Defending a Computer System using Autonomous Agents

Technical Report No. 95-022

Mark Crosbie, Gene Spafford
COAST Laboratory
Dept. of Computer Sciences
Purdue University
West Lafayette IN 47907-1398
{mcrosbie, spaf}@cs.purdue.edu
(317) 494-7825

11 March, 1994

Abstract

This report presents a prototype architecture of a defense mechanism for computer systems. The intrusion detection problem is introduced and some of the key aspects of any solution are explained. Standard intrusion detection systems are built as a single monolithic module. A finer-grained approach is proposed, where small, independent agents monitor the system. These agents are taught how to recognise intrusive behaviour. The learning mechanism in the agents is built using Genetic Programming. This is explained, and some sample agents are described. The flexibility, scalability and resilience of the agent approach are discussed. Future issues are also outlined.

1 Introduction

Because of increased network connectivity, computer systems are becoming increasingly vulnerable to attack. These attacks often exploit flaws in either the operating system or application programs. The general goal of such attacks are to subvert the traditional security mechanisms on the systems and execute operations in excess of the intruder's authorisation. These operations could include reading protected or private data or simply doing malicious damage to the system or user files.

The degree of protection from such malicious actions depends on the amount of time and effort spent building and maintaining the system's security defenses. By building complex tools, which continually monitor and report activities, a system security operator can catch potentially malicious activities as they occur. However, this involves a large expense in terms of time and money in both building and maintaining such a monitoring system. The monitoring will also impose a performance penalty on the system being protected — something which the users may object to.

This paper proposes a mechanism for building such a monitoring system which does not involve a significant amount of effort. It utilises many small processes to achieve the overall goal of system monitoring. Each process (or *agent*) operates independently of the other agents, but they all cooperate in monitoring the system. This approach has significant advantages in terms of overhead, scalability and flexibility.

2 Intrusions and Intrusion Detection

An *intrusion* can be defined as [1]:

any set of actions that attempt to compromise the integrity, confidentiality or availability of a resource.

and they can be catagorised into two main classes:

- *Misuse* intrusions are well defined attacks on known weak points of a system. They can be spotted by watching for certain actions being performed on certain objects.
- *Anomaly* intrusions are based on observations of deviations from normal system usage patterns. They are detected by building up a profile of the system being monitored, and detecting significant deviations from this profile.

As misuse intrusions follow well-defined patterns they can be detected by doing pattern matching on audit-trail information. For example, an attempt to create a setuid file can be caught by examining log messages resulting from system calls. This can be done using a pattern matching approach such as in [3].

However, anomaly intrusions are harder to detect. There are no fixed patterns that can be monitored for and so a more "fuzzy" approach must be taken. Ideally we would like a system that combined human-like pattern matching capabilities with the vigilance of a computer program. Thus it would always be monitoring the system for potential intrusions, but would be able to ignore spurious false intrusions if they resulted from legitimate user actions. It would rely on heuristics to decide this — they could either be pre-specified (by a human operator) or learned by the system over time. However heuristics will not always guarantee perfect accuracy, so another goal is to minimise the probability of incorrect classification.

2.1 Intrusion Detection

An *intrusion detection system* (IDS) must [2]

identify, preferably in real time, unauthorized use, misuse, and abuse of computer systems.

An intrusion detection system does not attempt to stop an intrusion as it occurs. Its role is to alert a system security officer that a potential security violation is occurring. As such it is a reactive, rather than proactive, form of system defense.

An intrusion detection system can either be *host-based* or *network-based*. Often a system is a hybrid of the two approaches. A host based system will monitor all the activity on a single host computer. It will ensure that no user operations are violating the site security policy. A network based system monitors on a net-wide basis — it will consider actions occurring on the network and analyse them as to whether they constitute potential security violations.

In either case, the resulting system is often a large monolithic module. This module performs all of the monitoring, data gathering, data manipulation and decision making for the whole system. It either sits in, or on top of, the system kernel. Depending on the data being gathered, it can monitor system audit logs, user activities and system state. From these observations, it will deduce some metrics about the system's overall security state, and decide whether an intrusion is currently occurring.

