

Detection of Computer System Intruder Access via Profiling of Kernel Memory and User Behaviour

Dr. Eric Cole

ABSTRACT

Granting computer account access based on initial authentication does not sufficiently address dangers from insiders or compromises due to software vulnerabilities or exposure of sensitive pass phrases. We present a technique that leverages the benefits of profiling unique aspects such as biological data without the use of specialized biometric hardware to better address this threat. This technique continually inspects the dynamics of the operating system as well as the users' actions to protect a system beyond initial access. Detection mechanisms described in this paper are unique in that they are capable of identifying intrusions even when installed after-the-fact of a compromise. In addition, they do so without the use of "known" signatures with means they are capable of detecting never before seen implementations. To illustrate its practicality, selective components have been prototyped with successful results using the Linux operating system.

KEY WORDS

Intrusion detection, operating systems, insider attack

1. INTRODUCTION

Gaining legitimate access to a device on a computer network typically requires submission of a username and pass phrase at the start of each session. Lax security procedures exercised by those who leave their systems unattended during an ongoing session allow malicious users to gain unauthorized access, an exercise in insider attack. In addition, user accounts can be compromised through vulnerabilities in application and operating system programs, or through the disclosure of sensitive pass phrases via carelessness, keyboard loggers, or network traffic collectors (*i.e.*, *sniffers*). Standard security practices in existence today do not address these scenarios.

One option to address weaknesses associated with this design is to incorporate the use of biometric devices, which base access decisions on something that the user *is* versus something that they *know*. Biometrics is structured around profiling human characteristics such as fingerprints, hand geometry, and speech recognition as compared to traditional methods which generally rely on the possession of a pass phrase or token. Biological characteristics are difficult to forge, which results in high rates of detection. However, biometrics devices consist of specialized hardware that can be expensive as well as cumbersome to operate and interface with. Concerns over human viruses and bacteria have also resulted in an increasing reluctance to physically touch a biometric reader when a large number of people have come in contact with it previously. This collection of detailed

personal physical attributes also brings concerns over user privacy and the potential for misuse of collected data.

Techniques described in this paper present a method to protect against intrusion beyond capabilities in existing solutions. It identifies reliable profile inputs that can be gathered with existing hardware in a manner that is completely invisible to users. By continually gathering these characteristics, authentication of the system can take place silently throughout the entire user session instead of only at the initial login. In addition, through prototype we have demonstrated that all sensing can be accomplished through the use of a single loadable kernel module, so no significant changes are necessary to the operating system.

We will first start by describing the types of threats that this system concentrates on, followed by related work that addresses aspects of this threat, and concluded with our solution and results obtained during testing.

2. INTRUDER THREATS

The continual increase of exploitable software and advancements in attack methodologies [X] has led to an epidemic of malicious activity by hackers and an especially hard challenge for computer security professionals. Compromises of computer systems can generally be observed by manipulation to the operating system itself to hide access, or impersonation of user accounts. Each threat is described more in detail below.

2.1 Operating System Compromises

Operating system compromises are the most sophisticated method of intruder access. They are particularly problematic because they corrupt the integrity of the very tools that administrators rely on for intrusion detection. These attacks are distinguished by the installation of *rootkits*. A rootkit is a common name for a collection of software tools that provide an intruder with concealed access to an exploited computer [1].

Contrary to the name, rootkits are not used to gain root access. Instead they are responsible for providing the intruder with the capability to hide processes, network connections, and files. Rootkits are either implemented at the user application level or within the kernel. Application rootkits are commonly referred to as Trojans because they operate by placing a "Trojan Horse" within a trusted program (*i.e.*, *ps*, *ls*, *netstat*, etc) on the exploited computer. Popular examples of application rootkits include TOrn [2] and Lrk5 [3]. Kernel rootkits consist of programs capable of directly modifying the running kernel itself and are much more powerful and difficult to detect because they subvert the underlying kernel functions. Popular examples of kernel level rootkits include SucKit [4] and Adore [5]. To the user, the behavior and properties of both application and kernel

level rootkits are identical; the only difference between the two is their implementation.

Rootkits are often used in conjunction with sophisticated command and control programs frequently referred to as *backdoors*. A backdoor is the intruder's secret entrance into the computer system that is usually hidden from the administrator by the rootkit. Ddb-ste [6] is an example of a TCP shell backdoor that is launched by an ICMP trigger packet.

2.2 User Account Compromises

A second class of access utilized by attackers is to impersonate legitimate users on computer systems. Identifying the existence of this can be particularly difficult because no files or areas of memory need to be compromised in order to accomplish this. Instead, an attacker must simply have privileged authentication information (i.e., usernames and pass phrases) needed to gain access. The only way to detect this method of intrusion is to be able to differentiate between legitimate access and fraudulent access, which is a difficult, but not impossible challenge.

3. INTRUSION DETECTION

Intrusion detection methods are algorithms of anomaly identification. Models or requirements are established and any deviation indicates an anomaly. Techniques are based on the set of all anomalous instances (negative detection) or all allowed behavior (positive detection) [7]. Much debate has taken place in the past over the benefit of positive versus negative detection methods, and both have been implemented with reasonable success [8].

