Automated diagnosis for computer forensics

Christopher Elsaesser and Michael C. Tanner
Cognitive Science & Artificial Intelligence Center
The MITRE Corporation
7515 Colshire Drive, McLean, VA 22102-7508

Abstract - Upon discovery, security administrators must determine how computer system intrusions were accomplished to prevent their reoccurrence. This paper describes an automated diagnosis system designed to focus investigation on the evidence most likely to reveal a hacker's method. The system takes as input victim configuration and vulnerability information and a description of the unauthorized access gained by the attacker. With this information and templates describing hacker exploits and computer actions the system generates possible attack sequences. Because it is impossible to know everything the attacker might be aware of or have done, attack hypotheses can include assumptions where there is no apparent action to accomplish part of an attack. The hypothetical attacks are next simulated on a model of the victim network. Successful simulation indicates a feasible means of accomplishing the unauthorized access. The simulation generates representative log entries that a pattern matching subsystem compares to system records. Close matches are indicators that the associated hypothesis was the means of attack.

Key words: computer security, abduction, plan recognition, heuristic search

Problem description

Computer hackers have become so capable, and networked computer systems so riddled with vulnerabilities and exposures, that it is almost impossible for such systems to be simultaneously secure and useful. When an intrusion is noticed, a security administrator must become a detective, developing and testing theories about the intruder's *modus operandi* in order to prevent the attack from recurring.

One problem today's computer sleuth does *not* face is a lack of data. Modern systems are monitored by Intrusion Detection Systems (IDS) that routinely log suspicious activities.[1] Most installations also log many (apparently) legitimate activities such as connections to services, user commands, and so on. These logs are the main sources of evidence for the investigator.

Dealing with the volume of potentially relevant data is a problem for an investigator. A large enclave logs millions of activities each day.[2] Identifying the log entries relevant to a particular intrusion is a daunting task. Many steps of an attack are indistinguishable from authorized activity. Even when one knows what to look for, the evidence is difficult to find without a good idea about the details.

This paper describes a system designed to aid an investigator to determine how a computer intrusion was accomplished. Our approach is to hypothesize attacks that could

[1] In most cases, IDSs are blocking filters with log files. Unsuccessful attacks—true positives—and misconstrued legitimate activities—false positives—are logged. Successful, undetected intrusions—false negatives—are discovered through their side effects, not by the IDS.

[2] IDSs on MITRE's DMZ log more than three million alarms per day, not including routinely logged events that did not trigger alarms.

account for an intrusion, simulate the attacks to verify that they could succeed on the target systems, and match the simulated side effects of apparently viable attacks to log files.

The following sections describe our approach and status of an experimental prototype. We begin by listing our assumptions. Next, we give an overview of computer-aided diagnosis and describe how it matches the computer forensics activity. A description of our system follows. We conclude with a status of our implementation and identify areas for extension.

Assumptions

The objective of our research is a decision support system for computer intrusion diagnosis. In designing the system, we assumed:

1. An up-to-date list of an enclave's configuration (hosts, local area networks, routers, firewalls, etc.). Getting up-to-date configuration information might require running vulnerability scanning and mapping systems *after* an intrusion is discovered. In investigations that preclude doing so, our system will use whatever information is available, with concomitant effects on the validity of a diagnosis.
2. Logs of activities on the systems that were attacked were recorded during the attack.
3. The diagnosis system can examine the enclave's records for evidence the attacker executed hypothesized attacks.

We did *not* assume:

1. The intrusion started at an identifiable border, e.g., from outside the enclave's firewall.
2. Computing the transitive closure of the system's vulnerabilities and exposures is sufficient to deduce how the incursion occurred. This implies that our system must be able to hypothesize an attacker with special information such as an account password.

Overview of computer-assisted diagnosis

Diagnosis is the process of deducing the most likely mechanism that caused an observed condition. Figuring out how a hacker created an unauthorized computer account is an example of diagnosis. Diagnosis is a type of *abduction*.

Abduction is inference that begins with data describing some state and produces an explanation of the data [Josephson & Josephson, 1994]. Medical doctors perform abductive inference when they determine that a patient has a disease. Normally they mean that disease, among all diseases, best explains the patient's pains, test results, radiology findings, etc.

So what is an explanation? Explanations give causes; to explain something is to assert its cause. What do we mean by the "best" explanation? Obviously the best explanation is the true one. Without knowing the true cause of the state (which would eliminate the need for inference) one must weigh an explanation's plausibility. The plausibility of an explanation depends on how much better it is than the alternatives, how good it is independent of the alternatives, how reliable the data is, and the breadth of search for

alternatives.[3] Accepting an explanation is a decision made under risk. One must weigh the costs of being wrong, the benefits of being right, and the expected value of a conclusion made immediately compared to the expected value after delaying the decision, perhaps to gather more information for a possibly better answer [Raiffa, 1968].

