ODISET: On-line Distributed Session Tracing using Agents *

Salvador Mandujano (smv@itesm.mx) Arturo Galván (agalvan@itesm.mx)

Center for Intelligent Systems
Instituto Tecnológico y de Estudios Superiores de Monterrey
Monterrey, Mexico

Abstract

When a security incident occurs it is sometimes necessary to identify its causes for legal and cautionary purposes. In an attempt to hide the origin of her connection, a malicious user may have jumped from a source host h_s into a series of hosts $h_i \in H = \{h_1, h_2, ..., h_n\}$ before breaking into final target h_t. This connection sequence describes a path that makes it difficult to find h_s given h_t due, in part, to the prohibitive amount of cooperation and synchronization that is required in practice by administrators. This paper describes a distributed rule-based model that automates this tracing process on-line with a $O(|H|))$ worst case scenario. Autonomous agents collaborate on the tracing and detection of the origin of an interactive connection using a loop unwinding technique and incorporating public key cryptography to create ciphered channels that allow them for secure communication. To meet the challenges of minimum system workload and improved robustness, the prototype features lightweight design and implementation as well as a dynamic port-allocation scheme to prevent sniffing and denial of service attempts. We describe the proposed model and present the experimental results obtained with the prototype system ODISET.

1 Introduction

Unauthorized access to computer systems is increasing in parallel with the growth of the Internet ([Rapalus, 2002]). For purposes of protection, accountability and liability, the need for connection back-tracing is a reality in the areas of cybernetic-law enforcement and computer security. The authentication structure of our networks as well as their distributed nature complicates the task of spotting the host from which a connection sequence stems. If some sort of anomalous activity is observed from one of our users, it may be necessary to trace the origin of the session in order to identify the person behind it. In most cases, this is not easy to accomplish [Yoda and Etoh, 2001].

Within the domain of a LAN, it is rather straightforward to perform this tracing given that system administrators usually have access to all the hosts within their LAN. Unfortunately, not all sessions come from the inside. If a connection is received from the outside, we will have to contact the administrator of a remote host h_r in order to discover the identity of the user or the next link of the connection path or *chain*. Our fellow administrator will have to look into her system to realize that, perhaps, it has been compromised as well. It is possible that the user we are looking for is using h_r as a step-stone to reach our server. In that case, we will have to contact some more system administrators within the country and/or abroad [Cheswick and Bellovin, 1994] [Stoll, 1987]. This has obvious implications time and cost-wise.

Once a system has been compromised, the perpetrator may have left back-doors installed to facilitate future access to the system. Using interactive sessions, she may visit the system again even when the software has been patched and her back-door remains untouched. It is not necessary for the user to be logged on in order to be detected. Audit trail files can be used to do off-line tracing but this can be a tough task since audit logs may have been tampered with or deleted.

In the case of connections to web servers through `HTTP` and `HTTPS`, for instance, the hosts that compose the connection chain between client and server cannot be easily modified. Some components of the routing system would have to be modified in order to force the packets to follow a particular path. In the case of interactive sessions, however, applications such as `telnet`, `login`, `rlogin` and `ssh` allow a user with suitable access permissions to connect from host to host back and forth creating loops at will to scramble her connection path a little. This significantly complicates the structure of the chain and the amount of effort required to detect the root host h_s from where she is actually starting everything up. In fact, the complexity of back-tracing a connection across a set H of n hosts maybe small for a small n and a small number of connection hops, but it rapidly increases as n becomes larger due to the number of hosts that a chain may contain.

One of the challenges of intrusion detection is to accu-

*This project was supported by the INSYS Computer Security Research Grant (*Cátedra INSYS de Investigación en Seguridad Informática*) of Instituto Tecnológico y de Estudios Superiores de Monterrey

rately identify the main causes of an incident. The ultimate goal is to catch intruders in real time and our objective is to help intrusion detection systems do the part of session tracing on-line. There are some intrusion detection and intrusion response tools [Yoda and Etoh, 2001] [Asaka *et al.*, 1999] [Wang *et al.*, 2001] that provide some sort of off-line connection tracing that also performs data correlation with previously recorded user activity information.

