Detecting Malicious Groups of Agents

Sviatoslav Braynov
Department of Computer Science
University of Illinois at Springfield
Springfield, IL 62703

Murtuza Jadliwala
Department of Computer Science and Engineering
State University of New York at Buffalo
Buffalo, NY 14260

Abstract

In this paper, we study coordinated attacks launched by multiple malicious agents and the problem of detecting malicious groups of attackers. The paper proposes a formal method and an algorithm for detecting action interference between users. It has to be pointed out that some members of a malicious group may not necessarily perform illegal actions, for example, they can prepare and organize an attack without taking active part in the actual attack execution. In addition, members of a malicious group may not necessarily know each other. The method we propose tries to solve these problems by building a coordination graph which includes all users who, in some way or another, cooperate with each other, i.e., the maximal malicious group of cooperating users including not only the executers of the attack but also their assistants. The paper also proposes formal metrics on coordination graphs that help differentiate central from peripheral attackers.

1. INTRODUCTION

Cooperation and coordination have been subjects of continuous interest in multiagent systems for many years. A large number of architectures, protocols, algorithms, and mechanisms have been developed, allowing a group of intelligent agents to work together towards a common goal. Instead of focusing on the problem of how to make agents better cooperate, this paper addresses the problem of detecting malicious cooperation, that is, detecting a group of adversarial agents.

Coordinated attacks launched by multiple adversarial agents are usually beyond the power of a single attacker. Such attacks are normally implemented by organizations having the power and resources to train and equip a task force capable of attacking and destroying information infrastructures from both afar and on location. A recent CERT report [17] concludes that modern attack tools are rapidly evolving, and are becoming more sophisticated. Unlike earlier attacks, launched by a single attacker against a single victim, recent attacks are better coordinated and more difficult to discover.

Detecting coordinated attacks and adversarial groups is of special importance to critical infrastructure protection [1]. Because of the magnitude of the potential damage, one cannot rely on the detect-respond paradigm which is currently used by Intrusion Detection Systems (IDSs). If a critical infrastructure is taken down, the effect will be more than apparent, the damage will be devastating, and IDSs will be of little help. Instead of IDSs, one needs systems for early detection and prevention of attacks against critical infrastructures.

In a world of ubiquitous computing, we can expect a coordinated attack to include not only humans, but also intelligent software agents acting on behalf of humans, intelligent sensors, and various intelligent handheld or embedded devices acting as a team by resource sharing, task allocation, and synchronization. A recent study by Honeynet researchers [36] reveals that criminals (commonly known as carders) make use of software robots called bots to automate merchant site identification, target exploitation, card validation, and card verification. The bots were capable of remotely accessing a common database containing vulnerable target merchant websites. The robots also had access to a database of known exploits that could be used to compromise a website.

Most previous work on plan and goal recognition has assumed cooperative or neutral agents [22, 33, 7, 6, 21]. Cohen, Perrault and Allen [8] distinguish between two kinds of plan recognition: keyhole and intended. In keyhole recognition, the agents being monitored are neutral to the recognition process. For example, they may not care or may not be aware that their actions have been observed. In intended recognition, agents take actions intending to be understood. More recent work on adversarial modelling goes further, by assuming that the adversary may try to conceal some of their actions [16, 15]. The applicability of such models, however, is limited to well-defined settings in which the adversary is well known and can be recognized (the opponent

0-7803-8799-6/04/$20.00 ©2004 IEEE.

team, for example), and his top-level goals and intentions are known (to win the game, to win an auction, to defeat our team, etc.). In such settings, the adversary does not try to hide or change his identity, the conflict is visible, and the incentives are more or less known. Examples include robocup soccer, games playing, or military simulations.

Many real-life problems, such as computer security, physical security, information warfare, forensics research, and antiterrorism, are much more complex. A computer hacker, for example, tries to conceal both his identity and his actions. He could be either an unauthorized outsider, or a legitimate insider who has worked several years for a company. A hacker can spoof his identity and impersonate another individual as in the notorious Mitnick attack [30]. The hacker's motives are not always identifiable. An attacker may steal a database of credit cards for an economic profit, a firm may launch a DoS attack against a competitor's website, and a punk may deface a website for personal satisfaction, or for the sake of proving his hacking skills. Examples of using malicious intelligent agents for distributed coordinated attacks are given in [3, 4, 5].

Several alert correlation methods have been proposed in the literature. Some of them [37, 40] correlate alerts based on similarity between alert attributes. Other correlation methods [12, 13] rely on a set of known attack scenarios, where some of the scenarios are dynamically learned. Correlation attack languages [14, 40, 25] have recently been proposed, in an attempt to specify relationships among attacks and detect coordinated attacks. Correlation languages use semantically rich alerts generated by an intrusion detection system in order to recognize the global state of the system. Attack graphs and trees are another class of tools used to describe logical steps and strategies behind attacks. Philips et al. [31] proposed a graph-based vulnerability model, where nodes identify system states, and arcs represent atomic attacks. Similar models have been used by Jha et al. [20, 19] and Schneier [35].

Although effective for correlating some alerts, most correlation methods cannot discover the causal relationships between alerts. As the very name of coordinated attacks suggests, the attack steps usually follow a special causal order. For example, one step can prepare or modify the result of another. It is only recently that attempts have been made to use causal correlation of alerts. Templeton and Levitt [39] proposed a method in which the postconditions of an action are used as preconditions of another action. Methods that correlate alerts based on the prerequisites and consequences of intrusions have been developed by Ning et al. [27, 29, 28], and Cuppens et al. [11, 10].

