Watcher: The Missing Piece of the Security Puzzle

John C. Munson
Computer Science Department
University of Idaho
Moscow, ID
jmunson@cs.uidaho.edu

Scott Wimer
Software Systems International, LLC
121 Sweet Avenue
Moscow, ID 83844
scottw@softsysint.com

Abstract

Modern intrusion detection systems are comprised of three basically different approaches, host based, network based, and a third relatively recent addition called procedural based detection. The first two have been extremely popular in the commercial market for a number of years now because they are relatively simple to use, understand and maintain. However, they fall prey to a number of shortcomings such as scaling with increased traffic requirements, use of complex and false positive prone signature databases, and their inability to detect novel intrusive attempts. The procedural based intrusion detection systems represent a great leap forward over current security technologies by addressing these and other concerns. This paper presents an overview of our work in creating a true procedural Disallowed Operational Anomaly (DOA) system.

1. Introduction

All modern software design methodologies have their origin in the very primitive standalone computer environments. These computer systems were not hooked up to the Internet nor even simple local networks. Typically one job ran on the computer at a time and jobs were batched and run sequentially. In this environment, the only threat to a program's integrity while it was executing was provided by the ineptitude of the computer operator. Life was simple enough that the execution of the program could be monitored as it ran through light displays on the operator's console. Control of the program was provided by the computer operator. Over time, control of program execution gradually shifted from the operator to the operating system. A human being simply could not respond in a timely fashion to the demands of a program running at electronic speeds.

From a computer security standpoint, the computer hardware, the operating system, user applications and the computer operator were all well contained within the confines of a room whose perimeter could be secured with existing security technology. In this limited environment, there was little need to consider invasive forces from outside. That only came when the complete secure environment was compromised by attaching it to a completely unregulated and potentially hostile information ether.

The evolution of the original computing environment was almost exactly duplicated by the evolution of the personal computer. Originally these systems were self contained computers. They were used for entertainment and, to a limited extent, for business. Then everything changed (except for the underlying security premise). People attached modems to their PC's and connected to the Internet. The software that they were running was designed to run in a contained and completely encapsulated environment.

The Internet was not a major factor in the design of early Windows O/S's. It was certainly not even considered when the architectural framework for UNIX was laid down. It still isn't a design consideration in the evolution of legacy code for managing a personnel system. We have a security problem today because the same naive premise about the operating environment of most our software is still in effect. That is, the software will operate in a closely confined hardware facility and if it is to be exposed to a more hostile world, a big brother security system will protect it.

The literature and media abound with reports of successful violations of computer system security by both external attackers and internal users [7]. These breaches occur through physical attacks, social engineering attacks, and attacks on the system software. It is this later category of attack that is the focus of this paper. During an attack, the intruder subverts or bypasses the security mechanisms of the system in order to gain unauthorized access to the system or to increase their current access privileges. These attacks are successful when the attacker is able to exploit a vulnerability in the software to cause it to execute in a manner that is typically inconsistent with the software specification and thus lead to a breach in security [1]. Intrusion detection systems monitor traces of user activity to determine if an intrusion has occurred. The traces of activity can be collated from audit trails or logs [11,17], network monitoring [9,15] or a combination of both. Once the data regarding a relevant aspect of the behavior of the system is collected, the classification stage starts.

Although taxonomies that are more complex exist [3,9], intrusion detection classification techniques can be broadly catalogued in the two main groups: *misuse intrusion detection* [12,13] and *anomaly intrusion detection* [10]. The first type of classification technique searches for occurrences of known attacks with a particular "signature" and the second type searches for departures from normality. Some of the newest intrusion detection tools incorporate both approaches [2,16].

Most current intrusion detection techniques examine the input to software or the output from software. NIDS (Network-based Intrusion Detection Systems) tools fit directly into this category. Log scanning tools also fall into this category. Other tools examine the input and output of the system calls made by programs. These I/O driven intrusion detection techniques have built in limitations. These limitations are discussed below because they represent the primary challenges successful new intrusion detection techniques need to overcome.

As we will see in the behavioral software model, program modules are distinctly associated with certain functionalities and operations that the program is capable of performing. As each operation is executed, a subset of software modules is executed which creates a particular and distinct *signature* of module executions [15]. An alternative approach examines sequences of system calls as indicators of system behavior [cf. 18]. What is missing in these approaches is an understanding that a program is being driven, ultimately by a human being whose activity set constitutes behavior. Thus, we can say that this user behavior induces characteristic behavior on the part of the program. Further, when users are behaving perfectly normally, this normal user behavior induces normal or nominal behavior on the part of the program. As we come to understand the nominal behavior of a system as it is executing its customary activities we can *profile* this nominal system behavior quite accurately. Departures from the nominal system profile represent potential malicious activity on the system.