This paper proposes an on-line approach that makes use of intelligent agents to identify the origin of a live session by unwinding connection loops. Instead of using historical audit trails, this model relies on evidence captured on operating system tables that are continuously refreshed by the system. For this reason, it is not likely that these tables experience changes that go unnoticed. The communication structure of the agents uses RSA public-key cryptography [Rivest *et al.*, 1979] [Schneier, 2001] to implement an encryption model that protects all communications between agents. One of the main issues of the multi-agent approach to automatic security monitoring is the overhead imposed by this sort of tools [Weiss, 1999] [Spafford and Zamboni, 2000]. Our prototype system *ODISET* was built from a lightweight design that does not represent a performance threat, making it ideal for wide deployment through an open source operating system. We present the design ideas behind the model, the tracing and communication techniques, the loop-unwinding method, as well as some experimental results of the first release of the prototype.

The rest of this paper is organized as follows: Section 2 explains the problem of connection back-tracing. Section 3 discusses relevant related work in the area. Section 4 is a description of our solution approach and the implementation issues of the prototype. Section 5 includes experimental results and findings. Conclusions and future work are in section 6 right before the listing of bibliography and references.

2 The Problem of Connection Back-tracing

Given a set of hosts $H = \{h_1, h_2, ..., h_n\}$, with $n \gg 2$, a user could create a connection chain of m hops as follows: from host h_i into h_{i+1}; from h_{i+1} into h_{i+2}; from h_{i+2} into h_{i+3}; and so on, until finally connecting to target host h_{i+m}. The problem of connection back-tracing is: given the set H, the target host h_{i+m}, and username u connected to h_{i+m}, find the hosts that compose the connection chain and identify its root host h_i.

For a small $n = |H|$ this problem is straightforward. Just a few hosts need to be analyzed in order to identify the source of a connection sequence. However, as n becomes larger (e.g. $n \gg 2$), the complexity of the problem increases since a larger number of hosts in H multiplies the possibilities of creating a more confusing connection path.

System administrators have limited access to information on who is connecting to one of their servers. On a LAN, administrators typically have login access to all servers, which makes the tracing process easier; however, for connections coming from the outside of the LAN, they will be unable to freely access all other servers for scrutiny. The visibility of this type of connections reduces to one link of the connection chain. That is, an administrator can only see the terminal number or the IP address/host-name of the immediate host from where the users are logging on. She is unable to tell, however, whether that host is in fact h_i (the machine from where the user is actually typing in commands). In order to find out whether that is the root host or not, she will have to contact the administrator of the remote system who, in turn, will have to perform a similar checking.

This job is cumbersome and requires time, communication, and trust from the parties involved. For the case of a connection that has been closed already, audit information will have to be examined in order to find the nodes of the chain that lead to the origin of the connection. This off-line sort of tracing has an important flaw: it counts on the existence and integrity of audit log files. For clean systems this would not be a problem but, for systems that have been compromised, and, in the case of a connection chain created to perpetrate an attack this could probably be the case, these files could have been tampered with during a previous break-in [Yoda and Etoh, 2001]. On the other hand, if a connection is still alive, i.e. there is a flow of packets between source and destination, information contained on system tables and network packets can be used to trace the user while she is on-line. That is, precisely, our approach.

3 Related Work

There are some projects that touch on the problem of connection route tracing. One of them is the method proposed in [Yoda and Etoh, 2001]. This model implements network traffic monitors that perform session tracing once a server has been compromised. A system that computes deviations from the traffic observed at two different hosts helps determine if these hosts were used in the same connection chain. Since checking network traffic at a node involves huge amounts of data, multiple packet monitors permanently filter and record specific system activity to build their own logs. These files are analyzed by a data correlator to identify the hosts of a connection chain. It is important to notice that this system deals with the problem of deleted logs by creating its own records. These records, however, may be the target of the same type of attack.

