Policy-controlled Event Management for Distributed Intrusion Detection

Christian Kreibich
University of Cambridge
Computer Laboratory
christian.kreibich@cl.cam.ac.uk

Robin Sommer
Technische Universität München
Computer Science Department
sommer@in.tum.de

Abstract

A powerful strategy in intrusion detection is the separation of surveillance mechanisms from a site's policy for processing observed events. The Bro intrusion detection system has been using the notion of policy-neutral events as the basic building blocks for the formulation of a site's security policy since its conception. A recent addition to the system is the ability to exchange events with other Bro peers to allow distributed detection. In this paper we extend Bro's existing event model to fulfill the requirements of scalable policy-controlled distributed event management, including mechanisms for event publication, subscription, processing, propagation, and correlation.

1. Introduction

In the light of ever-increasing numbers of security incidents in computer infrastructures, the field of intrusion detection has received a significant amount of interest in recent years. The goal of these systems is to detect malicious activity as quickly as possible in order to allow swift response and to provide substantial forensic evidence to understand the damage inflicted. This effort has led to the development of a variety of different intrusion detection systems (IDSs) [1]. One of the lessons learnt in the field is that IDSs operating individually do not understand the "big picture" — activity in a networked computer infrastructure is too multi-faceted to permit this. In addition, security and operational considerations do not allow a stand-alone setup for organisations for all but the smallest LANs. As a consequence, distributed IDSs have been proposed that allow the various detection systems to communicate in order to increase the collective field of vision.

In this application setting, the event-based communication paradigm is an obvious match: an individual IDS observes elementary events that are processed either in situ or forwarded to another system for analysis, possibly generating new higher-level events for which the process repeats, until the system at large has determined with sufficient certainty that a *security-relevant event* is occurring. This event is then logged, presented to the analyst, or triggers an autonomous response. Traditionally, the communication model such distributed IDSs have employed is a straightforward sensor/manager architecture where events are propagated unidirectionally from "dumb" sensors to "smart" manager nodes which abstract the elementary events into semantically richer composite events. DIDS [14] was the first to employ a such model. Today, many commercial systems follow a similar model. Other systems, such as Emerald [13], build up a hierarchical structure and propagate information up to the root level. Similarly, AAFID [16] builds on autonomous agents which communicate their results to hierarchically organised monitors. NetSTAT [17] preconfigures a set of probes with attack scenarios and distributes them throughout a network. If a probe is not able to detect an attack by itself due to the characteristics of its environment, it communicates its analysis to other probes as appropriate.

The Bro IDS [11] has always relied heavily on the idea of *policy-neutral* events as vehicles to convey the occurrence of interesting activity. The site-specific security policy is formulated in a domain-specific policy scripting language and defines the interpretation of such events. A major improvement of the system was the introduction of a communications framework that allows multiple Bro *peers*[1] to exchange state information [15]. While our initial experiments with distributed event management have been very encouraging, it also became apparent that more powerful event processing mechanisms than we had at our disposal were conceivable and in fact necessary to satisfy our requirements.

In this paper, we present the event model we are developing for the Bro IDS to support scalable policy-controlled distributed event analysis. The remainder of this paper is structured as follows. We specify the requirements of our

[1] We will refer to communicating Bro IDSs as *peers* when context refers to a group of nodes engaged in mutual communication, and as *nodes* when the focus is on members of a network in general.

event communication framework in Section 2. We then outline Bro's architecture in Section 3. Our event model provides a flexible group-based publish/subscribe mechanism, supports arbitrary event diffusion patterns, and allows for complex abstraction of event occurrence into composite events. We present the model in detail in Section 4. Distributed intrusion detection is a highly complex problem setting, and turning the Bro system into a fully distributed IDS is an ongoing process. We therefore discuss our assumptions, experiences, and expectations for future development at length in Section 5 before we conclude the paper with Section 6.

2. Event Communication Requirements

As a starting point, we begin by identifying the requirements of our event model.