Some unwanted activity may be understood from previous assaults on the system. We can store profiles and recognize these activities from our historical data. What historical data cannot do is to permit us to recognize new assaults. An effective security tool would be designed to recognize assaults *as they occur* through the understanding and comparison of the current behavior against nominal system activity. This can only be accomplished if the program is instrumented and its activity is closely watched. Monitoring and controlling program execution at run time through behavioral control is the missing piece in the security puzzle.

2. Current state of the art

In the current state of the art of intrusion and misuse detection, the software system is regarded as a black box. The misuse determination is made either on the basis of data inputs to the system or data outputs from the system. For any given program, the set of all possible input data is enormous. This holds true for all software: applications, servers, compilers, and operating systems. The set of all valid input is usually substantially smaller than the total input set. A useful example would be a web server that only knows how to parse input that is in accordance with the HTTP specification. A string of 3000 'a's would not constitute valid input for that web server. The web server will behave differently when handling GET request than when it is passed the huge string of 'a's. The set of invalid input consists of all members of the total input set that are not members of the valid input set.

Each program has a set of potentially destructive, intrusive and malicious input. Rule and signature based intrusion detection techniques exploit this set to spot attacks. These systems scan the input data for occurrences of items from the known malicious set and raise an alarm when a match is found. The discovery of new elements for the malicious set is a difficult and time consuming task. Research has established that variations on known malicious input slip past many signature based IDS tools [8,17]. Therefore each of these variations must then be added to the set of known malicious input. This growth challenges the ability of signature based techniques to scale.

A second drawback to examining the input for instances of malicious data is the "post attack" nature of the discovery process. The set of malicious input is not known a priori for each program. Rather, elements are added to the set as new attacks are observed, studied and de-constructed. Attacks that security professionals do not get to study may never result in additions to the known malicious input set. Therefore, the discovery process is dependent on successful attacks, and thorough forensic work following these attacks. This leads to a race between the attackers and the security community.

Intrusion detection tools can monitor the output data for matches against the known malicious output set. These tools raise an alarm if a match is found. By adding fingerprint type patterns to data that should be kept private, these tools can also function as a simple access control list, raising the alarm when the sensitive data are leaked. Tools that function this way have the same primary limitations as tools that look at the input data to programs. The information provided is useful but not timely if the goal is to prevent attacks from succeeding. Knowing that an attack has succeeded may better than not knowing at all, but the information is not available in time to take action.

From our perspective, the input to a program determines the execution behavior of the program. In fact, a specific change in execution behavior to compromise the system in some way is the goal of many attacks and exploits. The ability to identify these behavioral changes would be valuable. Even more valuable would be the ability to stop these behavioral changes. Preventing these changes would thwart many types of attacks.

3. The software behavioral model

In order to understand running software it will be necessary to build a metaphor that describes what the software is doing in relation to the user who is causing it to perform useful work. Software systems are constructed to perform a set of operations for their customers, the users. An example of such an operation might be the activity of adding a new user to a computer system [1]. At the software level, these operations must be reduced to a well-defined set of functions. These functions represent the decomposition of operations into sub-problems that may be implemented on computer systems. The operation of adding a new user to the system might involve the functional activities of changing to current directory to a password file, updating the password file, establishing user authorizations, and creating a new file for the new user. During the software design process, the basic functions are mapped by system designers to specific software program modules. These modules will implement the functionality.

From the standpoint of computer security, not all operations are equal. Some user operations may have little or no impact on computer security considerations. Other operations, such as, system maintenance activities, have a much greater impact on security. System maintenance activities being performed by systems administrators would be considered nominal system behavior. System maintenance activities being performed by dial-up users, on the other hand, would ***not*** be considered nominal system behavior. In order to formalize this decomposition process, a formal description of these relationships will be established [5].

Software systems are generally designed to implement a set of functional requirements or functionalities *F*. Thus, if the system is executing a functionality $f \in F$ then it cannot be expressing elements of any other functionality in *F*. Each of these functionalities in *F* was designed to implement a set of business requirements. From a user's perspective, this software system will implement a specific set of operations, *O*. This mapping from the set of user perceived operations, *O*, to a set of specific program functionalities is one of the major functions in the software specification process. In Table 1 we can see how two hypothetical user operations are mapped by this process onto a set of four functionalities. In this table there is a T in the intersection row and column for operation, o_1, and functionality f_1 indicating that functionality f_1 implements requirement (operation) o_1.