The most apparent problem with this approach to building an IDS is the overhead it imposes on the system being protected. Often, a single system is analysed by another system due to the added overhead of the IDS. If audit logs are being analysed, the kernel must generate audit information for all the actions it performs. This results in a large amount of information, which must be stored at least semi-permanently on some storage medium. Generating these detailed logs consumes both disk-space and CPU time. Once these logs are generated, the IDS must read and interpret them, attempting to correlate activities with other system information.

An IDS may also perform its own system monitoring — it may keep aggregate statistics which give a system usage profile. These statistics can be derived from a variety of sources — CPU usage, disk I/O, memory usage, activities by users, number of attempted logins etc. These statistics must be continually updated, and correlated with an internal model. This model may describe a set of intrusion scenarios or perhaps it encodes the profile of a "clean" system. Either way, significant processing must occur in matching the observed profile to an internal model.

The monolithic approach also presents some practical problems. If a new intrusion is discovered which utilises a system capability which was previously not monitored, then the IDS must be completely rebuilt to handle this. This is certainly not a trivial task. More importantly, the monolithic IDS represents a *single point of failure* — attack and destroy the IDS and system security is greatly reduced.

Finally, the monolithic approach does not scale well. It attempts to perform all the operations of an IDS in one module. In a networked environment it would be useful to distribute the IDS functionality across multiple machines. Not only would this reduce the overall load per individual machine, but if done correctly, could guarantee graceful degradation in the face of network partitioning.

3 A Finer-grained Approach

Instead of one large monolithic IDS module, we propose a finer-grained approach — a group of free-running processes which can act independently of each other and the system. These are termed *Autonomous Agents*[1]. They are trained to observe system behaviour, and cooperate with each other, so as to flag any behaviour that they consider to be anomalous.

Each agent is a lightweight program — it observes only one small aspect of the overall system. A single agent alone cannot form an effective intrusion detection system — its view of the overall system is too limited in scope. However, if many agents all operate on a system, and cooperate together, then a more complicated IDS can be built. However, agents are independent of each other. They can be added to and removed from the system *dynamically*. There is no need to rebuild the whole IDS in order to add new agents.

There are a number of advantages to having many small agents as against a single monolithic IDS. A clear analogy can be drawn between the human immune system and this proposal. The immune system consists of many white blood cells dispersed throughout the body. They must attack anything which they consider to be alien before it poses a threat to the body. Sometimes it takes more than just one white cell to actually destroy the attacker. By having a large number of cells, the body is always able to defend itself in the most efficient way possible. If an infection occurs in one area, then cells will move to that area so as to fight it.

Considering the small, lightweight agents in comparison to a monolithic approach, the following advantages can be seen:

- **Easily Tailored**
  By having many small agents which observe system behaviour the detection system can be tailored to a system's needs in the most efficient way possible.

- **Trainability**
  The ability to be trained is an advantage in that the human operator can identify major threats to be monitored and teach the agents to recognise these threats above all others. Once the major threats have been identified, the agents are free to evolve mechanisms to monitor for other, less obvious threats.

- **Efficiency**
  Obviously, users do not want a degradation in the performance of their system. The individual agents must be optimised to perform their monitoring in the most unobtrusive way possible. The primitives used by the agents are very simple and can interface cleanly with an existing network-layer interface. Once the training phase is complete, the agents will impose a low overhead on the system.

- **Fault Tolerance**
  If the system they were monitoring were to fail, the agents would not lose any state. As they encode their behaviour internally as actual code, restarting the agents would leave them in exactly the same state as before. They can resume monitoring the system without any degradation in performance.

[1] See the article by Maes [8] for an introduction to autonomous agents.

- **Graceful degradation**
  Similarly, if some agents are compromised, the system's defenses don't disappear. A graceful degradation in the system's ability to defend itself occurs - the best that can be expected in a case such as this.

- **Resilience to subversion**
  If a defense system is subverted by an attacker it is worse than useless - it gives a false sense of security. But knowledge of a particular agent on a system does not give knowledge of the operation of other agents - they all evolve under different conditions. Moving over to another system means that the agents there are slightly different so it is not a simple matter to subvert them. This is an important advantage.

- **Extendible**
  The agents could easily be modified to operate in networked environment where they actually migrated from system to system over the network. They could track anomalous behaviour over the network, and also move to systems where they would be most useful.

