Towards Detecting Intrusions in a Networked Environment

L. Todd Heberlein

Division of Computer Science
Report No: CSE-91-23
June 1991

Abstract

To date, current authentication and access control mechanisms have been shown to be insufficient for preventing intrusive activity in computer systems. Frequent media reports, and now our own research, have shown the widespread proliferation of intrusive behavior on the world's computer systems. With the recognition of the failure of current mechanisms to prevent intrusive activity, a number of institutions have begun to research methods to detect the intrusive activity. The majority of research elsewhere has focused on analyzing audit trails generated by operating systems. The University of California, Davis, on the other hand, has chosen to analyze the traffic on computer networks. In this thesis, I present both a method to model the traffic on the network and a method to analyze the model in order to detect intrusive activity. A prototype software package has been developed to test the model, and I discuss some of the surprising results from this study.

Chapter 1

Introduction

Computers are the targets of attacks [5]. Reports appear in the media almost weekly about outsiders breaking into computers, employees misusing computers, and rogue viruses and worms penetrating computers. Incidences such as the internet worm of 1988 [5], the Wank worm [5], and the Netherland hackers have gained international recognition, and they serve to emphasize the vulnerability of computer systems around the world.

These reported incidents are cases of intrusive behavior in our computer systems. Intrusive behavior can be defined as any attempt, successful or not, to compromise the integrity of data on a computer system, to disclose any data on the computer against the wishes of the owner of that data, or to deny use of the computer system or data by legitimate users [6]. General examples of intrusive behavior include [3]:

- *Attempted break-in*: This is an attempt to enter a computer system without authorization.
- *Masquerading* (or *successful break-in*): This occurs when someone uses an account on a computer system which belongs to someone else. This can occur if the account's password is compromised or a user leaves a terminal without logging out.
- *Penetration by legitimate user*: This occurs when a legitimate user on the system attempts to gain access to data to which he does not have expressed consent by the owner of that data.
- *Leakage by legitimate user*: This occurs when a legitimate user discloses data to a another individual without authorization to access that data.
- *Malicious code*: This includes both Trojan horses and viruses. Malicious code is code which, when executed, does more than it is advertised to do. Although the code may not have been written with malicious intent, the results of the actions taken by the code may damage the computer system.

Authentication and access control mechanisms are designed to guard against intrusive activity; however, these mechanisms have not been wholly successful. Failure of these mechanisms are due in part to the ease by which passwords can be compromised, failure by system administrators and users to properly use access control mechanisms, poorly designed operating systems, and bugs in the operating system.

The failures of authentication and access control mechanisms are compounded by the decentralization of computer systems and the increased access to a computer system by computer networks. The decentralization of computer systems is the movement away from a single mainframe computer to multiple workstations and personal computers. The movement is fueled by the increasing power and decreasing costs of workstations and personal computers. The result of decentralization is a type of computer system which is administered by people, usually the user community, with little or no formal training in system administration or computer security. This in turn results in a greater chance for poorly configured authentication and access control mechanisms.

Connecting a computer to a network also increases the chances of intrusive behavior occurring on that computer since this process increases the number of people who can potentially access it. Connecting a computer to a network provides a path to that computer for every user with access to the network. If the network is part of the internet, essentially everyone with access to a telephone has a path to that computer.

With the realization that current authentication and access control mechanisms have not provided adequate security against intrusive behavior, institutions which use computers and computer networks have become interested in detecting the intrusive activity which is occurring. If an intrusion can be detected, an institution can at least know from where intrusive activity is coming, how the activity is being perpetrated (and therefore, hopefully how to stop it), and what data have been compromised. The need to detect intrusive behavior has created a new field under the area of computer security called "intrusion detection."

1.1 Intrusion Detection

"Intrusion detection" means an attempt to detect any intrusive behavior occurring on computer systems. The first published paper in the field of intrusion detection, "An Intrusion-Detection Model," laid the groundwork for much of the future effort in the field [3]. The paper's model is based on the hypothesis that intrusive behavior would appear abnormal compared to typical behavior. The behavior of an object (e.g. users, programs, and terminals) is determined by the examination of system resources used by that object. Examples of resource usage include the number of processes started, the number of CPU cycles used, and the number of files opened. A statistical profile is created for "typical" behavior, and when a new behavior is observed which does not match the profile, it is reported as possibly intrusive. Much of the research in the field has focused on the representation of profiles and the comparison of behaviors based on those profiles [3,10,15,18,19,20,21,22,23].

Another approach used by intrusion detection systems is to use an expert system to determine if computing policy has been violated or to look for tell tale signs left by intrusive activity [18,20,21].

1.1.1 Audit Trail Analysis

The majority of intrusion detection systems use the operating system's audit trails as their source of information about system resource utilization. A typical audit record includes information such as which subject attempted to do what action to which object. Information is extracted from the audit records to record the system resource utilization for different objects such as users, terminals, and the system as a whole.

Although the audit-trail-based analysis has provided a measure of success, a number of limitations exist with this method. First, audit trails traditionally do not provide much of the information necessary to perform security analysis. This is due in part to the

historical purpose of audit trail collection - the billing of customers. Second, audit trails tend to be system specific. Each operating system provides a different set of information in a different format. An intrusion detection system designed to work on a Multics operating system's audit trails would need a great deal of restructuring to operate on another operating system. Third, the collection of audit trails is expensive in terms of CPU usage and storage utilization. Many organizations, even those working in the field of computer security, turn off auditing on their machines to avoid the resource penalty. Fourth, the audit trails themselves can be the target of an intruder. Intruders have been observed turning off auditing on machines in order to hide their tracks. Fifth, and last, the delay in the actual recording and analysis of the audit information can allow an intruder to do damage and exit the machine long before the intrusion is noticed [22]. So, although there exists a strong desire for immediate notification of intrusive activity, audit mechanisms can introduce a delay factor.

Due to these drawbacks, as well as the desire to explore other possible sources of information, the University of California, Davis embarked on an effort to build an intrusion detection system which uses network traffic as its source of information on resource utilization [10,11,12].

1.1.2 Network Traffic Analysis

By taking advantage of the broadcast property of a local area network (LAN), the analysis of network traffic can solve a number of the drawbacks associated with audit trail based analysis. First, network standards exist by which a variety of hosts can communicate. An intrusion detection system based on network traffic can therefore simultaneously monitor a number of hosts consisting of different hardware and operating system platforms. A network consisting of a number of possibly different computer systems is referred to throughout this thesis as a heterogeneous network. Second, the collection of network traffic does not create any performance degradation on the machines

being monitored, so network monitoring is more attractive to a user community which places importance in the performance and responsiveness of their machines. Third, since a network monitor can be logically isolated from the computer environment, its analysis cannot be compromised by an intruder. Typically, the intruder has absolutely no way of knowing that the network is being monitored. And fourth, since a network monitor draws its information directly from the network, no delay occurs from the time an intrusion occurs and the time the evidence is available. Instead, intrusive activity is observed as it occurs.

The original work in this type of network monitoring was based on simple traffic analysis: modelling the flow of data among the different machines [11,12]. In this work, network traffic is modelled with a concept called a data path. A data path is a method by which one machine can communicate with a second machine. A data path is defined by the three-tuple <src_host, dst_host, network_service>. If the traffic flow shifted (e.g. a new data path is observed) at any point, this information would be reported as a possible intrusion. For example, a particular host initiating a login to a host to which it has never logged into before would be considered suspicious. This work was based on Denning's hypothesis that intrusive activity would manifest itself as anomalous behavior.

Although this method showed early promise, a major drawback quickly became apparent: the information available from simple network packet analysis was at a level much too low to detect subtle intrusive activity. For example, an intrusion over a commonly used data path would not be detected. Unfortunately, this is often the case when the intrusion is being perpetrated by an insider. Furthermore, services which typically produced unpredictable activity, such as remote fingers, are not helpful in the detection of intrusive activity. Because such a high percentage of network activity by these serves are anomalous, they simply produce too many incorrect reports of intrusive activity, or false positives (see section 5.2).

The problem is not unlike the problem facing the prisoners in Plato's "The Allegory of the Cave" [17]. The prisoners are provided with very low level information about the

real world - simply the shadows of the actual activity. Although the information is at a fairly low level, patterns of light and dark on the wall, the information represents a very complex and dynamic world. The prisoners can detect major changes in the light and dark patterns on the wall (e.g. a shadow where there is almost always light) without too much difficulty; however, to have a better understanding of what is truly going on in the world, the prisoners must infer complex concepts from the shadow patterns. Plato's prisoners do not perform this abstraction, so when one of them is finally shown the actual world, he is blinded and then dumbfounded by what he sees. Only then does he realize how little he understood by simply observing the shadows.

Our monitor, like the prisoners, can only see the shadows of the real world activity. The shadows, network packets, provide some capability of determining when something is wrong (e.g. packets being exchanged between hosts which have never communicated before); however, to provide a better understanding of the networked environment being monitored, and therefore to be able to detect the more subtle intrusive behavior, the monitor must be able to abstract out the complex behavior of the networked environment from the packets. Unfortunately, unlike Plato's prisoner, our monitor cannot be unshackled to directly observe the objects creating the shadows. Instead, the objects must be inferred and understood by only observing the low level network packets.

1.2 Goal

The motivating goal behind the work presented in this thesis is to design a system capable of detecting intrusive activity in a heterogeneous network. The design is constrained by the fact that only the low-level information of packets are directly observed by the system; however, to detect subtle intrusive behavior, more complex phenomena must be inferred and analyzed. The behavior of individual network connections, individual hosts, network services, and other high-level objects must be scrutinized in order to detect

intrusiveness. Thus the design must be capable of inferring complex objects and then detecting intrusiveness in these complex objects.

The problem of detecting intrusive behavior in a heterogeneous network through the observation of packets can be abstracted to one of detecting behaviors in complex systems from the analysis of low level information. If this generalized problem can be solved, then the specific problem of detecting intrusiveness in computer networks can be reduced to the application of the generalized solution to this specific problem. Furthermore, the generalized solution can be applied to other specific problems such as detection of intrusiveness in a single computer system, detection of component failure in computer networks, and detection of component malfunction in complex subsystems (e.g. spacecraft).

Chapter two of this thesis presents a meta-language to model complex systems from the observation of low level information. Chapter three presents a generalized solution for detecting a particular behavior in a system represented by the meta-language presented in chapter two. These two chapters offer a solution to the generalized problem of detecting a behavior in a complex system. Chapter four applies this generalized solution to the original problem of detecting intrusive behavior in a heterogeneous computer network. This application fulfills the original goal of this thesis: the design of a system to detect intrusive activity. As proof of the functionality of the design, chapter five presents the results of a prototype network based intrusion detection system. Conclusions and future research are presented in chapter six. And finally, Appendix A provides a brief description of the installation and operation of the working prototype.

Chapter 2

System Description Language

As mentioned previously, the problem of detecting intrusive activity in a heterogeneous network of computers through the observation of network packets can be generalized to the detection of a behavior in a complex system from the analysis of low level information. The complex system is composed of a variety of components each of which in turn may be composed of other components, but only the simplest of components, the lowest levels of information, are directly visible to a monitor. Unfortunately, to detect the behavior of interest, the complex components which are not directly observed, as well as the low level components, must be examined for the manifestation of the behavior.

To provide for a mechanism to infer the complex components of a system, I have defined a meta-language, called the system description language (SDL), to describe the relationship among components of a system. The description of a system with this language is called its system language definition. As the low level information is observed, the system language definition is used to infer the existence of the complex objects and the relationships between them. A snapshot of all the low level objects and the inferred complex objects and their relationships to each other represent a model of the actual system at a particular moment in time. It is this model which will be examined for the manifestation of the behavior of interest.