Table 1. Mapping operations to functionalities

$O \times F$	f_1	f_2	f_3	f_4
o_1	T	T		
o_2		T	T	T

From a computer security standpoint, we can envision operations as the set of services available to a user (e.g., login, open a file, write to a device) and functionality as the set of internal operations that implement a particular operation (e.g., user-id validation, ACL lookup, labeling). When viewed from this perspective, it is apparent that user operations that may appear to be non-security relevant may actually be implemented with security relevant functionalities (sendmail is a classic example of this, an inoffensive operation of send mail can be transformed into an attack if the functionalities that deal with buffers can be overloaded).

Table 2. Mapping functionalities to modules.

$O \times F$	m_1	m_2	m_3	m_4	m_5	m_6
f_1	T	T		T		
f_2	T		T		T	
f_3	T		T			T
f_4	T		T		T	

The software design process is strictly a matter of assigning functionalities in *F* to specific program modules $m \in M$ the set of program modules. The actual granularity of the notion of a module is not a significant factor in this discussion. The granularity should be sufficient to provide the resolution needed for subsequent measurement purposes. For our purposes in this discussion, a module will be a C function. The design process may be thought of as the process of defining just how the functional requirements will be implemented is a set, M, of program modules. In Table 2, we can see the hypothetical mapping of the set of four functionalities established in Table 1, to a specific set of program modules.

As a program executes it will make many transitions from module to module. Each of these transitions will represent one epoch, or time increment. For the purposes of measurement, we are interested in the relative frequency of execution for each module during a fixed number of epochs or dump interval. We will accumulate these data in a module profile. A module profile for an n module system will be an n element vector. Each element of the vector will contain a tally of the number of instances that each module has executed during the current dump interval.

We can see that there is a distinct relationship between any given operation, *o*, and a given set of program modules. That is, if the user performs a particular operation then this operation will manifest itself in certain modules receiving control. We can tell, inversely, which program operations are being executed by observing the pattern of modules executing, i.e. the module profile. In a sense, then, the mapping of operations to modules and the mapping of modules to operations is reflexive.

It is a most unfortunate accident of most software design efforts that there are really two distinct set of operations. On the one hand, there is a set of explicit operations O_E. These are the intended operations that appear in the Software Requirements Specification documents. On the other hand, there is also a set of implicit operations, O_I, that represent unadvertised features of the software that have been implemented through designer carelessness or ignorance. These are not documented, nor well known except by a group of knowledgeable and/or patient system specialists, called hackers.

The set of implicit operations, O_I, is not well known for most systems. We are obliged to find out what they are the hard way. Crackers and other interested citizens will find them and exploit them. What is known is the set of operations O_E and the mappings of the operations onto the set of modules, *M*. For each of the explicit operations there is an associated module profile. That is, if an explicit operation is executed, then a well defined set of modules will execute in a very predictable fashion. We can use this fact to develop a reasonable profile of the system when it is executing a set of operations from the set of explicit operations. We can use this nominal system behavior to serve as a stable platform against which we may measure intrusive activity. That is, when we observe a distribution of module profiles that is not representative of the operations in O_E then we may assume that we are observing one or more operations from the set O_I; we are being attacked.

When a program is actually executing, we will observe its behavior from two different sources. The first behavioral aspect is the mapping between the user operations and the sequence of program module executions. So if a user executed the following sequence of operations from the set shown in Table 1, { o_1, o_1, o_1, o_2, o_1, o_2} then we might observe the following execution sequence of modules, {m_1, m_2, m_1, m_3, m_1, m_2, m_1, m_6, m_1, m_5}. The point, here, is that there is that there is a distinct relationship between what the user is doing, the sequence of module executions that we can observe. This constitutes the ***behavior*** of the program.

The second behavioral aspect of program execution has to do with the way that program modules interact when the program is executing. We can see, for example, that functionality f_3 is implemented in modules m_3 and m_6. We are very interested in the way that m_3 and m_6 are invoked in the implementation of that functionality. It may well be that m_3 calls m_6 and whenever m_3 is invoked m_6 is also always invoked. In this case there is little or no information in the call to m_6 and we can learn to ignore it. A very different circumstance arises when module m_3 is sometimes called when f_3 is invoked and other times only m_6 is called. In this case, the operation of the two modules is almost completely independent.