- **Scalability**
  The agents approach scales nicely to larger systems - simply add more agents and increase their diversity. Taking the whole notion to a network level also leads to an interesting insight - network agents which migrate around large networks and monitor network traffic for suspicious behaviour.

Of these the most important are, we believe, the ease of tailoring agents to your system, the resilience to subversion exhibited by agents and the highly scalable nature of the agents approach.

There are some drawbacks to the autonomous agent approach. They impose an **overhead** on the system as they will consume both memory and CPU cycles in order to monitor for intrusions. This is a cost of any intrusion detection system however, and the cost must be weighed up against the benefits of having a protection mechanism in place. **Training** the agents to monitor the system takes time. Unlike a solution which aims to be generic for every system, the autonomous agents will be tailored specifically for the system being monitored. This means that time must be spent analysing what is to be monitored before the agents can be placed in the system. The possibility of **false positives** must be minimised so as to make the intrusion detector a useful security tool. As in any intrusion detection system, if the agents are **subverted** then the intrusion detector becomes a security liability. Because the agents are distributed throughout the system and monitor many different system parameters, they are more immune to this sort of attack.

4 Architecture of an Agent

The guiding concept behind the design of each agent is *simplicity*. The agent should do one job well, do it in cooperation with the other agents, and do it as efficiently as possible. Each agent will be structured to monitor audit logs and other system data streams for a small subset (possibly even one) of activities. When an activity is detected (e.g. a login attempt, a network connection, excessive CPU usage etc.), the agent will simply inform the other agents on the system that it suspects an intrusion. Naturally this agent could mistaken. However, the other agents will take this into account in their own operation. Once these broadcasts exceed some threshold, a message can be sent to a human operator via some monitoring process.

This illustrates how a intrusion decision is reached. No one agent has the final vote to decide on a potential intrusion. It is a collective consensus between all the agents on the system. If only one agent suspects an intrusion, then it will be ignored. If many of them suspect anomalous behaviour, then there probably is a potential intrusion. Certain events may be more "important" in this scheme than others — for example 50 failed login attempts on root would be more significant than an ftp connection from a machine outside our domain. If two different agents detect these events, the message from the first agent detecting failed login attempts could be given more "weight" than that of the second. Other agents would then react differently to seeing the more "important" message — possibly monitoring for slightly different activities as a result.

The internal design of an agent can be based on a variety of paradigms. A simple static pattern matching system could be used — this could detect known misuse anomalies. Or a rule-based system could match observed events and have trigger actions for every matched rule. Instead of having all the rules encoded in one large rulebase (see, for example, the IDES system [6]), a small subset of rules pertaining to a single event could be encoded. A more flexible scheme still could use a dynamic learning system, such as a neural network or classifier system[2]. Another approach would be to use *Genetic Programming* [5] to evolve small programs which detected very simple activities on the system.

5 Prototype Solution

To illustrate how we envision such a system operating, consider a small local-area network of workstations running Solaris 2. These stations are behind a simple firewall, and they are protected by a collection of agents as we have described. They will be trained to detect anomalous activity in this traffic by being subjected to a training phase by a human operator. The operator will present different styles of network traffic (both intrusive traffic and neutral traffic) and guide the learning of the agents. Note that the agents use some learning paradigm internally, the operator does not have to explicitly adjust the operation of any of the agents. Once the training phase has been complete, they will be allowed to run continually on the system, observing activities and cooperating to decide whether these activities form part of an intrusion attempt. They will report their suspicions to a human operator.

We propose using the *Genetic Programming* (GP) [5] paradigm as a basis for the internal design of the agents. This is a powerful machine-learning paradigm which allows both feedback learning (i.e. human-guided learning) and discovery (i.e. finding new combinations of activities to monitor for). In a GP system, populations of programs are evolved to solve a specific problem. The problem often has no singular correct solution, or the solution is very expensive to compute. The possible solution programs are represented as parse trees for a simple meta-language and these parse trees are manipulated by operations similar to those found in natural genetics. After time the population of programs converges on a particular program (or set of programs) which gives the optimal solution to the problem.