The SDL, system language definition, and the snapshot of an actual system have a direct resemblance to the definition of a traditional programming language. The SDL provides a functionality similar to that of the BNF meta-language. The system language definition is similar to a traditional program language definition (e.g. Pascal). And the

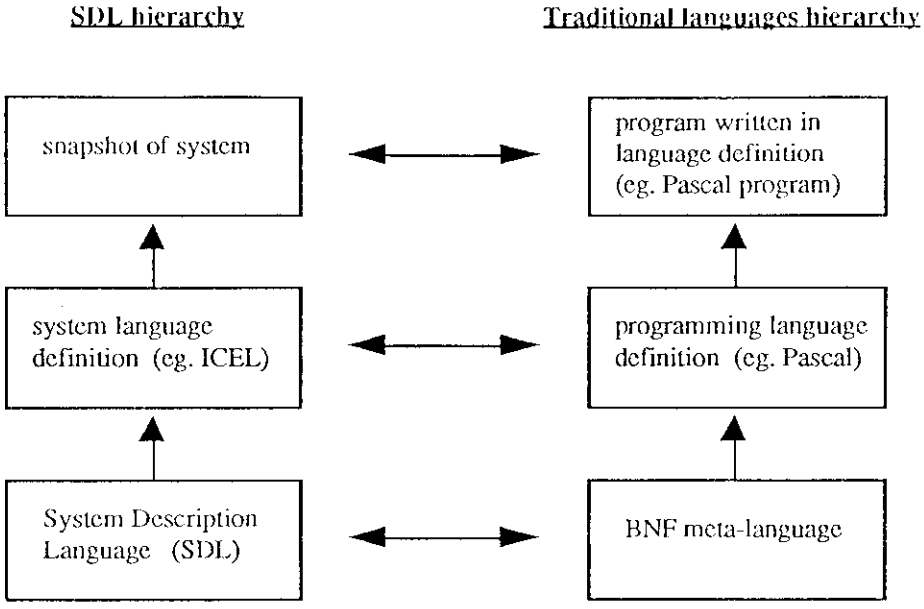


Figure 1

snapshot of a system is similar to a program defined by a traditional programing language. This relationship is shown in figure 1.

The system description language is the focus of this chapter. Section 2.1 introduces the issues which must be addressed by the system description language. Section 2.2 presents a review of attribute grammars, the ancestor of the system description language. And section 2.3 discusses the actual system description language.

2.1 Issues to be Addresses by the SDL

To design a language which can be used to describe and model complex systems from the observation of low level information, a number of issues must be addressed. First, how are the low level, simple components of the system detected, and how are the attributes of each low level object determined? I have chosen to not address this issue in this thesis, and it is not part of the language definition. The low level components are detected, and their associated attributes are determined by a preprocessor. This is not unlike the design of conventional programming languages which assume the presence of a lexical analyzer to detect tokens, and, if necessary, determine their attributes.

The second issue is identification and representation of components of the system which are not observed directly. In fact, a complex object which does not have a real world counterpart may be desired. For example, my model for the computer network environment includes an object called a "service-set." The service-set object does not exist in the actual system, but its presence is helpful in analyzing other components such as network connections. The system description language must provide a mechanism for inferring the existence of these unobserved, perhaps nonexistent, objects. Furthermore, the language must provide mechanisms to determine enough information about these abstract objects so they can be analyzed for the behavior of interest.

The third issue concerns the transitory nature of many of the objects in a system. Systems such as a heterogeneous network have a number of components which exist for a time, and then disappear. For example, network connections are created and destroyed continuously. The system description language must be able to handle the creation and destruction of components, and the system description language must provide information to determine when a component should be created or destroyed. Thus the model of an actual system, as determined by the system language definition, can change over time.

In summary, the system description language assumes the low level, simple components and their attributes are provided to it. From these simple components, the systems description language must provide a mechanism to infer the existence of, and the relationships between, complex objects. The system description language must provide mechanisms to determine enough information about the complex objects to analyze the objects for the presence of the behavior of interest. Finally, the system description language must provide a means both to determine when a component to the system is created or destroyed and to modify the model of the system due to the creation or destruction of a component.

2.2 Attribute Grammars

The system description language which satisfies the above requirements is built upon the concept of attribute grammars. A quick introduction to attribute grammars is provided below. Readers already familiar with this subject may want to skip to section 2.3.

An attribute grammar describes both the strings accepted by a language (e.g. the syntax of the language) and a method to determine the "meaning" of those strings (e.g. the semantics of the language). An attribute grammar consists of a context-free grammar, a set of attributes for each symbol in the grammar, and a set of functions defined within the scope of a production rule in the grammar to determine the values for the attributes of each symbol in that production [1]. The following example of an attribute grammar for the definition and interpretation of binary numbers* will be used to clarify the relationships between these three components of an attribute grammar.

The context-free grammar for our language of binary numerals is defined by G = (**V**,**N**,**P**,**S**) where **V** is the set of symbols, **N** is the set of nonterminal symbols, **P** is the set of production rules, and **S**, an element of **N**, is the start symbol. The set of terminal symbols, a subset of **V**, is {1,0,.}. These are the ASCII characters one, zero, and period. The set of nonterminal symbols, **N**, is {B,L,N}. They represent the abstract objects bit, list of bits, and number. The start symbol for our attribute grammar for binary numbers is N, the abstract number. The set of production rules relating these symbols and providing the definition of acceptable strings is given in figure 2A.

---

* This example is taken from [14].

$N \rightarrow L.L$	$N \rightarrow L_1.L_2$	$v(N) = v(L_1) + v(L_2)/2^{l(L_2)}$
$N \rightarrow L$	$N \rightarrow L$	$v(N) = v(L)$
$L \rightarrow LB$	$L_1 \rightarrow L_2B$	$v(L_1) = 2v(L_2) + v(B)$, $l(L_1) = l(L_2)+1$
$L \rightarrow B$	$L \rightarrow B$	$v(L) = v(B)$, $l(L) = 1$
$B \rightarrow 1$	$B \rightarrow 1$	$v(B) = 1$
$B \rightarrow 0$	$B \rightarrow 0$	$v(B) = 0$
A	B	

Figure 2

By this context free grammar, we can see that the string 11.01 is an acceptable binary number. The parse tree for this string is given in figure 3A.

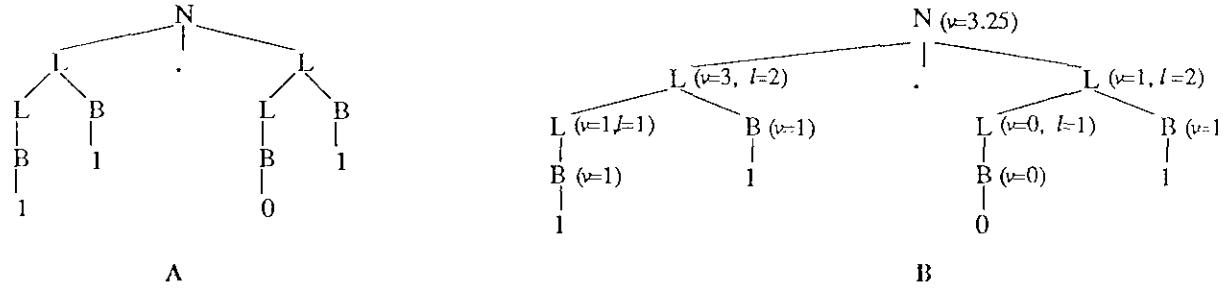


Figure 3

The context free grammar can be used to build a parse tree of a string and determine whether the string is valid in the language; however, the context free grammar cannot be used to determine the meaning of the string. The addition of attributes and attribute functions are necessary to determine the meaning of the string.

The set of attributes, A, for each nonterminal are given as follows: $A(B) = \{v\}$, $A(L) = \{v,l\}$, and $A(N) = \{v\}$. The attribute v is the value of a symbol, and the attribute l is the length of a symbol.

The set of functions defined within the scope of each production rule is given in figure 2B.

By using the attributes for each symbol and the attribute functions, we can now assign meaning to each symbol in the parse tree (see figure 3B). For our language of

binary numbers, the most important meaning is that of the start symbol N. Our string 11.01 now has the meaning of 3.25.

2.3 System Description Language

This section introduces the system description language, an extension of attribute grammars. This system description language provides a structure by which a system's components and relationships between components can be described. The description, or system language definition, of a system can be used to both infer the existence of complex objects (e.g. determine the syntactic structure of the system) and assign "meaning" to these objects (e.g. the semantic information about the system). The meaning of an object, the values of its attributes, will be used to determine if the behavior of interest is present in any of the components of the system.

Similar to an attribute grammar, a system language definition written in the SDL consists of a structural grammar, a set of attributes for each object, or symbol, in the structural grammar, and a set of functions defined within the scope of a production rule of the structural grammar which determine the attribute values for each object in that production.

2.3.1 Objects

Objects are the components of the system which will be modelled. These objects may or may not have real world counterparts. Two variety of objects exist: basic objects and complex objects. Basic objects are the low-level components which are directly observed. These are similar to terminal symbols in traditional programming languages. Complex objects, on the other hand, are not observed and must be inferred from the observation of basic objects. Complex objects are similar to non-terminal symbols in traditional programming languages. These two objects are discussed further below.

2.3.1.1 Basic Objects

Basic objects are the only observed components of a system. They are simple, indivisible components of the complex system being modelled; they are detected and their attributes determined by a preprocessor. This preprocessor performs the job of a lexical analyzer in traditional programming languages. Basic objects are treated as events; they only exist for the moment at which they are observed. For example, in the networked system, packets are basic objects. Basic objects for other systems may be an audit record from an operating system, a message to a spacecraft component, or a sampled data point from some measuring instrument.

A basic object type is defined by a name and a list of attributes. The name format for my system is the same as the standard C identifier. Attributes will be discussed in section 2.3.2. An example basic object representing a possible audit record is:

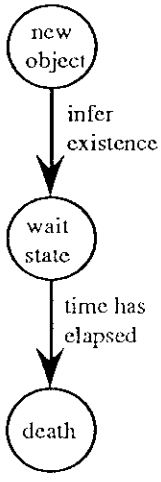
```
basic: audit_record {
            attribute list
       }
```

The keyword **basic** states that the following object type is a basic object, and the object type's name is audit_record. Attributes for this object will be discussed later in section 2.3.3.

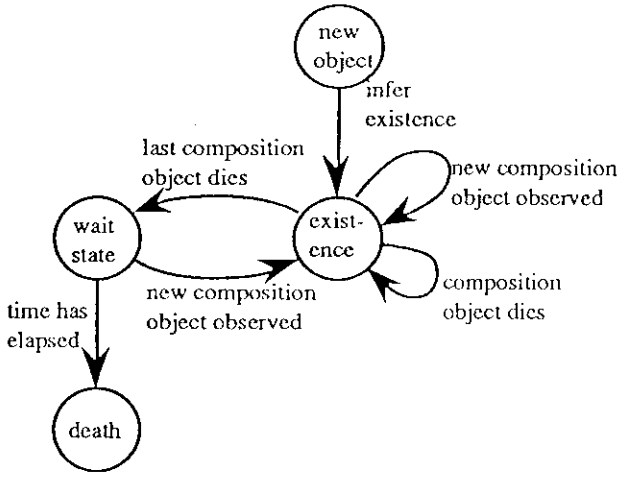
2.3.1.2 Complex Objects

As mentioned previously, complex objects are components of a system which are not directly observed by the monitor, so they must be inferred from the observation of the basic objects. A complex object is composed of basic objects and/or other complex objects. For example, a complex object type called process may be defined for an audit trail based monitor. Although processes are not directly observed by the monitor, information about them can be inferred from the audit records. Therefore, in our model,

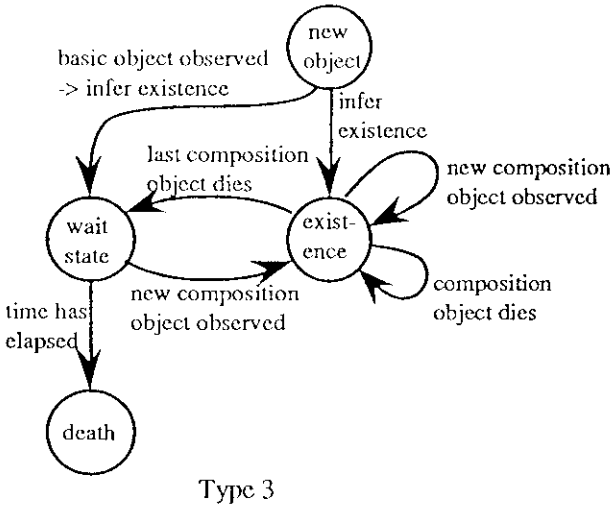
processes are composed of audit records. This composition will be discussed further in section 2.3.3.