- Expressiveness: Events must be expressive enough to embody arbitrary types of activity, and must be structured enough to permit type-safe processing.
- Policy-controlled Event Processing: All aspects of the event model must be configurable in a site's policy, including local event handling as well as event delivery & forwarding schemes. *Cooperative* policy configuration is of course desirable and in fact necessary to achieve best results, but remote nodes must not be able to interfere with local policy.
- Selective Receiver Interest & Notification: It must be easy for Bro peers to indicate interest and end of interest in both local and remote events. Peers must not be bothered with events they are not interested in. At the same time, a node must be given enough flexibility to allow for suitable communication patterns (e.g., broadcast or request/reply).
- Event Abstraction: The policy language must allow the definition of *event patterns* and consequential abstraction into higher-level events.
- Scalability and Performance: The model must be highly scalable and lend itself to a high-performance implementation. Target environments are both local networks and Internet-wide cooperations.
- Secure Communication: Peers must authenticate themselves to their peers before exchanging events. Other hosts must not be able to eavesdrop into or tamper with the flow of events.
- Homogeneous Language Extension: The changes to the Bro scripting language necessary to support distributed event communication should blend in with the existing syntax and semantics as much as possible.

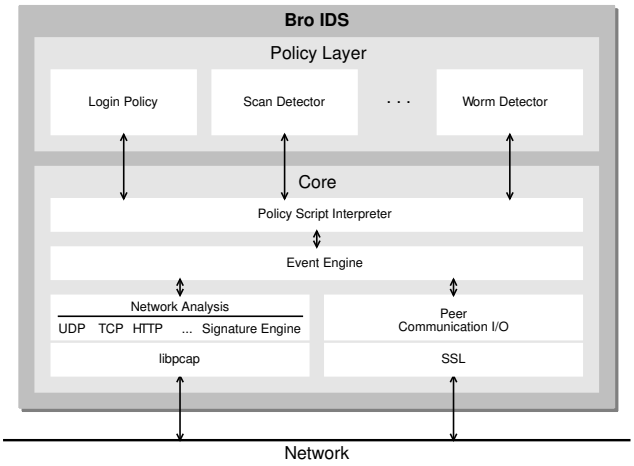

Figure 1. Architecture of the Bro IDS.

3. Architecture of the Bro IDS

Bro's architecture is described in detail in the original paper [11] and remains basically unchanged, with the exception of support for communication among Bro nodes. We repeat the architecture's key elements here in condensed form to put in context the event model and its implications for distributed event processing. Figure 1 illustrates Bro's architecture.

3.1. Separation of Mechanism from Policy

The core idea of Bro is to split event detection mechanisms from event processing policies. Event generation is performed by *analysers* in Bro's core: these analysers operate continuously based on input observed by Bro instances and trigger events asynchronously when corresponding activity is observed. Due to its strong basis in network-based detection, Bro's core contains analysers for a wide range of network protocols such as RPC, FTP, HTTP, ICMP, SMTP, TCP, UDP, and others. These analysers trigger events whenever interesting activity in a network flow is observed, for example when a new TCP connection is established or an HTTP request is made. Besides that, a signature engine allows typical misuse-based intrusion detection: it matches byte string signatures against traffic flows and triggers events whenever a signature matches. Once an event is triggered, it is passed to the *policy layer* which then takes care of processing the events. Care is taken to minimise CPU load: only analysers responsible for triggering the events used at the policy layer are actually enabled.

3.2. Policy Configuration

Each Bro peer runs a policy configuration in its policy layer. This policy embodies all or part of a site's security policy, expressed in scripts formulated in the special-purpose Bro scripting language. To understand the significance of this approach it is important to realise that the relevance of an event varies from site to site — some sites may consider the detection of a Microsoft IIS exploit attempt on a pure UNIX network a threat, while others may not. Bro's policy language is strongly typed, procedural in style, and provides a wide range of elementary data types to facilitate the analysis of activity on a network. Elements in the policy can be tagged with *attributes*, these attributes serve to define meta-information about these elements, such as state expiration, persistence for on-disk storage of state across instances, etc. The basic building blocks of a policy are *event handlers* which process occurring events in the fashion called for by a site's security policy.

3.3. Communication Framework

Bro's communication framework allows the transmission of arbitrary kinds of state between Bro instances. The driving idea behind its design was to allow the realisation of *independent state*: state accumulated at the policy layer should no longer be thought of as a local concept but rather as information dispersed throughout the network. The communication model imposes no hierarchical structure. Examples of exchangeable state include triggered events, state kept in data structures in policies, and the policy definitions themselves. For the purpose of this paper it is sufficient to think of the entities exchanged between peers as events, though that ignores a large part of its flexibility.

Peer communication can but need not happen over the same network that Bro's network analysers are monitoring. Since Bro peers exchange highly security-relevant information, it is important to ensure that only authorised peers are allowed to talk to each other and that no other hosts can eavesdrop into the conversation. Therefore, while event communication itself remains loosely coupled, the peers' identities are controlled tightly: peers authenticate each other using certificates and communicate over encrypted channels, using the SSL protocol. To conserve bandwidth, connections can optionally be compressed.