The problem we raise in this paper is the following. Given a security mechanism that detects a single malicious action, how can one identify all users which directly or indirectly contribute to a joint malicious activity? In other words, we want to identify all members of a malicious group and their actions.

To illustrate the problem, think of the following example. Consider a large crowd (users, processes, hosts, threads) with a few attackers hiding inside and executing a coordinated attack plan. Due to the large crowd, it is practically impossible to discern individual attackers, many of which could be performing seemingly innocuous or perfectly legitimate actions. Once the attack succeeds it is clear who the attackers are, but at this point it is too late to make a difference.

Another scenario includes a very long sequence of legitimate actions performed by different users preparing for a very short attack consisting of one or two atomic actions. The long sequence by itself may not constitute an attack signature, because it might be applied in perfectly legal alternative scenarios.

One problem with current intrusion detection systems is that they detect only the immediate executers of an attack and not their assistants, i.e. the agents who prepared the attack by taking perfectly legal actions. Another problem is that the causal relationships between actions in a single attacker scenario differ significantly from the causal links between several attackers. For example, cooperation and correlation allows a group of attackers to perform actions which are beyond the power or capabilities of a single attacker. Moreover, in Section 3, we will show that a group of attackers can perfectly cooperate without showing any correlation between single attackers' actions. For example, a malicious group can divide a large task into a set of independent subtasks so that each subtask does not correlate with the other tasks.

The method we propose tries to solve these problems by building a cooperation graph which includes all users who, in some way or another, cooperate with each other, i.e. the maximal malicious group of cooperating users including not only the executers of the attack but also their assistants.

The paper proposes a formal method and an algorithm for detecting action interference between users. It has to be pointed out that some members of a malicious group may not necessarily perform illegal actions, for example they can prepare and organize an attack without taking active part in the actual attack execution. In addition, members of a malicious group may not necessarily know each other.

The paper is organized as follows. Section 2 introduces a formal model of distributed monitoring. Section 3 discusses different types of malicious cooperation and their detection. A formal model of coordination detection is proposed in Section 4. Section 5 describes a detection algorithm and its implementation. Finally, Section 6 completes the paper with an analysis of coordination graphs.

2. DISTRIBUTED MONITORING

2.1. Formal setting

A distributed system consists of a set of agents (including both human and software) and a set of passive objects. The agents produce events by performing actions on objects and other agents. All actions in a system can be totally ordered [24], and the history of the system is represented as a sequence of actions called *system trace*:

$$t = t_1, t_2, t_3, ..., t_{m-1}, t_m, ...$$

The starting and the finishing time of each action t_m are denoted respectively by $s(t_m)$ and $f(t_m)$.The system trace is totally ordered:

$$s(t_k) < s(t_m) \qquad \text{for all} \qquad k < m$$

An action could be either individual or collective and it moves the system from one state to another. The system state is the collection of all volatile, permanent, and semi-permanent data of the system at a specific time [32]. Let $S = \{S_1, S_2...\}$ denote the set of system states, $U = \{u_1, u_2, ..., u_n\}$ stand for a finite set of agents, and A_i be the set of actions available to agent u_i.

The system state is described with a complete set of ground literals. The joint action space is $A = A_1 \times A_2 \times ... \times A_n$. That is, each joint action $\bar{a} = (a_1, a_2, ..., a_n), a_i \in A_i$, is a combination of individual actions performed simultaneously by each of the agents. Throughout this paper, we assume that an agent can perform at most one action at a time. Some of the actions could be ϵ, a null or no-op action. We assume that $\epsilon \in A_i$ for $i = 1, ..., n$. In other words, some agents could be idle at some moments.

In the paper, we follow the STRIPS action representation augmented with a concurrent action list to handle concurrent actions [2]. Each action, individual or joint, is described by a set of preconditions and a set of postconditions. Figure 1 shows a generic action schema representing a class of actions. When an action schema is instantiated, all variables must be bound to constants. That is, an action is a fully instantiated action schema.

Given a state s, a joint action $\bar{a} = (a_1, a_2, ..., a_n)$ can be executed iff the preconditions of all elements of $\bar{a}$ are satisfied in s. The resulting state is obtained by taking the union of the postconditions of each of the elements of $\bar{a}$ and applying it to s. This implies that the pre- and postconditions of the elements of $\bar{a}$ are jointly satisfiable. The concurrent action list is a list of action schemata and negated action schemata, specifying which actions must be simultaneously executed or not executed for a given action to have its intended effect.

In the paper, we restrict our attention only to actions with fully observable effects. That is, whenever an action occurs, it is clear what the action is. We use $agents(\bar{a})$ to denote the agents involved in a joint action $\bar{a}$, i.e, the agents performing non-epsilon actions.

action <first-order predicate>
  :parameters
    <list-of-free-variables>
  :preconditions
    <conjuctive-list-of-predicates>
  :concurrent <conjuctive-list-of-action-names>
  :postconditions <conjuctive-list-of-predicates>

Figure 1. Action schema

The sequence of actions, including epsilon actions, performed by agent u_k is called the agent u_k trace:

$$t^k = a_1^k, a_2^k, ..., a_m^k, ...$$

Apparently, each agent trace is a subsequence of the system trace. In the paper, we address the question of monitoring a system of agents, called the target, for malicious cooperation. The target could involve both human and software agents, the threads of the software agents, the hosts on which agents execute, etc. Monitoring an active subject means analyzing the system trace of the subject.