3.1 Negative Detection

Negative detection models operate by maintaining a set of all anomalous (non-self) behavior. The primary benefit to negative detection is its ability to function much like the biological immune system in its deployment of specialized sensors. However, it lacks the ability to *discover* new attack methodologies. Signature based models such as Chkrootkit [9] are implementations of negative detection. Chkrootkit maintains a collection of signatures for all known rootkits (application and kernel). This is very similar to mechanisms employed by popular virus detectors. Although successful against them, negative detection schemes are only effective against known rootkits. This means that these systems are incapable of detecting new rootkits that have not yet had signatures distributed. Also, if an existing rootkit is modified slightly to adjust its appearance it will no longer be detected by these programs. Users of this type of system must continually acquire new signatures to defend against the latest rootkits which increases administrator workload rather than reduces it. Because attacks in the wild evolve rapidly, this solution will never be complete and users of this model will always be behind offensive technologies.

3.2 Positive Detection

Positive detection operates by maintaining a set of all acceptable (self) behavior. The primary benefit to positive detection is that it allows for a smaller subset of data to be stored and compared, however accumulation of this data must take place prior to an attack for integrity assurance.

One category of positive detection is the implementation of change detection. A popular example of a change detection algorithm is Tripwire [10]. It operates by generating a mathematical baseline using a cryptographic hash of files within the computer system immediately following installation (i.e., while it is still "trusted"). It assumes that the initial install is not infected. Tripwire maintains a collection of what it considers to be self, and anything that deviates or changes is anomalous. Periodically the computer system is examined and compared to the initial baseline. Although this method is robust because unlike negative detection it is able to *discover* new rootkits, it is often unrealistic. Few system administrators have the luxury of being present to develop the baseline when the computer system is first installed. Most manage systems that are already loaded, and therefore are not able to create a trusted baseline to start with. This approach is incapable of detecting rootkits "after the fact" if a baseline or clean system backup was not previously developed. In an attempt to solve this limitation, some change detection systems such as Tripwire provide access to a database of trusted signatures for common operating system files. Unfortunately this is only a small subset of the files on the entire system.

Another drawback with static change analysis is that the baseline for the system is evolving. Patches and new software are continually being added and removed from the system. These methods can only be run against files that are not supposed to change. Instead of reducing the amount of workload for the administrator, the constant requirement to re-baseline with every modification dramatically increases it. Furthermore, current implementations of these techniques require that the system be taken offline for inspection when detecting the presence of kernel rootkits.

4. METHODS OF PROFILING

The concept of utilizing behavioral profiling such as keystroke dynamics as an authentication system has had much attention in the field of biometric and computer security. One of the first studies related to the behavioral aspect of keystrokes occurred in 1980 when Gaines et al. [2] conducted an experiment in which seven secretaries typed the same three 300-400 lowercase word blocks of text two different times separated by four months. Unfortunately the sample count for this study was small, and even all of the seven were not able to participate in both sessions which left the experimenters with only 11 typing samples in the end. Latencies between the typing of adjacent keys was analyzed and merged into 27 (the number of lowercase letters on a keyboard plus the space key) x 27 digraphs. A latency digraph is simply a representation of the time that it takes between the typing of two keys. For example, figure 1 demonstrates an

observation and measurements between two users as they type the word “authentication”.

To condense the results, Gaines et al [2] only included the digraphs that were repeated 10 or more times per individual per sitting. A two-sample t-test was carried out over the data from both sittings and the experimenters found that the two samples had a pass rate between 80% and 95%. In addition, the experimenters used this same data to demonstrate the potential of keystrokes as an authentication mechanism. Using the sessions and digraphs described above, 55 tests were conducted from persons desiring to be authenticated to the system. The impostor pass rate was zero and the false alarm rate was about 4%. Following analysis, the experimenters discovered that the core digraphs *in*, *io*, *no*, and *ul* yielded perfect results with zero impostors and zero false positives [2]. Although this was a tremendous milestone, these results cannot be considered as the basis for an extensive system because their sample population was too small and their required collection data was too large.

Following these early results Leggett, Williams, and Umphress conducted similar studies with 17 programmers [3] [4] [5] with the addition that they measured both the mean of all keystrokes latency as well as digraph latencies as discussed above. Unlike the results of Gaines et al. [2], Leggett and Williams [3] found in tests using 12 different digraph selections including the experimentation of the inclusion and exclusion of the space key that their best results were obtained by using all 27 x 27 digraphs. In fact, their results using the core digraphs identified by Gaines et al. [2] instead resulted in an impostor pass rates above 17% and a false alarm rate above 30% [3]. Their best rates using all digraphs and a maximum latency of 500msecs yielded an impostor pass rate of 5% and a false positive rate just over 5%. While these best rates are substantially better than the rates derived using the core digraphs, they are still too low for general practice. In addition, both experimentation methods required the users to type an unreasonable minimum of 300 characters per each request for authentication.