4. Bro's Event Model

We will now describe the event model we are currently developing for Bro. Bro events capture asynchronously the occurrence of specific activity. An event in Bro has a *type* that is defined by a *name* and the sequence and types of the event's *arguments*. An event materialises by assigning corresponding values to the event arguments. All state associated with a Bro event is *mobile*: the values of the event's arguments are always distributed alongside the event itself and are not bound to any particular node's address space. Once instantiated, all event handlers defined for its type are triggered. This triggering can happen inside the Bro core, or in a policy script using the `event` statement. In both cases, all event handlers see the same version of the event, i.e., event handlers cannot modify the event before another event handler processes it. However they can of course trigger different follow-up events according to their implementation. Events are also available to event handlers as first-order data structures in the policy language: instances of record type `event` contain elementary event parameters such as the time of creation, a description of the Bro peer that triggered the event, and more. This event-specific variable is made available to the event handlers through the special `this` variable, similar to the `this`/`self` concepts in many object-oriented languages.

Figure 2 gives an example of an event handler. `conn_stats` is a user-defined record type that holds per-connection accounting information. The `default` attribute causes record fields to be initialised to zero. `active_conns` is a table indexed by connection identifiers (`conn_id` records) and yielding `conn_stats` records. The `write_expire` attribute causes entries that remain unmodified for more than 5 minutes to be removed. The event handler for the `connection_established` event has a single parameter, a `connection` record. When triggered, the handler inserts a new `conn_stats` record with the current time taken from the last network packet's timestamp into `active_conns` for the connection provided as an event argument. [2]

4.1. Event Subscription

In addition to the local event processing mechanism just described, triggered events can also be delivered to other Bro instances. We use the widely used publish/subscribe model [5] for letting each node specify in its policy which events it would like to receive from its peers: a node requests events by providing a regular expression that matches the event names of interest. Only peers that have indicated interest in an event type are delivered such events. To allow for simpler expressions, the matching can be restricted to event names in a given namespace. For example, `Worm::*` stands for all events in the `Worm` namespace and `Worm::*infectee*` represents all events in the same namespace whose names contain the substring "infectee".

[2] The "$" operator selects a field of a Bro record; similar to the "." operator in C.

```
type conn_stats: record {
        start_time: time;
        num_pkts: count &default = 0;
        num_bytes: count &default = 0;
};

global active_conns: table[conn_id] of conn_stats
        &write_expire = 5 mins;

event connection_established(c: connection) {
        local stats: conn_stats;
        stats$start_time = network_time();
        active_conns[c$id] = stats;
}
```

Figure 2. Example of a simple event handler.

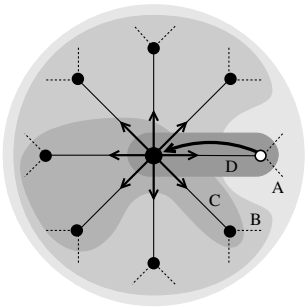

Figure 3. Various event dispatching policies.

4.2. Event Dispatching

One of our first observations with distributing events was that while broadcasting events to all subscribers is often appropriate, some events require a different approach: one of our standard events is a heartbeat in the form of a "ping" event to a peer, which is answered with a corresponding "pong" event. In the standard model, multiple clients pinging a single peer at the same time will cause the entire subscriber set of the pong event to receive a copy of every pong event, even though each subscriber is only interested in the pong event associated with its own pings. Clearly, the policy layer needs to be able to specify the event dispatching mechanism more precisely. Our goals are exemplified in Figure 3: the white node sends an event to the central node whose processing triggers another event to which all of the ring nodes are subscribed. Depending on the central node's dispatching policy, different subsets of the subscriber set are included: (A) shows full broadcast, (B) limited broadcast excluding the originator, (C) an arbitrary subgroup, and (D) the originator only. Besides those, many other scenarios are conceivable, such as anycast variations or event sampling. Note that this means that we do not guarantee that every subscriber sees *every* instance of event types it subscribed to — that selection is up to the policy of the dispatching node.

```
global pong(seq: count): event &dispatch=sender;

global signature_match(state: signature_state,
                       msg:   string,
                       data:  string):
       event &dispatch=broadcast;

global heartbeat(): event &autotrigger=5mins;

function sender(e: event): set[event_peer] {
    local receivers: set[event_peer];
    add receivers[e$peer];
    return receivers;
}

function broadcast(e: event): set[event_peer] {
    return e$subscribers;
}
```

Figure 4. Example attributes.