The behavioral data on the interaction of program modules can be gleaned from the profiles of module execution. The sequences themselves do not well disclose system behavior. To extract the actual behavioral data we will tally the frequency of execution of each module during a fixed number of program epochs, or module calls. In this case, at fixed intervals we will dump a profile vector containing the frequency of module executions. This vector will contain the essential information as to the precise nature of specific module interactions. It is the interaction of the modules that reveal behavior, not the sequences module executions.

4. The missing piece in the security puzzle

In our approach to anomaly detection or software misuse, we are interested in the behavior of the software system. We actually open up the running software system and measure its behavior while it is running. In this context the data may be seen as stimuli for the program that exhibits some activity in response to each datum. Data in the normal range of user activity will induce normal behavior on the software. Data outside of this range will induce different or unusual behavior on the system that may be readily observed. In this approach the focus shifts from trying to model and understand the data space to which a program may be subjected to the behavior of the program in response to the data input.

The central issue here is that once we understand the notion that a program exhibits behavior that can be measured, we can begin to assert behavioral control on the program execution. We can certify certain behaviors as nominal and reject behaviors that are outside of this range. We can do this in real time because we are monitoring the activity of the system in real time.

Control is the central issue in computer security. It has long been accepted that data control in the form of encryption is a necessity to preserve the integrity of information flowing from one agency to another. Controlling access to system resources has also shown great value for imposing a security regime.

Access control has been used over time as a means of attaining some modicum of security. In the middle ages, castles were constructed to limit the access of marauding bands of itinerant soldiers to the populace of a region. These castles were effective if and only if they were sufficiently strong. This made them a nightmare for the occupants. The castles were cold, drafty and very restrictive in terms of the movement of their inhabitants. With the advent of the trebuchet and the cannon, even these imposing and uncomfortable structures became obsolete. There are other real good examples of failed access control in the Great Wall of China or the Maginot Line constructed before World War II to defend France against the Germans. Access control is a deterrent but not a solution. Trusted computer operating systems have all of the user comfort of medieval castles. They are a classical example of the castle architecture carried into computer operating system design.

The big downfall of all access control technology is the inherent vulnerability built into the system from the start. The Maginot Line did not surround France. The Germans simply went around it to the north when the time came to invade France. All software systems contain similar vulnerabilities. They work well as long as the enemy is cooperative and does not exploit the intrinsic vulnerabilities in the system. For very large software systems, however, it is virtually impossible (and completely unnecessary) to know and remove all vulnerabilities. Access control is useful to build specific well understood defenses around specific system resources such as files or system services. That is all.

The missing piece of the computer security puzzle is that of behavioral control. It is simply not possible to build systems that are free from vulnerabilities. That should never be an objective of software development. Normal users of a system doe not exploit vulnerabilities. Only the deliberate misuse of systems will exploit vulnerabilities. This misuse can be detected and acted on immediately. To demonstrate this concept we have tested a web server that contains perhaps one of the most vulnerable builds of RedHat 6.2 Linux running on the Internet which we have invited people to attack. With our technology in place we have bee able to identify these assaults and stop them before they can exploit the known vulnerabilities in this operating system.

5. Watcher

The Disallowed Operational Anomaly computer security solution is a technology based on our behavioral control methodology. The DOA methodology is embodied in the Watcher for Linux product. The Linux operating system was selected as the first expression for this technology for a number of reasons. The principle reason was that it was a sufficiently large and complex piece of software. While the core DOA technology is applicable to any piece of software, the Linux kernel provided a good demonstration ground. Implicit in this is the fact that the source was available so that it could be easily instrumented. We chose to instrument an OS kernel because doing so imposed a high reliability requirement on the instrumentation and profiling process we developed. When instrumenting other applications, such as `apache`, there is not such a high reliability requirement.

The Linux kernel source was altered in four ways. First, about 3300 instrumentation points were inserted throughout the source for the kernel. These instrumentation points, or sensors, are the source of the execution behavioral data that is stored into the baseline and measure. The sensors placed in code are principally used to determine whether a given code segment has been reached [cf. 14]. Secondly, a few elements were added to the task`_struct` and `sk_buff` structures in the kernel. These were employed to identify the cause of the behavior observed at the instrumentation points. Thirdly, code was added to start the profiling process when a process is executed or an IP packet is handed from the device driver to `ip_rcv()`. And finally, a quick check was added in `ip_rcv()` against a table of banned IP addresses. This permits packets to be dropped from banned hosts very early.