A patent introduced by Garcia [6] presents one solution to the impractical requirement of large data sets.

In the apparatus described in this patent, only the keystroke dynamics of the username are considered. The Mahalanobis distance function is used to determine if the supplied username data set passes based on the corresponding reference data. Unfortunately no evidence is provided on the accuracy of this method, but it is claimed that the impostor pass rate is 0.01% and that the false alarm rate is 50%. Similarly, Joyce and Gupta [8] experiment with a slightly more complex name based method that requires a modified login procedure. Their data reference points include a mean latency associated with the typing of the username, password, first name, and last name of a subject. Prior to authentication they require each new user to type the above data points eight times. Latencies from a submitted authentication request are compared to the mean reference signature and the magnitude of the difference is determined. When this L_1 norm is within the threshold variability of the reference signature than access is granted. This research introduced the key concept that users with very little variability in their signatures should have low thresholds for acceptance against authentication requests. They report an impostor pass rate of less than 1% when the impostor was supplied with the subject’s username, password, first name, and last name. The false alarm rate was just over 13%.

A second enhancement came from Young and Hammon [8] in which they propose that identification should take place continuously as a dynamic component instead of just initially as a static function. In addition, they include additional data points such as the speed in which a user types common keywords, the speed in which a user types a pre-defined number of keystrokes, etc. The Euclidean distance between the sample and reference vectors is then compared. However, no information is provided on the implementation specifics or its accuracy.

More recently, building off of this dynamic aspect many neural network approaches such as Primeaux et al. [9], Bleha et al [10], Brown and Rogers [11], and Alexandre [12] have been presented. However, with the

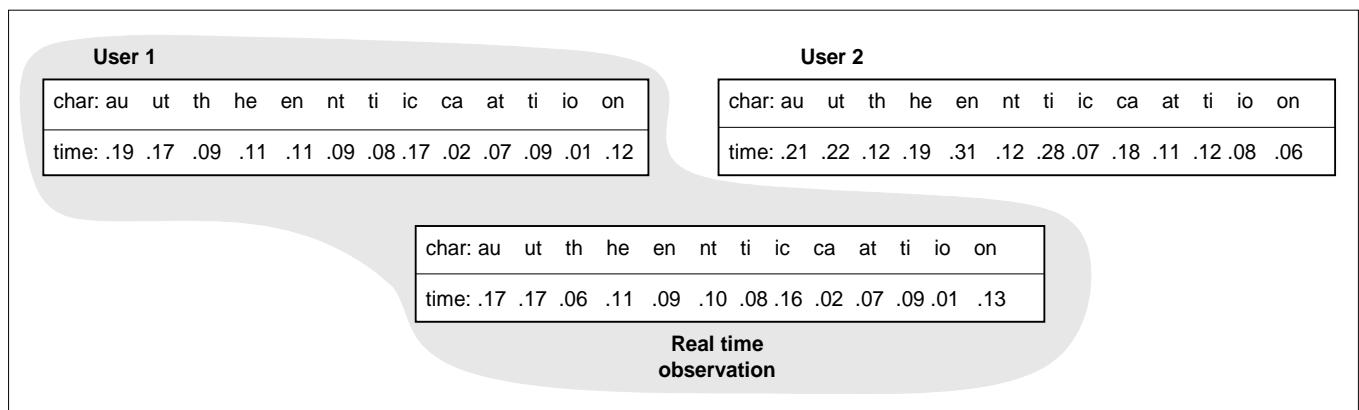


Figure 1. Comparison between previously gathered digraphs for two users show that observation values more closely match those of User 1

inherent limitation of neural networks such as dilution, speed, and retraining present obstacles for large-scale usage [13].

5. PROFILING – OUR APPROACH

The approach outlined in this paper is to expand the concept of profiling to build an intrusion detection system that is specialized to focus on the threat of insider attack. It does this by monitoring the behaviour of both the operating system and the users that access it.

We invisibly collect sensing information from within the kernel of the operating system and from the behaviours of the users as they go about their normal activities. Changes in either indicate a potential compromise. The larger the number of anomalies detected, the greater the certainty that a compromise has taken place.

Conceptually our system is comprised of two components: an operating system profile monitoring module and a user behaviour observation system.

5.1 Operating System Profiling

Determination of an acceptable profile within the operating system is based on the following five anticipated behaviors:

- 1: All kernel calls should only reference addresses located within normal kernel memory
- 2: Memory pages in use indicate a presence of functionality or data
- 3: A program executing on the processor should be visible in user space
- 4: All unused ports can be bound to
- 5: Persistent files must be present on the file system media

Any noncompliance indicates the presence of either an application or kernel modification (i.e., a rootkit) that is attempting to alter the integrity of the operating system by changing its behavior. Implementation approaches of each within the Linux operating system is described in detail in the following sections.

5.1.1 System Calls

The system call table is composed of an indexed array of addresses that correspond to basic operating system functions. Because of security restrictions implemented by the x86 processor, user space programs (ring 3) are not permitted to directly interact with kernel functions for low level device access (ring 0). They must rely on interfacing with interrupts and most commonly, the system call table to execute (see figure 2).