Our model uses a combination of event handlers, event record types, element attributes, and subscriber sets. Bro already treats event handlers as first-order elements of the scripting language, i.e., they can be assigned to variables, called indirectly, etc. We extend this approach by tagging event handlers with `dispatch` and `autotrigger` attributes. Using `dispatch`, the user can specify a *dispatcher* callback that serves as an *active forwarding filter*: the dispatcher is passed the event as an instance of the `event` record type and returns an instance of type `set[event_peer]` which contains all subscribed peers that event will be forwarded to. The evaluation of this function and the actual event forwarding happen after the event is triggered and before the event handlers are executed. Dispatchers can thus employ all set operators offered by the Bro language. Since a dispatcher could theoretically return arbitrary nodes in the resulting recipients set, the set is automatically intersected with an event type's actual subscriber set to make sure the recipients remain within the subscriber set. The `autotrigger` attribute specifies a temporal value and causes events to be triggered periodically once the specified period of time expires. We expect that further event attributes will be added in the future.

Note that an event type can have multiple event handlers. Allowing every event handler to use a different dispatcher could lead to conflicts: for example, one event handler could request that an event never be forwarded to any of the subscribers, while another could request full broadcast. We solve this problem by providing one event dispatcher per event *type*: dispatcher assignment happens when an event type is *declared*.

Figure 4 illustrates these concepts: `pong` events use the `sender` function as their dispatcher, which selects only the peer who originated the event in the resulting event peer set. `signature_match` events use the `broadcast` dispatcher which simply uses the entire subscriber set provided

with the event. Finally, `heartbeat` events will be triggered automatically every 5 minutes.

4.3. Event Chains

Thanks to the `event` statement, an event handler can trigger the occurrence of another event, leading to *chains* of events (and consequentially, of event handler invocations). This poses questions regarding the propagation of properties from one event to the next. We currently track event chains only locally, i.e., among sequences of event handler invocations on the same host. Note that a single event can lead to multiple event chains because of the possibility of multiple event handlers per event type: each new event can start a new set of chains. In the light of our event dispatcher model, we are primarily interested in the *original* source of the event and the time of creation. We propagate these properties and leave the dispatch policies unchanged. That way, in an event chain $A \rightarrow B \rightarrow C$ of three events, C's event handler could still implement the dispatch policy `sender` shown in Figure 4 that would cause event C to be sent only to the peer from which the node originally received A.

4.4. Composite Events

Often we are not interested in the occurrence of a single event but in the occurrence of a *pattern of events*. For example, successful attacks often involve multiple stages: a reconnaissance phase in which the attacker scans for vulnerable machines, the break-in itself into one or more victim machines, and finally the attempt of removing any evidence and maintaining access to the machines. These steps need not occur sequentially, and many of them have different incarnations. However, each stage can generate events. Therefore, detection accuracy benefits from combining such patterns into semantically more high-level *composite events* [12]. In general, composite events consist of a set of *input events*, a number of *conditions*, and an *output event* which is triggered if the conditions are met. In the intrusion detection domain, many approaches have been devised to correlate lower-level events into composite events. The STAT suite models attack scenarios with state machines, using a custom language [4]. Emerald features P-BEST [8], a production-based expert system. Other correlation schemes include identifying groups of events which share common attributes [3]; transforming high-level attack descriptions into acyclic directed graphs [7]; and matching consequences of one step with the prerequisites of others [10].

Implicitly, Bro has always used composite events: its event handlers can keep state across invocations, allowing policies that track precisely the type and parameterisation of previous events. For instance, Bro's scan detector tracks connection attempts per originator. If a certain scanning threshold is reached, a new event is raised which reports the detected scanner. Thus, the detector implicitly defines a pattern of connection attempt events that eventually trigger a composite event. With the development of the communications framework we have made the representation of composite events more explicit by the introduction of *triggers*. Triggers follow the *event-condition-action* paradigm [9, 2] and consist of small self-contained programs that encode the logic necessary to detect composite events. Triggers consist of a set of *input event handlers* which evaluate the composite event's *triggering condition* which, if met, leads to the triggering of possible *output events*. The key difference to other policy statements is that triggers are self-contained and mobile. Thus, we can initialise triggers as we see fit and *transfer* them from one peer to another[3] while any output events are sent to the peer that originated the trigger. The state required for correlating input events is maintained inside the trigger's scope. No external variables may be referenced inside triggers, making trigger state similar in spirit to closures in functional programming.

```
trigger scanner raises { successful_scanner }

global scanners: set[addr] &write_expire = 1hr;

event is_scanner(ip: addr) {
    add scanners[ip];
}

event connection_established(c: connection) {
    if ( c$orig_h in scanners &&
         c$resp_h in local_nets )
        event successful_scanner(c$orig_h);
}
```

Figure 5. Example of a trigger.