Type 1



Type 2



Type 3

Complex object type 1: composed of only basic objects

Complex object type 2: composed of only complex objects

Complex object type 3: composed of both basic and complex objects

Figure 4

A major difference between complex objects and basic objects is that complex objects have persistence. Whereas basic objects are treated as events, complex objects are treated as persistent elements which are created and possibly destroyed. The creation of a complex object occurs as soon as it can be inferred. The destruction of an object is

considerably more complex and depends on the both the definition of the complex object and the existence of objects which compose the complex object.

First, if any object **A** exists and is part of an object **B**'s composition, then object **B** should continue to exist. Second, if the last object which is part of object **B**'s composition is destroyed, then object **B** will be destroyed after a specified time delay, Δt, unless another object which is part of **B**'s composition is created or observed. This specified Δt is the value of a function associated with the object, and it may depend on the object's attributes. This function is discussed in section 2.3.4.

Complex objects can be composed of only basic objects, only complex objects, or a combination of basic and complex objects. The three state machines describing the life cycle of a complex object are shown in figure 4.

Complex object types are defined in my system by one of the following forms depending on their composition:

complex type 1: *name* {
attribute list
}

complex type 2: *name* {
attribute list
}

complex type 3: *name* {
attribute list
}

2.3.2 Attributes

As mentioned previously, each object has a set of attributes associated with it. These attributes provide a "meaning" to each object. It is the attributes which will be used to determine if the object is associated with a particular behavior. These attributes are also used, along with the production rules described in 2.3.3, to determined if an object **A** is part of object **B**'s composition.

Each attribute consists of a name and type. The name is used to reference the value, and the type determines the variable type which can be assigned or retrieved from the attribute. For example, "int value" would describe an attribute of type "int" which is referenced by the name "value." Attribute types may be complex structures defined in the same format as complex types are described in the C language [13]. The extent of the complexity depends on the final implementation. For simplicity, I allow attributes types to be as complex as any in the C language, and they are therefore defined in the C style.

Many of the attribute values of an object will be assigned by the monitor. For example, when the existence of a new host is inferred, a host object is created and its internet address is immediately assigned by the system. The values of other attributes, however, are determined by attribute functions. Attribute functions, described in section 2.3.4, take as input attribute values associated with the object and possibly attributes of other objects associated with it by the production rules (see section 2.3.3).

A complex object type to represent a process which is composed of audit records can now be defined as follows:

```
complex type 1:    process{
                       int      process_id
                       int      creation_time
                       char*    user_name
                       int      num_of_files_opened
                   }
```

This simple definition of a process has a simple identifier, *process_id*, a time at which the process was created, *creation_time*, the user who owns the process, *user_name*, and a record of the number of data files the process has opened, *num_of_files_opened*. The set of attributes for an object **O** can be defined as $A(\mathbf{O}) = \{a_1, a_2, ..., a_n\}$. For example, A(process) = {process_id, creation_time, user_name, num_of_files_opened}.

2.3.3 Productions

Productions define the relationship between the different object types of a system. They define which types of objects compose a complex object, and they indicate how to determine which set of objects from an object type compose that object. A production rule has the form:

complex_object_type -> list of *object_composition*

The complex_object_type is simply the name of a complex object type (e.g. process). An object composition is a set defined by a tuple of the form <object_type restrictions>. The object_type is simply the name of one of the defined object types (basic or complex), and the restrictions determine which of all possible objects of type object_type are actually used to compose the complex object.

For example, let the complex object type called process be define as above, and let the object type called audit_record be defined as follows:

```
basic:      audit_record {
                              int       audit_number
                              int       process_id
                              int       action
                              char*     object_name
                              int       error_code
                          }
```

A production rule for the process object can now be defined as follows:

```
process -> audit_record
    where for all e ∈ audit_record
        e.process_id = process.process_id
```

Finally, each element of audit_record which composes a particular process object is called a sub-component of the process objects, and the process object is called a super-component of the audit_record objects. The concepts of sub-components and super-components will be used in section 3.2 to define integrated object analysis functions.

2.3.4 Attribute Functions

The attributes of a complex object which are not static (e.g. the internet address of a host is determined at the moment it is inferred, and this value does not change) are defined by attribute functions. The attribute functions for a structural language are defined as they are for attribute grammars; however, special attention must be given to the format of the production and the restriction for the production. For example, an attribute function to determine the value for the attribute "num_of_files_opened" of a process object could be as follows:

process.num_of_files_opened = | S |
where S = {e ∈ audit_record |
(e.action = OPEN_FILE) & (e.error_code = NONE)}

Each e ∈ audit_record is assumed to be a sub-component of the process object as defined by the restrictions in the production rule for process objects.

Chapter 3

Detecting Behaviors in Systems

Once the structural grammar, attributes, and attribute functions have been defined, a second set of functions, called behavior-detection functions, must be defined for each object in the structural grammar. Behavior-detection functions determine whether an object is associated with the particular behavior of interest. Because a behavior may manifest itself differently or more clearly in different object types, each object in a system parse tree (the snapshot of the system) must be examined for the behavior by particular behavior-detection functions designed for that object type. For each type of object, there will be two behavior-detection functions: the isolated behavior-detection function, and the integrated behavior-detection functions. These two function types are discussed below.

3.1 Isolated Object Analysis

An isolated behavior-detection function for an object uses the attributes of that object to calculate the probability that the object is associated with the behavior of interest. In short, an isolated behavior-detection function is a classifier. With some preprocessing to transform the attribute types, a large number of classifiers can be used.

Unfortunately, classifiers generally have to be trained with sample data, and the behavior of interest is often quite rare. There are at least two possible solutions to the problem of lack of sample data: expert systems and single behavior classifiers. An expert system, designed by people knowledgeable about the problem domain, can use heuristics to determine how close an object's behavior is to the behavior of interest. A single behavior classifier is built around the assumption that a rare behavior will be significantly different than normal behavior. If this is true, a single classifier can profile normal behavior, and then it could report any behavior which does not strongly resemble normal behavior. Work on such single behavior classifiers have been done by SRI for IDES [15]

and Los Alamos National Laboratories for Wisdom and Sense [22]. For our particular problem environment, we combined the efforts of both an expert system and a single behavior classifier.

3.2 Integrated Objects Analysis

An integrated behavior-detection function for an object modifies the result of the isolated behavior-detection function for the object by including the analysis of the isolated behavior-detection functions for sub-components and super-components of that object. The modification by an integrated behavior-detection function allows the inclusion of both the results of aggregated analysis (those from super-components) and the results of more detailed levels of analysis (those from sub-components). The integrated behavior-detection function can be implemented by a weighted average function such as:

$$\frac{W_1*\text{Object\_if} + W_2*\text{Super\_if} + W_3*\text{Sub\_if}}{W_1 + W_2 + W_3}$$

Where Object_if is the value calculated by the objects isolated behavior-detection function, Super_if is the average isolated behavior-detection function value for all the super - components, Sub_if is the average isolated behavior-detection function value for all the sub-components, and W_1, W_2, and W_3 are the weights.

The relationship between an object's attributes, the isolated behavior-detection functions, and the integrated behavior-detection functions can be seen in figure 5. In this example, we are interested in analyzing the object B_1 for a particular behavior. The object B_1 is composed of objects C_1 and C_2, and it is part of the object A_1. Result B_{1V} is the analysis of object B_1 in isolation, and result $B_{1V'}$ is the result after combining the result of B_{1V} with the results from objects C_1, C_2, and A_1.

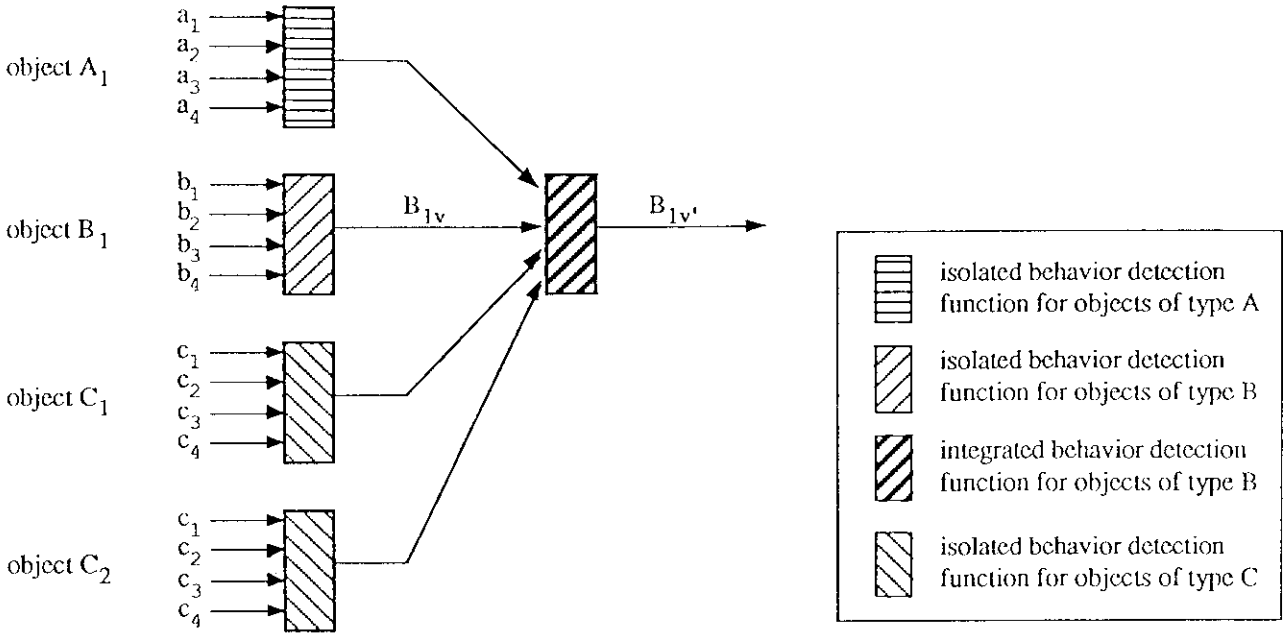


Figure 5

3.3 Examples

This section provides some examples which take advantage of the of the integrated behavior-detection functions. These examples, taken from actual events recorded by the NSM implementation, show how the analysis of an object can be enhance by the including the analysis of higher-level and lower-level objects. The objects in question are hosts and connections, and their relationship is given by the production rule

HOST -> CONNECTION

The first example involved an intruder who exploited a poor design in an operating system: the ability to use a file transfer service called tftp (trivial file transfer protocol) to copy the password file from a host without having to login to that host. The intruder on host A initiated a tftp connection to host B to acquire the password file. Several minutes later the intruder initiated a successful login from A to B by using a cracked password.