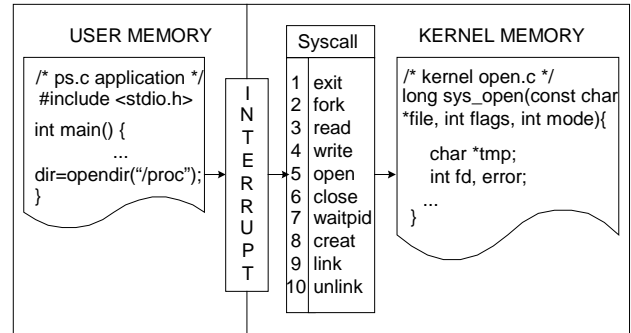


Figure 2. User to kernel function transition.

When the user space program desires access to these resources in Linux an interrupt 0x80 is made and the indexed number of the system call that corresponds to the desired function is placed in a register. The interrupt transfers control from user space to kernel space and the function located at the address indexed by the system call table is executed. This transition can be seen by observing trace or truss as demonstrated below:

```
brk(0x805d000) = 0x805d000
open("/bin", O_RDONLY | O_DIRECTORY) = 3
fcntl64(3, F_SETFD, FD_CLOEXEC) = 0
```

Most kernel level rootkits operate by replacing the addresses within the system call table to deceive the operating system into redirecting execution to malicious functions. The result is a lack of integrity across the entire operating system. Anomalies in the system call table can be detected by first deriving a non-biased address for the table through pattern matching a CALL following an interrupt 0x80 request as demonstrated below:

```
p = (void *)int80;
for (i=0;i<50;i++) {
    if ((p[0] == '\xff') && (p[1] ==
        '\x14') && (p[2] == '\x85')) {
        (ulong *)sct = *(ulong *) (p+3);
        break;
    }
    ++p;
}
```

This is necessary to ensure that the addresses retrieved from the system call table are authentic and are not based on a mirror image of the system call table maliciously created by an intruder. Once this address has been acquired, the system uses generalized positive anomaly detection based on the first derived law of the operating system that all kernel calls only reference kernel memory. On Linux the starting address of the kernel itself is always located at 0xc0100000 [12]. The ending address can be easily determined by the variable `_end` and the contiguous range in between is the kernel itself. Although the starting address is always the same, the end changes for each kernel installation and compilation. On some distributions of Linux this variable is global and can be retrieved by simply creating an external reference to it, but on others it

is not exported and must be retrieved by calculating an offset based on a global variable. In this implementation the global variable `__strtok` which is one address below `_end` in memory was used as the bases for the offset. Below is an example of anomaly detection as well as the offset calculation for `_end`:

```

ulong *end = (ulong *)&__strtok + 0x1;
for (i = 1; i < table_size; i++)
    if ((uint)sys_call_table[i] > *end)
        // deviation found
    
```

The name of the flagged system call is displayed along with the address that it has been redirected to. Following complete analysis of the table the system determines an estimate for the range of the redirection by determining the highest and lowest patch address. This range is identified as a subset of address space within a malicious kernel rootkit. The range calculation will identify any rootkit that dynamically patches into the system call table.

This model errors on the conservative side and will not detect changes in addresses that are physically located within the kernel memory itself. To do so, an intruder would have to overwrite actual kernel functionality in memory, or replace the kernel itself.

However, theoretically not all kernel rootkits must patch over the system call table to alter operating system behavior. System calls serve as pointers to functionality needed by applications. An intruder could design a system capable of changing the functions themselves. The first anticipated behavior that the kernel should not directly make calls outside of kernel space still holds, and call graphs that originate with the system call should not lead to any dynamically allocated memory outside of the normal range (see figure 3).

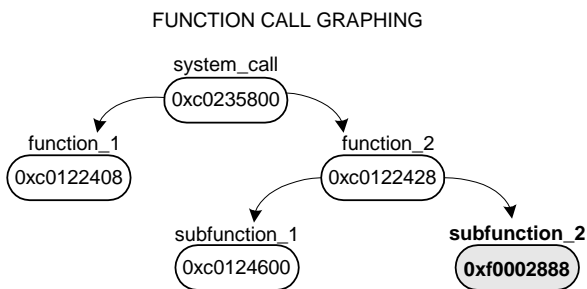


Figure 3. Call graphing to identify malicious functions

In this case, the investigation begins with a `system_call` and identifies that `subfunction_2` has been patched into a new memory range. In addition, page tables can be analyzed individually to identify anomalous behavior that violates anticipated behaviors.

5.1.2 Loadable Modules

As kernel modules are loaded on the operating system they are entered into a linked list located in kernel virtual memory used to allocate space and maintain administrative information for each module. The most

common technique for module hiding is to simply remove the entry from the linked list (see figure 4).