As an example, assume that we want to detect scanners which have successfully set up a connection into our internal network shortly after performing their scans. More precisely, the system is to raise the event `successful_scanner` if (1) the event `is_scanner` is raised for an IP address X; and (2) within the next hour, the event `connection_established` is raised for a connection between X and a local responder. Figure 5 shows such a trigger: the `is_scanner` handler inserts detected scanners into the `scanners` set. The `connection_established` handler raises the new event if such a known scanner has successfully setup a connection to a local host.

[3] Bro already supports the serialisation of scripts into a binary representation [15]. Therefore, sending a trigger presents no technical difficulty.

5. Discussion

5.1. Characterisation of Events

Given the vast application space of event-based systems, it behooves us to consider carefully the characteristics of events in the intrusion detection space. Successful intrusion detection needs to cope with an extraordinary range of environmental context, policy implications, semantical ambiguities, and technical limitations. We believe this has two implications relevant to our discussion. First, there is strong reason to believe that IDS events need to be just as semantically variable as their problem setting. This implies that the mechanisms for moving events around have to be rich in terms of type information. Second, IDS events require a great deal of context to allow justifiable decisions in the analysis stage. The value of IDS events varies tremendously with their context. This implies that particularly the propagation of IDS events of high abstraction level will require nontrivial optimisations to deal efficiently with the amounts of state involved.

5.2. Threat Model

An attacker could attempt to abuse the event model to cause harm to both local and remote Bro nodes. An attacker without direct access to a Bro node could try to exploit a mechanism known to trigger events, in order to clog a connection. The attacker could wait until this point to perform more sensitive activity that would not end up reported. However, events typically are small (few hundreds of bytes, at most), the attacker can abuse only a single connection, and the receiver would notice the onslaught of events and could react accordingly (for example disconnecting or resolving to sampling events, etc). We believe this risk to be manageable. Impersonating a Bro node is harder for the attacker: as mentioned in Section 3.3, we require mutual authentication from peers upon connection establishment and encrypt connections, using SSL. PKI-based authentication is our main trust hurdle; once an attacker manages to break into a host running Bro and manages to authenticate itself successfully to the Bro peers, the node could be used to willfully send fake or broken messages.

5.3. Topic- vs. Content-based Subscription

Our subscription mechanism at this point is topic-based: each event type's name defines a topic that subscribers indicate interest in. However, support for content-based subscription is provided with triggers because they can encode arbitrary predicates for filtering event delivery. Note that the content-based approach can be considered complementary to our dispatchers: we let the event-triggering node's policy govern which subscribers receive the event, whereas triggers represent the potential recipient's event interest.

5.4. Incorporating Type Information

We believe that extending Bro's record types with object-oriented features would improve our event processing repertoire considerably. While type-based publish/subscribe has been discussed before [6], there remain interesting possibilities for event management. For example, we could leverage event argument types for more fine-grained event handler selection: multiple event handlers for the same event type could be provided with different subtype selections for the event's argument set. This opens many possibilities; one is the use of this type distinction to restrict the invocation of event handlers to the handler matching the provided types most closely.

Location dependence of a record's method implementation is another technique. When a record is transmitted between peers, the implementation of the type's methods need not necessarily be the same in both peers (unless the implementation is specifically sent along). Such "spatial polymorphism" can be used to vary the behaviour of an event argument depending on the event's location. We believe this will have many uses, particularly given the fact that Bro's event handlers do not return a value while a record type's methods can.

5.5. Performance Considerations

While our dispatcher scheme provides great flexibility, attention needs to be paid to performance: low-level events can occur hundreds of times per second. Dispatcher evaluation for each of those events could quickly lead to missed events in other parts of the system. We tackle this problem from two directions: first, we believe that in a large majority of cases, only a small number of simple dispatchers will be used. We can implement them in the Bro core, avoiding the overhead of script interpretation. Second, additional language attributes could be used to indicate that a dispatcher's evaluation does not change, hence the resulting subscriber set could be cached after a single evaluation.