These programs are no different from code written in a traditional language such as C or Pascal. However, their syntactic elements are drawn from a specially created language that allows a simple representation of important system parameters. For example, instead of having to calculate average CPU usage, this may be available to the programs as a variable `CPU_USAGE`. A layer of code between the agents and the system makes such abstractions available.

Figure 1 shows a simple parse tree for an agent. This parse tree corresponds to the following block of pseudo-code:

```
for-each-packet do
  if( ip-destination-address-of-packet
           is-not-equal-to my-ip-address )
      then generate-a-suspicion-broadcast
  endif
endfor
```

The *Terminals* in the parse tree (the primitives `IP-DEST`, `MY-IP` and `RAISE`) obtain their values from the abstraction layer beneath the agents (see Figure 2). In this simple example, the primitive `IP-DEST` would obtain the IP Destination address for the current packet from the abstraction layer and then the `IP-NEQ` function would compare that address to the IP address of the system (given by the `MY-IP` primitive).

What this simple agent does is to inform the other agents on the system if it sees a packet that arrived at this machine, but had a different IP destination address from the one on this system. This may or may not be a useful thing to do, but it may perform some function in conjunction with the other agents on the system at the time.

As another example, this shows how agents can detect if a packet arrives that is destined for a subnet outside the firewall. Not only that, if the packet is of type UDP, and destined to port 520 (the RIP update port), then

[2] See Goldberg [4] for more on Classifier Systems.

