# Flexible, Mobile Agent based Intrusion Detection for Dynamic Networks

Christopher Krügel Thomas Toth

Distributed Systems Group, Technical University Vienna
A-1040 Argentinierstrasse 8, Vienna
Tel: +43 1 58801 18451 / Fax: +43 1 58801 18491
{chris, ttoth}@infosys.tuwien.ac.at

**Abstract**

*Intrusion detection systems (IDS) are security tools that attempt to detect malicious activities which are targeted against a network and its resources. Similar to virus scanners, IDS sensors need regular updates of their attack databases to detect novel intrusion attempts. Additionally, the identification of distributed intrusions requires cooperation of different sensors.*

*Mobile devices present a new challenge to intrusion detection. Whenever such a machine (e.g. laptop) is plugged into the network, it introduces a potential entry point that might be exploited by hackers. Ideally, the newly inserted device immediately participates in the intrusion detection process. This would require an instant integration into the detection process and an automatic update of the attack database. Such requirements are not addressed by current ID systems that assume a static environment.*

*We present Sparta, a mobile agent based ID system that allows to detect network intrusions in a changing, dynamic environment. Each participating device only requires an agent platform as well as generic sensor support. When connected to the network it becomes automatically and smoothly integrated into the ID process.*

## Introduction

The constant increase of attacks against networks and their resources causes a necessity to protect these valuable assets. Firewalls and intrusion detection systems are two complementary techniques to increase the security of a network. While a firewall acts as a first barrier to repel hackers, ID systems watch inside the network for attacks that have penetrated the first defense perimeter.

Intrusions can consist of several steps that occur at different hosts and therefore cannot be identified locally by a single sensor. This makes it necessary to consider multiple distributed events at the same time and deduce knowledge that is beyond the scope of an individual event. In order to be capable of detecting distributed attacks, each sensor has to participate in a coordinated process and needs enough information of its collaborators to communicate interesting observations. The process of detecting a related set of events with certain properties is called *correlation*.

Sensors need up-to-date attack databases to identify local events that might be part of a multi-step intrusion. It is insufficient to have knowledge of the steps of an intrusion when sensors do not extract the relevant basic events from their input data.

Current ID systems that meet these requirements are not constructed in a way that allows to deal with difficulties introduced by the use of mobile devices. As soon as a mobile device is connected to the network protected by the IDS, it should participate in the detection process. That implies that it has to be aware of events that might indicate an attack and that have to be extracted from the sensor's input stream. Additionally, the device needs enough knowledge of its peers and the network topology to engage in the correlation process.

As mobile devices are not permanently (a static) part of the network they have to learn enough of their environment to collaborate with remote ID sensors and retrieve a current attack database each time they are reconnected. It is too tedious for the user of a mobile device to perform this task manually an the security of the complete network is lowered when the integration is forgotten.

*Sparta* (an acronym for Security Policy Adaptation Reinforced Through Agents) is the name of a system architecture which is capable of monitoring a network to detect network intrusions and security policy violations. The event correlation is performed in a distributed fashion using mobile agents. While performing their task, these agents do not need a global view of the protected network and are therefore naturally suitable for operating in changing, dynamic environments. Instead of collaborating directly with its peers or a central server unit, devices only provide an execution environment for mobile agents. Sparta uses a configuration mechanism that allows to update sensors and synchronize their attack databases remotely over the network.

The combination of a decentralized detection scheme and the remote management of sensors enables our design to successfully deal with the challenges which are faced by ID systems that intend to operate in a changing environment caused by the utilization of mobile devices. In addition, the distributed detection process has superior scalability and fault-tolerance properties when compared to traditional systems with only one single correlation unit.

# 1 Functional Description

Sparta allows to identify and relate interesting events that occur at different hosts on a network. A single event is something of interest for our intrusion detection process which is further detailed by specifying values for its attributes. A set (or pattern) of events describing a multi-step attack scenario is defined by relating basic events. These connections are introduced as temporal or spatial relationships between events or by imposing certain constraints on their attributes. When dealing with complex patterns, it is not sufficient to select events based on their content alone. They have to be considered in context.