Software execution can be observed through a variety of techniques. Some techniques are more invasive than others, source instrumentation vs. library interposing, for example. The granularity of execution information available also varies depending on the technique chosen. Some techniques can only provide information at the system call level [cf. 6, 7, 11, 18], others can provide it at instruction level in the monitored program. These techniques are collectively called "sensors" in this document. Sensors are the methodologies for observing information from running software. Sensors provide the necessary telemetry to observe the execution behavior of a program.

5.1. Behavioral tagging

Watching execution behavior through the interaction of the modules in *M* only tells us part of the story. Knowing what behavior or operations in the set *O* are occurring leads directly to the question of attribution. The problems is to know who or what is causing the behavior being observed right now. Several sensor techniques described above lend themselves to tagging the cause of the behavior to the behavioral data. This information is referred to as "tag data." Common tag data from our work with the Linux kernel and library interposing on Solaris 7 are: IP addresses, process IDs, TCP session IDs, socket file descriptors, user IDs, etc.

Source code instrumentation and library interposing are two methods we have used that support tagging rather easily. Implementing behavioral tagging using other sensor techniques may be more difficult. Behavioral analysis becomes a powerful security technique by making use of the behavioral tag data.

5.2. Behavioral baseline

Behavioral data can be accumulated into a set representing a baseline of program behavior. The type of behavior captured in the baseline determines its utility. If the normal behavior of a program is stored in the baseline, then it can be used to detect abnormal program usage. If the baseline is generated by testing the program, then it can be used to detect untested behavior. For this paper, our interest is security rather than reliability; we focus on detecting abnormal program usage rather than untested behavior.

Creating a baseline for normal program usage is simple. The data from the sensors simply needs to be stored. However, for a complex program such as a web server or an operating system, the baseline can become very large. A program's sensors can quite easily produce, on average, half a million data points per minute. At times of high system activity, we have seen an instrumented Linux kernel emit over 50,000 data points each second. Real-time analysis techniques were developed that could deal with the enormous volume of behavioral data.

Obviously the entire collection of behavioral data cannot be used in real-time for analysis without very powerful hardware. A compact model that completely represents the baseline will be required for both speed and brevity. This model must be constructed so that real-time comparisons between it and the profiles emitted by the running program can be performed. To solve this problem, the sensors for each program store data into a profile. The profile has a dump interval that specifies the amount of epoch to gather before the profile is emitted. The sensor in each program module is given a numerical value called a "click ID." These start at 1 and count up incrementally.

5.3. The profiles

For a program that has 100 points of instrumentation, the profile is an array of 100 integers. The click IDs are used as indexes into the profile array. When the path of execution passes over a sensor, the value at `profile[clickID]` is incremented. The profile is handed off for processing once a fixed number of epochs have been recorded into it. Each profile can be viewed as a point in a 100 dimensional space. The baseline then is a collection of points in 100 dimensional space. For real programs, there are usually several hundred points of instrumentation. In the Linux kernel we instrumented, there are just over 3000 points of instrumentation. For the kernel's baseline, this means a collection of points in a 3000 dimensional space.

By treating each profile as the coordinates of a point, we have made the behavior visual, and reduced the bandwidth of behavioral data emitted by the program. While working in the 3000 dimensional space is easier from a bandwidth point of view, it is still too computationally intensive for real-time application. We must reduce the dimensionality to a manageable level for this approach to work.

5.4. Problem simplification

By looking at the data in each profile and the corresponding instrumented source code it is clear that certain modules always are invoked together. Modules identified by click IDs of 7, 8, 13, and 74 may always be called together for example. Through the use of a statistical filter we are able to establish a mapping vector that maps each actual program modules to a virtual module in a much smaller profile. This mapping process is shown graphically in Figure 1. For the Linux kernel, the virtual profile tends to have between 80 and 120 virtual modules. This means that Watcher is not processing the profiles from the 3300 points of the Linux kernel but is processing the set of much small dimensionality, the virtual profiles. The size of the virtual profile depends on the variety of different tasks performed by the program. In general, single purpose programs have smaller virtual profiles than general purpose programs whose behavior repertoire is much larger. The underlying structure of the virtual modules will depend very much on the diversity of the activity performed. In general the larger the set of operations actually selected by the user from *O*, the greater the number of virtual modules