Definition 1 ***(Filter of traces)*** *The function $F[P]$ maps a trace t to a subtrace t' that satisfies the predicate P.*

For example, if $P = \{agent(a_k) = u_1\}$, then $F[P](t)$ is the agent u_1 trace. Our definition of filter is a generalization of the definition provided by Ko et al. [23]. They define the predicate P on the set of a single action attributes, whereas in our definition, P is defined on the overall sequence of actions. The major advantage of our definition is that it allows us to filter traces based on inter-action correlation which was not possible with the previous definition.

One of the goals of this paper is to define a set of predicates that can be used for detecting correlation among actions in the system trace. In other words, for a given correlation predicate P, the filter $F[P]$ returns all subtraces that satisfy the correlation pattern defined by the predicate.

3. TYPES OF COOPERATION AND THEIR DETECTION

Our detection model is based on the idea that in order to identify a malicious group, one should be able to detect links, relationships, and cooperation between the members of the group. In general, there are two main reasons to cooperate:

Reason A: Cooperation allows attackers to perform actions which are beyond the power or capabilities of a

single attacker. That is, cooperation allows a group of attackers acting together to achieve what single attackers cannot achieve by acting alone.

Reason B: Even when an attacker is capable of executing his tasks alone, cooperation allows attackers to achieve their goals better, i.e. it could decrease costs, improve quality, increase speed, etc.

In general, one could identify two types of cooperation:

Cooperation through action correlation in which agents' actions interfere with one another. A particular type of interference occurs when an agent can perform an action that enables a future action to be performed by another agent who is otherwise incapable of enabling it. For example, an insider could start a XWindows server which can be used by an outsider to start a XWindows attack [9]. The XWindows exploit is based on a XTest protocol vulnerability that allows an outside agent to create a one-way tunnel into a network from the outside, and gain control over the network. In this example, the insider enables the outsider to exploit a vulnerability.

Cooperation through task correlation in which agents actions do not interfere with one another. Instead, cooperation is achieved by dividing a large task into a set of independent subtasks:

$$T_1 \wedge T_2 \wedge ... \wedge T_k \rightarrow T$$

In this case, the execution of tasks $T_1, T_2, ..., T_k$ implies task T. Apparently, each subtask can be assigned to a different attacker who can execute it independently of other attackers. As an example, consider parallel port scanning in which, in order to avoid detection, a large set of ports is divided into small subsets, each subset being assigned to a different attacker.

In this paper, we study cooperation of type A, and detection through action correlation. There are several reasons for constraining our framework. First, the problem is extremely difficult to be generally approached, and there is no indication that a general solution exists.

Second, cooperation through task correlation is difficult to discover. It is often the case that sensor data is insufficient to find correlation between agents' tasks. Detecting task correlation requires information about agents' goals and intentions. Agents' intentions are, in general, not directly observable and a system trace could be intentionally ambiguous. The problem is further complicated by the presence of a strategic adversary who is aware that he has been monitored. To illustrate the complexity of the problem, consider an agent who has executed task T_1 and task T_2. Without additional information, it is impossible to differentiate between the following cases:

- The agent's intention was to execute both T_1 and T_2.
- The agent intended to execute only T_1. Task T_2 was executed as a "noise" with the intention of confusing the detection system.
- T_2 was intended and T_1 was "noise".
- Both T_1 and T_2 were "noise".

An additional problem is that we do not know when an attacker has achieved his goal. For example, an attacker may compromise a host for the sole purpose of using it as a platform for a further attack.

Third, type B cooperation only improves efficiency, without extending agents' capabilities. Apparently, in order to detect such cooperation, one needs a clear understanding of agents' incentives, benefits, and criteria of efficiency. As yet, there are no formal models of hackers' incentives in computer and network security.

In cooperation through action correlation, agents' actions interfere with one another. By interference we mean the fact that an action can affect the outcome of another action. In other words, the intended effect of a single action may depend on other action(s) taken previously or concurrently with the given action. An action performed by a single agent or a group of agents could modify the outcome of an action performed by another group, thereby invalidating the outcome, or improving it.

In general, we have two types of interference: positive and negative. Positive interference occurs when one action enables another action, or improves its results. Negative interference takes place when an action invalidates the result of another action or merely disables it. Two dimensions of interference can be identified: interference between the actions of the same agent, and interference between actions of different agents.

We view cooperation between attackers as an interference of their actions. The main objective of cooperation is to avoid negative interference and take advantage of positive interference between attackers' actions. This observation serves as a starting point for cooperation detection. To detect cooperation between attackers we look for patterns of forward enabling.

Definition 2 *In **forward enabling**, a group of agents G brings about the preconditions of an action to be performed later by agent u_i who is not capable of bringing about the preconditions by himself.*

The idea behind forward enabling is that an agent's actions are goal-oriented and form a sequence in which former actions prepare later actions. If an action is beyond the capabilities of a single agent, then the agent could ask another agent or group of agents to take the action.