The fundamental functionality can be described as follows. Basic events are specified in an *event definition language* (EDL). These EDL specifications are then used to configure sensors to extract the corresponding, interesting events which are locally collected and stored. Currently, we use sensors that can monitor network traffic and system log files. The probes are remotely managed and set up in a way that allows them to detect occurrences that conform to events described in EDL. A user may issue queries in our *event query language* (EQL) to search for a set (pattern) of events that fulfill his desired constraints. In addition, the system can also be used to gather statistical information. It is possible to find the number of pattern instances at each host and to calculate the maximum or minimum for event attribute values as well as their sums over a set of hosts. The query is then carried out by mobile agents.

For our intrusion detection system, a failed authentication attempt or the start of a root shell might be examples of interesting events. Sparta allows to count the number of failed telnet logins for a certain user throughout the network (to detect distributed door knob rattling attempts) or to find tree-like connection patterns between hosts (to identify a spreading worm). It is important to notice that event correlation might yield information that is impossible to gain by just looking at a single node. Consider an intruder who tries to cover his tracks by performing several consecutive telnet logins (i.e. producing a **telnet chain**). This is an often observable behavior that exploits the fact that different machines are administered by different people and don't have synchronized local clocks. Tracing an attacker by having to look at all these logfiles is rather difficult. On each local machine only a simple incoming and outgoing connection is noticed but when looking at the entire network the offending pattern becomes evident. GrIDS [12] is a well known ID system which bases its detection solely on looking for connection patterns but uses a different mechanism to collect and relate data.

## 2 System Architecture

The system consists of a set of hosts connected by a network. Each host is running a Sparta instance that consists of *local sensors* (i.e. event generators), an *event storage component* and a *mobile agent platform*. Additionally, one node has to run the *management console*.

The local event generation is done by sensors which monitor interesting occurrences on the network (network based) or at the host itself (host based detection). The exact types of events and their attributes are determined by the application's needs and specified in EDL. EDL is a typed language that follows a basic object oriented design. Each event type is represented as a class. It is possible to derive a new event class by extending (subclassing) an existing one and add the desired additional information. This allows to write patterns that relate high level events and have the system automatically consider all actual instances (i.e. subclasses) of such generic events.

Sensors store their generated data in a local data storage component, preferably a database. The data storage component must be able to support the inheritance relationship of events. When queries specify parent class events, derived events have to be returned as well.

The mobile agent subsystem provides a communication layer to move the state and the code of agents between different hosts and an environment to execute them. Additionally, it implements protection mechanisms against security threats involved when using mobile code. Agents are encrypted while transferred over the network and authenticated on arrival. The agent platform also provides clock synchronization with a maximum guaranteed deviation. This is needed to temporally relate events at different nodes.

The management console is a central application that is needed to configure and manage the system. Notice however that it is not involved in the actual detection process which is carried out by collaborating mobile agents. The console allows users to specify queries and claim their results. It initiates the launch of appropriate agents and provides a way for them to communicate back their findings. As mentioned above, such queries are written in EQL, which is designed to conveniently specify patterns that describe security violations.

An important task of the management console is the provision of a membership database (directory service). When agents have to look for event patterns, they need access to a list of all hosts with installed agent platforms. The membership database keeps track of all hosts running a Sparta instance within the network. The membership database is updated when a new device is added to or leaves the set of hosts that execute the distributed detection process. Whenever a host intends to participate in the ID process, it registers itself at the management console using a simple multicast protocol. In order to be able to remove stale entries from the database, each machine has to periodically announce its status to the management console. Whenever a host should be disconnected it needs to unregister there.

The management console is also responsible for the configuration of remote sensors. The update of event specifications at each sensor can be manually initiated at the management console and an automatic synchronization is performed each time a device is introduced into the system and registers itself.

# 3 Specification Languages

The basic building block of a pattern is a set of local events. Each local event is specified by a description written in our event definition language (EDL). The specification of an event in EDL includes its name, its class type and constraints on a set of its member attributes. Each device has to provide basic sensor support to gather event data. Every sensor is capable of producing events of certain basic types where each type has a list of attributes (with their corresponding types). Sensors can be able to extract and interpret packets from the network (network based) or read and parse log file entries (host based). Not all events (i.e. all network packets or log file entries) that are monitored by sensors are useful for the subsequent intrusion process. EDL defines interesting events as instances of basic event classes by imposing certain constraints on their attributes.