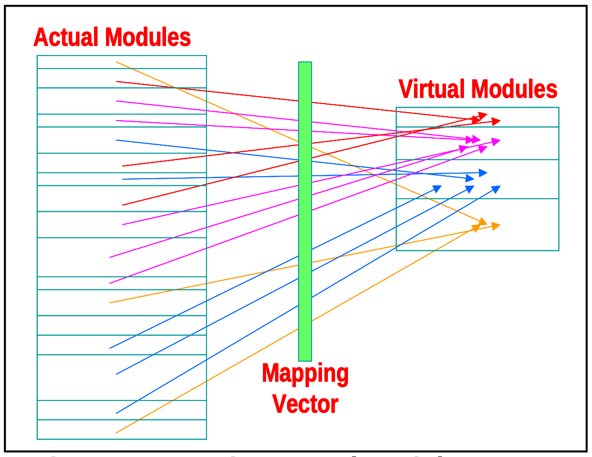

Figure 1. Mapping Actual Module Counts to Virtual Modules

5.5. Creating a model of normal behavior

Each virtual profile can be viewed as a point in an approximately 100 dimensional space. By plotting these points we observe something rather remarkable. The points form natural clusters. The complete plot of the baseline represents all of the behavior from the baseline generation period. The entire cluster can then be represented by its center and some radius, epsilon, about the center point. These are called centroids. By storing the mapping vector and the list of centroids, we are able to represent the behavior from the baseline very succinctly.

The size of the model is determined primarily by the program's range of behavior, rather than the amount of data collected in the baseline. The model representing the behavior of the Linux IP stack is roughly 30k, even when the baseline ranges from 2 to 60 MB. These centroids allow Watcher to represent nominal behavior very succinctly and thus enable real-time comparisons with new behavior. The actual centroids representation for a typical calibration of the Linux kernel is shown in the Figure 2. This succinct representation of normal behavior permits the rapid computation of distances for new emerging virtual profiles.

5.6. Behavioral Measurement

The behavioral baseline will serve as a reference point to identify when a program is behaving abnormally. Abnormality, however, is not a simple binary condition. Rather, it is a continuous function. What is needed is technique for measuring the abnormality of a profile when compared against the baseline data.

As each profile is emitted by the sensors, the virtual profile for it is a point in the same space as the model built from the baseline. The distance between the new point and the closest centroid can be calculated. This distance is a scalar measurement of the normality of the behavior stored in the profile. If the distance is less than the epsilon radius used in creating the model, then the behavior is normal. If the distance is greater than the epsilon radius, then the distance answers the question of the how abnormal the behavior was. When an attack changes the behavior of the program, the module sensors emit a profile whose distance is greater than the epsilon. When testing attacks that would normally succeed, these attacks impact the behavior of the targeted program dramatically. They are very visible.

The distance values describe the normality of the current program behavior. By measuring the behavior of the program we open the door for enforcement of normal behavior. Ideally, an administrator should be able to force all important programs to execute in their normal, approved manner. When an attack cannot change the behavior of the targeted program, the attack fails. Stopping the attack as it starts is the goal of behavioral control. Doing this requires two additional steps, establishing thresholds and defining control policy.

As long as the distances are less than the epsilon radius, the behavior of the program is normal. Administrators may choose to label distances slightly greater than the epsilon normal also, because they are nearly normal. A threshold distance can be established which separates normal and allowed behavior from abnormal behavior. Enforcing normal behavior requires that abnormal behavior be stopped.

The threshold value is a variable under the control of the administrator. In periods of heightened threat, the threshold can be lowered for more rigid behavioral control. In periods of reduced threat, the threshold can be raised, creating a less restrictive, but less secure, behavioral environment for the program.

5.7. Policy

Each profile contains tagging information in addition to the behavioral information. After calculating the distance the tag information becomes very useful. If the distance for a profile is over the anomaly threshold, the behavior may need to be stopped. If the tag is a process ID, the process can be stopped or killed by sending it a signal. If the tag is a socket's file descriptor, `shutdown(2)` can be called to close the socket. If the tag is a source IP address, incoming packets from that IP can be ignored in a number of ways.

The actions taken in response to anomalous behavior are determined by a variety of factors. For critical programs or servers, the responses chosen will probably be draconian. The responses are controlled by the administrator. The behavioral measurement techniques we have developed address the question of where and when to take action.

This approach can spot and stop attacks that share two key characteristics. First, the attack must effect the execution of the program. Second, the attacked program must generate enough behavior to fill more than a single profile. If an attack can completely express itself in less than a single profile, our approach will spot the abnormality and the cause, but will not be able to prevent the attack. However, we have yet to observe any successful attacks that can completely express themselves in a single profile.