4. A FORMAL MODEL OF FORWARD ENABLING

Let $precond(a)$ denote the preconditions of action a. In general, $precond(a)$ is a conjunction of predicates:

$$precond(a) = p_1 \wedge p_2 \wedge ... \wedge p_k$$

Agent u_i is capable of performing action a^i_m at moment $s(a^i_m)$ if all preconditions evaluate to true at $s(a^i_m)$. Let agent i's trace be:

$$a^i = a^i_1, a^i_2, ..., a^i_m, ...$$

Apparently, if agent u_i intends to execute action a^i_m at moment $s(a^i_m)$, the agent must take preliminary actions to bring about $precond(a^i_m)$. That is, every ground literal in $precond(a^i_m)$ is a postcondition of some action previously executed by agent u_i:

$$\forall \text{ m} \quad precond(a^i_m) \subseteq \bigcup_{f(a^i_k)<s(a^i_m)} postcond(a^i_k) \qquad (1)$$

where $f(a^i_k)$ is the finishing time of action a^i_k, and $s(a^i_m)$ is the starting time of action a^i_m. **Forward enabling could be detected when agent u_i attempts to perform a^i_m without preparing for all preconditions in $precond(a^i_m)$.** Let

$$precond(a^i_m) = precond^+(a^i_m) \wedge precond^-(a^i_m)$$

where $precond^+(a^i_m)$ are the preconditions brought about by agent u_i, and $precond^-(a^i_m)$ are the remaining preconditions. That is:

$$precond^-(a^i_m) \bigcap (\bigcup_{f(a^i_k)<s(a^i_m)} postcond(a^i_k)) = \varnothing$$

In this case, some other agents must have helped agent u_i by bringing about all remaining preconditions $precond^-(a^i_m)$. To ensure that this is a case of cooperation, we require that agent u_i is not capable of bringing about $precond^-(a^i_m)$. That is, for every precondition $p' \in precond^-(a^i_m)$ there is no action a' in the action set A_i of agent u_i that brings about p':

$$\forall p' \in precond^-(a^i_m) \quad \nexists a' \in A_i.(p' \in postcond(a'))$$

To find the agents who have helped agent i, for every precondition $p' \in precond^-(a^i_m)$, we look for agents who have brought about p' as a postcondition of one of their actions taken prior to action a^i_m. That is, agent u_j helped agent u_i, if u_j brought about some precondition $p' \in precond^-(a^i_m)$. More formally, agent u_j helped agent u_i, if the precondition $p' \in precond^-(a^i_m)$ is logically derived from postconditions $p_1, p_2, ..., p_n$:

$$p_1, p_2, ..., p_n \rightarrow p'$$

and for some $p_k \in \{p_1, p_2, ..., p_n\}$ there exists an element t_k of the system trace

$$t_1, t_2, ..., t_k, ...t_m, ...$$

such that:

$$(f(t_k) < s(a^i_m)) \wedge (p_k \in postcond(t_k)) \wedge (u_j \in agents(t_k))$$

This observation helps us detect both the helper (agent u_j) and the moment when the help was rendered (moment $f(t_k)$). If action t_k is a joint action , then several agents have helped agent u_i.

With some abuse of notation, we write $p' \in postcond(t_k)$ to mean that p' can be derived from $postcond(t_k)$. Space limitations do not allow us to elaborate on the details of matching preconditions with postconditions. We assume that there is an underlying reasoning mechanism that: (i) derives all logical consequences of a postcondition, (ii) unifies a postcondition with a precondition through a most general unifier.

It has to be pointed out that, if some agent u_j has helped agent u_i, this is not always strong evidence of cooperation. For example, it could be the case that agents u_j and u_i have similar tasks, and agent u_j, by executing his task, unintentionally and accidentally helped agent u_i.

To rule out such cases we look at agent u_j's trace. If agent u_j's action t_k, taken to help agent u_i, is irrelevant to agent u_j, then apparently the sole purpose of this action is to help. More formally, suppose that the help was rendered by agent u_j by performing action t_k. The action t_k is irrelevant for agent u_j if it does not prepare any future action of agent u_j:

$$\nexists t_o \quad (s(t_o) > f(t_k)) \wedge (u_j \in agents(t_o)) \wedge$$

$$(precond(t_o) \cap postcond(t_k) \neq \varnothing)$$

5. DETECTION ALGORITHM AND ITS IMPLEMENTATION

To illustrate the algorithm, we first describe a simple coordinated attack that we have designed and implemented in RedHat Linux 6.2.

5.1. Coordinated Attack Using the Restore Exploit in RedHat Linux 6.2

RedHat Linux 6.2 contains a utility for backup/restore called *restore* (version 0.4b15-1). The problem with *restore* is that it can execute an external program with *suid* privileges. *Restore* is used in our exploit by running it in an interactive mode (*/sbin/restore -i*) [34]. In this mode *restore* provides a shell interface that allows a user to move around the directory tree selecting files to be restored. To determine

which shell to execute (*rsh, ssh* etc.), *restore* uses an environmental variable *RSH*. In our exploit, an attacker sets *RSH=/tmp/execute_me*, thereby forcing restore to run a non-privileged program *execute_me* with suid privileges.