5.6. Event Composition & Routing

Event triggers retain the comparatively low-level approach to expressing event patterns through state in script variables. We are investigating the use of more high-level language elements to ease this task; we envision a language model similar to the one used in [12] but based on Petri nets.

Our event model is currently not very routing-friendly: when node A subscribes to an event type at peer B, this implies a direct connection between A and B. The only

routing mechanism provided is the configuration of *explicit* forwarding policies. This has to date not presented a problem and we defer more advanced routing support until we experience stronger need. The same applies to the extension of node-local event chain tracking to a global one, which would allow the introduction of route tracing functionality such as "source routing" or "record route" across event instantiations.

5.7. Node Flexibility

Bro is a fairly heavyweight application that may not be the best choice for the purposes of all nodes in a Bro network. We have implemented a lightweight and highly portable library supporting Bro's communication protocol called *Broccoli* [4] that we use to let other systems partake in the event communication. Broccoli nodes can request, send, and receive Bro events but do not support Bro's policy language. The policy has to be implemented directly in the code or through mechanisms such as configuration files.

6. Summary

We have presented our steps towards an event model for the Bro IDS that we expect to provide the flexibility and control necessary for creating large-scale distributed IDSs based on Bro. Indeed, we believe that the combination of Bro's communication framework with policy-configurable event processing allows us to think of Bro not only as a distributed IDS but also as a more general system for highly configurable distributed event processing. We plan to experiment with increasingly large networks of Bro nodes, possibly spanning the current Bro deployments at institutions such as ICIR, Lawrence Berkeley National Laboratory, and Technische Universität München.

7. Acknowledgments

This work has been carried out in collaboration with Intel Research Cambridge and ICIR. We would like to thank Jon Crowcroft, Holger Dreger, Anja Feldmann, and particularly Vern Paxson for interesting discussions and valuable feedback.

References

[1] S. Axelsson. Intrusion Detection Systems: A Survey and Taxonomy. Technical Report 99-15, Depart. of Computer Engineering, Chalmers University, Mar. 2000.

[2] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. *Lecture Notes in Computer Science*, 1995, 2001.

[3] H. Debar and A. Wespi. Aggregation and Correlation of Intrusion-Detection Alerts. In *Proc. of Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[4] S. Eckmann, G. Vigna, and R. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[5] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 31:114–131, jun 2003.

[6] P. Eugster, R. Guerraoui, and O. . C.H. Damm. On Objects and Events. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Tampa Bay, Florida, USA, Oct 2001.

[7] C. Krügel, T. Toth, and C. Kerer. Decentralized Event Correlation for Intrusion Detection . In *Proc. of Information Security and Cryptology*, volume 2288 of *Lecture Notes in Computer Science*, 2001.

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

[9] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proc. of AAAI, Orlando, Florida*, July 1999.

[10] P. Ning, Y. Cui, and D. S. Reeves. Constructing attack scenarios through intrusion alerts. In *Proc. 9th ACM Conference on Computer and Communications Security*, 2002.

[11] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23-24):2435–2463, 1998.

[12] P. R. Pietzuch, B. Shand, and J. Bacon. A Framework for Event Composition in Distributed Systems. In *Proc. of the 4th ACM/IFIP/USENIX Int. Conf. on Middleware (Middleware '03)*, pages 62–82, Rio de Janeiro, Brazil, June 2003. Springer.

[13] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *National Information Systems Security Conference*, Baltimore, MD, October 1997.

[14] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C.-L. Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur. DIDS (Distributed Intrusion Detection System) – Motivation, Architecture, and an Early Prototype. In *Proc. 14th NIST-NCSC National Computer Security Conference*, 1991.

[15] R. Sommer and V. Paxson. Exploiting Independent State For Network Intrusion Detection. Technical Report TUM-I0420, TU München, 2004.

[16] E. H. Spafford and D. Zamboni. Intrusion detection using autonomous agents. *Computer Networks*, 34(4):547–570, 2000.

[17] G. Vigna and R. A. Kemmerer. Netstat: A network-based intrusion detection system. *Journal of Computer Security*, 7(1):37–71, 1999.

[4] *Broccoli* is the healthy acronym for "Bro Client Communications Library", found at `http://www.cl.cam.ac.uk/~cpk25/broccoli/`.