IDA (Intrusion Detection Agent system) [Asaka *et al.*, 1999] is another system that detects the origin of an information exchange related to an incident. It is primarily an intrusion detection system that employs mobile agents to detect local attacks. From a main host, investigation agents are sent over to the requester that needs an integrity check. These agents use the MLSI (*Marks Left by a Suspected Intruder*) strategy to reduce the amount of data needed to flag system activity as anomalous. In order to collect further information on a break-in, IDA does passive route tracing using audit log data. The structure of the tracing method is not specified by the authors but its agents broadcast connection evidence to their peers as a way of identifying possible hosts involved in the attack.

The *TBAIR* system (*Tracing-based Active Intrusion Response*) [Wang *et al.*, 2001] is a network-based intrusion detection tool that, unlike most traditionally-passive intrusion

detection systems, adds active response to security events using a technique called *Based on Sleepy Watermark Tracing*. TBAIR tries to attack the root of the vulnerability exploiting problem by locating the originators of the incident to hold them accountable for their intrusions and their method is based on the analysis of evidence collected from the network.

Thumbprinting [Staniford-Chen and Heberlein, 1995] is a technique that places several processes across the network in order to capture activity signatures or "thumbprints". This method does not require to have processes on all hosts and, like in the case of [Yoda and Etoh, 2001], it replicates packet information for future review. This system is based on the fact that the packet content of a session passing though a set of hosts is invariant and can be compared in order to identify a host in the chain. This system is capable of detecting the root of a connection but it is not designed to identify the exact hosts that are part of the chain.

The approach that we present here through the implementation of the ODISET system differs from these previous projects in the following manner. It performs on-line rather than off-line tracing (i.e., it does not use audit logs). The system does not incorporate agent mobility. Instead, it uses a distributed model using agents that communicate over secure channels for sharing information. Unlike some other models, ODISET is able to not only identify the origin of the connection but also all the hosts that are part of the connection chain.

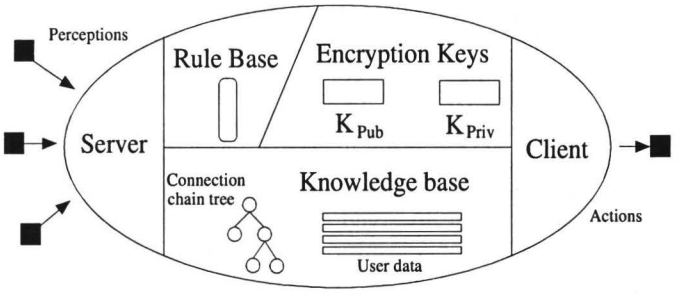

Figure 1: Basic composition of an ODISET tracing agent: public and private encryption keys, client functionality, server functionality, rule base, and knowledge base (connection chain tree + user data).

4 Solution Approach using Agents

"We require systems that decide for themselves what they need to do in order to satisfy their design objectives. Such computer systems are known as agents" [Weiss, 1999]. The use of agents for performing system assurance activities [Spafford and Zamboni, 2000] that typically require the skills of a human (e.g. inference, learning and decision making) makes possible the development of automatic or semi-automatic tools that aid system administrators in their task of securing a system.

The distributed nature of agents allows for more efficient, parallel data processing. We take advantage of this feature that perfectly fits into a network environment [Huhns and Singh, 1998] in order to speed up connection back-tracing using autonomous agents. This approach eliminates the need for centralized data collection and analysis, a method that is still in use and that represents a design deficiency by putting the availability of a system in risk (monolithic systems typically suffer from having a single-point-of-attack).

In order to provide the agents with survivability features that make them more resistant to attack, each agent owns a digital certificate to create private communication channels when interacting with its peers. With the intention of reducing the probability of *Denial of Service (DoS)* attacks targeted toward the agents, a port-allocation scheme is used. This scheme dynamically changes the socket port numbers used by the agents in order to avoid being monitored or receiving unfinished-protocol requests.

This system features rule-based agent behavior through which individual agents are capable of detecting connection loops in a chain. The approach deals with the problem of deleted or damaged audit information by not using historical audit data. Agents look up information on dynamic session tables in order to extract data relevant to the tracing process. The process table, for instance, is not easily modifiable as audit logs are. The system might go down due to inconsistency if changes are made to this type of file, which would make the change evident.