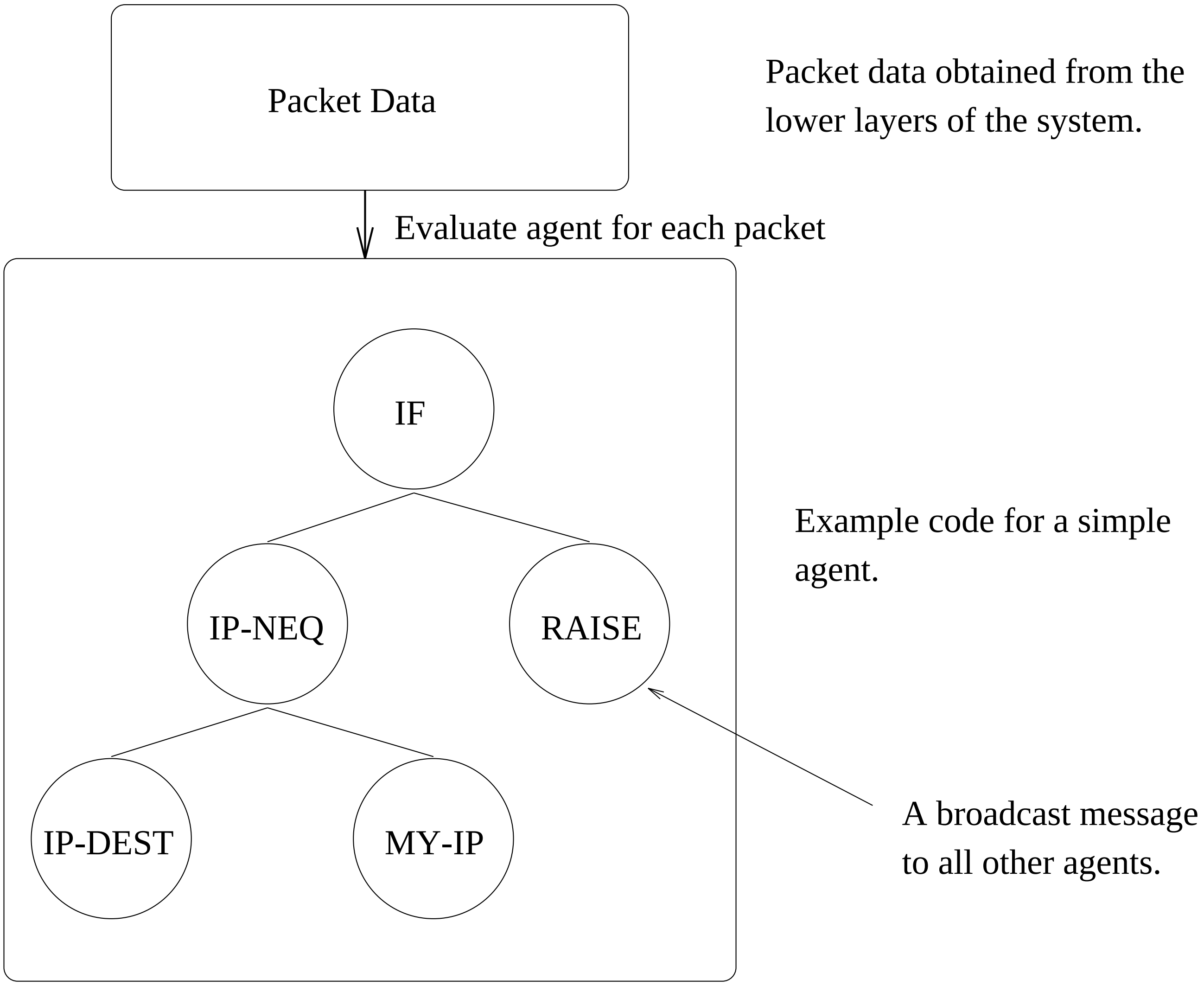

Figure 1: Sample internal parse tree for an agent

generate an additional suspicion message. This illustrates some of the flexibility in encoding the activity patterns as parse trees. Not only that, but it gives the overall system more flexibility — we could have a single agent replicated throughout the entire network monitoring for this activity, or perhaps a single agent on the firewall machine.

```
for-each-packet do
  if( get-subnet-part(ip-destination-address-of-packet )
           is-not-equal-to my-subnet-address )
      then
     generate-a-suspicion-broadcast
     if( (packet-protocol equals UDP) and
      (udp-dest-port equals 520)) then
      generate-a-suspicion-broadcast
     endif
  endif
endfor
```

By distributing this agent throughout the entire network, we have gained a significant advantage if the network is *partitioned*. Despite the loss of connectivity between machines, the agents can continue to operate independently and monitor for suspicious activities. This is an example of graceful degradation in the system which would not have occurred if a monolithic IDS was being used. If a single IDS was hosted on the firewall machine, then a network partition could leave machines behind the partition with little or no intrusion detection defenses.

Figure 2 gives an overall view of how the agents operate. At the lowest level is the raw network interface itself. In this prototype implementation, this is the Sun DLPI interface [7]. It provides an interface to allow programs to transmit and receive raw datalink-level frames. This system does not generate any new network data, so only the receive capabilities are used. The system can gather data from the network and encapsulate it in a form that can be presented to the agents.

Above this lies the Network Primitives layer. This takes the raw network data from the DLPI interface and encapsulates it in such a way so as to allow the agents to handle it. The agents will require the values of various fields in the network packet header, plus a variety of aggregate values, such as average packet size, inter-packet arrival times and time-of-day. These values must be either derived from the packet data or from outside system sources.

The agents operate above the Network Primitives layer. Each agent is actually a program which can be represented as a parse tree for a simple language. This language allows the agents to inspect the contents of network packets and perform operations based on this information. The network packet information is obtained from the underlying network primitives layer. The actual mechanisms of this, plus an example of an agent are described in the *Internal Design of the agents* section.

Above this lies the Training Module. Before the agents are allowed to monitor a system they must be trained to correctly respond to intrusions. They must also be trained to minimise the number of false positives (spurious intrusion reports) generated. This involves human interaction with the agents via this module. Once the agents have been trained they can be placed in a system without this module in place. The training is by a feedback mechanism — the operator provides an input describing whether the agents' actual behaviour was close to the desired behaviour for the given traffic pattern presented to them. It is similar to the training phase in neural networks.