If an attack is simply ignored by the program, then its behavior is not likely to be effected. If an attack does not require that the execution of a program be changed, then behavioral analysis is unlikely to spot it. For example, `ftpd` does not behave differently when downloading `/etc/passwd` as opposed to any other file, say,`/tmp/harmless`. Other security techniques address the problem of unauthorized but programmatically allowed behavior. Access Control Lists (ACLs) seem to be the most powerful method for addressing this class of security problem.

6. Experimental Results: Victim

For experimental purposes, we have chosen to demonstrate the capabilities presented through execution behavioral measurement and control by connecting a highly vulnerable version of the Linux kernel on a computer connected to the Internet. We then published the URL for the machine to various sites to attract the attention of crackers to the machine. As additional incentive we published the fact that we would ship the computer to the first person who succeeded in rooting the operating system on this computer. The challenge provided intensive interest from a number of commercial organization and individual crackers.

To prepare victim for its role as a web server, we generated an "Everything" install of RedHat 6.2; installed Watcher kernel plug-in with the 2.2.18 kernel; put a test web site on victim; requested pages from the test web site; turned on nearly every service listed in `inetd.conf`; set up `cron` jobs to restart services that get killed; set up `cron` jobs to build log summary files; and set up a `cron` job to force `httpd` and `inetd` to be restarted. While other security systems rely on either specific or general knowledge about attacks, our approach is very different. We calibrated the system for its normal behavior as a web server and also some standard system administration activity are what constitute normal behavior. The behavior of the system was compared in real-time against the behavior expressed by the baseline.

Watcher was the only security control on victim. Victim demonstrated that this technology is not only viable, but directly applicable when it comes to rejecting intrusion attempts. The victim challenge was equivalent to parking a car in New York city with the keys in it, windows rolled down the doors unlocked and open, and the car running with a sign on it saying "Steal Me", and an ad in the paper saying where the car is and to how steal it. In our security metaphor, when an attempt is made to steal this car (trash victim) the car simply vanishes from their sight. The security vulnerabilities

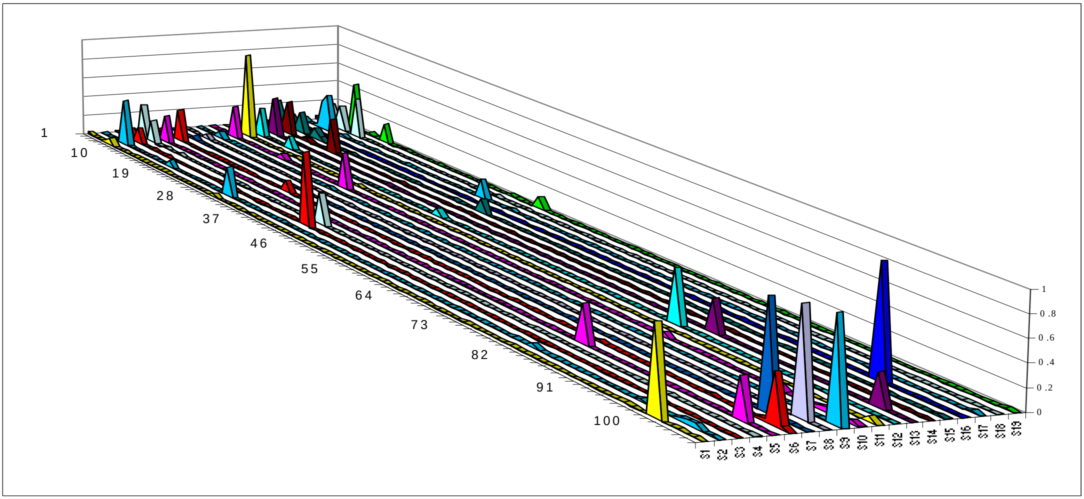

Figure 2. Nominal Behavior of Linux Kernel

are not the problem. Dealing effectively with the assault is the solution.

Anomalous and normal behaviors are a function of the role a system has been deployed in. For victim, the normal behavior is serving web pages. Additionally, it is normal for system administrator to be tailing log files, and building the auto-generated portions of the victim site. That is essentially the extent of "normal" behavior. Abnormal behavior is anything else. The difference between normal and abnormal is not a bold black line though. What is really of interest is the difference between normal behavior, and the behavior currently occurring. This is where the behavioral measurement aspect of our approach applies. Thresholds are established on the allowed difference from normal; anything over the thresholds is stopped. For victim these were clearly set thresholds rather low.