4.1 Proposed Model

Tracing method and security

The system is composed of a number of tracing agents (Figure 1) that communicate with each other sharing information on user connections. There is an agent running at each host h_i and the ideal situation is having an agent running on every host of the network. Every agent has client and server capabilities that enable it to request information from other agents and to supply information to them. An agent is not useful by itself, it needs to collect information from the others in order to identify the links of a connection chain. The agent at the target host will eventually collect information from a number of its peers and will inform to the local administrator about the origin of the connection.

The perceptions of a tracing agent are received from two main sources: 1) the state of the system regarding user sessions and connections, and 2) messages from other agents requesting or providing information.

The environment an agent inhabits is highly dynamic and provides the communication channel necessary for an agent to stay in touch with its peers over the network. This environment can also represent a threat to the tracing system if malicious users gain access to it.

As a result of the changes perceived by an agent, it can perform a number of different actions: a) send messages to an agent, b) request messages from an agent, c) broadcast messages to all agents, d) generate encryption keys, e) encrypt messages, f) decrypt messages, g) identify connection loops, h) retrieve system status information, and h) save its knowledge to a safe location.

The knowledge of an agent is stored on a repository known

as *knowledge base (KB)*. This KB includes tracing host information and user data. A tree structure stores the information on the hosts that are part of a connection chain. Information regarding user names, login times and dates is stored on a separate structure. These two elements represent all the knowledge an agent has about the state of its environment. Should this knowledge be lost, the agent would lose track of previous events and would have to be updated by its peers.

The actions an agent takes depend on the information of the state of the system and its perceptions [Russell and Norvig, 1995]. Embodied into the agent are a set of rules that enable the agent to work autonomously deciding what to do. As new knowledge is fed into the KB more rules from the *rule base* need to be analyzed before making any conclusion and proceeding (the rule base is composed of 23 rules). The type of perceptions and actions do not change but the state of the environment does evolve hereby modifying the behavior of agents. The type of rules that make up the rule base are of the form:

```
  R1:
IF ((HOW-CONNECTED(User)=Console) AND
(WHAT-DOING(User)=Ssh-to(Target)))
⇒ TRACING-DONE();
  R2:
IF ((MESSAGE-TYPE(Message)=Tracing-
request) AND (VALID-PUBLIC-KEY(P-
key)=False))
⇒ DENY-CONNECTION() AND BROADCAST-
TREE() AND ALERT();
```

In order to protect their information exchanges, all agents have an RSA key-pair that is used to negotiate symmetric session keys before encrypting their messages. Encryption at this level guarantees that the information will not be visible at any point before reaching the top of the IP stack of the recipient party. This will deter sniffing attacks that are otherwise possible on plain text agent communications [Jansen *et al.*, 2000].

The tracing process starts when a connection needs to be monitored. Agent A at target host h_t receives local information from the system and checks for remote connections. If there is a connection from a remote machine h_r, A will contact agent B at host h_r in order to research the connection. For this purpose, a private channel will be opened using the asymmetric encryption keys of the agents. A symmetric session key will be agreed upon and all messages will be encrypted with it for that exchange. The same procedure will be followed if there are relevant connections from outside of the host of h_r coming into the server. A set of secure channels will be setup by the agents in order to share the information they have regarding connections and user activity. One hop at a time, the links composing a connection chain will be found.

The tracing information obtained by an agent is shared with the rest of the agents through *broadcast* communication (i.e., an agent sends a message to all existing agents). This guarantees that the KB's of the agents are consistent with the state of the environment and that, in the event of a security incident, the knowledge acquired by an agent will not be lost as it has been replicated and enriched by other agents.