The analysis of the second connection, the login connection, is very important. Although the anomaly detection component of the isolated behavior-detection function (see section 4.2 for more information on the isolated behavior-detection function) provided a

fairly high warning level for this connection, the isolated behavior-detection function ranked this connection as suspicious as several other unusual login connections that day. However, because host A was marked as suspicious due to the fact that a password file had been shipped to it, the integrated behavior-detection function raised the suspicion level for the connection above that of other login connections-a correct analysis. Therefore, although the isolated behavior-detection function rated the connections the same, by using the integrated behavior-detection analysis the NSM was able to successfully rate the intrusive connection as more suspicious than other connections.

The second example involved an intruder trying to gain access to computers by trying default passwords. The intruder would try the account "guest" with password "guest," the account "root" with password "root," and the account "field" with password "service." Because this is similar to a thief going down a street and trying each door to see if it is unlocked, computer security experts call this a "door knob attack." This attack was successful on two machines.

In this particular example, the intruder on host C attempted a number of login connections to several machines. Because each connection was marked as suspicious, the integrated behavior-detection function increased the suspicion level for host C. Thus, host C was rated as being more likely to be involved with intrusive activity than any other host.

In both of these examples the analysis of the isolated behavior-detection functions were enhanced by including other information. In the first example, the suspicion level of a connection was increased by including the analysis of one of it's super-components, the host A. Likewise, in the second example, the suspicion level of a host was increased by including the analysis of the sub-components, the connections.

Chapter 4

Network Security Monitor

What has been presented so far are a method for modelling systems and a method for analyzing them for particular behaviors. This chapter applies these methods to the LAN environment to create a monitor capable of detecting intrusive behavior in that environment. Section 4.1 uses the system description language to model the network, and section 4.2 defines some simple, but effective, behavior-detection functions to look for intrusive behavior.

As evidence to support the methodologies presented in this thesis, and to support the network model in particular, a prototype monitor, called the Network Security Monitor (NSM), has been implemented using the model described in this section. The results from this implementation follow in chapter 5.

4.1 Interconnected Computing Environment Language

Using the system description language format, I define a system language definition, called the Interconnected Computing Environment Language (ICEL), to model the heterogeneous network. The objects of the ICEL are described in section 4.1.2; their attributes are presented in 4.1.3; productions for the complex objects are given in 4.1.4; and the attribute functions for each complex object are presented in 4.1.5.

4.1.2 Objects

4.1.2.1 Basic Objects

The ICEL has only one basic object: the packet. The packet represents a standard internet packet as defined in [2]. Packets, the only component of the ICEL directly

observed, are given to the NSM as complete units with all their attributes precomputed. These attributes are presented in detail in section 4.1.4.

Packets are the only means of communication on the network. When a process on machine **A** wants to send some information to a process on machine **B**, it must do so by encapsulating the information in an internet packet and placing that packet on the network.

4.1.2.2 Complex Objects

The ICEL has a number of complex objects. These complex objects include streams, connections, hosts, service-sets, and the network-system. Streams, connections, and hosts have direct analogies in the real system being monitored. Service-sets and the network-system do not have such equivalences; however, they are used to provide better analysis of the network environment.

A stream is a unidirectional flow of data from a process on one machine to a process on another. It is constructed by assembling together the packets the first process places on the network for the second process. The data inside the packets can be strung together, so patterns which may span several packets can be detected. For example, the word "hello" may be distributed in five different packets, but a stream has the capability of detecting the existence of that word.

A connection is a bidirectional flow of data between two process on two machines. It consists of two streams - one representing an input stream and the other representing an output stream. For example, for a login connection one stream may represent the keystrokes a user sends to the remote process, and the other stream may represent the information the user sees (e.g. results from doing an "ls").

A host represents a node on the network with an internet address such as a computer or printer. A host is identified by the connection made from or to it. Thus a host is constructed by assembling all the connections associated with that host.

A service-set is a fictitious object representing the overall behavior of a particular network service. It is constructed by assembling all of the connections of a particular service type together. For example, a service-set exists representing all telnet connections. A service-set essentially aggregates the warning values for all connections of a particular service. Thus the object is useful for determining whether or not a flaw in a particular service is being mechanically exploited on a number of machines.

A network-system is a fictitious object representing the entire state of the networked computing environment. Only a single instance of the network-system exists (e.g. it is the start symbol for the system language).

4.1.3 Attributes

The attributes for each of the objects are given in the following tables.

Attributes for a PACKET

Attribute	Description
src_addr	the internet address of the host which generated the packet.
dst_addr	the internet address of the host to which the packet is destined.
protocol	the protocol used (e.g. TCP/IP or UDP/IP).
src_port	the port number on the source host which generated the packet. This is used to help identify the processes which are exchanging the packet.
dst_port	the port number on the host to which the packet is destined. This is used to help identify the processes which are exchanging the packet.
num_of_bytes	the number of bytes in the data portion of this packet.
list_of_bytes	the actual bytes in the data portion of this packet.
time	the time at which the packet was observed.

Attributes for a STREAM

Attribute	Description
src_addr	the internet address of the host from which the packets in this stream were generated.
dst_addr	the internet address of the host to which the packets in this stream are destined.
protocol	the protocol used (e.g. TCP/IP or UDP/IP).
src_port	the port number on the source host for this stream. This is used to help identify the processes which are exchanging the packet.
dst_port	the port number on the destination host for this stream. This is used to help identify the processes which are exchanging the packet.
start_time	the time of the first packet observed for this stream.
last_update_time	the time of the most recent packet observed for the stream.
total_packets	the number of packets transmitted on this stream.
total_bytes	the number of bytes of data in the packets.
string_matcher_list	a list of string_matchers. A string_matcher is a complex data structure used to look for a particular string in the data portion of consecutive packets. It is represented by the 3-tuple <count,state,dfa>. The count is the number of times the string has been matched. The state is the current state of the dfa. And the dfa is an automaton designed to match a string in a stream. The current implementation is based on the Knuth-Morris-Pratt algorithm [16]

Attributes for a CONNECTION

Attribute	Description
initiator_addr	the internet address of the host which initiated the connection.
receiver_addr	the internet address of the host to which the connection is made.
protocol	the protocol used (e.g. TCP/IP or UDP/IP).
initiator_port	the port number on the host initiating the connection.
receiver_port	the port number on the host receiving the connection. This is used to determine the network service type of the connection.
service	the network service (e.g. telnet) used for the connection
start_time	the time the existence of the connection is first inferred. This is the time of the first packet observed for this connection.
last_update_time	the time of the most recent packet observed, from either stream, for this connection.
pkts_from_initiator	the number of packets transmitted from the initiator host to the receiver host.
bytes_from_initiator	the number of bytes of data transmitted from the initiator host to the receiver host.
strs_matched_from_initiator	a list of numbers indicating the number of times each string being searched for has been matched in the data from the initiator host.
pkts_from_receiver	the number of packets transmitted from the receiver host to the initiator host.
bytes_from_receiver	the number of bytes of data transmitted from the receiver host to the initiator host.
strs_matched_from_receiver	a list of numbers indicating the number of times each string being searched for has been matched in the data from the receiver host.

Attributes for a HOST

Attribute	Description
host_addr	the internet address of the host.
pkts_from_host	the number of packets generated from the host since its existence was first inferred.
bytes_from_host	the number of bytes in the packets from the host since its existence was first inferred.
strs_matched_from_host	a list of numbers indicating the number of times each string being searched for has been matched in data from the host.
pkts_to_host	the number of packets sent to the host since its existence was first inferred.
bytes_to_host	the number of bytes in the packets sent to the host since its existence was first inferred.
strs_matched_to_host	a list of numbers indicating the number of times each string being searched for has been matched in data to the host.
n_min_connection_num_from	the number of connections initiated from the host in the last *n* minutes.
current_connection_num_from	the number of connections from the host which currently exist.
n_min_connection_num_to	the number of connections made to the host in the last *n* minutes.
current_connection_num_to	the number of connections made to the host which currently exist.

Attributes for a SERVICE-SET

Attribute	Description
protocol	the protocol used (e.g. TCP/IP or UDP/IP).
service_id	the network service (e.g. telnet) represented by this service-set.
total_pkts_from_initiator	the number of packets generated by the initiator host for all connections of this service type.
total_bytes_from_initiator	the number of bytes generated by the initiator host for all connections of this service type.
strs_matched_from_initiator	a list of numbers indicating the number of times each string being searched for has been matched in the data from the initiator host for all connections of this service type.
total_pkts_from_receiver	the number of packets generated by the receiver host for all connections of this service type.
total_bytes_from_receiver	the number of bytes generated by the receiver host for all connections of this service type.
strs_matched_from_receiver	a list of numbers indicating the number of times each string being searched for has been matched in the data from the receiver host for all connections of this service type.
n_min_connection_num	the number of connection of this service type which have been generated in the last *n* minutes.
current_connection_num	the number of connections of this service type which currently exist.

Attributes for a NETWORK-SYSTEM

Attribute	Description
total_pkts_from_initiator	the sum of all packets from initiator streams since the network-system's existence was first inferred.
total_bytes_from_initiator	the sum of bytes in the data portion of the packets from initiator streams since the network-system's existence was first inferred.
strs_matched_from_initiator	a list of numbers indicating the number of times each string being searched for has been matched in streams from initiator hosts.
total_pkts_from_receiver	the sum of all packets from receiver streams since the network-system's existence was first inferred.
total_bytes_from_reciever	the sum of bytes in the data portion of the packets from receiver streams since the network-system's existence was first inferred.
strs matched_from_receiver	a list of numbers indicating the number of times each string being searched for has been matched in streams from receiver hosts.
n_min_connection_num	the total number of connections which have been started in the last *n* minutes.
current_connection_num	the current number of connections which exist on the network
current_host_num	the current number of hosts which are known to exist on the network.

4.1.4 Productions

The productions for the different complex objects are described below:

```
STREAM -> PACKET
    where for all p ∈ PACKET
        (STREAM.src_addr = p.src_addr,
        STREAM.dst_addr = p.dst_addr,
        STREAM.protocol = p.protocol,
        STREAM.src_port = p.src_port,
        STREAM.dst_port = p.dst_port)
```

CONNECTION -> STREAM
    where for all s ∈ STREAM
        (CONNECTION.protocol = s.protocol,
        ((CONNECTION.initiator_addr = s.src_addr,
        CONNECTION.receiver_addr = s.dst_addr,
        CONNECTION.initiator_port = s.src_port,
        CONNECTION.receiver_port = s.dst_port)
        or (CONNECTION.initiator_addr = s.dst_addr,
        CONNECTION.receiver_addr = s.src_addr,
        CONNECTION.initiator_port = s.dst_port,
        CONNECTION.receiver_port = s.src_port)))

HOST -> CONNECTION
    where for all c ∈ CONNECTION
        ((HOST.host_addr = c.initiator_addr)
        or (HOST.host_addr = c.receiver_addr))

SERVICE-SET -> CONNECTION
    where for all c ∈ CONNECTION
        (SERVICE.protocol = c.protocol,
        SERVICE.service_id = c.service)

NETWORK-SYSTEM -> HOST SERVICE-SET

4.1.5 Attribute Functions

Certain attributes of an object are assigned by the monitor system when the object is created. The functions to determine the values for the attributes which are not assigned by the monitor are provided below. The functions for each attribute of each complex object follow the complex object's production description.