The exploit we designed works as follows. The first step includes creating a fake shell (C code which sets effective uid and gid to 0, i.e. root, and then executes the shell */bin/sh*) and compiling it somewhere on the system (in our case in */tmp*). Let the source file of the fake shell and the executable be called *cool.c* and *cool*, respectively. To get root shell access to the system, *cool* should be owned by root and world executable. The change of ownership can either be done by root himself or by a program or system utility like *restore* which runs external commands with setuid (root) privilege. If *restore* is forced to execute a carefully crafted ownership and permission change script (*/tmp/execute_me*) when it is setuid, the desired change in ownership of */tmp/cool* will be accomplished.

In our scenario, a group of three attackers try to exploit the vulnerability in *restore*. The first user is an Insider with access to most of the programming tools (like compilers and linkers). There are two other attackers, Guest1 and Guest2, which have guest accounts with very limited access and utilities at their disposal. Every attacker for some reason or another may not be able or may not be willing to execute the entire exploit by himself. The insider might not be willing to execute the exploit for reasons of avoiding being caught. Likewise, the guest users cannot exploit the entire vulnerability because they cannot compile */tmp/cool.c* to create */tmp/cool*. Therefore, the users need to coordinate in some way to perform the exploit. The basic idea for coordination is that the insider performs all the preparation for the exploit, leaving the actual attack to Guest1 and Guest2. In other words, the Insider writes code for *cool.c* and *execute_me*, and compiles *cool.c* to produce the fake shell *cool*. Having set the environment variable *RSH*, Guest1 is waiting for the Insider to produce *cool*. As soon as *cool* is produced Guest1 executes *restore* by firing the command */sbin/restore -i*. This forces *restore* to execute *execute_me*. After that Guest1 waits for 4 seconds to allow restore to change the permission of */tmp/cool* (the fake shell), making it root owned and world executable. Guest1 exits the system as soon as the permissions of the file */tmp/cool* have been changed. The interesting part is that the Insider and Guest1 have performed perfectly legal operations; i.e. they have not tried to overflow any buffers, to access anything they are not supposed to, or to elevate privileges. No harm was done on the system. The attack has been prepared and now it is ready to be launched.

After the preparation of the attack, Guest2 observes the exit of Guest1, and concludes that the preparation has finished. Guest2 now executes the command */tmp/cool* to get the root shell. The problem is that most IDSs will capture only the last step of the attack where Guest2 executes *cool*. After several experiments with the Linux Security Module (LSM) [26] and Snare [18], we found that they detect only the final step of the attack, thereby leaving most of the attackers (the Insider and Guest1) and their preparatory actions undetected.

5.2. Coordination Detection Tool (CDT)

We have built the Coordination Detection Tool (CDT) which includes an auditing tool, a rule based Intrusion detection system, and a system for detecting and identifying malicious groups and malicious cooperation. CDT has a forward enabling coordination detection algorithm and a backward changing algorithm built into it for detecting malicious cooperation.

CDT is built on top of the Linux Security Module for Linux (LSM) [26]. LSM monitors the execution of system calls in the Linux kernel and builds corresponding logs. LSM also provides C2-style (Sun's Basic Security Module [38]) log records for Linux systems. However, the auditing of LSM is very awkward, producing huge irrelevant logs generated by even simple user actions.

CDT uses the logging facility provided by LSM. CDT tracks each and every request sent to the kernel in the form of a system call and logs it, if the call falls in the class of events CDT is configured to log. CDT has its own auditing tool which allows for purging unwanted records from the log file, and aggregating low-level records into more abstract records. This reduces significantly the volume of data, thereby allowing all detection algorithms to work on selectively chosen and small datasets. For example, CDT can be configured to track specific users, system calls, or relationships between system calls.

The *restore* exploit executed on a system running LSM produced almost 1600 records for the duration of the exploit, which lasted for less than one minute. The CDT tool filtered all irrelevant logs, thereby reducing the number of records from 1600 to 250.

5.3. Detecting forward enabling

In this section, we describe an algorithm for detecting forward enabling. Because of space limitations, the algorithm is only schematically sketched using pseudocode in Figure 2.

The algorithm uses a data structure called a postcondition list. The postcondition list is an ordered sequence of quintuples:

$$< postcondition\_name, postcondition\_type, resourse,$$

$$created\_by, used\_by >$$

for $every\ p \in PostconditionTable$
do if $created\_by \subset used\_by$
then for $every\ agent\ u_1\ such\ that$
$(u_1 \in created\_by) \wedge (u_1 \notin used\_by)$
do for $every\ agent\ u_2\ such\ that$
$(u_2 \notin created\_by) \wedge (u_2 \in used\_by)$
do if $p \notin AccessControlList[u_2]$
then if $Helps(u_1, u_2)$ *does not exist*
then $create\ Helps(u_1, u_2)$
else $counter(Helps(u_1, u_2)) = counter(Helps(u_1, u_2)) + 1$

Figure 2. Detecting forward enabling

The postcondition list represents postconditions of all actions in the order of their occurrence.

The postcondition list produced by CDT for the *restore* exploit is shown in Table 1. Every postcondition belongs to a certain class, such as *File_create*, *File_Change_permission*, etc., shown in the second column of the table. In addition, every postcondition affects a system resource or a group of resources by changing their current status (the status of a resource is described by a list of features and their values). The resources affected by each postcondition are shown in the third column of Table 1. Every postcondition is also associated with two sets of users. The set *created_by* includes the users that directly produced the postcondition, while the set *used_by* includes the users that directly used it as a precondition for a future action. To avoid regress, if a postcondition is a logical consequence of several other postconditions, only the agents from the last stage of the inference appear as creators of the precondition. That is, every postcondition is associated only with its immediate creators.