Consider the case of a network based sensor like Snort [11] that can read and interpret traffic from the network. It can process IP based traffic and produces events of the types `IP`, `ICMP`, `TCP` and `UDP`. As a TCP/UDP packet is encapsulated (contained in the payload) of the IP packet, the corresponding `TCP`/`UDP` event classes have to inherit the attributes from the `IP` event class. The basic event classes and their respective attributes must be known to the sensors and the management console in advance. An brief excerpt from the `IP` and `TCP` class definitions are given below.

```
class 'IP' {
   ...
   IPAddress source_addr;
   IPAddress dest_addr;
}
```

```
class 'TCP' extends 'IP' {
   int16 source_port;
   ...
   boolean syn_flag;
}
```

Although it is be possible to store each received event in the local storage component for subsequent processing, the required capacity would soon exceed the available resources. Therefore, a filter has to be applied to extract only events might be part of a larger pattern (i.e. multi-step attack). EDL is used to specify such filters (i.e. event definitions) in a sensor independent way. These specifications are sent to the remote hosts where they are stored in the attack database and are used to configure the local sensors. Each sensor must process the filter definitions for all types of events that it can produce. This usually involves a translation from the sensor-independent EDL specification into a specific configuration suitable for the probe. The translation is done remotely at each sensor, the management console is only concerned with EDL specifications.

The following descriptions show two filters named `'udp-filter'` and `'tcp-connect'` written in EDL. The first one matches UDP packets originating from IP address 192.168.0.1 sent to port 7 while the second matches TCP packets that initiate a virtual connection (TCP 3-way handshake). Notice that the source address attribute is accessible in `'udp-filter'` because the `UDP` class is derived from `IP`. The translation into the format used by Snort (which is utilized in our system) is shown as an example below. The variable `host` is always defined as the IP address of the host where the sensor is installed.

```
EDL:   'udp-filter': UDP {
           source_addr = '192.168.0.1';
           dest_port = 7;
       }

       'tcp-connect': TCP {
           syn_flag = true;
       }

Snort: alert udp 192.168.0.1/32 any -> $host/32 7 (msg:'udp-filter')
       alert tcp any any -> $host/32 any (flags:S; msg:'tcp-connect')
```

Every packet that matches a filter has to be stored in the storage component and its data is used to fill the fields specified by the event's class definition.

EQL allows to formulate a query that describes sets (patterns) of basic events with their relationships. In order to limit the amount of data that has to be transferred between hosts by mobile agents, our query language does not allow arbitrary patterns. When a system uses mobile code (i.e. mobile agents), it should aim at performing flexible computation remotely at the location where the interesting data is stored instead of abusing agents as simple data containers.

A pattern $\mathsf{P}$ basically consists of one or more sets of events (called *event sets*). Each event set describes the sequence of events that have to occur locally at a certain host in order to satisfy the pattern. A pattern that relates events that occur at n distinct hosts consists of n sets of events, one for each node.

A connection between events at different hosts (i.e. between different event sets) is established by connection events. A set of events $\mathsf{S}_A$ at host $\mathsf{A}$ is linked to a set of events $\mathsf{S}_B$ at host $\mathsf{B}$, if and only if $\mathsf{S}_A$ contains a send event and $\mathsf{S}_B$ contains the corresponding receive event.

A pair of corresponding send (at node $\mathsf{A}$) and receive (at node $\mathsf{B}$) events are created whenever $\mathsf{A}$ opens a channel to $\mathsf{B}$ to transmit data (e.g. open a TCP connection, send a UDP packet, send an Ethernet frame). Two predefined labels called `send` and `receive` are used to identify the send and receive events, respectively, in the event sets of the pattern's EQL specification.

All nodes except one (called *root node*) must have exactly one send event. The root node itself contains no send event. This limitation allows only tree-like pattern structures with the root node (the one without a send event) being the root of the tree. Although this restriction seems harsh at first glance, most desirable situations can still be described. Usually, activity at a target host only depends on events that have occurred earlier at several other hosts, a situation that can easily described by such patterns. The major advantage of the proposed limitation is the possible implementation of an efficient search algorithm (for details, see Section 4).