5.1 Cooperation of multiple agents

The importance of cooperation is illustrated by this example. In this case, the agents are monitoring a far more varied selection of system parameters. Agents are distributed both across the network and on the host itself. There are three subsystems being monitored in the system — the network, the NFS device driver and the disk subsystem. The network connection has an agent which monitors the source address of incoming connections. If

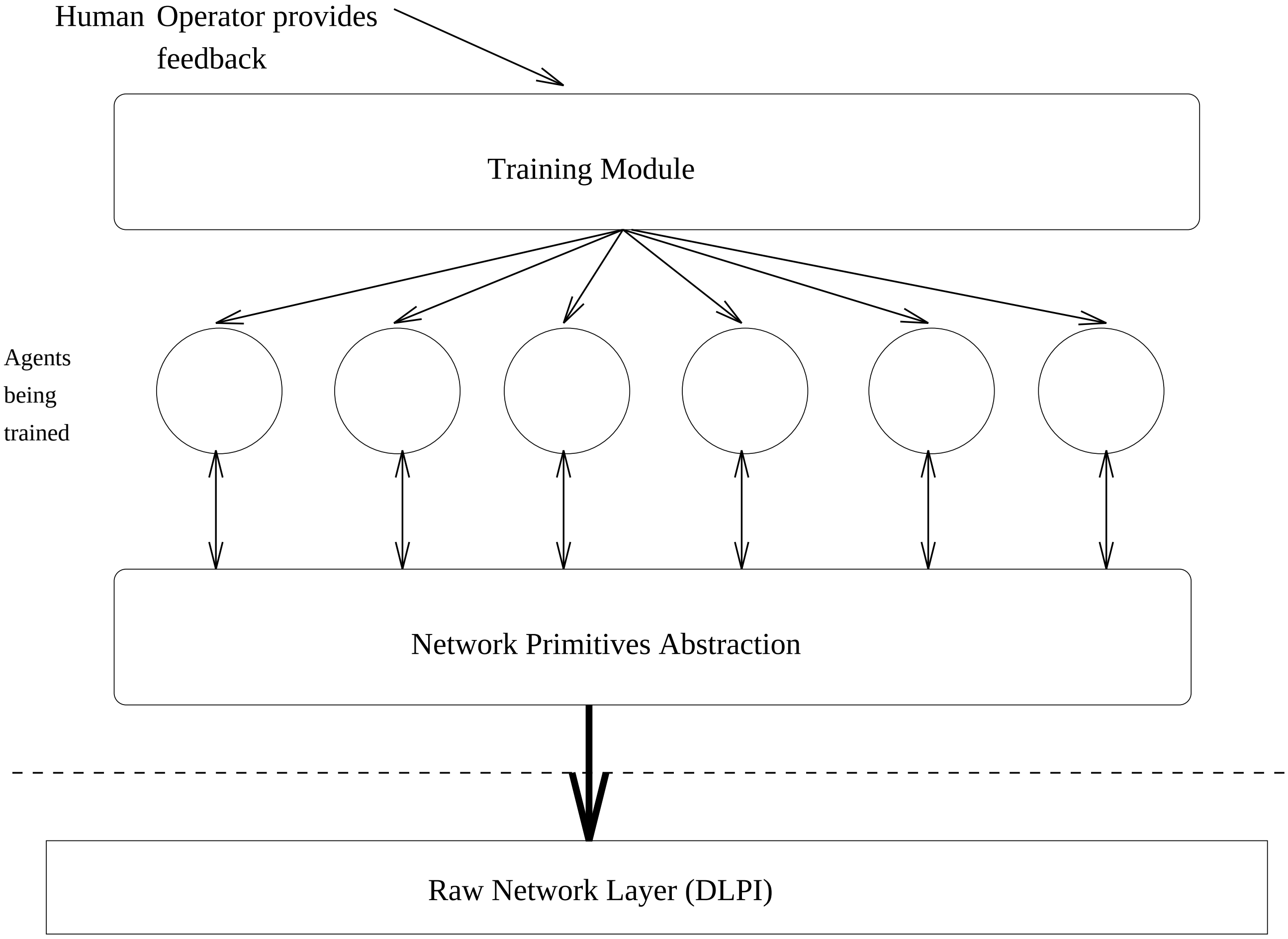

Figure 2: Architectural Overview of Agents in the system

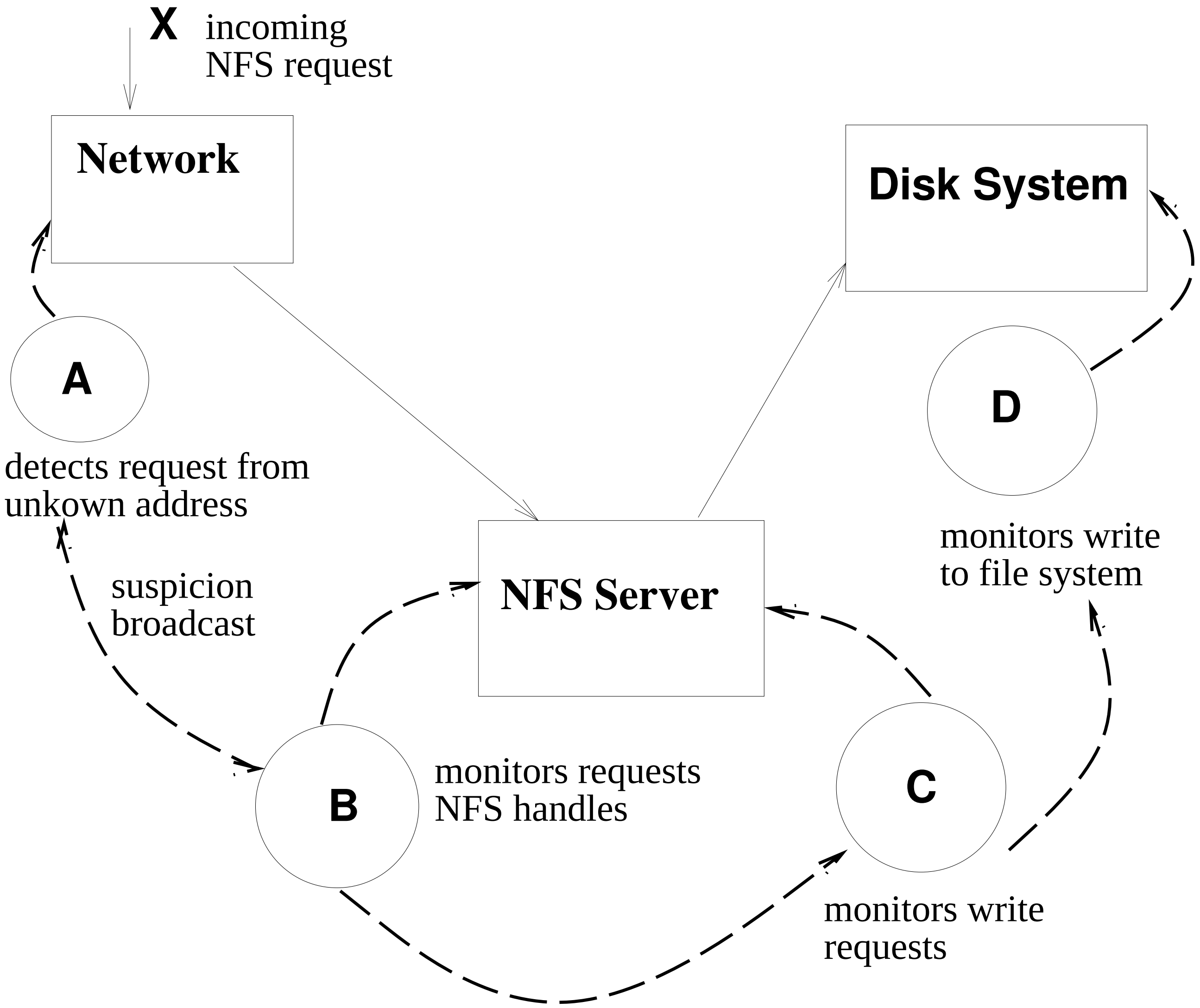

Figure 3: Agents monitoring a NFS write request

it sees one it has not seen before, it considers this as suspicious behaviour. Two agents are monitoring the NFS server. One of them analyses requests for NFS handles and another is monitoring all write requests. Finally an agent is monitoring the disk subsystem itself for writes to specific system directories. This is shown in Figure 3. Here an intruder is attempting to use a valid NFS handle to write to a system directory on the local disk. The intruder is coming in over the network from a previously unrecognised machine.

In this scenario, a write request comes in from a system X which agent A has never seen before. This causes the agent to become suspicious — it raises its suspicion level and sends a message out to other agents on the system. In this case the network connection is to the NFS server. This in itself is not enough to make agent A trigger an intrusion alert. However, agent B is monitoring requests for NFS handles and has received A's notification of suspicion. This, couples with its observation of the NFS request from X makes agent B increase its suspicion level, and broadcast this to the other agents.