The argument supporting a diagnostic conclusion is as follows:

1. Something abnormal is observed.[4]
2. There are several possible explanations for this abnormal state. [Something interesting enough to be explained will usually have a number of possible explanations.]
3. Some of the explanations are eliminated as impossible because their preconditions were not present.
4. Some of the explanations are judged to be implausible because their anticipated consequences[5] are not observed or because they cannot explain other important data. [Sometimes we have data in addition to the original problem that we want our diagnostic hypothesis to explain.] If examining the data does not find such side effects, then that hypothesis can be ruled out by *modus tollens*.
5. Of the remaining plausible explanations, the best is the diagnostic conclusion.

Our Approach

Follows the steps of abduction, here we describe our approach to computer forensic investigation:

1) Observe abnormality: Abnormalities include altered or unreadable files, inability of authorized users to use the system, etc. Explaining such an abnormality is the diagnostic problem.

2) Generate possible explanations: We use a planning system to generate sequences of actions that could have caused the abnormality. The abnormality, represented as a goal state, is passed to the planner along with a description of the state of the system presumed to hold before the attack. Any plans generated are potential explanations.

3) Eliminate impossible explanations: The planning system is first limited to generating plans for which the situation includes their necessary preconditions. For example, an action might require access to a modem. If no modem is connected to the network, no explanation that includes that action will be generated. It is possible that in some situations no explanation can be generated. In such cases, the planning system is able to hypothesize explanations that include assumptions.[6] When it does, it is up to the user to determine when to eliminate an explanation based on an assumption.

4) Eliminate implausible explanations: Executing a diagnostic plan on the victim system generally is not feasible. Criminal investigators often proscribe tampering with a

[3] An alternative way to determine the "best" explanation is Occam's Razor: the best one is the "simplest" explanation—perhaps the explanation with the fewest clauses and logical connectives.
[4] The reason for diagnosing abnormal values is that one may presume that something out of the ordinary had to have happened for the abnormality to come about.
[5] Possibly, side effects.
[6] For example, security experts know that just because they are told there are no modems on a system does not mean that is the case. This example might be a good candidate for an assumption.

compromised system. More prosaically, the current state of the system often will not be the same as when the attack occurred. For example, if the hacker created a new account, then that account was not present when the attack began. To address such situations, we test hypothesized attacks in a simulator. Simulation can eliminate plans that are not feasible as well as generate side effects a planner is unable to foresee [Shoham, 1988]. To prune the possibilities further, we compare simulation traces to records from the compromised system to find evidence of the events and their side effects.

5) Rate the remainder: Abduction often produces more than one plausible explanation. For example, in computer forensics we cannot rule out plans that fail to exactly match the logs, because logs are usually not complete. In such cases we rate the explanations to focus the investigator on the most likely ones. If the attack involved actions that the planner knows, then the true cause will be among the explanatory hypotheses, because our hypothesis generation technique considers all possibilities.

A system for diagnosis of computer intrusions

This section describes a prototype computer intrusion diagnosis system. The prototype consists of: (1) an interface that converts data from the target enclave to a representation suitable for attack planning and simulation, (2) a planning system that generates possible explanatory plans from an observed symptom of an intrusion, (3) a simulator that can execute the attack plans on a model of the victim system, and (4) a pattern matching system that searches system log files for actions and side-effects of a successfully simulated attack.

Representing the domain

Domain representation is, of course, the key aspect of any reasoning system. In our case, the representation language and the elements of the domain that are represented must be chosen so that the hypothesis generation system and the simulation agree.

Our planning system is general purpose— it has no domain-specific object classes. Instead, it creates domain classes and state descriptions from a user's description of the domain of interest. The language it uses is the Planning Domain Description Language (PDDL) [Gallab, 1998]. We found this language to be as suited for defining a simulation as for planning, and use it for both subsystems.

Before we describe a problem situation, we describe the domain for the planner based on object classes and propositions from the simulation. That is, the planner's object classes are those implemented in the simulation (Figure 7) and the propositions describe states in terms of relations implemented in the simulation. Figure 1 is a representative fragment of such a description.

```
(define (domain network)
  (:extends computer)[7]
  (:types network domain – simulation-object
          router – host
          firewall - router)
  (:predicates
   (connected-to ?node - object
                 ?net – simulation-object)
   (part-of ?net - network ?domain - domain)
     …)
```

Figure 1: Example domain description

The :types clause of a domain definition describes an object class hierarchy. The domain description also defines a list of :predicates that define the state representation. These predicates are used for situation, action, and problem descriptions. Action descriptions are discussed in some detail below.