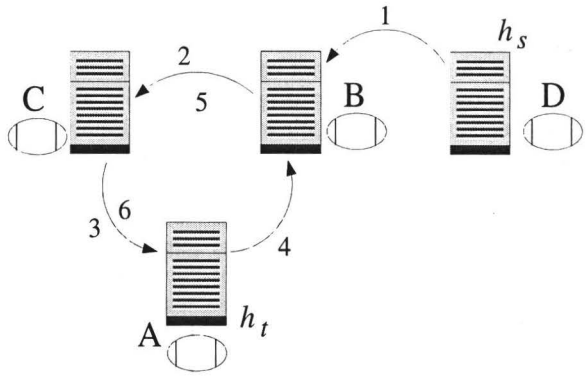

Figure 2: Loop unwinding. If a user connects from host h_s into host h_t doing a loop through hops 1, 2, 3, 4, 5 and 6 as shown above, agent A at h_t will start the tracing process and will be able to identify the loop to reduce tracing time in the future.

If we have a set H of n hosts, and a user creates a connection chain C of m hops, the system will behave well with a $O(n)$ worst case scenario. Suppose $m \gg n$. If the number q ($0 < q < m$) of connections that go from host h_x into host h_y – or vice versa – form a loop, the arrangement of the agents will find at most two relevant connections between the hosts hereby unwinding the loop. This is due to the fact that an agent is capable of identifying repeated connections toward itself by storing information on what users are connected. For the case of loops passing through k hosts, the chain will be reduced to at most $2k$ hops. This makes a worst case scenario of $O(n)$ with $n = |H|$.

Several methods exist (see [Wolf and Lam, 2000] for a reference) for loop unwinding, mostly in the compilers and discrete mathematics areas. In our model, we utilize a simple technique based on observed activity. An agent first stores user name and host-name information on its KB. When a tracing process is started, an agent will engage in communication with several neighbor agents in order to collect data that allow them to draw conclusions. Each time agent A receives new information, it checks its KB for loops. This is done as follows (see Figure 2). All tracing requests have an ID number w. Agent A initiates a tracing round. If while waiting for news on a particular w the agent is being asked by a cooperating agent B for information to complete tracing round w as well, the agent knows it is part of a loop since it is receiving information on a tracing round that it started (all agents, including initiators have this property). The trace is stored on its KB tree T and, if the user being traced repeats a loop that passes through the same host where agent A is located, this agent will reduce the tracing time by omitting a further requests to follow a loop.

Dynamic port-allocation protocol

Network service daemons usually work on a particular port listening for requests. Port-scan attacks try to reveal what are the services listening on the ports of a host. Once this is done, they look for vulnerabilities present on one of those services and, if they find one, they are in a position to exploit such a

flaw on the port they have previously identified. DoS attacks are started with port-scans to send a large number of request for service to a vulnerable port so that, at the end, there are so many requests pending to be answered – usually incomplete requests at protocol level – that the machine slows down and needs to be restarted. Additionally, if the service on that port does not support encryption, the traffic going through it may be observed by a network sniffer.

All network services using sockets on a port may suffer from this sort of attacks. We propose a model to minimize the likelihood of being a target. As we mentioned, all communications between agents is encrypted, so sniffing of plain text is impossible in practice. In order to deter attacks at the service port, we propose a technique that can be used in real life with radio equipment: channel switching (this is currently being implemented on the prototype). Initially, all agents listen on a well-know port p. The requests for service on that port do not allocate significant system resources according to the implementation. This basic port is used just to agree on the actual communication port q for the exchange ($q \neq p$). For this, the receiver proposes a valid port number that is sent to the other party over a secure channel. They immediately switch to that port for exchanging information. This port switching is repeated during the session within a fixed short period of time. In order to defeat this protection mechanism, a port scan will have to be run again and, by the time it succeeds, the communication may have moved to a different port. Frequency of switching and encryption this mechanism possible.