STREAM -> PACKET:
    STREAM.start_time = min {p.time | p ∈ PACKET}
    STREAM.last_update_time = max {p.time | p ∈ PACKET}
    STREAM.total_packets = |PACKET|
    STREAM.total_bytes = $\sum_{p \in PACKET}$ p.num_of_bytes
    STREAM.string_matcher_list = KMP(STREAM.string_matcher_list,p.list_of_bytes}

CONNECTION -> STREAM:

CONNECTION.start_time = min (s.start_time | s ∈ STREAM}

CONNECTION.last_update_time = max {s.last_update_time | s ∈ {STREAM))

CONNECTION.pkts_from_initiator = s.total_packets
where CONNECTION.initiator_addr = s.src_addr

CONNECTION.bytes_from_initiator = s.total_bytes
where CONNECTION.initiator_addr = s.src_addr

CONNECTION.strs_matched_from_initiator = s.string_matcher_list
where CONNECTION.initiator_addr = s.src_addr

CONNECTION.pkts_from_receiver = s.total_packets
where CONNECTION.receiver_addr = s.src_addr

CONNECTION.bytes_from_receiver = s.total_bytes
where CONNECTION.receiver_addr = s.src_addr

CONNECTION.strs_matched_from_receiver = s.string_matcher_list
where CONNECTION.receiver_addr = s.src_addr

HOST -> CONNECTION:

$$\text{HOST.pkts\_from\_host} = \sum_{c \in C1} c.\text{pkts\_from\_initiator} + \sum_{c \in C2} c.\text{pkts\_from\_receiver}$$

$$\text{HOST.bytes\_from\_host} = \sum_{c \in C1} c.\text{bytes\_from\_initiator} + \sum_{c \in C2} c.\text{bytes\_from\_receiver}$$

$$\text{HOST.strs\_matched\_from\_host} = \sum_{c \in C1} c.\text{strs\_matched\_from\_initiator} + \sum_{c \in C2} c.\text{strs\_matched\_from\_receiver}$$

$$\text{HOST.pkts\_to\_host} = \sum_{c \in C2} c.\text{pkts\_from\_initiator} + \sum_{c \in C1} c.\text{pkts\_from\_receiver}$$

$$\text{HOST.bytes\_to\_host} = \sum_{c \in C2} c.\text{bytes\_from\_initiator} + \sum_{c \in C1} c.\text{bytes\_from\_receiver}$$

$$\text{HOST.strs\_matched\_to\_host} = \sum_{c \in C2} c.\text{strs\_matched\_from\_initiator} + \sum_{c \in C1} c.\text{strs\_matched\_from\_receiver}$$

HOST.n_min_connection_num_from = | C3 |

HOST.current_connection_num_from = | C4 |

HOST.n_min_connection_num_to = | C5 |

HOST.current_connection_num_to = | C6 |

where C1 = {c ∈ CONNECTION' | c.initiator_addr = HOST.host_addr}

and C2 = {c ∈ CONNECTION' | c.receiver_addr = HOST.host_addr}

where C3 = {c ∈ CONNECTION' | ((CURRENT_TIME - c.last_update_time) <= n_min) & (c.initiator_addr = HOST.host_addr)}

where C4 = {c ∈ CONNECTION | c.initiator_addr = HOST.host_addr}

where C5 = {c ∈ CONNECTION' | ((CURRENT_TIME - c.last_update_time) <= n_min) & (c.receiver_addr = HOST.host_addr)}

where C6 = {c ∈ CONNECTION | c.receiver_addr = HOST.host_addr}

Note: CONNECTION' is the set of all elements which have ever been a member of CONNECTION.

SERVICE-SET -> CONNECTION:

SERVICE-SET.total_pkts_from_initiator = $\sum_{c \in C1}$ c.pkts_from_initiator

SERVICE-SET.total_bytes_from_initiator = $\sum_{c \in C1}$ c.bytes_from_initiator

SERVICE-SET.strs_matched_from_initiator = $\sum_{c \in C1}$ c.strs_matched_from_initiator

SERVICE-SET.total_pkts_from_receiver = $\sum_{c \in C1}$ c.pkts_from_receiver

SERVICE-SET.total_bytes_from_receiver = $\sum_{c \in C1}$ c.bytes_from_receiver

SERVICE-SET.strs_matched_from_receiver = $\sum_{c \in C1}$ c.strs_matched_from_receiver

SERVICE-SET.n_min_connection_num = | C2 |

SERVICE-SET.current_connection_num = | CONNECTION |

where C1 = CONNECTION'

and C2 = {c ∈ CONNECTION' | (CURRENT_TIME - c.last_update_time) <= n_min}

NETWORK-SYSTEM -> HOST SERVICE-SET:

NETWORK-SYSTEM.total_pkts_from_initiator = $\sum_{s \in S1}$ s.pkts_from_initiator

NETWORK-SYSTEM.total_bytes_from_initiator = $\sum_{s \in S1}$ s.bytes_from_initiator

NETWORK-SYSTEM.strs_matched_from_initiator = $\sum_{s \in S1}$ s.str_matches_from_initiator

NETWORK-SYSTEM.total_pkts_from_receiver = $\sum_{s \in S1}$ s.pkts_from_receiver

NETWORK-SYSTEM.total_bytes_from_receiver = $\sum_{s \in S1}$ s.bytes_from_receiver

NETWORK-SYSTEM.strs_matched_from_receiver = $\sum_{s \in S1}$ s.str_matches_from_receiver

NETWORK-SYSTEM.n_min_connection_num = $\sum_{s \in SERVICE\text{-}SET}$ s.n_min_connection_num

NETWORK-SYSTEM.current_connection_num = $\sum_{s \in SERVICE\text{-}SET}$ s.current_connection_num

where S1 = SERVICE-SET'

4.2 Detecting Intrusive Behavior

As was mentioned earlier, an isolated behavior-detection function is currently an area of important study. A generic classifier such as the statistical anomaly detector component of the IDES system [15] which could be used for all object types, would be a nice solution; however, this ignores possibly important semantic information about the individual object types. Instead of using a single generic statistical tool to implement the

isolated behavior-detection functions, I have concentrated on a behavior-detection tool for only a single object type: connections.

I have chosen to build the isolated behavior-detection function for connections out of three separate components: an anomaly detector, an attack model, and an expert system. Each component produces a warning value between zero and ten, and the final warning value is simply a weighted average of these results.

Initially the weights for all three components were equal; however, through experimentation and use I found that the weight for the expert system component should be much higher than that of the anomaly detector component; the expert system produced much better results in isolation than did the anomaly detector. A method to determine the correct balancing for these weights will be a part of future research.

4.2.1 Anomaly Detector Component

The anomaly detection component of the isolated behavior-detection function for connections is based on the concept of data paths (see 1.1.2). A warning value for each connection is based on the probability of observing a connection on that data path, <src_host, dst_host, service>. If the probability of observing such a connection is high, then the warning value would be low, and conversely, if the probability of observing the connection is low, then the warning value would be high.

The probability of observing a connection on the data path is based on whether a connection was observed on the data path on each of the 'n' previous days. This function, taken from [15], is defined as:

$$P_r = \frac{\sum_{k\geq 1}^{n} W_k 2^{-bk}}{\sum_{k\geq 1}^{n} 2^{-k}}$$

where

- k is the index of days. k = 1 is the most recent day, and k = n is the oldest day for which data is known.
- W_k is an indicator function that returns 1 if there was a connection on the data path for the k^{th} day, and it returns 0 otherwise.
- b is the half-life of the data paths. For the NSM implementation, I used a half-life of seven days (a single week), so b = 1/7 = 0.143.

The choice of a half-life value was arbitrary; however, a method to calculate a half-life which will result in the most accurate probability prediction will be a part of future research.

4.2.2 Attack Model Component

The attack model component is based in part on the work by Gihan Dias [7]. This work tries to determine the probability of a connection being an attack based on the hosts and network service involved.

First the attack model assigns to every host a value between zero and ten to represent a security level for the host. A host with a security level of zero is perceived to be very insecure, and a host with a security level of ten is perceived to be very secure. Furthermore, the model assigns to every network service two numbers between zero and ten representing the level of authentication required to use the service and the capabilities of the service.

The attack model assumes that an attack will more likely be from an insecure host to a secure host than from a secure host to an insecure host. The model also assumes that an attacker would prefer to use a service with little authentication requirements (e.g. password requirements) but strong capabilities. These are captured by the equation:

$$\text{Warn\_level} = \frac{(10 + \text{dst\_sec\_level} - \text{src\_sec\_level}) + (10 + \text{cap} - \text{auth})}{4}$$

where

- dst_sec_level is the security level of the destination host.
- src_sec_level is the security level of the source host.
- cap is the capability value of the service used.
- auth is the authentication required by that service.

4.2.3 Expert System Component

The expert system component of the attack model is based on the tell tale signs I regarded that someone being intrusive might leave behind. For example, if the person generated a high number of "Login incorrect" or "Permission denied" messages, the expert system would increase the warning level of a connection. Examining password files, attempts at exploiting known operating system flaws (e.g. accessing the password file with the trivial file transfer protocol service, tftp), and attempts at trying default accounts and passwords (e.g. trying the guest account with password "guest") would also increase the warning level of a connection.

Despite the rule base being very small (on the order of a dozen rules), the expert system component of the NSM proved to be most successful at both detecting actual intrusions and reporting the fewest false positives.

Chapter 5

Model Results

An implementation of the model presented in chapter 4 has been tested at the University of California at Davis for several months. This implementation, called the network_analyzer, is part of a suite of tools called the Network Security Monitor (NSM). The other components of the NSM (network_recorder, transcript, and playback) are discussed in [10].

Section 5.1 discusses the early successes from the first few months of testing. Section 5.2 discusses some of the knowledge gained which will help guide future research.

5.1 Results

I ran the NSM (including the network_analyzer) on the Electrical Engineering and Computer Science LAN at UCD for a period of approximately three months. During this time over 400,000 connections were identified and analyzed by the NSM, and among these connections, over 400 were identified as being associated with intrusive behavior. If the connections for each day were ordered by the warning level assigned to them by the network_analyzer, the expected position for the identified intrusive connections would be in the top 0.5% of the connection. In other words, if there were 3500 connections analyzed in one day, approximately half of the connections associated with intrusive behavior could be found by examining only the top seventeen most suspicious connections.

Intrusive connections were associated with over half of the hosts on the LAN. These hosts included a variety of hardware and operating system platforms. The intrusive activity ranged from simple doorknob attacks (trying default passwords on machines) to complex attacks requiring the coordinated use of several services and several hosts. For example, one attack required the tftp, finger, telnet, and login services and two different

target hosts for a complete success. Roughly half of the intrusive activity was associated with outsiders and half with insiders.

For comparison, of the intrusive activity associated with the 400 connections identified correctly with the aid of the NSM, only eight attempted logins were detected by system administrators or the user community. Thus, for every intrusive connection detected by administrators and users, roughly forty went undetected.

5.2 Lessons Learned

The testing of the model on an actual network provided me with concrete results which help substantiate the model as well as determine areas of future research. First, I wanted to examine if the profile of network connections grew unbounded. Figure 6 shows the size of the profile (determined by the number of data paths present) for a period of slightly more than three weeks. As can be seen, the profile size reached a size limit after a little more than a week of analysis, so experimental results show that network activity does not produce an unbounded profile.

Second, I wanted to determine how useful the detection of anomalous activity was at detecting intrusive activity in connections. Unfortunately, strict anomaly detection was shown to be a poor means of intrusion detection. Mail connections, which frequently were made to very unusual sites, tended to dominate the connections which were reported as possibly associated with intrusive activity; however, telnet connections made up the majority of actual intrusive connections. Figure 7 shows the average probability of occurrence calculated by the network_analyzer for connections associated with a number of network services. This graph shows the average mail connection (smtp) had a lower average probability of observation than did the average telnet connection. In other words, mail connections tended to be more anomalous than telnet connections. In the future, effort will be made to normalize anomalous activity for services based on the rate of anomalous activity generated in the past.