Initialization

The first step of a diagnosis is to collect facts about the situation. Situation descriptions are input to both the hypothesis generation system and the simulation. Our system uses off the shelf tools to create a description of a victim system's configuration and state in PDDL. If Figure 2 were the victim system, Figure 3 would be its PDDL description.

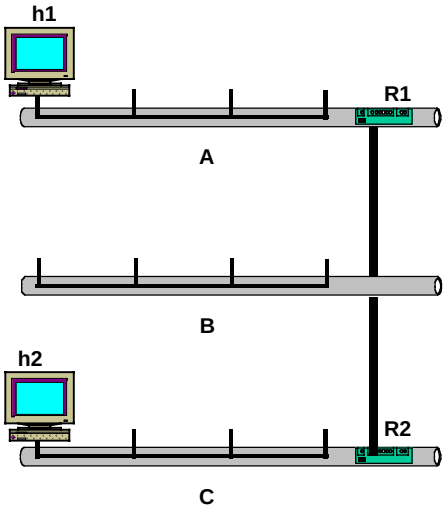

Figure 2: A sample network

```
(define (situation example)
 (:domain network)
 (:objects
  R1 R2 - Router
  h1 h2 - unix-host
  A B C - Network)
 (:init
  (connected-to h1 A)
  (connected-to h2 C)
  (connected-to R1 A)
  (connected-to R1 B)
  (connected-to R2 B)
  (connected-to R2 C)
  (offers-service h2 r-commands)
  (found + h2 /etc/hosts.equiv)))
```

Figure 3: Description of the sample network

[7] The Planning Domain Description Language permits hierarchical definition of domains with inheritence.

The ":objects" field names the parts of the enclave and specifies their class. The ":init" field is a list of predicates that describe the configuration of networks and hosts. For example, `connected-to` predicates specify which hosts (including routers) are connected to which networks. `offers-service` specifies that a host provides a network service. `found` specifies contents of files. In the example in Figure 3, the "+" wildcard in `/etc/hosts.equiv`, exposes `h2` to a well-known UNIX exploit.

Creating a situation snapshot of an arbitrary enclave is a large undertaking, and our prototype only scratches the surface. So far we include network connectivity and trust relationships in .rhosts, .xhosts, and hosts.equiv files. Shared files are also noted. Each host is initialized with a standard file system, and user and group information is used. In the future we intend to integrate several of the emerging off-the-shelf security analysis tools to fill in details.

Generate attack hypotheses

Hypothesis generation is the most important part of automated diagnosis. As we stated, we use a general purpose planning system for this task. Its main inputs are a situation description like the one in Figure 3 (with much more detail, of course), and a problem description such as the one in Figure 4.

```
(define (problem login-as-guest)
  (:domain network)
  (:situation example)
  (:goal (has-shell hacker admin.stu.edu guest)))
```

Figure 4: Example problem statement

The problem is specified by the user based on an observed symptom such as unauthorized access to particular resources. The :goal clause describes the symptom as a state. Given a situation and problem, the role of hypothesis generation is to generate a sequence of actions starting in the situation that could result in the goal state.

General-purpose planners create plans from action templates. In this domain, actions usually represent computer commands (Figure 5).

```
(:action login
  :parameters (?user - user
               ?host - host
               ?uid  - account)
  :precondition
     (and (know-password ?user ?host ?uid)
          (offers-service ?host console))
  :effect (has-shell ?user ?host ?uid))
```

Figure 5: Planning template representing a computer command

Many state-based planning systems are based on the concept of "means-ends analysis." Means-ends planning proceeds by chaining backward by matching effects to subgoals, starting with the problem goal. Preconditions (means) of applicable actions are turned into subgoals that preceding actions must establish with their effects. The process

terminates when all the preconditions of actions are either established by preceding actions or are found in the input situation.

It has occurred to other that means-ends planning is suited to computer security applications [Roberts, 1995; Zerkle 1996; Ho, 1998]. But we found means-ends search difficult to control in this domain, resulting in combinatorial explosion for all but the simplest examples. This occurs because is difficult in means-ends planning to incorporate domain knowledge to guide search, such as well-known procedures that exploit particular vulnerabilities and exposures. Without such knowledge, planning is made tractable by searching for the shortest plan that accomplishes the goal. But it is sometimes the case that the shortest plan is not an accurate means of diagnosis, because many hacker engage in stealth that, while it makes their attacks take more steps, have advantages hard to qualify in a state-based representation.