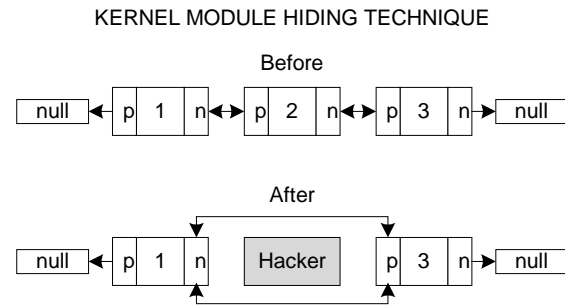


Figure 4. Method to unlink kernel modules for hiding.

Removing the entry does not alter the execution of the module itself, but prevents an administrator from locating it. Even though the module is unlinked it remains in the same position in virtual memory. This estimated physical location falls within a predictable range that is based on a function of the page size, alignment, and size of all previously loaded modules (see figure 5).

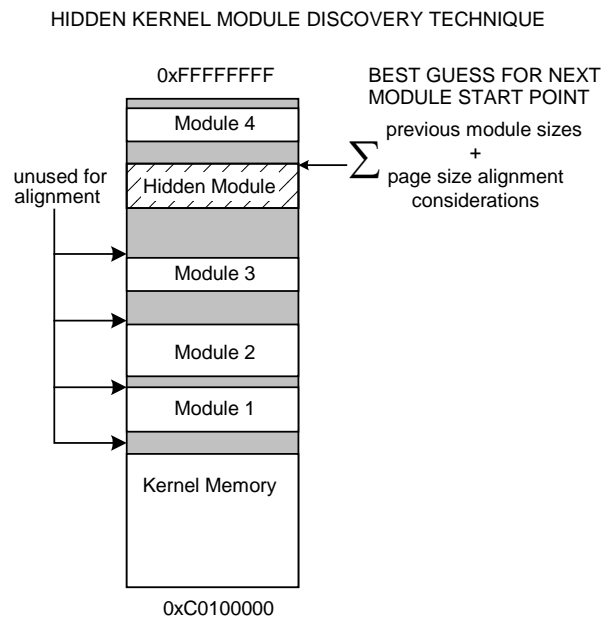


Figure 5. Best guess technique for module prediction

In addition, each module begins with a common structure that can be used to pinpoint the exact starting address of the module within the predicted range. Therefore, even without relying on the kernel's linked list, this calculation can generate a complete listing of all loaded modules. By using the second anticipated behavior that memory usage indicates the existence of functionality or data, virtual memory is searched page by page within this predicted range to identify pages that are marked as "active". There should be no active memory within these pages. Any active pages within the gap that contain a module

structure indicate the presence of a kernel implant that is loaded and executing, but has been purposefully removed from the module list.

This detection method can be expanded in the future by incorporating the ability to search the kernel for other functions that reference this malicious address range (indicating what if anything the kernel module has compromised). This is an example of how the system adapts from a generalized positive detection scheme to a negative detection scheme based on sensed need.

5.1.3 Executing Processes

The system is capable of identifying running processes that are hidden by either user space or kernel space rootkits. It accomplishes this by employing two different sensors. The first sensor is based on the third behavior that a process visible in kernel space should be visible in user space. This sensor executes a ps command to observe an untrusted user space view of the running processes. Following this it manually walks each element in the task structure from the kernel to generate a second view of running processes. The two views are then compared and anomalies are identified which indicate that a process is hidden. This sensor can discover all process hiding techniques that rely on using PID 0, hiding done by system calls, and hiding done by user space rootkits or Trojans.

Although not tremendously stable, it has been demonstrated through implementation that a process can execute without being present in the task queue once it has been scheduled. To detect this hiding technique, a second negative sensor is deployed to investigate the presence of anomalies within process ids that are not present within the task queue. This sensor is based on the second behavior that memory pages in use indicate the presence of functionality or data. Process file system entries are specifically searched one by one to identify the presence of a process in memory within the gap. This detects all process hiding techniques that operate by removing the process from the task queue for scheduling.

5.1.4 Network Connections

Similar to hidden process detection, port detection operates by observing both a trusted and untrusted view of operating system behavior. This model is based on the fourth behavior that all unused ports can be bound to. The untrusted view is generated by executing netstat, and the trusted view is accomplished by executing a simple function that attempts to “bind” to each port available on the computer.

5.1.5 Network Traffic

As a method of redundancy that increases confidence in results and avoids false positives, this system is also capable of detecting when incoming packets are destined to a process that has been hidden. This is accomplished by inserting a packet handler into the stack as demonstrated:

```
remediate.type = htons(ETH_P_ALL);
remediate.func = receive_packet;
dev_add_pack(&remediate);
```

Using this new handler named “remediate”, each incoming TCP packet is briefly inspected to identify the process ID that it is associated with. Identification takes place by acquiring a pointer to the sk buffer using the tcp_hash structure shown below:

```
hash = tcp_hashfn(iph->daddr,
                 ntohs(tcp->dest),
                 iph->saddr,
                 tcp->source);

head = &tcp_ehash[hash];
read_lock(&head->lock);
if (head->chain)
    sk = head->chain;
read_unlock(&head->lock);
```

Packets that are identified as belonging to a hidden process are flagged, and could be terminated automatically if desired by setting the sk->state to TCP_CLOSE. In addition, packets that match this profile could also be recorded to establish additional forensics evidence.