The constraints on basic events are expressed by relating the event's attributes to constant values using one of the standard operators $(=, != , <, >, \leq, \geq)$ with their usual semantic meaning. Relationships between different events and their attributes are realized with variables. The value of a certain event attribute can be assigned (bound) to a variable. This variable may then be used in relations with other event attributes thereby effectively creating a link between the attributes that define and use the variable. The value of a variable may be read arbitrarily often but it may be bound to an attribute only once.

The following example shows a telnet chain pattern that describes a connection from Node1 to port 23 at Node2 and from there to port 23 at Node3 (which is the pattern's root node). We use the basic `tcp-connect` event and the predefined variable `host` as explained above.

```
Node1 {
  send(node2): tcp-connect [ source_addr = $host; ]
}
Node2 {
  receive(node1): tcp-connect [ dest_addr = $host; port = 23; ]
  send(node3): tcp-connect [ source_addr = $host; ]
}
Node3 {
  receive(node2): tcp-connect [ dest_addr = $host; port = 23; ]
}
```

## 4 Pattern Detection

Usually, patterns are discovered by gathering event data from distributed nodes at a central host where pattern matching algorithms distill the desired information. Our approach differs from the usual setup by detecting patterns in a fully distributed way using mobile agents. Mobile agents roam the network to search for suspicious events to start a more detailed investigation. When an agent spots the mark of a possible intrusion, it decides which data to carry with it on its next hop and which place to visit next.

The advantages of such a pattern detection scheme base on the fact, that no central entity is needed for data correlation. This increases the fault-tolerance of the system as no single point-of-failure is present. The scalability is improved by the fact that new hosts on the network won't automatically send additional traffic to the single server machine. The biggest advantage is the fact that no global information is used during correlation. After agents have started, they act autonomously and make local decisions. This allows the integration of new devices without the need of updating collaborating sensors (except the entry in the membership database at the management console).

The detection is done in the following way. An agent is started at the management console with a given pattern in EQL that it has to look for. It starts its task by contacting the directory service (membership database) to obtain a list of all hosts with an installed agent platform. These nodes are then visited in arbitrary order.

When an agent arrives at a host, it looks for events that fulfill the constraints given for the root node of the pattern it is currently investigating. In the case of the telnet chain introduced above, the agent would have to look for received TCP connections at port 23 (see Step 1 of Figure 1). The result of this process are a number of events (representing different instances of the pattern) which satisfy the root node constraint.

When no events are found, the agent immediately continues its journey. Otherwise, all receive events are identified. In the telnet chain case only receive events exist but more complex patterns are possible. For the distributed detection algorithm, a receive event is important because it establishes a relationship between nodes which is used to select a promising next place where an agent should look further. For each receive event, a helper agent is spawned which follows the link to the host with the corresponding send event (see Step 2 of Figure 1). By using the send event (determined by the receive event of the node the agent was coming from) it looks for events which fulfill the current part of the pattern. In the case of our telnet chain example, the agent already knows that the send event to Node 3 exists (as it is coming from there) and now searches for another accepted TCP connection at port 23 (from Node 1).

When the local event set contains receive events itself, the process recursively repeats by having the agent spawning helper agents and waiting for their return (see Step 3 of Figure 1). When the helper agents return, they report their findings (i.e. pattern instances) back to the agent waiting at the originating node. Helper agents only move over a single hop to a remote node and eventually