We found task decomposition planning better suited to generating intrusion hypotheses. Task decomposition makes it easy to represent abstract activities such as establishing user accounts and known “hacks” that exploit exposures due to promiscuous configuration settings (Figure 6). A task decomposition template tells the planner the subgoals that must be established –and roughly in which order—to accomplish the task. Just as in means-ends planning, task decomposition planning bottoms out in executable actions, retaining a convenient representation for hypothesis verification.

```
(:action rlogin-hack
   :parameters (?local ?victim - host
                ?uid ?root - account
                ?home - unix-directory
                ?user - user)
   :precondition
      (and (has-account ?victim ?uid user ?home)
           (found ?home ?victim /etc sharetab)
           (has-account ?local ?root superuser /))
   :expansion (sequential
                (has-shell ?user ?local ?root)
                (access ?local ?victim ?home)
                (account-exists ?local ?uid)
                (has-shell ?user ?local ?uid)
                (found ?local ?victim ?home rhosts)
                (has-shell ?user ?victim ?uid))
   :effect (has-shell ?user ?victim ?uid)
   :documentation "edit ?home/rhosts when ?home
                   directory exported")
```

Figure 6: Task decomposition templates can model hacker knowledge

Retaining control of the search process enables us to address two sources of complexity in the forensics domain: uncertainty about the attacker’s knowledge state, and the need to generate many alternative plans.

We implemented in our hypothesis generation system the capability to make (selective) assumptions about the attacker. For example, we can consider the possibility that the

attacker is an insider or used "social engineering" to obtain a password by inserting an assumption as an establisher of an attack precondition. Of course assumptions must be examined for plausibility by a human investigator.

Generating multiple plans is done by expanding several alternative subplans to accomplish any given subgoal. This allows us to consider the possibility that the attacker might have used an attack that, by planning standards, is less "efficient" than an alternative. Combinatorics are controlled by the planner by reusing subplans or actions where possible (i.e., an action can fulfill subgoals in several alternative plans), and by parameterizing the number of subplans that are expanded.[8]

The output of hypothesis generation is a set of possible attacks in the form of sequences of actions and assumptions that could account for the observed intrusion. Our system produces plan sequences as lists of commands in the syntax simulator can directly evaluate. An example of the syntax is given in the following section.

An important capability of our hypothesis generation system is the ability to identify the enabling conditions for an attack. Enabling conditions are propositions from the problem situation that are required preconditions of plan actions that are necessary for the plan—that is, there are no available alternatives—and are impossible to establish by alternate means.[9] For example, an enabling condition of certain attacks is promiscuous trust of other systems within an enclave, expressed by including a wildcard in a .rhost file.[10] Knowing these enabling conditions helps the system administrator prevent future incursions. Their identification can be valuable even when a hypothesized attack cannot be proven to be the one that led to the incursion. We recommend that enabling conditions should be the first thing the investigator examines.

Simulation

Each hypothesis about how an intrusion was accomplished must be tested to see if it is feasible. Deleting those that will not work because of some detail the planner did not consider simplifies the check for evidence. Our system uses simulation to test the feasibility of attacks and to generate side effects for comparison to records from the victim system.

Why simulate when you have a planner? In short, because it is possible to simulate effects of actions that would be impossible to note in a practical plan representation [Shoham, 1988]. In any domain, actions can have contingent effects that are apparently irrelevant to planning or unknowable by a planner but can affect the executability of *possible* subsequent plan steps. There are domains, and computer attack is one of them, where ramifications of actions can indirectly affect preconditions of subsequent actions through a chain of effects on processes outside the planner's control. For example, on some systems, an attempt to copy a password file will cause an alarm that leads to the

[8] Our system can interleave planning and simulated execution, allowing us to investigate unexpanded alternatives at a later time.

[9] Assumptions also represent enabling conditions that do not appear in the problem situation. Assumptions are only made when there is no other way to establish the problem goal.

[10] Doing this also makes life much easier for users. It is particularly useful when setting up a large military command and control system on short notice.

offending user to be knocked off the system.[11] There are simply too many alternatives to account for this possibility: that there is no IDS, there is an IDS but it does not have that rule, there is an IDS with that rule but it was disabled for a particular UID on a prior visit to the machine, and so on.

If a planner had to consider all the possible ramifications of every proposed action to ensure that none of them affect later steps of the plan, the search space would be doubly exponential (for each candidate action, there are exponentially many futures to consider). If, instead, the planner only concerns itself with action effects that are locally relevant, then its problem is tractable but it could generate plans that cannot execute. The purpose of our system is to test possible plans for feasibility, not to find a single, guaranteed to execute plan.

A simulation starts from a known initial state and computes effects of actions. While it is impossible to simulate every detail, a simulation can represent more than a planner can because the simulator does not have to search the resulting state space. Therefore, simulation can be used to check if an apparently feasible plan can execute. As important in our domain, there can be apparently insignificant side effects—things that would not be modeled for the purpose of planning—which provide evidence that an action was executed.