5.1.6 Files

Perhaps the most difficult aspect of rootkit discovery is successful detection of hidden files. This is difficult to implement because there are potentially hundreds of different hiding techniques, file systems do not always “remove” data from the media when a file is deleted, and the increasing storage sizes of media make for a tremendously time consuming search.

The model presented is based on the fifth anticipated behavior that persistent files must be present on the file system media. It operates by first observing a kernel space view of visible files. Each listing is then searched for in user space to determine if it is hidden. For speed, the current implementation conducts searches based on cached entries. In the future additional targeted negative detection sensors will be created and deployed to specifically search in areas that are known to store other malicious data such as the previously detected hidden process, kernel module, or files currently opened by them.

5.2 User Profiling

In addition to threats against the operating system designed to hide intruder access, a detection capability must be capable of identifying threats against user accounts which are not so easily identified. Similar to the profiling described of anticipated operating system behavior, we establish sensors to profile anticipated user behaviors. These profile characteristics include some aspects of past usage time and frequency statistics, but are primarily focused on habitual characteristics that would be difficult for an intruder to forge.

5.2.1 Keystroke Latency

Based on the research past research successes discussed previously [X], initial tests will be conducted using a 26 x 26 digraph which consists of lowercase letters alone. Scoring will be as follows:

$$L = \frac{\sum_{i=1}^{i=n} L_i}{n}$$

where n is the number of tokens in the net

As suggested by Joyce and Gupta [9], acceptance is based on the magnitude of the differences between L and the reference set.

5.2.2 Command Option Selection

The data points discussed above in previous research rely on the mechanics of *how* the user types and not *what* they type. These characteristics are heavily influenced by muscular fatigue, hand injury, and familiarity with the physical proportions of the keyboard interface. However, data collected through keystrokes also presents the ability to perform analysis based on more stable behavioral aspects such as user habits. This research suggests a set of data points based on this content as an avenue for providing secondary dynamic authentication of a user. In this case it is both what is typed and how it is typed. Initially the data points are only be gathered from command line actions and therefore do not take into account behaviors observed during purely textual environments such as word processors and email clients. However, as the experiment continues this authentication system will be applied to graphical environments.

Each analyzed component of data is referred to as a *token*. *Nets* of tokens are gathered each time the user presses the carriage return <CR> key.

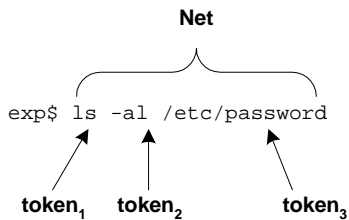


Figure 6. A net of tokens observed from a user

Individual options that are combined together such as in $token_2$ in the example above are treated as unique elements and are thus included as one token. This is done to differentiate between the above example and the logically identical one below by a different user behavior:

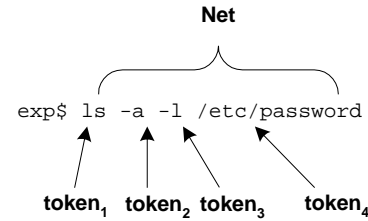


Figure 7. A slight lexical distinction in usage of the same command in a different user profile

The initial experimental data set includes: latency scoring, error scoring, and command scoring. The scores for each data set are combined together along with weights for each defined by individual user variances. Each weight is inversely proportional to the variance so that the score judged to have the most accuracy is given the most influence.

$$V = \frac{\sum_{i=1}^{i=n} S_i \omega_i}{n}$$

Normalized values between .85 and 1 represent continued validation, whereas values below that require a confirmation of identify by the primary means of authentication.

5.2.3 Command Repetition

Commands offer a unique insight into both the desires and habits of a user. For example, perhaps subject₁ repeatedly types “ls -al” out of habit following most commands whereas subject₂ is more apt to type “ls -S”, but only occasionally. Instead the second user primarily types “netscape” with no options. Behavior observed through commands is designed to be the core component of this design. Although initial implementation will be minimal, if results are promising this will be enhanced to include large interconnected characteristics of user activities. As it matures, this analysis data point could easily become a recognized method of secondary authentication and user intent. The potential of this concept is promising because it provides a unique characteristic that is forced as well as stable across users and will likely not be influenced by fatigue or injury like the other methods. In the initial design reoccurring commands, their options, and any relevant sequential data are stored for each user. Frequency statistics as well as relationship scores are awarded to each command that is typed by the user (see figure 8).

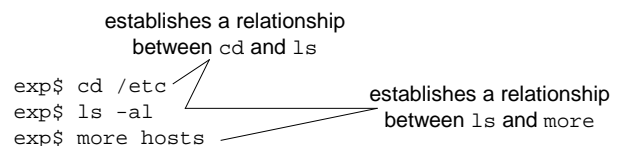


Figure 8. Relationships between commands

During initial testing it is assumed that each net will contain only one command. In addition to token relationships within nets, relationships between the tokens within adjacent nets are stored and analyzed through graphs. (see figure 9).