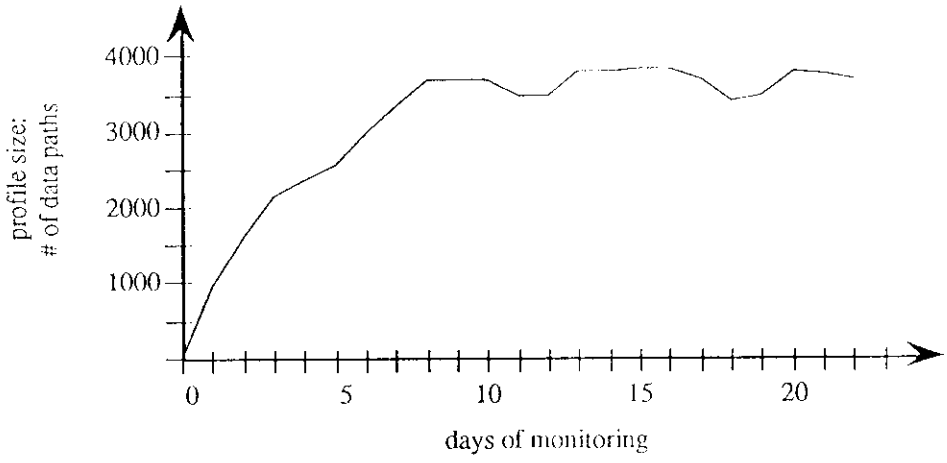


Figure 6

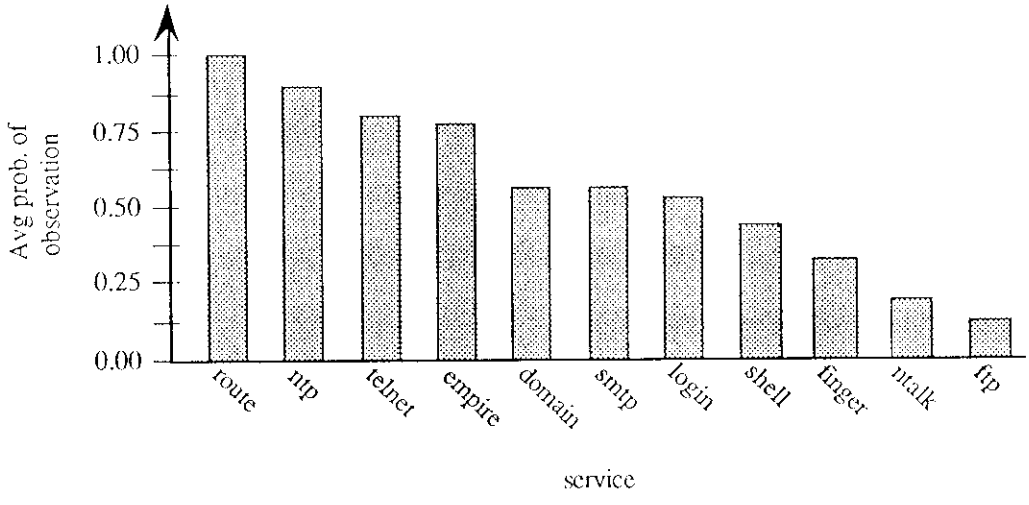


Figure 7

Chapter 6

Conclusions and Future Research

As government reports, recent books, and the popular media have stated, our computers systems are vulnerable to attack. Authentication and access control mechanisms to prevent intrusive activity have not been wholly successful, so a second layer of defense, intrusion detection, is needed. UCD took on the task of developing an intrusion detection system capable of simultaneously detecting intrusive behavior on many hosts running many versions of operating systems. To solve this problem, I developed a model to detect behaviors in dynamic and complex systems. I then mapped the interconnected computing environment into this model, and defined an isolated behavior-detection function to detect intrusive behavior in connections.

Results from a prototype of the model appear very promising. The NSM detected nearly 40 times the number of connections associated with intrusive behavior than did the system administrators and user community.

Despite these positive results, however, there are a number of issue brought up in this thesis which require further research. First, a rigorous method for determining the half-life of data path information is needed (section 4.2.1). One possible solution is to use a Newtonian search method to minimize the absolute value of actual probability - calculated probability. Another issue which I did not address is the possibility that the data paths for different services may have a different half-lifes.

A second area in need of further research is the expert system component of the isolated behavior-detection function for connections. The rule base should be expanded to capture all known specific attack methods, and if possible, generalized rules to detect new and unknown attacks should be designed.

A third area of research is a method for choosing the weights used to combine different components of the isolated behavior-detection function for connections.

Questions such as whether the calculated warning values of the components are independent or not must be addressed.

A fourth area of research is in the area of generalized isolated behavior-detection functions. I mentioned earlier that I had my doubts whether a generalized function was capable of being successful; however, I made no attempt to prove or disprove this. Future research could include the testing of different intrusion detection algorithms by using them as isolated behavior-detection functions for different objects and comparing their performances.

Finally, a fifth area of research involves the testing for properties of system language definitions. Since the SDL, a system language definition, and a snapshot of a system are related to a traditional programming language hierarchy (see figure 1), many of the same questions about programming languages can be asked of my system. For example, can a language be shown to be complete? That is, can all the attributes for all objects always be calculated? Or, are the models of a system (snapshots of the system) always unique? This is similar to the question of whether a parse tree for a programming language is unique. Can a system language definition be recursive? For example, are there restrictions which would allow the production rules

A -> B

B -> A

to still produce complete results?

References

1. G.V. Bochmann, "Semantic Evaluation from Left to Right," Communications of the ACM, vol. 19, no. 2, pp. 55-62, Feb. 1976.

2. D.E. Comer, Internetworking With TCP/IP, 2nd ed., Englewood Cliffs, New Jersey: Prentice Hall, 1991.

3. D.E. Denning, "An Intrusion Detection Model," *IEEE Trans. on Software Engineering*, vol. SE-13, no. 2, pp. 222-232, Feb. 1987.

4. D.E. Denning, a conversation with L. Todd Heberlein, December 6, 1990

5. P.J Denning, ed. Computers Under Attack: Intruders, Worms, and Viruses. New York: ACM Press, 1990.

6. Department of Defense Trusted Computer System Evaluation Criteria, Dept. of Defense, National Computer Security Center, DOD 5200.28-STD, Dec. 1985.

7. G.V. Dias, K.N. Levitt, B. Mukherjee., "Modeling Attacks on Computer Systems: Evaluating Vulnerabilities and Forming a Basis for Attack Detection," Technical Report CSE-90-41, University of California, Davis.

8. C. Dowell and P. Ramstedt, "The COMPUTERWATCH Data Reduction Tool," *Proc. 13th National Computer Security Conference*, pp. 99-108, Washington, D.C., Oct 1990.

9. D.B. Guralnik, ed. Webster's New World Dictionary, (Simon and Schuster, 1980).

10. L.T. Heberlein, B. Mukherjee, K.N. Levitt, D. Mansur., "Towards Detecting Intrusions in a Networked Environment," *Proc. 14th Department of Energy Computer Security Group Conference*, May 1991.

11. L.T. Heberlein, G.V. Dias, K.N. Levitt, B. Mukherjee, J. Wood., "Network Attacks and an Ethernet-based Network Security Monitor," *Proc. 13th Department of Energy Computer Security Group Conference*, pp. 14.1-14.13, May 1990.

12. L.T. Heberlein, G.V. Dias, K. N. Levitt, B. Mukherjee, J. Wood, D. Wolber., "A Network Security Monitor," *Proc. 1990 Symposium on Research in Security and Privacy*, pp. 296-304, May 1990.

13. B.W. Kernigan, D.M. Ritchie., The C Programming Language, 2nd ed. Englewood Cliffs, New Jersey: Prentice Hall, 1988.

14. D.E. Knuth, "Semantics of Context-Free Languages," *Math Systems Th. 2* (1968), 127-145. Correction appears in *Math Systems Th.5* (1971),95.

15. T.F. Lunt, et al., "A Real Time Intrusion Detection Expert System (IDES)," Interim Progress Report, Project 6784, SRI International, May 1990.

16. U. Manber, Introduction To Algorithms: A Creative Approach, New York: Addison-Wesley Publishing Company, 1989

17. Plato, The Republic of Plato, Trans. B Jowett. 1892.

18. M.M. Sebring, et al., "Expert Systems in Intrusion Detection: A Case Study," *Proc. 11th National Computer Security Conference*, pp. 74-81, Oct. 1988.

19. S.E. Smaha, "Haystack: An Intrusion Detection System," *Proc. IEEE Fourth Aerospace Computer Security Applications Conference*, Orlando, FL, Dec. 1988.

20. S.R. Snapp, J. Brentano, G.V. Dias, T.L. Goan, T. Grance, L.T. Heberlein, C. Ho, K.N. Levitt, B. Mukherjee, D.L. Mansur, K.L. Pon, S.E. Smaha., "Intrusion Detection Systems (IDS): A Survey of Existing Systems and a Proposed Distributed IDS Architecture," Technical Report CSE-91-7, University of California, Davis.

21. W.T. Tener, "Discovery: an expert system in the commercial data security environment," *Security and Protection in Informations Systems: Proc. Fourth IFIO TC11 International Conference on Computer Security*, North-Holland, May 1988.

22. H.S. Vaccaro and G.E. Liepins, "Detection of Anomalous Computer Session Activity," *Proc, Symposium on Research in Security and Privacy,* pp. 280-289, Oakland, CA, May 1989.

23. J.R. Winkler, "A Unix Prototype for Intrusion and Anomaly detection in Secure Networks," *Proc. 13th National Computer Security Conference*, pp. 115-124, Washington, D.C., Oct. 1990.

Appendix A

Network Security Monitor

a brief description

6 June 1991

L. Todd Heberlein

Installation

To install NSM from the tar file, enter the command

tar xvf nsm.tar

If the NSM is distributed on tape, enter the command

tar xv

Either of the commands will produce a new directory called *NSM* in your local directory. Under this directory are four more directories: *analysis*, *bin*, *src*, and *tmp*. If the computer you are using is a Sun-4 architecture, the binary programs should be fine. However, if you are running on a Sun-3, or if you simply want to recompile the programs, you will have to enter each of the program directories under *NSM/src* and remake the programs. To remake a program for a Sun-4, type the command

make

If you want to run on a Sun-3 architecture, modify the *makefile* by changing the -DSUN4 option in the CFLAGS to the -DSUN3 option.

Note: the program *network_capture* will not run on the computer unless the Networking Tools and Programs software installation option was installed (do a man on *etherfind* for more information). Basically, if *etherfind* works on your computer, so will *network_capture*.

The configuration file, *NSM/analysis/config.file*, will have to be modified to reflect your site's internet addresses and dail-up ports. For more information on this file, see the section File Descriptions.

File Descriptions

This section discusses the files used by the NSM. At the end of the appendix is a figure providing a diagram of the file structure.

NSM

(type-directory): *NSM* is the root directory for the NSM tools. Underneath the *NSM* are four directories called *analysis*, *bin*, *src*, and *tmp*.

NSM/analysis

(type-directory): *analysis* is the directory from which the user will do all of his or her work. This directory contains the data files which direct the the various programs of the NSM to do their work. Results from analysis (e.g.. connection files, profile files, and transcript files) are stored here as well.

NSM/bin

(type-directory): *bin* is the directory for all the executable programs for the NSM. The directory contains the programs *analyze*, *network_capture*, *top_con*, *transcript*, and *warn_sort*. This directory is static and should not change unless one of the programs is recompiled.

