demo

The IDIAN project has developed a demonstration of a subset of the protocol on a testbed consisting of two computers running Linux, shown in Figure 6.

Upon startup, the event generator sends an advertisement message to the analyzer, announcing the fact that it can generate notifications of stack overflow attacks. The analyzer negotiates with the event generator to monitor a set of programs for stack overflow. The specific set can be adjusted by the

Figure 6: The protocol testbed



user by using the GUI, and the analyzer will renegotiate the contract. Upon receiving a GIDO notification of a possible attack, the analyzer passes it to the GUI for display to the user.

The event generator monitors the system logs for messages from StackGuard [2] code that identifies possible stack overflow attacks. Any such messages are transformed into GIDOs, but those GIDOs are sent only if they match the filter in the contract which has been negotiated with the analyzer.

Currently, the demo uses a subset of the protocol and supports a single contract with a single agreement.

7 Conclusions and Future Work

This paper has presented the work of the IDIAN project in developing a protocol that allows ID components to negotiate over the provision of ID services such as detection capabilities and response mechanisms. The protocol enables components to create contracts which bind producer ID components (e.g., audit generators) to provide their services to consumers (e.g., analyzers).

The project has demonstrated the potential feasibility of this approach using a testbed consisting of two systems which negotiate over the set of audit data to be shipped.

Many open areas of research remain. For example, the current implementation of filters uses full Perl expressions in the filter GIDO to match data values. This approach could raise issues of security and efficiency. More work is needed in the area of filter design.

Another open area of research involves the measurement and expression of cost – the amount of resources a component requires to provide a set of services. Since there are so many resources involved in providing a service (memory, CPU time, network bandwidth, etc.) and since these resources can change dynamically, accurately estimating and communicating cost information is a major challenge.

Also, the IDIAN project plans to create a larger testbed involving more than two systems in order to test a complete version of the the basic protocol, as well as the hierarchical extension.

Finally, an interesting project would be to incorporate the IDIAN protocol into existing ID components to further explore the protocol's applicability.

References

[1] J. Balasubramaniyan et al. An architecture for intrusion detection using autonomous agents. Technical Report Coast TR 98-05, Department of Computer Sciences, Purdue University, 1998.

[2] C. Cowen et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, 1998.

[3] ISO/TC97/SC21. Information processing systems — Open systems interconnection — Estelle — a formal description technique based on an extended state transition model. IS 9074, International Organization for Standardization, Geneva, 1997.

[4] C. Kahn et al. A common intrusion detection framework. `http://seclab.cs.ucdavis.edu/cidf/papers/jcs-draft/cidf-paper.ps`, 1998.

[5] U. Lindqvist and P.A. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *Proc. 1999 IEEE Symposium on Security and Privacy*, Oakland, California, May 1999.

[6] A. Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection.* PhD thesis, Computer Science Institute, University of Namur, Belgium, Sept 1997.

[7] J. Thees and R. Gotzhein. Protocol implementation with Estelle – from prototypes to efficient implementations. In *Proc. Estelle '98*, Evry, France, 1998.