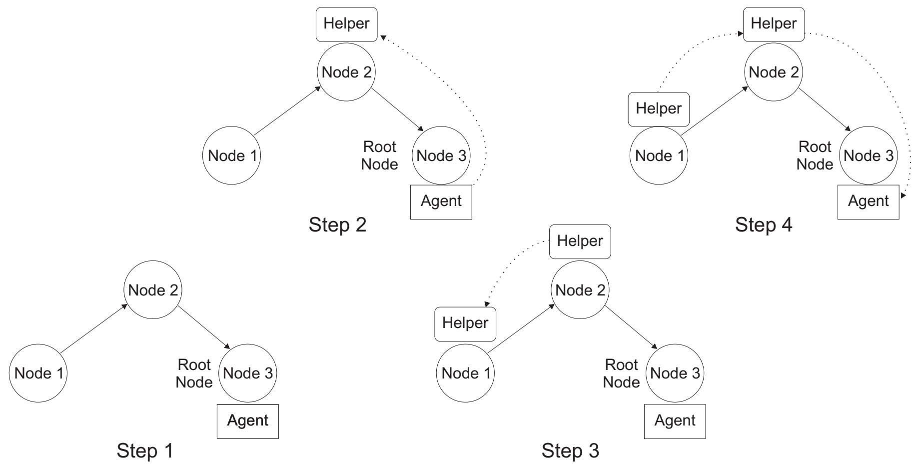


Figure 1: Pattern Detection

return back. When pattern instances are returned, the waiting agent processes them (e.g. match variables or perform statistical evaluation) and eventually continues. When all helper agents have returned a pattern might be detected by the agent waiting at the root node as all information is available (see Step 4 of Figure 1).

Usually, only a small amount of data has to be transferred as it is not necessary to transport all pattern instances themselves but merely time stamps or single attribute values which have been assigned to variables.

Variables are treated in the following way. When a variable has already been bound to a value, it is straightforward to use this value directly for the attribute's constraint. This is the case when a value is assigned to a variable at a node which is closer to the root than the node where it is used. On the other hand, when a constraint depends on a variable which has not been resolved yet, one has to temporary ignore that constraint. First, all actual bindings of the variable have to be determined and are then matched against the instances where the variable has not been assigned yet. This allows to filter out pattern instances which do not satisfy the previously ignored constraint. Notice that this may cause agents to transfer unnecessary pattern instances but the runtime complexity is still linear with regard to the number of found send/receive events.

Instead of having a single agent visiting all nodes sequentially, the task could be parallelized easily by partitioning the set of interesting hosts. Each partition is visited by a dedicated agent which all have to agree on a destination node, where they meet and merge their results.

## 5 Evaluation

While mobile agents often provide an elegant solutions for problems in distributed environments they also inflict additional costs onto the system in terms of potential lower performance and higher bandwidth usage. Agents do not run as native processes on hosts and in addition to the data, code has to be transported as well. In contrast to systems with a static sensor configuration, our dynamic, remote update of probes is also more costly. We measured configuration and detection process properties of our prototype system and concluded that our design is well suited for dynamic networks.

Our testbed consisted of three nodes with Sparta components (2 Pentium III with 550 MHz, 1 Pentium with 200 MHz) that were connected by a lightly loaded 10 Mbps network. One Pentium III ran the management console. All measurements have been performed 10 times and the average results are shown.

The configuration of the sensors involves several steps. First, interesting events have to be defined in EDL. We have developed a tool that uses the intrusion specifications of the arachNIDS database [1] as input and converts them into EDL filters. Our sample set consisted of 504 event description that were translated into EDL at the management console in 0.48 seconds. This step obviously has to be performed only once. The EDL descriptions are transmitted to sensors over the network where they have to be parsed into an internal data structure and added to the attack database. In order to configure the sensors, these descriptions need to be converted into the probe specific format. Then the sensor has to perform some initialization work on its own. The parsing and insertion of all 504 event descriptions lasted 2.21/3.43 (at a PIII/Pentium) seconds but one has to notice that extensive error checking was enabled. In addition, a merge between new EDL descriptions and specifications already in the database has to be performed. The translation into Snort rules (as Snort is used as our network sensor) took another 1.31/1.67 seconds. The time it took Snort to set up the appropriate network filters is shown for completeness and amounted to 0.83/1.02 seconds.

We used two different EQL queries to measure agent properties. One represents the simplest possible query and specifies only a single event at the root node. The other one is the telnet chain query used as an example in Section 3 which links attributes of events at different nodes. The parsing of the query at the management station took 0.01/0.03 seconds for the simple/complex query.

The following table summarizes our results for the execution of the detection process. The databases have been filled with events that allowed agents to detect 789 instances of the simple pattern and 37 instances of the complex one.

| Property | Simple | | Complex | |
|---|---|---|---|---|
| | PIII | Pentium | PIII | Pentium |
| Local Detection Time | 10ms | 30ms | 27ms | 80ms |
| Agent Size | 6080 bytes | | 26752 bytes | |
| Agent Size (encrypted) | 6918 bytes | | 27590 bytes | |

It took 0.34 seconds on the PIII and 0.54 seconds on the Pentium to perform the authentication of an agent. This time includes the retrieval of the certificate from the remote host and the decryption of the agent. When the certificate is already present, the time drops to 0.08/0.16 seconds.