NSM/src

(type-directory): *src* is the directory for the source code for all the executable programs in the *NSM/bin directory*. The source code for each executable program is in its own directory. The directory names are the same as the executables: *Analyze*, *Network_capture*, *Top_con*, *Transcript*, and *Warn_sort*.

NSM/tmp

(type-directory): *tmp* is the directory for the network traffic data files. These log files will be of the format *filename*YYMMDD.HH; where YY is the year, MM is the month, DD is the day of the month, and HH is the hour of the day. The choice for *filename* can be changed in the *config file*.

NSM/analysis/con_count file

(type-text data file): *con_count file* contains a single number representing the total number of connections analyzed by the *analyze* program. This number will be initially set to zero. If, for example, analyze was run on three days of which 4000, 4010, and 4020 connections were analyzed , then *con_count file* would contain the number 4000 after the first day, 8010 after the second day, and 12030 after the third day.

Example:

```
12030
```

NSM/analysis/config file

(type-text data file): *config file* is the most important data file in this directory, and almost every program of the NSM uses it. *config file* contains the information describing the local site's internet addresses (the class B networks for the site), the local dial-up ports, the place to store the data files (e.g.. NSM/tmp), and how often to switch data files (typically it is on every hour).

Example:

```
#num_of_local_class_B_nets      1
#class_B_net                    128.120.0.0
#num_of_local_gateways          4
#local_gateway                  128.120.2.251
#local_gateway                  128.120.2.253
#local_gateway                  128.120.2.254
#engr-dnet1                     128.120.59.29
#output_root_file_name          ../tmp/log
```

```
#minutes_between_files          20
```

Each line contains two pieces of information: a comment and a value. The comment has to be a single "word" (I read it in with a single scanf(...%s...)). The values in this file can be read as follows:

#num_of_local_class_B_nets = 1 - This number indicates how many class B networks are at the local site.

#class_B_net = 128.120.0.0 - There will be n lines of this form, where the number n is given in the previous line (in this case, 1). This value gives the internet address of the local class B network. In this case it is 128.120.*.*. Place zeros, where the stars are. Any packet which has a source or destination address will be considered a foreign packet.

#num_of_local_gateways = 4 - This number indicates how many local gateways or dial-ups there are. Any packet from/to these dial-up addresses is considered to a foreign packet.

#local_gateway = 128.120.2.251 - This is an internet number of a dial up

#local_gateway = 128.120.2.253 - This is an internet number of a dial up

#local_gateway = 128.120.2.254 - This is an internet number of a dial up

#engr-dnet1 = 128.120.59.29 - This is an internet number of a dial up

#output_root_file_name = ../tmp/log - This is the directory where the network traffic will be stored (../tmp) and the root file name for the data files (log).

#minutes_between_files = 20 - This indicates that the file name where traffic is stored will be changed every twenty minutes. It will be aligned on the hour.

NSM/analysis/connections.file

(type-text data file): *connections.file* is a log file of all the connections observed by the last run of the program *analyze*. The first line of the file specifies the number of strings searched for by *analyze*. The following lines are the strings searched for. After the strings are the actual logs of connections ordered by when they were closed (or terminated). Each log is on a single line, but each line typically wraps around three line on a normal 80 column display.

Example:

```
7
login: guest
Login incorrect
daemon:
passwd
login: root
Permission denied
CWD ~ROOT
 218 267389 8.944 5.778 10.000 10.000   128.120.2.251   128.120.57.60   6 25858
    23    telnet  Mon-Jun-03-18:12:03-1991   Mon-Jun-03-18:12:38-1991   35
    51     40     34     144 0-rec-1 1-rec-2
 199 267370 8.944 5.778 10.000 10.000   128.120.2.251   128.120.57.14   6 10498
    23    telnet  Mon-Jun-03-18:10:09-1991   Mon-Jun-03-18:10:36-1991   27
   126     93     71     278 0-rec-1 1-rec-5
 119 267290 8.944 5.778 10.000 10.000   128.120.2.251   128.120.57.67   6 11020
    23    telnet  Mon-Jun-03-17:59:48-1991   Mon-Jun-03-18:00:22-1991   34
   109     81     70     243 0-rec-3 1-rec-3
```

Although this file represents a connection file which has actually been sorted by waring value, the information in this data file is exactly the same as the original connections.file

The first line of the file contains a number indicating the number of strings which were searched for while processing this log file. In this case, seven strings were used. Following the number are the actual strings used. These strings are indexed from 0 to (n - 1) (in this case, 0 to 6). Therefore the string

login: guest

has the index number 0, and the string

CWD ~ROOT

has the index number 6.

Following the strings are the actual connection logs. The first connection log, although it is one line, is wrapped across three different text lines. The information in this first record can be interpreted as follows:

connection index for this particular processing = 218
connection index for all time = 267389
composite warning value = 8.944
attack model warning value = 5.778
anomaly detection warning value = 10.000
expert system warning value = 10.000
address of the host initiating the connection = 128.120.2.251
address of the host receiving the connection = 128.120.57.60
protocol of service (6 - TCP, 17 - UDP) = 6
port used by initiating host = 25858
port used by receiving host = 23
service name = telnet
start time of the connection = Mon-Jun-03-18:12:03-1991
ending time of the connection = Mon-Jun-03-18:12:38-1991
duration of the connection in seconds = 35
number of packets initiator sent = 51
number of bytes initiator sent = 40
number of packets receiver sent = 34
number of bytes receiver sent = 144
string match: (0-rec-1) [This states that the string index 0, "login: guest," was sent by the receiving host (rec -> receiving host, init -> initiator host), and it was matched 1 time]
string match: (1-rec-2) [This states that the string index 1, "Login incorrect," was sent by the receiving host 2 times]

NSM/analysis/host.file

(type-text data file): *host.file* contains a list of internet addresses and security level numbers. The data file is usually empty; however, it is possible to specify the security levels (between 0 and 10) of individual hosts. A telnet from a low security machine to a high security machine will be considered more suspicious than a telnet from a high security machine to a low security machine. This is used by the program *analyze* to calculate the attack model portion of the warning value. This file can be empty. Any local host not seen in this file will be given the default security level of 3, and any foreign host not seen in this file will be given the default security level of 1.

Example:

```
128.120.57.1      5
128.120.57.117    8
128.120.57.118    8
128.120.57.119    9
128.120.57.120    5
128.120.57.121    5
```

```
128.120.57.122      5
128.120.57.130      5
128.120.57.131      8
128.120.57.132      8
```

In this example the host 128.120.57.1 is assigned a security level of 5, and the host with the internet address 128.120.57.117 is assigned a security level of 8.

NSM/analysis/profile.file

(type-text data file): *profile.file* contains a record of observed past network activity. It is used by the program *analyze* to calculate the anomalousness of an observed connection. *profile.file* will initially be empty.

Example:

```
128.115.1.1  128.120.57.1  6  25  0  32
130.86.71.1  128.120.57.1  6  79  0  1
130.86.71.2  128.120.57.20  6  513  1  255
128.228.1.2  128.120.57.20  6  25  1  255
128.118.56.2  128.120.57.20  6  25  0  111
137.39.1.2  128.120.57.20  6  25  0  48
130.86.71.2  128.120.57.20  6  514  1  136
129.245.1.2  128.120.57.20  6  25  0  33
129.10.1.2  128.120.57.20  6  25  0  67
```

The first line of the profile represents a data path from the host 128.115.1.1 to 128.120.57.1 by the TCP/IP protocol (6 = TCP, 17 = UDP) and the service port 25 (electronic mail). The 0 indicates that there were no connections on this data path on the most recent day. The last number, 32, is translated to a bit list (00100000). A one indicates that a connection was observed on this data path during that day. The most recent day is represented by the most significant bit (left most), and the furthest day remembered is represented by the least significant bit (right most)

NSM/analysis/strings.file

(type-text data file): *strings.file* is perhaps the most useful file for detecting intrusions. *strings.file* contains a list of strings that the program *analyze* will search for in the network connections. For example "login: guest" can be searched for in the network connections. If it is matched, it usually implies that someone tried to login as guest. A number of stings are searched for (e.g.. "Login incorrect"), and the results of this search is used by *analyze* to calculate the expert system component of the warning value; however, the search for other strings is done by simply placing a new string at the end of the list.

Example:

```
login: guest
Login incorrect
daemon:
passwd
login: root
Permission denied
CWD ~ROOT
```

To find the results of a string match, examine the content of the file *connections.file*. For example to find all occurrences of the string "Permission denied," string index 5, simply grep for a occurrences of the strings "5-rec" and "5-rec." For example:

egrep "5-recl5-init" connections.file

NSM/analysis/tcp.file

(type-text data file): *tcp.file* contains a list of known tcp service ports and names. Also associated with each tcp service are two numbers representing the service's capability and authentication requirement (each number has a value between 0 and 10). For example, telnet has strong capabilities and strong authentication requirements (a password). The capability and authentication requirement values are used by *analyze* to determine the expert system component of the warning value.

Example:

7	echo	1	1
9	discard	1	1
11	systat	1	1
13	daytime	1	1
15	netstat	1	1
20	ftp-data	7	7
21	ftp	7	3
23	telnet	10	3
25	smtp	4	3
37	time	1	1
43	whois	1	1
53	domain	1	1
101	hostnames	1	1
111	sunrpc	8	3
77	rje	1	1
79	finger	4	1

In this example, the first service occupies port number 7, its name is "echo," its capabilities are listed at a strength of one, and its authentication is also listed at a strength of one.

NSM/analysis/udp.file

(type-text data file): *udp.file* contains a list of known udp service ports and names. Also associated with each udp service are two numbers representing the service's capability and authentication requirement (each number has a value between 0 and 10). For example, tftp has strong capabilities, but it has very weak authentication requirements (none). The capability and authentication requirement values are used by *analyze* to determine the expert system component of the warning value.

Example:

53	domain	1	1
111	sunrpc	7	7
69	tftp	8	1
123	ntp	1	1
512	biff	1	1
513	who	1	1
514	syslog	1	1
517	talk	1	1

In this example, the first service occupies port number 53, its name is "domain," it has a capability strength of one, and it also has a authentication strength of one.

NSM/bin/analyze

(type-binary program): *analyze* is the most important of the NSM programs; *analyze* is the program which actually detects and analyzes the connections in the network

traffic. This program requires as input the starting year, month, day, and hour of interest in network traffic data files and the total number of hours to search for data. The output of this program is the data files *connections.file*, *profile.file*, and *con_count.file*. The previous *connections.file*, *profile.file*, and *con_count.file* will be overwritten. In addition to the input mentioned above, *analyze* uses the following data files: *con_count.file*, *config.file*, *host.file*, *profile.file*, *strings.file*, *tcp.file*, and *udp.file*.

Example:

../bin/analyze 91 6 3 0 24

This command will analyze the data starting at 1991, June 3rd, and the 0th hour (midnight), and it will analyze 24 hours worth of data.

NSM/bin/network_capture

(type-binary program): *network_capture* pulls the network traffic off the network and places the packets into the data files in the directory *NSM/tmp*. Only a specific portion of the network traffic is recorded-the traffic between local hosts and hosts off site and the traffic to and from local dial-ups. Currently no FTP data is stored. The program has no input. The output are the log files in the *tmp* directory. *network_capture* uses the information in *config.file* to determine which traffic to capture and where to store it.

Example:

../bin/network_capture

That is it.

NSM/bin/top_con

(type-binary program): *top_con* takes a file of connection logs and creates a script file which when executed will create transcript files for the first *n* connections. The format is "*top_con input_file output_file n*." *input_file* is the name of the connection log file (a file such as *connections.file*); *output_file* is the name of the new script file; *n* is the number of connections for which transcripts will be created.