After looking at the precondition list in Table 1, the algorithm discovers several cases of cooperation. First, postcondition $p7$ is produced by the Insider and used by another user Guest2. The postcondition is brought about as a result of compiling *cool.c* to produce *cool*. The point is that the Insider never uses *cool* again. Instead, *cool* is needed and executed by another user Guest2. In search for stronger evidence, the detection algorithm checks whether Guest2 is capable of producing *cool* by himself. The algorithm looks up the linker *ld* and compiler *gcc* in the access control list and finds that they are accessible only to root and Insider. Therefore Guest2 needs *cool* but cannot compile *cool.c*. To help him the Insider compiles it, and never uses it again. Obviously the sole purpose of producing *cool* is to help Guest2.

The output of the algorithm is a coordination graph. The nodes in the graph represent agents, and the arcs represent coordination. There is an arc from agent u_1 to agent u_2 iff agent u_1 helps agent u_2. The arcs are labelled with frequencies showing how many times agent u_1 helped agent u_2. The coordination graph corresponding to the *restore* exploit is shown in Figure 3.

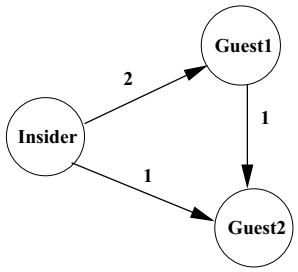

Figure 3. Coordination Graph

6. ANALYSIS OF COORDINATION GRAPHS

An attack is defined as a subgraph of a coordination graph.

Definition 3 ***An attack*** *A is a subgraph of a coordination graph G whose nodes, A_N, are the actual attack executers, and whose edges, A_E, are atomic attacks.*

The set of the **actual attack executers** consists of all agents which directly harm, break, block, or destroy a target. The attack executers achieve the final objective of a malicious group by taking actions directly on the target. In most cases, IDSs would classify these actions either as an anomaly or an attack signature.

The set of actual attack executers is expected to have one interesting property: the set is reachable from any node of the coordination graph.

Conjecture 1 *If the coordination graph does not include decoy (fake) attacks, then there is a path from every member of the malicious group to at least one attack executer.*

Conjecture 1 is based on the assumption that in a coordinated activity, attackers' actions are linked by a causal relationship in which one action prepares another. Consider a path in which attacker u_1 helps u_2, u_2 helps u_3 and so on. If the path cannot be extended beyond attacker u_n, i.e. if there are no outgoing links from u_n , then there is no further activity to be prepared, and u_n directly contributes to the attackers' goal. That is, he is one of the actual attack executers who participates in the final stage of the attack.

Conjecture 1, however, does not hold if the attackers deploy one or more decoy attacks. A decoy attack is an attack launched for the sole purpose to mislead and confuse IDSs

Name	Condition Type	Resource	Created by User(SysCall,Command)	Used by User(SysCall,Command)
p1	File_Change_permission	/dev/pts/0	guest2(AUE_CHMOD,login)	Not Found
p2	File_Create	/dev/null	insider(AUE_OPEN_WTC,bash)	Not Found
p3	File_Change_permission	/dev/pts/2	insider(AUE_CHMOD,login)	Not Found
p4	File_Create	/tmp/cool.c	insider(AUE_OPEN_WTC,sh)	Not Found
p5	File_Create	/tmp/ccYUGCj0.s	insider(AUE_OPEN_WTC,cc1)	Not Found
p6	File_Create	/tmp/ccbSeKNw.o	insider(AUE_OPEN_WTC,as)	Not Found
p7	**File_Create**	**/tmp/cool**	**insider(AUE_OPEN_WTC,ld)**	**insider(AUE_OPEN_RW,ld), guest2(AUE_EXECVE,cool)**
p8	**File_Change_permission**	**/tmp/cool**	**guest1(AUE_CHMOD,chmod)**	**guest2(AUE_EXECVE,cool)**
p9	**File_Create**	**/tmp/execute_me**	**insider(AUE_OPEN_WTC,sh)**	**guest1(AUE_EXECVE,execute_me)**
p10	File_Create	/tmp/t24603-sh	insider(AUE_OPEN_WTC,sh)	Not Found
p11	**File_Change_permission**	**/tmp/execute_me**	**insider(AUE_CHMOD,chmod)**	**guest1(AUE_EXECVE,execute_me)**

Table 1. Post Condition List

[3]. On one hand, the attack is real in the sense that it aims at a real target. On the other hand, it is not intended to be finalized. One can think of a decoy attack as an attack interrupted in the middle of its execution. Obviously, a decoy attack will produce one or more "dead ends" in the coordination graph.

A similar argument leads to the following conjecture.

Conjecture 2 *If the coordination graph does not include decoy (fake) attacks, then any node without outgoing links corresponds to an actual attack executer.*

One drawback of existing intrusion detection systems is that they can detect only the immediate executers of an attack, and not their assistants, i.e., the agents who prepare the attack. The assistants are usually the agents who organize, prepare, and make the attack possible, without taking active part in it. We call such agents **shadow agents**, because they usually perform legitimate actions that cannot be captured by current intrusion detection systems. Shadow agents present a real threat because they remain unrecognized after the attack, and can prepare and launch future attacks.