Agents C and D have received these previous broadcasts and take them into account when they monitor actions. When the intruder at X issues a write request, agent C will have sufficient evidence to raise its suspicion level and broadcast this. Finally, when agent D sees a write to a system directory, its suspicion level has gone above a threshold value (due to all the earlier broadcasts from the other agents) and it will inform the operator of a possible intrusion.

This shows how the agents can cooperate to achieve the final goal of detecting an intrusion. Notice how each agent monitors for very common activities — agent B is monitoring NFS handle requests, a very common occurrence in a networked environment running NFS. However, it is only when a sufficient weight of evidence is gathered by all the agents working together that an alarm is raised.

6 Conclusions

We feel that the agents approach is a very powerful design abstraction. It is very scalable to as new agents can be continually added to a system. It is a simple enough concept so that it can be extended in a wide variety of ways — perhaps a hybrid approach using both agents and a monolithic IDS.

It also the flexibility to be adapted to a variety of network architectures. For example, in a network distributed system, agents could be distributed over all the systems in use. The agents could migrate from system to system, possibly based on each host potential for attack. This would be similar to antibodies moving into a location susceptible to a viral infection. The agents are small enough to allow this to happen in a clean, efficient way. Furthermore, the agents could replicate in a controlled way. If, for example, an intruder has managed to halt the operation of a particular agent, other agents could detect this and restart a new copy of that agent (or perhaps multiple copies). Thus the potential intruder is faced with an ever changing security defense system.

If the agents were distributed on a network-wide basis they could have greater scope for monitoring system usage. The network monitoring agents could work closely with the host-based agents to determine if an intrusion is occurring. For example, if network based agents were to broadcast an intrusion suspicion, the host-based agents could take steps to be extra vigilant — increased audit trail detail, or more thorough monitoring of network ports for example. Thus the whole network would become a single defensive unit. Each host on the network would be contributing to the defense of the overall whole.

Another extension would be to have the agents take some *active* measures in defending the system other than simply reporting on its state. For example, if they detect a suspicious connection, they may attempt to slow down the connection in order to forestall any damage while waiting for the human operator to take action. The more agents that became suspicious about such a connection, the greater the amount of slowdown. This idea could be extended even further — the agents could actively kill connections they did not trust, and hamper user activity which was outside acceptable bounds. These are extreme examples, and currently we feel that leaving the response to an intrusion in the hands of a human operator is a safer solution.

We propose building a prototype in order to test these concepts. Initially our prototype will monitor network traffic on a small local network behind a firewall. The primary goal of this prototype will be to prove that the concept of autonomous agents applied to intrusion detection works. We will examine the overhead such a system imposes on both the hosts on the network and the network itself. We will investigate ways to make the learning/training phase as simple as possible for any operator to perform. We also want to identify ways in which the agents approach can be generalised to larger network systems with heterogeneous machines.

References

[1] R. Heady, G. Luger, A. Maccabe, M. Servilla. *The architecture of a network level intrusion detection system.* Technical Report, University of New Mexico, Department of Computer Science, August 1990.

[2] B. Mukherjee, L. Todd Heberline, Karl Levitt, *Network Intrusion Detection.* IEEE Network, May/June 1994, pages 26-41.

[3] Sandeep Kumar, Gene Spafford. *A Pattern Matching model for Misuse Intrusion Detection*, Proceedings of the 17th National Computer Security Conference, October 1994.

[4] David Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addision-Wesley, 1989.

[5] John Koza. *Genetic Programming: On the Programming of Computers by means of Natural Selection.* MIT Press, 1992.

[6] T. Lunt, H. Javitz, A. Valdes et al. *A Real-time Intrusion-Detection Expert System (IDES)*, SRI International Technical Report, SRI Project 6784, February 28, 1992.

[7] Neal Nuckolls, *How to use DLPI*, Internet Engineering, SUN Microsystems.

[8] Pattie Maes *Modeling Adaptive Autonomous Agents*, Artificial Life, Vol 1 No. 1/2, Ed: Christopher Langton, MIT Press, 1993.