Our simulation takes as input a situation description like the example in Figure 3, and a sequence of commands that result from parsing a plan representing an attack hypothesis.

The first thing the simulation does is create a model of the enclave described in the situation description. The simulation reads PDDL situation syntax and creates the objects specified in the description. In contrast to a general purpose planning system, our simulation is designed specifically for this domain. Therefore, each object type in a situation specification must exist in the predefined object class hierarchy shown in Figure 7.

There are several classes of particular interest here. The network class models connections between hosts. User instances represent persons who use the computer systems (hacker, system administrator, etc.). Each instance holds information about the current identity of that person on each host, the services in use, and the person's history during the session.

[11] Examples of this problem involve actors outside the control or knowledge of the planner. The classic example is the Yale Shooting Problem, which points out that it is impossible for know the effect of pulling the trigger of a gun that was loaded some time before it was picked up and pointed at a victim.

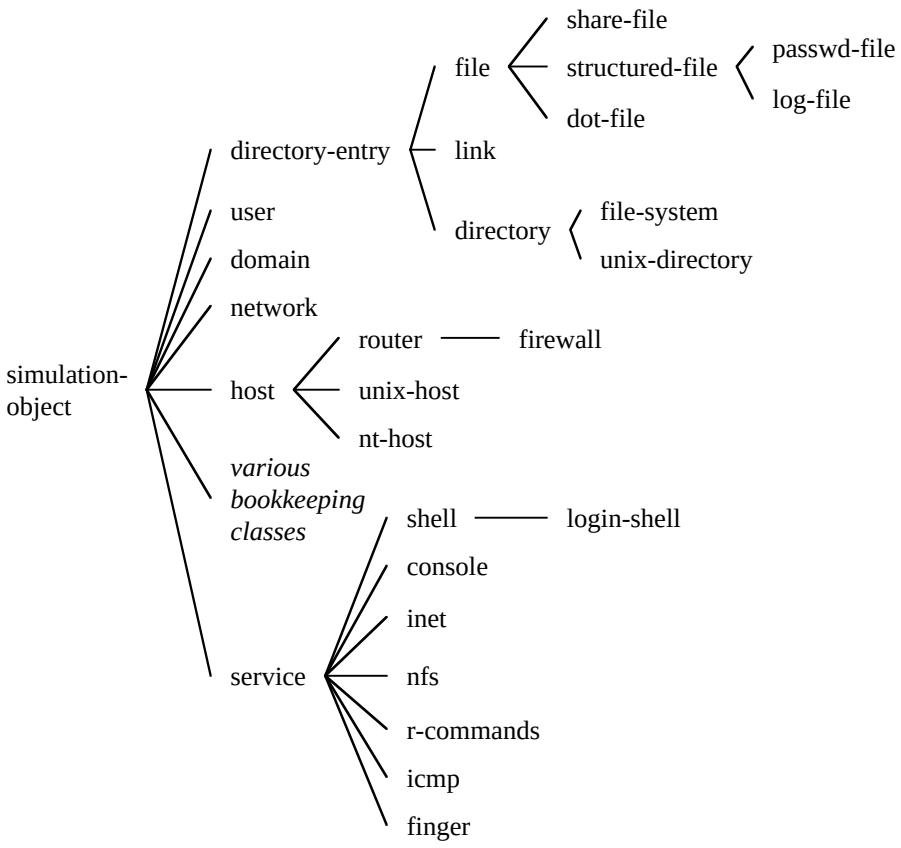

Figure 7: Simulation object class hierarchy

The main loop of the simulation interprets and executes commands on simulated host computers. "`command`" is the application program interface function that fields all simulation directives. A list of such commands is the output of the planning system and one of the inputs to the simulation.

Directives represent computer commands that are enabled by services. There are default services for some classes of host, but most must be specified in the situation description.

Suppose an attack plan calls for a user called "*hacker*" who has access to host h1 on network A to access host h2. He might first log in to h1 at a console, then rlogin to h2. The simulation directives from such a plan would be as follows:

```
( …
 (command hacker login root h1.A)
 (command hacker rlogin root h2.A)
…)
```

When the simulation interprets the "login" command, it creates a console service on the host h1, then dispatches the arguments "root" and "h1.A" to the login method. Login checks h1's /etc/passwd file[12] to determine that there is an account called root and

[12] Because h1 is an instance of a Unix machine, the method implementing login will be the one designed to simulate Unix. Other operating systems have their own versions of the login method that function in the appropriate, system-specific manner.

whether the user knows the password for that account. Password knowledge is represented by a list of passwords that each user knows. In this example, if hacker knows the password of the root account on h1, the login method will create a login-shell service and establish hacker's identity on h1 to be root.