Shadow agents are those members of a malicious group who are located away from the main stream of attack, i.e, from attack subgraphs.

Definition 4 *The set of shadow agents consists of all agents s whose distance from the attack subgraph exceeds a certain domain dependent threshold d_0:*

$$\min_{a \in A} d(s, a) > d_0$$

where the minimum is taken over all members, a, of the attack subgraph, A, (the actual attack executers).

Shadow agents are the opposite of the attack executers who are directly exposed to detection and prosecution. An agent could be a shadow member for several reasons. One reason is to avoid detection. If an attack is detected and investigated, staying away from the main attack stream offers protection by making it difficult to unravel the conspiracy. Second, an agent may not be able to contribute to the attack preparation and execution due to limited resources, tools, and capabilities. In this case, staying in the "shadow" is not a deliberate choice, but the result of one's own limited ability to participate actively in the attack.

In the *restore* attack Guest1 and the Insider are shadow agents. The attack signature consists of the action performed only by Guest2. Existing intrusion detection systems, using anomaly or misuse detection, will identify the attack signature and raise an alert. The problem is that the attack signature is only part of the attack. The Insider and Guest1, who prepared the attack by taking legitimate actions, will remain undetected. The problem can be solved by finding a maximal coordination graph. The graph explains all steps of an attack, starting with its early preparation, and finishing with its execution.

Every coordination graph represents an illegal network. Illegal networks differ from legal networks in several important ways. First, participants in illegal networks must conduct their activities in secret. The need for secrecy should lead attackers to create sparse and decentralized coordination graphs. In order to avoid detection, attackers can deploy a series of "buffers" in which a single task is intentionally and artificially divided into smaller subtasks assigned to different attackers. The "buffers" decrease the amount of direct involvement in an attack by replacing a direct coordination link with a chain of indirect links. Second, attackers

face a difficult dilemma if they want to design a coordination graph that maximizes both concealment and efficiency. The problem is that a malicious group of attackers has a task to accomplish, and these tasks must be performed efficiently.

Coordination graphs can be used in off-line automated forensics analysis, for discovering attackers' traces, and in on-line intrusion detection for neutralizing shadow agents. In automated forensic analysis, the whole postcondition list is available, and the algorithm produces a complete coordination graph. In on-line intrusion detection, the coordination graph is built incrementally and an instant alert could be raised as soon as suspicious cooperation is detected. Another advantage is that shadow agents can be discovered and neutralized on-line, i.e., as soon as the main stream of the attack is discovered.

After a potential coalition of agents has been detected, an automated response can be triggered, depending on the type of coordination pattern. The model presented in this paper allows for selective generation of responses, commensurate with the characteristics of a coordinated attack. For example, one can apply graph-theoretic measures to find the structure of a malicious group, and to identify the **central attackers** and the **peripheral attackers**. The concept of centrality reflects an attacker's position in the coordination graph. Centrality is the result of how each attacker resolves the dilemma of concealment versus coordination. A central player is actively involved in the coordination and uses less "buffers" to avoid detection. In contrast, a peripheral attacker is only indirectly involved in the preparation of the attack.

One centrality measure which can be used in intrusion response is the number of the direct coordination links for an attacker:

$$Centrality(u_i) = \sum_{k \in S} link(u_i, u_k)$$

where S is the set of all neighbors of user u_i. For example, in the coordination graph in Figure 3, the Insider has centrality 3, whereas the centrality of Guest1 is 1, and the centrality of Guest2 is 0. The fact that the Insider has the highest centrality indicates that he plays a central coordination role. A closer look at the coordination graph reveals that the Insider is indeed the most powerful user who prepared the major part of the attack.

It has to be pointed out that the notion of centrality is complimentary to the notion of the immediate attack executers. An attacker may be a central player without taking part in the actual attack execution, as is the case in the coordination graph in Figure 3. An effective intrusion detection and response requires quick localization and neutralization of both immediate executers and central attackers. In our example, the first two users that have to be neutralized are Guest2 and the Insider.

7. CONCLUSIONS

In this paper, we studied coordinated attacks and the problem of detecting malicious networks of attackers. The paper proposed a formal method and an algorithm for detecting action interference between users. The output of the algorithm is a coordination graph which includes the maximal malicious group of attackers including not only the executers of an attack but also their assistants. The paper also proposed a formal metric on coordination graphs that help differentiate central from peripheral attackers.

The methods and the algorithms proposed in the paper can be used in off-line automated forensics analysis, for discovering attackers' traces, and in on-line intrusion detection for neutralizing shadow agents and central attackers.

Because the methods proposed in the paper allow for detecting interference between perfectly legal actions, they can be used for detecting attacks at their early stages of preparation. For example, coordination graphs can show all agents and activities directly or indirectly related to suspicious users. This could be potentially useful in systems for early detection and prevention of attacks against critical infrastructures.

References

[1] National Strategy to Secure Cyberspace. The White House, February 14, 2003.

[2] C. Boutilier and R. Brafman. Partial-order planning with concurrent interacting actions. *Journal of Artificial Intelligence Research*, 14:105–136, 2001.

[3] S. Braynov. On future avenues for distributed attacks. In *Proceedings of the 2nd European Conference on Information Warfare and Security (ECIW)*, Reading, UK, 2003.