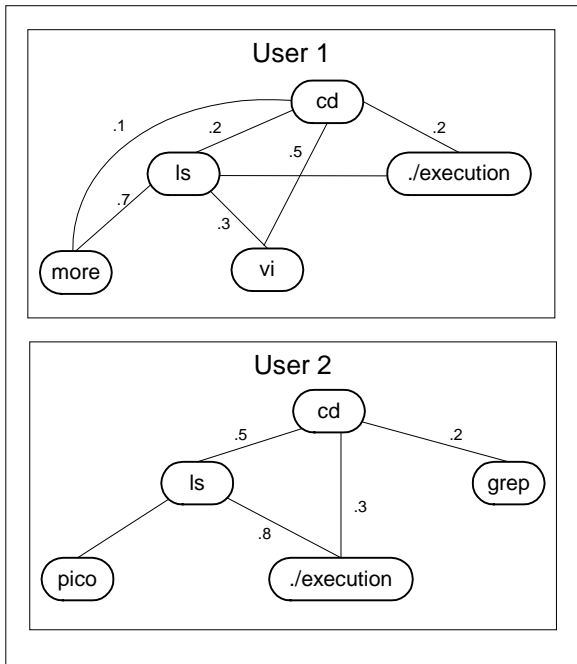


Figure 9. Weighted relationship graphs that differ among users

To maintain these graphs the following elements of data must be stored for each user:

```

struct options {
    // option string
    char[MAXLENGTH] element;
    // count of reoccurrence frequency
    unsigned int frequency;
    // count of session occurrences
    unsigned int count;
    struct options *next;
}

struct command {
    // per element read/write lock
    rwlock_t lock;
    // the command
    char[MAXCOMMAND] action;
    // pointer to the command options
    struct options *options;
    // count of reoccurrence frequency
    unsigned int frequency;
    // count of session occurrences
    unsigned int count;
    struct error *next;
}

struct commands {
    // per structure read/write lock
    rwlock_t lock;
    // user command high score
    unsigned int h_score;
}

```

```

// user command low score
unsigned int l_score;
struct command *entry;
}

```

5.2.4 Spelling and Typing Errors

Reoccurring typing and spelling errors may be a stable source of user behavior. As an example, subject₁ may frequently transpose the command “more” as “mroe”. On the other hand, subject₂ may not have ever mistyped any command within a history of 500 or more logins. Likewise, within documents repetitive or learned spelling errors such as confusion between *i* and *e* ordering in words like “their” may also be used to distinguish one user from another. While errors of this nature are not forced behavior and therefore cannot be used to initialize authentication and grant immediate access, a sudden deviation in user behavior can be used indicate an impostor (see figure 10).

User 1		
Top Misspelled Words	Count	% of occurrence
thier	23	0.23
recieve	10	0.14
persue	5	0.07

User 2		
Top Misspelled Words	Count	% of occurrence
irresistable	2	0.0002
harrass	2	0.0002
concensus	1	0.0001

Figure 10. Habitual misspellings by different users

The primary feature of interest is the similarity between behavior related to the error action and not the error itself. To gather this, a Euclidean distance measure is conducted between the two pattern vectors as defined below:

Let $D = [d_1, d_2, \dots, d_n]$ and $U = [u_1, u_2, \dots, u_n]$ then

$$E(D,U) = \left[\sum_{i=1}^{i=n} (d_i - u_i)^2 \right]^{1/2}$$

The initial data points for the vectors in this set are the number of errors per time period, number of errors per data set, and number of errors per session. Special consideration is applied to reoccurring errors and the following applies a weighted value to the distance calculation based on the frequency of error by the individual and the probability of the error occurring. The score of the Euclidean distance is applied to the entire net of tokens whereas the frequency/probabilities values represent the mean of scores related to the specific tokens within the net.

$$S = E(D,U) \frac{\alpha}{\beta} \text{ where } \alpha = \text{frequency and} \\ \beta = \text{probability}$$

Note that probability is defined by likelihood of occurrence within the sample set and not within the set of available commands.

Alternatively, the weight may be based on the Gaussian probability density of a pattern vector given below:

$$P_i(E) = (2\pi)^{-\frac{n}{2}} |C_i|^{-\frac{1}{2}} \exp \left[-\frac{1}{2} (E - m_i)^t C_i^{-1} (E - m_i) \right]$$

whereas E is the pattern, m is the mean vector, and C is the covariance matrix

Decisions are based on deviations in either direction of the reference cases T as follows:

$$\|S - T\|_1 \text{ whereas } \sum_{i=1}^{i=n} |s_i - t_i|$$

To implement this system the following elements of reference data, T must be stored for each user:

```
struct error {
    // per element read/write lock
    rwlock_t lock;
    // string containing error
    char[MAXVAL] value;
    // count of reoccurrence frequency
    unsigned int frequency;
    // count of session occurrences
    unsigned int count;
    struct error *next;
}

struct errors {
    // per structure read/write lock
    rwlock_t lock;
    // user error high score
    unsigned int h_score;
    // user error low score
    unsigned int l_score;
    struct error *entry;
}
```

Errors within initial testing are limited to command errors which are identified by trapped return values from the command execution. If this is successful, future experiments will include the evaluation of textual errors such as those found within word processing documents and originating emails. It has not been concluded yet if it will be more advantageous to evaluate entire words as tokens or to further subdivide the tokens into smaller core units.