4.2 Implementation of the Model

As a proof of concept, the model was implemented on a prototype system named *ODISET* (On-line DIstributed SEssion Tracing). Every agent is a stand-alone process implemented with sockets for providing client and server functionality. It can request information from others and it can share information as well. A 1024-bit RSA key-pair $\{K_{Pub}, K_{Priv}\}$ is generated before the agent is launched. Once the agent is started up, it is ready to collaborate in the tracing of a session. When contacting a peer agent, it exchanges public keys and verifies the signature on the received key. The certification must come from the ODISET master key whose public key is accessible to all agents. In order to speed up message encryption, and given that encryption with public-key methods is much slower than the one using private-key algorithms [Schneier, 2001], the agent that is to send a message does the following: 1) it generates a session key K_{DES} for symmetric encryption, 2) it then encrypts the message M using K_{DES}, 3) it encrypts K_{DES} using the public key of the recipient, and, finally, 4) it sends the encrypted session key and the encrypted message to their destination. The implementation of the RSA and DES algorithms is based on version 2.0 of the RSAREF™cryptographic library by RSA Laboratories.

The first release of the ODISET prototype was developed on a RedHat Linux 8.0 box. Although agent templates can be easily obtained from multiple agent-generator systems and for different compilers and interpreters, they typically put unnecessary functionality into the agent that produces heavy agents. For this reason, and with the intention of providing the best performance possible, the implementation was made in C by minimizing the number of libraries to include. Considering that programs and data files may be damaged after a security incident and that a security tool can not rely on them, our agents do not use the output of programs such as `w`, `who` and `ps` to read system tables like `utmp` and `wtmp`. They incorporate this functionality into its own body, which gives them extended independence. The size of each agent is around 20 Kbytes to which we add two encryption keys that need to be uploaded at certain point (around 4 additional Kbytes) as well as the knowledge and rule bases that are never uploaded entirely into memory.

The knowledge base of each agent includes a tree structure T where all connection chains are stored. Whenever a host has received all the information regarding a particular session, it stores the connection chain $c = [h_1 \rightarrow h_2 \rightarrow ... \rightarrow h_n]$ into T and sends it to all available agents. Every agent reads c and finds the hosts in c that match its own tree. It then creates new branches to keep its KB up-to-date. The rule base is encoded along with the body of the agent. Being an static structure, it is not necessary to keep it as a separate entity.

5 Experimental Results

In order to evaluate the efficiency of the method, multiple experiments were prepared. After some implementation corrections, all of them were successful after the unwinding technique was incorporated. The testing facility is a set of seven Linux machines running kernels 2.1 and above on Redhat 7.2, RedHat 7.3, RedHat 8.0 and Mandrake 8.0 operating system installations. The general structure of the experiments consists of creating connection chains of length $m \in \{2, 4, 8, 16, 32\}$ using n hosts where $n \in \{3, 5, 7\}$. This maximum number of machines was selected since our experience tells it is highly improbable that, for session performance purposes, an attacker uses much more step-stone hosts. All cases where there is no loop in the chain, that is, where $m = n$ are easily resolved so we prepared connections that include a series of loops along the chain. For instance, in the three-server setup, the connection goes in circles from host h_1 into h_2, from host h_2 into host h_3, from host h_3 back into host h_1 and so on for a total number of m hops. The experiments show that the algorithm effectively unwinds the loops. When an agent finds the same loop repeated several times, it will not try to solve it over and over again. Its KB contains user and connection information that allows it to conclude that a user is creating a loop.

Part of the contribution of this method is the fact that by using loop unwinding, the performance of the algorithm has an upper threshold that is, at most, linear on the number of hosts n through which the user connects. A set of n agents will be in charge of tracing the connection and even for a very long chain, an agent located on each hosts guarantees that loops will not delay the tracing process in any way.

Table 1 includes tracing numbers with and without encryption. Both columns indicate that the growth of the tracing time is below linear time with respect to the number of hops m (this makes us think this model should escalate well to large networks). The time difference from tracing with and without

Table 1: Tracing time with and without encryption (RSA key exchange + DES chaining block ciphering) with seven hosts (n=7); m corresponds to the number of connection hops.

m	Time without encryption (sec)	Time with encryption (sec)
2	0.03	0.84
4	0.10	1.13
8	0.62	3.09
16	0.69	4.20
32	0.73	3.99

encryption is significant. This only reminds us of the performance cost of using cryptographic algorithms to protect our data. It is not likely that a connection chain extends over more than thirty hosts, for instance. It follows that, even with 1024-bit keys, the use of encrypted communication is the way to go regarding agent communication for this problem.