Example:

../bin/top_con warn91-6-3 top-10 10

This will take a connection file called warn91-6-3 (same format as connections.file) as input, and it will create a shell script file called top-10 which will include the commands to generate transcript files of the first 10 connections in the file warn91-6-3. To execute this shell script, simply type "top-10."

NSM/bin/transcript

(type-binary program): *transcript* takes a connection file and a connection number as input, and it creates two transcript files (one for the input stream and one for the output stream) for the that connection. The names for these transcript files will be *connection_file_name.number*.init and *connection_file_name.number*.dest where *connection_file_name* is the name of the connection file supplied as input and *number* is the connection number. The file ending in init is the data sent by the host initializing the connection, and the file ending in dest is the data sent by the host to which the connection was made.

Example:

../bin/transcript warn91-6-3 199

This will take the connection file named warn91-6-3 and look for connection index number 199 in it. When it finds this particular connection log, it will generate two transcript files (warn91-6-4.199.init and warn91-6-3.199.dest) representing the

data flowing from the host which initialized the connection and the data from the destination host.

NSM/bin/warn_sort
(type-binary program): *warn_sort* takes as input an unsorted connection file and outputs a sorted (by warning value) connection file.

Example:

../bin/warn_sort connections.file warn91-6-3

This example will take the connection log file called connections.file and generate a connection log file sorted by warning value called warn91-6-3.

NSM/src/Analyze
(type-directory): *Analyze* is the directory for all the source code for the program *analyze*. To re-make the program *analyze*, simply type the command "make" from inside the *Analyze* directory.

NSM/src/Network_capture
(type-directory): *Network_capture* is the directory for all the source code for the program *network_capture*. To re-make the program *network_capture*, simply type the command "make" from inside the *Network_capture* directory.

NSM/src/Top_con
(type-directory): *Top_con* is the directory for all the source code for the program *top_con*. To re-make the program *top_con*, simply type the command "make" from inside the *Top_con* directory.

NSM/src/Transcript
(type-directory): *Transcript* is the directory for all the source code for the program *transcript*. To re-make the program *transcript*, simply type the command "make" from inside the *Transcript* directory.

NSM/src/Warn_sort
(type-directory): *Warn_sort* is the directory for all the source code for the program *warn_sort*. To re-make the program *warn_sort*, simply type the command "make" from inside the *Warn_sort* directory.

NSM/tmp/logYYMMDD.HH
(type-binary data file): logYYMMDD.HH is the general form for the binary data files for the network traffic. For example, log910530.00 is the network traffic for 1991 May 30 and the 0^{th} hour (e.g.. midnight until 1:00 am). The name "log" can be changed by modifying the data file *analysis/config.file*.

Basic Operations

This section describes the simple operations needed to keep a moderate handle on network security at a particular site. I assume that a single person (or perhaps a small group of people) will be in charge with examining the network activity regularly for possibly intrusive data.

The first step is to start the collection of data. Assuming the config.file is set up correctly (see **Installation** and **File Descriptions**: analysis/config.file), the collection of data is started simply by typing the following command (as root) from the analysis directory:

../bin/network_capture

This command will start the collection of data. It should run continuously, and if desired, this process can be placed in the background. The CPU overhead of this process is barely noticeable; however, the disk usage can be expensive. Keep an eye on the disk space. Finally, I strongly recommend that the data be kept on a local hard disk; shipping the data across the network to another disk will reduce your network bandwidth.

The second step is to analyze the data. The following list of commands, which can be put into a shell script (e.g.. my_shell_script), are all that is needed to produce transcript files for the days ten most intrusive looking connections:

```
../bin/analyze 91 6 3 0 24
../bin/warn_sort connections.file warning.list
../bin/top_con warning.list top-10 10
top-10
```

The execution of these commands will produce ten pairs of transcript file of the form warning.list.###.init and warning.list.###.dest (### will be the connection number). A security officer need only verify whether these connections are legitimate or not by examining them with any word processor.

To modify the date to process new data, simply modify the first command. For example, to process the data the next day simply change the first line to:

../bin/analyze 91 6 4 0 24

The following two files are actual transcript files generated by the above method for a particular connection. The first file is the data sent by the destination host back to the hacker. The second file is the data sent by the hacker to the target computer.

First file:

```
TRANSCRIPT

For connection file:    warn91-6-3
and connection index:  218

Initiating host:    128.120.2.251
Destination host:  128.120.57.60
Service:              telnet

Start time:  Mon-Jun-03-18:12:03-1991
End time:     Mon-Jun-03-18:12:38-1991

Warning level:  8.944
words matched from initiating host:
words matched from destination host:
        Login incorrect     2
        login: guest    1
```

```
Data from destination host
-------------------------------------
}}{{~}

SunOS UNIX (surya)


{~login: guest
Password:
Login incorrect
login: uucp
Password:
Login incorrect
```

Second file:

```
TRANSCRIPT

For connection file: warn91-6-3
and connection index: 218

Initiating host: 128.120.2.251
Destination host: 128.120.57.60
Service:          telnet

Start time: Mon-Jun-03-18:12:03-1991
End time:   Mon-Jun-03-18:12:38-1991

Warning level: 8.944
words matched from initiating host:
words matched from destination host:
        Login incorrect  2
        login: guest  1

Data from initiation host
-------------------------------------
{}}{{guest
guest
uucp
```

In summary, the following operations are needed:

../bin/network_capture

my_shell_script

The dates in the shell script, my_shell_script, will have to be changed, but that is the only modification which is needed. Since network_capture needs to be only started once, the only program which needs to be executed on a daily basis is my_shell_script.

Advanced Operations

This section describes a few extra operations a security officer may wish to perform to provide extra security. First, if intrusions are suspected from a particular host, a security officer may want to find all connections from or to that host. This can be accomplished by simply grep-ing for the host's internet address in the connection file. This will produce all the connection records associated with that host. For example:

grep 141.225.1.2 connections.file

may produce the following results:

```
2340 267290 8.944 5.778 10.000 10.000   141.225.1.2   128.120.57.120   6 11020
    23      telnet    Mon-Jun-03-17:59:48-1991    Mon-Jun-03-18:00:22-1991      34
    109      81       70      243 0-rec-3 1-rec-3
2345 267295 8.944 5.778 10.000 10.000   141.225.1.2   128.120.57.121   6 1235
    23      telnet    Mon-Jun-03-18:01:00-1991    Mon-Jun-03-18:02:22-1991      82
    218     165      140     612 0-rec-1
```

This result indicates that there were two connections from 141.225.1.2: connection 2340 and connection 2345. The first connection matched the string "login: guest" three times and the string "Login incorrect" three times. The second connection matched the string "login: guest" once. To generate the transcript reports for these two connections, simply type:

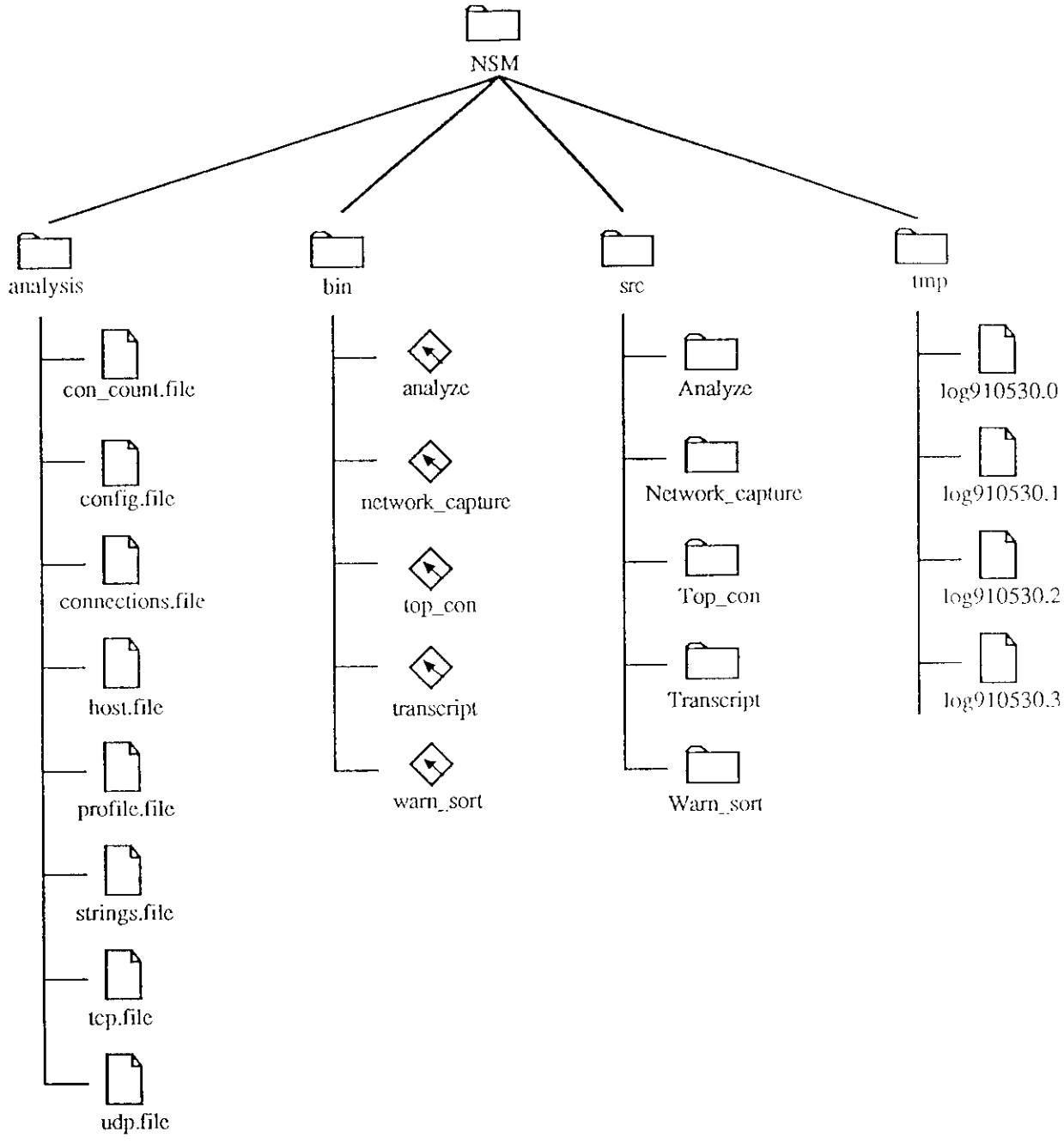
../bin/transcript connections.file 2340
../bin/transcript connections.file 2345

Another operation which might be useful is to search for a particular key word or string. For example, suppose the string "CLASSIFIED" is contained in all classified documents. To search for such a string in all network traffic, Add the string to the end of the *strings.file*, and then process the data with *analyze*. If this string is the 8th string in the file, all occurrences of it can be searched for with the command

egrep "7-init|7-rec" connections.file

(note that the strings are counted from 0 to (n-1). Thus the eigth string is actually indexed as string 7). If this string was matched in any of the connections, the connection records will be printed. To retrieve the actual transcript of the connections, use the ../bin/transcript command as in the above example. This is a useful exercise to determine what sensitive data is being shipped off site.

NSM Layout


NSM
analysis
bin
src
tmp
con_count.file
config.file
connections.file
host.file
profile.file
strings.file
tcp.file
udp.file
analyze
network_capture
top_con
transcript
warn_sort
Analyze
Network_capture
Top_con
Transcript
Warn_sort
log910530.0
log910530.1
log910530.2
log910530.3