The command interpreter logs every directive executed on each host. This includes access to the log files on the hosts, which attackers can modify. The simulation also includes a "ground truth" log that is not accessible to user directives. This log, an example of which is shown in Figure 8, is used for searching the real system logs for evidence of an attack, as described in the next section.

```
8/20/2001 14:44:14 EDT, icmp: anybody ping butterfinger.stu.edu successful.
8/20/2001 14:44:14 EDT, icmp: anybody exit successful.
8/20/2001 14:44:24 EDT, inet: anybody network-services successful.
8/20/2001 14:44:24 EDT, inet: anybody exit successful.
8/20/2001 14:44:35 EDT, finger: anybody finger guest nil successful.
8/20/2001 14:44:35 EDT, finger: anybody exit successful.
8/20/2001 14:44:46 EDT, nfs: anybody showmount -e butterfinger successful.
8/20/2001 14:44:46 EDT, nfs: anybody exit successful.
8/20/2001 14:44:59 EDT, nfs: anybody mount butterfinger.stu.edu (/ export foo) (/ foo) successful.
8/20/2001 14:44:59 EDT, nfs: anybody exit successful.
8/20/2001 14:45:51 EDT, r-commands: anybody rlogin butterfinger guest successful.
8/20/2001 14:46:03 EDT, login-shell: guest ls nil successful.
8/20/2001 14:46:14 EDT, login-shell: guest cat (important) successful.
8/20/2001 14:46:25 EDT, login-shell: guest write important trash t successful.
8/20/2001 14:46:35 EDT, login-shell: guest logout successful.
8/20/2001 14:46:35 EDT, login-shell: guest exit successful.
8/20/2001 14:46:35 EDT, r-commands: anybody exit successful.
8/20/2001 14:47:06 EDT, console: anybody login root butterfinger.stu.edu successful.
8/20/2001 14:47:18 EDT, login-shell: root cat (/ var log) successful.
```

Figure 8: Sample log generated by the simulation for one host

In addition to executing directives on simulated hosts, the simulation allows one to query the state of the simulation. There are queries to list commands available to a user, determine whether an account exists on a host, determine whether a host is listening on a port, and determine whether a file exists and has certain information in it. This capability might be used if one wishes to interleave planning with execution.

Verifying hypotheses by searching log files

The final step of diagnosis is to match simulated log entries from viable attack hypotheses to records from the victim system.

One way system administrators diagnose attacks is by running a tool such as Ethereal[13] or Review[14] on tcpdump logs to reconstruct each session and examining the resulting command-line traces to try to find the attack and how it was accomplished. One problem with this approach is that the logs can include hundreds of potentially relevant sessions.[15]

---

[13] http://www.ethereal.com/.

[14] http://www.net.ohio-state.edu/security/talks/1997-06_review_first/paper/paper.html.

[15] In general, it is infeasible to keep network traffic forever, but it is possible to keep some of it. Some sites keep several days' worth. So we assume the user or system administrator detected the problem soon enough after the attack occurred that this site still has the data.

We aim to simplify this task by automatically matching simulated logs to reconstructed sessions.

One issue to address in the log matching process is partial ordering of attack steps. In general, potential explanations generated by the planner will be partially ordered. For example, if several files are to be modified, the order in which they are changed usually does not matter. During planning it is more efficient not to commit to a particular order.

Even though the planner might be indifferent to the order of certain steps, some order must be specified for execution. This presents a design choice. If the simulation is given only one possible linear order of a plan, then the pattern matcher becomes responsible to search for all the possible orders of the resulting log entries. To do so, the pattern matcher would need to be able to reason about the ordering constraints. Likewise, if the simulation were given a partially ordered plan, it would have to be able to generate the alternative linearizations. We decided that the best alternative is making the planner send to the simulation all possible linearizations of each plan. The simulation then passes the results of successfully simulated executions to the pattern matcher. This makes pattern matching much simpler. Because some linearizations turn out not to be executable, this design also reduces the number of passes through the log files.

Intelligent pattern matching involves two aspects: exploiting serial position in the pattern being matched, and partial pattern matching.

Exploiting serial position in the example logs is straightforward. It is enabled by the assumption that the input—i.e., the output from the simulator—is in the exact order it must be for that explanation of the attack to be valid. For example, if the explanation indicates that someone first copied the shadow password file and then accessed a user's .rhosts file, then seeing the order reversed in the system logs eliminates that explanation.