[4] S. Braynov. On detecting malicious networks. In *Proceedings of the 3nd European Conference on Information Warfare and Security (ECIW)*, London, UK, 2004.

[5] S. Braynov and M. Jadliwala. Representation and analysis of coordinated attacks. In *Formal Methods in Security Engineering: From Specifications to Code*, Washington D.C., 2003.

[6] H. Bui, S. Venkatesh, and G. West. Policy recognition in the abstract hidden Markov models. *Journal of Artificial Intelligence Research*, 17:451499, 2002.

[7] E. Charniak and R. Goldman. A bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79, 1993.

[8] P. Cohen, C. Perrault, and J. Allen. Beyond question answering. In W. Lehnert and M. Ringle, editors, *Strategies for Natural Language Processing*, pages 245–274. 1981.

[9] E. Cole. *Hackers be ware*. New Riders, 2002.

[10] F. Cuppens, F. Autrel, A. Miege, and S. Benferhat. Recognizing malicious intention in an intrusion detection process. In *Second International Conference on Hybrid Intelligent Systems (HIS 2002)*, 2002.

[11] F. Cuppens and A. Miege. Alert correlation cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 202–215, 2002.

[12] F. Cuppens and R. Ortalo. LAMBDA: A language to model a database for detection of attacks. In *the ACM Workshop on data mining for security applications*, pages 1–13, 2001.

[13] O. Dain and R. Cunningham. Fusing a heterogeneous alert stream into scenarios. In *DARPA Information Survivability Conference and Exposition*, pages 146–161, Anaheim, California, 2001.

[14] S. Eckmann, G. Vigna, and R. Kemmerer. STATL: An attack language for state-based intrusion detection, 2000.

[15] J. Eilbert, D. Carmody, D. Fu, T. Santarelli, D. Wischusen, and J. Donmoyer. Reasoning about adversarial intent in asymmetric situations. In *In Proceedings of the AAAI Fall Symposium on Intent Inference for Users, Teams, and Adversaries*, North Falmouth, Massachusetts, 2002.

[16] C. Geib and R. Goldman. Plan recognition in intrusion detection systems. In *DARPA Information Survivability Conference and Exposition (DISCEX II'01)*, pages 46–55, 2001.

[17] A. Householder, K. Houle, and C. Dougherty. Computer attack trends challenge internet security. *Security and Privacy*, 1, 2002.

[18] Intersect Alliance. Snare host based intrusion detection agent for Linux, 2004.

[19] S. Jha, O. Sheyner, and J.Wing. Two formal analysis of attack graphs. In *Computer Security Foundations Workshop (CSFW)*, pages 49–63, Nova Scotia, Canada, 2002.

[20] S. Jha and J. Wing. Survivability analysis of networked systems. In *International Conference on Software Engineering*, pages 307 – 317, Toronto, Canada, 2001.

[21] G. Kaminka, D. Pynadath, and M. Tambe. Monitoring teams by overhearing: A multi-agent plan-recognition approach. *Journal of Artificial Intelligence Research*, 17:83–135, 2002.

[22] H. Kautz and J. Allen. Generalized plan recognition. In *In Proceedings of AAAI-86*, pages 32–37, 1986.

[23] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *IEEE Symposium on Security and Privacy*, pages 175 – 187, 1987.

[24] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, pages 558–565, 1978.

[25] U. Lindqvist and P. Porras. Detecting computer and network misuse through the production-based expert system toolset (p-best). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California, 1999.

[26] Linux Security Module. Information-technology promotion agency, Japan, http://www.ipa.go.jp/stc/ida/download.html, 2003.

[27] P. Ning, Y. Cui, and D. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *The ACM Conference on Communication and Computer Security*, 2002.

[28] P. Ning, D.Xu, C. Healey, and R. Amant. Building attack scenarios through integration of complementary alert correlation methods. In *in Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*, pages 97–111, 2004.

[29] P. Ning and D. Xu. Learning attack strategies from intrusion alerts. In *The ACM Conference on Communication and Computer Security*, pages 200–209, 2003.

[30] S. Northcutt and J.Novak. *Network Intrusion Detection*. New Riders, 2002.

[31] C. Phillips and L. Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 Workshop on New Security Paradigms*, pages 71–79, 1998.

[32] P. Porras. STAT: A state transition analysis tool for intrusion detection. Master's Thesis, Computer Science Department, Univ. of California, Santa Barbara, 1992.

[33] D. Pynadath and M. Wellman. Probabilistic state-dependent grammars for plan recognition. In *Proceedings of the 16th Annual Conference on Uncertainty in Artificial Intelligence*, pages 507–514, 2000.

[34] Restore Exploit. http://www.securiteam.com/exploits/6u0061f0ak.html, 2004.

[35] B. Schneier. Attack trees: Modelling security threats. *Dr. Dobb's Journal*, December, 1999.

[36] L. Spitzner. Honeypot Technologies Inc. Personal communication, 2003.

[37] S. Staniford, J. Hoagland, and J. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10:105–136, 2002.

[38] Sun Microsystem. Sun Security and Auditing, SunSHIELD Basic Security Module, November 1993.

[39] S. Templeton and K. Levitt. A requires/provides model for computer attacks. In *New Security Paradigms Workshop*, pages 31–38, 2000.

[40] A. Valdes and K. Skinner. Probabilistic alert correlation. In *In Proceedings of RAID 2001*, pages 54–68, 2001.