The transfer time of an agent between two hosts amounted to 11 milliseconds which is rather independent of the actual agent size as most time is spent during connection setup.

# 6 Related Work

The idea of correlating events which occur at different places in a network is implemented in a number of academic intrusion detection systems. ID systems like EMERALD [9], NetStat [13] or AAFID [3] gather and relate data from different sources. They have a static, hierarchical architecture where sensors located at different hosts collect data and send it to a central entity where events are related. In contrast, our distributed design with mobile agents allows the use of the system in dynamic environments (as created by the use of mobile devices). Additionally, scalability and robustness properties are improved in comparison to traditional approaches. The systems above also suffer from the drawback that their sensors need to be configured in advance and changes in the network topology often result in (partially) manual readjustments.

Commercial intrusion detection systems like Network Flight Recorder [8], Real Secure [10] or Net Ranger [7] perform their analysis on packet level by monitoring network traffic. As only local analysis is done, these products do not have to address the problems of changing network configurations.

Only a few intrusion detection systems have tried to use mobile agents so far. While IDA [2] mainly uses them to trace a possible attacker back to its origin, Micael [4] pursues a more ambitious aim where each system component is realized as a mobile agent. Unfortunately, only a high-level system design has been presented and no details have followed so far. The potential advantages of mobile agents in intrusion detection systems are summarized in [5] and [6].

# 7 Conclusion

The automatic update and integration of new devices into the intrusion detection process is vital for maintaining an adequate security standard in dynamic network environments. When devices or sensors have to be manually updated or integrated into a running ID system, no consistent event correlation process that includes all nodes can be established.

We present a solution where mobile agents perform the task of correlating intrusion data in a fully decentralized manner. In addition, the attack databases of probes are automatically updated when they join the federation of distributed ID sensors. This allows the implementation of an efficient intrusion detection system in varying environments caused by the use of mobile devices.

## References


[1] arachNIDS: advanced reference archive of current heuristics for Network Intrusion Detection Systems. http://www.whitehats.com/ids, 2001.

[2] Midori Asaka, Atsushi Taguchi, and Shigeki Goto. The implementation of ida: An intrusion detection agent system. In *Proceedings of the 11th FIRST Conference*, June 1999.

[3] Jai Sundar Balasubramaniyan, Jose Omar Garcia-Fernandez, David Isacoff, Eugene Spafford, and Diego Zamboni. An architecture for intrusion detection using autonomous agents. In *14th IEEE Computer Security Applications Conference*, December 1998.

[4] Jose Duarte de Queiroz, Luiz Fernando Rust da Costa Carmo, and Luci Pirmez. Micael: An autonomous mobile agent system to protect new generation networked applications. In *2nd Annual Workshop on Recent Advances in Intrusion Detection*, September 1999.

[5] Wayne Jansen, Peter Mell, Karygiannis, and Don Marks. Applying mobile agents to intrusion detection and response. Interim Report (IR) 6416, NIST, October 1999.

[6] Christopher Krügel and Thomas Toth. Applying mobile agent technology to intrusion detection. In *ICSE Workshop on Software Engineering and Mobility*, 2001.

[7] NetRanger. http://www.cisco.com/univercd/cc/td/doc/product/iaabu/netrangr/, 2001.

[8] Network Flight Recorder. http://www.nfr.net/, 2001.

[9] Phillip A. Porras and Peter G. Neumann. Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th NIS Security Conference*, October 1997.

[10] RealSecure. http://www.iss.net/customer_care/resource_center/product_lit/, 2001.

[11] Martin Roesch. Snort - lightweight intrusion detection for networks. In *USENIX Lisa 99*, 1999.

[12] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. Grids - a graph based intrusion detection system for large networks. In *Proceedings of the 20th National Information Systems Security Conference*, volume 1, pages 361–370, October 1996.

[13] G. Vigna and R. Kemmerer. Netstat: A network-based intrusion detection system. In *Proceedings of the 14th Annual Computer Security Applications Conference*, December 1998.