Partial pattern matching is required for two reasons. First, the simulated log entries will not match the actual session log exactly, even when the hypothesis is correct. For example, if someone logged in as a trusted user—perhaps by surreptitiously obtaining a password—then the hypothesis generator may only be able to guess the UID. Second, the session including the attack might include steps that are not necessary for accomplishing the goal; these steps will not be generated by the planner.

Belief computation is required when partial pattern matching is necessary. The planning system we are using computes probabilities for each of the propositions contributing to a plan [Seligman, 2000]. This information could be used to weigh partial matches, and can be augmented with heuristics about how good each individual and sequence of matches is. We are investigating this issue and are not prepared to comment on its solution as yet.

Related Research

Other Diagnostic Methods

Artificial Intelligence researchers have built a number of abductive diagnosis systems, including the earliest expert systems [Josephson & Josephson, 1994]. Most diagnostic systems use either a heuristic or a model-based approach.

Heuristic approaches are exemplified by early rule-based expert systems such as MYCIN [Shortliffe, 1976]. In these systems, rules encode expert diagnostic knowledge. Rule-based systems were found to be plagued by maintenance difficulties. Rules interact in unexpected ways and new rules, or new clauses in existing rules, have unpredictable consequences. This made adding new knowledge very difficult. Dealing with uncertainty by associating belief measures with rules turned out to lead to incorrect conclusions [Wise, 1986], precluding an important aspect of diagnosis.

Model-based diagnosis attempts to deal with some of the problems encountered by heuristic approaches. Model-based diagnosis is based on the assumption that it is easier to understand how a system works than to understand how to diagnose faults in it. Thus, a system based on knowledge derived from more reliable sources than human experts would be able to handle novel problems. An example of model-based reasoning is GDE [deKleer, 1987]. GDE generates diagnoses that account for differences between behavior predicted by the model, representing an ideal, and observed behavior. Reiter [1987] describes a diagnostic method similar to GDE but based on first-order logic as a formalized way of doing model-based diagnosis.

The main problem with model-based diagnosis is its computational complexity. For example, Reiter's method requires generating all possible diagnoses—there are exponentially many of them—and proving the consistency of each one—a semi-decidable problem. To make this approach practical it is necessary to make a number of assumptions about the problem or the domain. One assumption is that a single fault caused the problem. Another assumption, used in GDE, called minimality, is that supersets of diagnoses are also diagnoses and therefore the minimal set can stand for all of them. However, minimality implies independence and monotonicity, i.e., that a diagnosis that includes more faults explains more symptoms. It turns out that only experts can tell whether problems in a domain arise from single faults, or whether diagnostic hypotheses are monotonic. Thus, model-based diagnosis has many of the same problems as heuristic diagnosis, transformed into new concepts. See Bylander [1991] for more discussion of these issues.

Our approach is more model-based than heuristic. We address the computational complexity of the model in two ways. First, we only use the model to test candidate explanations generated by the planner, not to generate the candidates. Second, our model is an event-driven simulation, not logic-based. Therefore, it only has to predict states that are important for rating the plausibility of the candidate explanations, not all possible states.

AI approaches to intrusion analysis

Ho, et al. [Ho, 1998] suggest combining partial order planning and executable Petri nets to generate attack signatures that allow unordered events in the action sequence. Their idea is to use a planning system to construct a Petri Net representation of intrusion scenarios and use a "searching agent" to determine whether any of the intrusions are in progress. The benefit of partially ordered scenarios is to create a more general signature that is possible with state transition based signatures. While this approach seems to be a promising advance over exhaustive enumeration of possible attack sequences, the authors do not address the matching process. As we note, giving a matcher a partially ordered

signature requires that the matching subsystem be capable of reasoning about possible linearizations. The resulting computational complexity would make real time intrusion detection impossible.

NetKuang [Zerkle, 1996] proposes to find vulnerabilities on networked computer systems created by poor system configuration. It resembles our planning step but uses a backward, goal-based search of the actual host computers to find the transitive closure of vulnerable systems. Goals are privileges—such as becoming a member of a group that would then have access to a file. Computing the transitive closure of a vulnerability or exposure does little to help discover how an intrusion was accomplished. The means-ends style of planning, added to the fact that NetKuang bases its planning on direct access of configuration information (i.e., each action must be executed on the real hosts), mean this approach could get out of control, essentially creating a worm.[16] It certainly would not be suitable for computer forensics.

Roberts [Roberts, 1995] created a plan-based simulation of malicious intruders to generate realistic audit logs that illustrate malicious behavior. This is similar to the purpose of our simulation. The "plan-based" notion comes from using means-ends analysis to assemble sequences of actions to accomplish a (malicious) goal. This simulation is geared to training systems administrators to be able to recognize patterns of malicious behavior by showing them logs of actions that represent the behavior. Our use of simulation is geared to helping them find these patterns when they are "a needle" in the "haystack" of a typical week's logs from a large enclave.