There have now been several thousand assaults on the victim machine. The results of this experiment look like this:

Attack	**Rejection rate**
Port Scans	100%
Buffer Overflows	100%
Worms	100%

Within the domain of behavior control and vulnerability exploits, the software on victim can easily manage 100% of the misuses to which it has been subjected to date.

7. Summary

While the existing paradigms of computer security are still very useful and serve perfectly well in their capacities, there has existed a gap in the computer security space. Our technology and approach fills that gap by providing procedural based intrusion detection and response. We suggest that this gives Watcher the unique ability to detect and halt completely novel attacks that have yet to be seen on the Internet, and better yet, we have the ability to protect the first person to see a new attack or exploit. No one needs to be sacrificed to the new virus or worm anymore.

In essence, we have learned to solve the right problem. Removing all software vulnerabilities is clearly an unsolvable problem. Providing restrictive and onerous barriers to software use makes the software uncomfortable and difficult to use. Monitoring and controlling program execution at run time through behavioral control is the missing piece in the security puzzle. The complete puzzle has three pieces; data control (encryption), access control, and behavioral control.

8. References

[1] J. Alves-Foss, D. Frincke and J. Munson. Measuring Security: A Methodological Approach, *International Workshop on Enterprise Security*, Stanford, CA, June 1996.

[2] D. Anderson, T. Frivold and A. Valdez: Next-generation intrusion detection expert system (NIDES). Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA, SRI-CSL-95-07, May 1995.

[3] M. Bishop: A Taxonomy of UNIX and Network Security Vulnerabilities," M. Bishop, Technical Report 95-10, *Department of Computer Science, University of California at Davis*, May 1995.

[4] D. Denning: An intrusion-detection model. *IEEE Transactions on Software Engineering*, Vol.13, No:2, pp.222-232, February 1987.

[5] S. G. Elbaum and J. C. Munson, "Intrusion Detection through Dynamic Software Measurement", *Proceedings of the USENIX Workshop on Intrusion Detection and Network Monitoring* , Santa Clara, CA, April 1999.

[6] A.K. Ghosh, C. Michael and M. Schatz: A real-time intrusion detection system based on learning program behavior, *Proceedings of theThird International Workshop, RAID 2000* , Springer-Verlag, Toulouse, France, pp. 93-109, October 2000.

[7] A. K. Ghosh, A. Schwartzbard and M. Schatz: Learning program behavior profiles for intrusion detection. *Proceeding of the USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, CA, April 1999.

[8] L. R. Halme and R. K. Bauer: AINT misbehaving - a taxonomy of anti-intrusion techniques. *Proc. of the 18th National Information Systems Security Conference*, pp. 163-172, October 1995.

[9] J. Hochberg, K. Jackson, C. Stallings, J. F. McClary, D. DuBois and J. Ford: NADIR: An automated system for detecting network intrusion and misuse. *Computers & Security*, Vol.12, No:3, pp.235-248, May 1993.

[10] H. S. Javitz and A. Valdes: The SRI IDES statistical anomaly detector. *Proc. of the IEEE Symposium on Research in Security and Privacy*, pp.316-326, May 1991.

[11] A. P. Kosoresow and S. A. Hofmeyr, "Intrusion Detection via System Call Traces", *IEEE Software*, Septemeber/October 1997, pp. 35-42.

[12] S. Kumar and E. H. Spafford: A pattern matching model for misuse intrusion detection. *Proc. of the 17th National Computer Security Conference*, pp. 11-21, October 1994.

[13] S. Kumar and E. H. Spafford: A Software Architecture to Support Misuse Intrusion Detection, *Proc. 18th National Information Systems Security Conference*, pp.194-204, 1995.

[14] The Linux Kernel Instrumentation Project, http://sourceforge.net/projects/kip/

[15] J. C. Munson, "A Software Blackbox Recorder." *Proceedings of the 1996 IEEE Aerospace Applications Conference*, IEEE Computer Society Press, Los Alamitos, CA, November, pp. 309-320, 1996.

[16] A. P. Porras and G. P. Neumann: EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. *National Information Systems Security Conference*, 1997.

[17] M. Sobirey, Richter and H. Konig. The intrusion detection system AID. Architecture, and experiences in automated audit analysis. *Proc. of the International Conference on Communications and Multimedia Security*, pp. 278-290, September 1996.

[18] C. Warrender, S. Forrest, and B. Pearlmutter: *Detecting intrusions using system calls: alternative data models*, IEEE Symposium on Security and Privacy, IEEE Computer Society Press, pp. 133-145, 1999.