The strengths of this model are its unwinding algorithm, the possibility of performing on-line tracing and the structure of an autonomous lightweight agent. The proposed dynamic port-allocation method can effectively deter communication sniffing and promises to be effective for the case of agent-based applications as well. A disadvantage of the system is that it works exclusively with alive connections. If, for one reason, the user disconnects the session from our host, the agents will go blind and will not be able to trace the intruder. The model, however, can be easily extended toward off-line tracing, but there are already other systems, like the ones highlighted in Section 3, that cover that case. Another improvement area is the fact that an agent will see only users that are connected to a host through an interactive session using commands that update system tables. If an intruder exploits a vulnerability and gets to spawn a shell session at the host without having to run a remote connection command, she will not be seen. That is because these connection commands like `rlogin` and `ssh` write system activity information to system tables. If none of this programs is used by an intruder, the host will not record her presence and she will be able to go unnoticed.

6 Conclusions and Future Work

This model proves the feasibility of performing on-line connection back-tracing using lightweight autonomous agents in a distributed fashion. It also proposes two security mechanisms that can be implemented on other software agents to make their communication structure more robust (these mechanisms are random port-switching and encrypted agent communication using certificates). We conclude that the adoption of a tracing system like this by multiple operating systems through a sort of *agent sand box* per host would indeed contribute to solve security incidents more rapidly.

Future work includes a) the integration of the ODISET tracing tool into an agent-based intrusion detection system, b) the development of survivability methods for agents using replication, mobility and zero-loss mechanisms, and c) the extension of the tracing space to include dial-up connections.

References

[Asaka *et al.*, 1999] M. Asaka, M. Tsuchiya, T. Onabuta, S. Okasawa, and S. Goto. Local attack detection and intrusion route tracing. *IEICE Trans. Commun. New Paradigms in Network Management*, E82-B(11), November 1999.

[Cheswick and Bellovin, 1994] W.R. Cheswick and S.M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison Wesley, 1994.

[Huhns and Singh, 1998] M.N. Huhns and M.P. Singh. Agents and multiagent systems: Themes, approaches, and challenges. *Readings in Agents*, pages 1–23, Morgan Kaufmann, San Francisco, CA 1998.

[Jansen *et al.*, 2000] W. Jansen, P. Mell, T. Karygiannis, and D. Marks. Mobile agents in intrusion detection and response. June 2000.

[Rapalus, 2002] P. Rapalus. Computer security survey 2002. Technical Report 1, Computer Security Institute, CSI, and the Federal Business of Investigations, FBI, April 2002.

[Rivest *et al.*, 1979] R.L. Rivest, A. Shjamir, and L.M. Adleman. On digital signatures and public key cryptosystems. *MIT Laboratory for Computer Science*, (MIT/LCS/TR-212, technical report), January 1979.

[Russell and Norvig, 1995] P. Russell and E. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[Schneier, 2001] B. Schneier. Managed security monitoring: Network security for the 21st century. *Computer Security Journal, 17.2*, 2001.

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

[Staniford-Chen and Heberlein, 1995] S. Staniford-Chen and L.T. Heberlein. Holding intruders accountable on the internet. IEEE Symposium on Security and Privacy, 1995.

[Stoll, 1987] C. Stoll. *The Cukoo's Egg*. Double Day, 1987.

[Wang *et al.*, 2001] X. Wang, D. Reeves, and S. Wu. Tracing-based intrusion response. *Journal of Information Warfare*, 1(1), September 2001.

[Weiss, 1999] G. Weiss. *Multiagent Systems: A modern Introduction to Distributed Artificial Intelligence*. MIT Press, 1999.

[Wolf and Lam, 2000] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. volume 2, pages 452–471, October 2000.

[Yoda and Etoh, 2001] K. Yoda and H. Etoh. Finding a connection chain for tracing intruders. 6th European Symposium on Research in Computer Security (ESORICS 2000), pp. 191–205, October 2001.