Discussion

This paper describes a way to use abstract models of computer systems to guide the search for causes of an intrusion. A planning process generates hypotheses about how an intrusion was accomplished. Abstract plan templates describe well-known hacker techniques and common system administrator actions that a hacker may use (e.g., create-new-user). Plan decomposition fills in subgoals with action templates that represent user commands.

A hypothesis generated by the planning system can include *assumptions*. This is where our approach differs from other planning-based approaches to intrusion analysis, and from most planning systems. Assumptions are necessary because it is generally impossible to have complete information about the state of the victim system before it was invaded or of the knowledge of the attacker. One important use of assumptions is to account for the possibility of insider attacks. Assumptions and the enabling conditions of hypothesized attacks should be the first thing the forensics expert should attend to.

Because it is impossible to anticipate and encode every detail of every action in this domain,[17] we use simulation to test hypotheses and generate detailed side effects. Using simulation allows us to avoid tampering with the victim system, although it will be most effective if current configuration data can be extracted to initialize the simulation. We simulate every possible linear ordering of the hypothetical intruder plan, collecting a log

[16] It would seem that to obtain access to some of the files needed to compute the transitive closure of promiscuous trust exposures, the NetKuang process would have to run at root (GUID=0) level.
[17] Or in any realistic domain [Shoham, 1988].

of the actions of each one that successfully executes. Plans that do not execute in the simulation are not viable explanations of the intrusion.

There are likely to be several viable hypotheses after simulation. The simulated logs from these are compared, via pattern matching, with sessions recreated from log files from the victim system. We are working on automated aids that should make this process much faster than human evaluation of logs.

We have a proof-of-concept implementation of the system described in this paper. Each part works on several representative attacks. Development effort remains in three areas:

1. System integration. The three parts of the system are not yet cleanly "glued together." There is also some work needed to integrate off-the-shelf tools used to collect system configuration and vulnerability information and to translate tcpdump files into session logs.
2. Pattern matching. Our pattern matcher is rudimentary. Extension is needed to account for partial matches—including belief management—and to better control repeated search of recreated sessions.
3. Knowledge engineering. The planning system needs many more templates representing attacks. One possibility of assisting this normally time consuming task is to use automatic text summarization techniques to convert descriptions such as CERT advisories into attack templates.

Acknowledgements

This research was funded by The MITRE Corporation. We thank the Chief Technical Officer, David Lehman, and the Chief Engineers for their support. We thank Dr. John Vasak for suggesting attack generation as a research objective. Dr. Todd Wittbold suggested the forensics application, helping us avoid wandering the wilderness of the information security domain.

References

Bylander, T., D. Allemang, M. C. Tanner, J. R. Josephson, "The computational complexity of abduction," *Artificial Intelligence*, 49:25–60, 1991.

de Kleer, J. and B. C. Williams, "Diagnosing multiple faults," *Artificial Intelligence*, 32(1):97–130, 1987.

Ghallab, M., A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, D. Wilkins, *PDDLæ the planning domain definition language*, Technical report, Yale University, 1998.

Ho, Y., D. Frincke, and D. Tobin, *Planning, Petri Nets, and Intrusion Detection*, Department of Computer Science, University of Idaho, Moscow, ID, 1998.

Josephson, J. R. and S. G. Josephson, *Abductive Inference*, Cambridge Univ. Press, 1994.

Raiffa, H., *Decision Analysis*, Addison-Wesley, 1968.

Reiter, R., "A theory of diagnosis from first principles," *Artificial Intelligence*, 32(1):57–96, 1987.

Roberts, C. C., *Plan-based Simulation of Malicious Intruders on a Computer System*, Naval Postgraduate School, Monterey, CA, 1995.

Seligman, L., P. Lehner, K. Smith, C. Elsaesser and D. Mattox, "Decision-Centric Information Monitoring," *Journal of Intelligent Information Systems*, Kluwer Scientific Publishers, 14(1), March 2000

Shoham, Y., *Reasoning about change*, The MIT Press, Cambridge, MA, 1988.

Shortliffe, E. H., *Computer-Based Medical Consultations: MYCIN*, Elsevier, 1976.

Wise, Ben P., *An Experimental Comparison of Uncertain Inferences Systems*, Carnegie Mellon University, Pittsburgh PA, 1986.

Zerkle, D., K. Levitt, "NetKuang—A Multi-Host Configuration Vulnerability Checker", *Proc. of the 6th USENIX Security Symposium*. San Jose, California, July 22–25, 1996, pp. 195–204.