5.2.6 Typing Corrections and Style

Error analysis can be extended to include statistical calculations and pattern matching of correction behaviors as well. For example, correction behaviors may identify that subject₁ generally takes the following actions to type the word **proprietary**:

P r o p r a <bs><bs>r i e t e <bs> a r y

This information is more colorful than the binary decision of if the word was or was not misspelled and therefore may prove to be an interesting and unique data point for user behavior.

5.2.7 Graphical Interface Usage

For users accessing a graphical environment mouse dynamics may also be another potential data point worth investigating. Specially, consider the following habitual behaviors graphs of two subjects as they attempt to correct the spelling of the word “misspelled” in Microsoft Word®. Subject₁ is not a knowledgeable user of the application and therefore:

- 1) uses the mouse to select [Edit] and determines that it is not correct
- 2) uses the mouse to select [Tools]
- 3) uses the mouse to select [Spelling and Grammer]

On the other hand, Subject₂ is an efficient user of the application and therefore has memorized the “shortcut” key [F7] and out of habit uses it to accomplish this task.

Following initial implementation of the design described in this paper and sufficient analysis these methods along with others yet to be considered will be investigated.

6. RESULTS

To demonstrate the effectiveness of profiling we have implemented each operating system sensor with success. Tests were run on computers with standard installations of the Linux 2.4.18-14 operating system. The actual execution of the detection system (not including hidden file detection) took less than one minute to complete and the incorporation of hidden file searching on a 60G drive extended the time to 15 minutes per trial. Tests consisted of repeated instances of a) a clean system, b) user space rootkit installation, c) publicly available Adore version 0.42 rootkit, and d) a newly discovered non-public kernel rootkit. All tests involving a clean system were reported as “non-infected” and no false positives occurred. All tests involving a user space rootkit which trojaned ps, ls, and netstat successfully identified hidden processes, hidden files, and hidden port listeners. No false positives were reported.

Tests involving the Adore kernel rootkit were conducted using both standard system call patching and task queue removal for process hiding. All 15 system calls, all hidden processes (both methods), the Adore

kernel module, and all hidden files were successfully discovered. The port hidden by Adore was 2222, which was discovered. However, because the implementation of Adore physically breaks netstat's ability to output to a pipe, there is no "untrusted" view to compare against. Therefore all bound ports were reported whether malicious or not.

The most important test was conducted against a newly discovered non-public kernel rootkit. Because no signatures have been derived for this rootkit, detection tests conducted by Chkrootkit failed. When tested against our system all seven of the patched system calls were discovered, all hidden processes were discovered, all hidden files were discovered, the module itself was discovered, and all hidden port listeners were discovered. Because this rootkit does not physically break netstat like Adore, no false positive port listeners were reported.

7. FUTURE WORK

In addition to detection, the techniques described in this paper can be leveraged to categorize, recover from, adapt, and remediate against intrusions. For example, the following decision tree demonstrates how this system can be used to classify the sophistication of the method of attack:

1. A user space "ls" is performed
2. The getdents system call is made

The results of actions 1 and 2 are compared and any anomalies between the two indicate that the "ls" binary has been physically trojaned by a user space rootkit.

3. The `sys_getdents()` function is called

Any anomalies between 2 and 3 indicate that the system call table has been patched by a kernel rootkit. The kernel will then be searched for other occurrences of addresses associated with the patched function to determine the extent of infection caused by the rootkit.

4. The `vfs_readdir()` function is called

Any anomalies between 3 and 4 indicate that the function `sys_getdents()` has been patched over by a kernel rootkit.

5. Raw kernel file system reads are made (currently implemented in this release)

Any anomalies between 4 and 5 indicate that `vfs_readdir` or a lower level function has been patched over by a kernel rootkit.

6. Raw device reads are made

Any differences between 5 and 6 indicate that a complex hiding scheme that does not rely on the file system drivers of the executing operating system has been implemented. The same series of decision trees can be built for the flow

of execution of all system calls. In addition, the design will be ported to Solaris, BSD, and Windows operating systems.

8. CONCLUSION

This research discusses a technique to detect against intruder compromises, including sophisticated kernel rootkits. It operates by sensing characteristics about operating system performance and user activity. Behavior is constantly compared to previously identified uniqueness, and any differences indicate a potential compromise. As the number of differences increases so does the confidence that a compromise has taken place.

To demonstrate our techniques, we have implemented profiling within the operating system with tremendous success. Without the use of signatures, this prototype was able to identify all publicly available, and internally developed user space and kernel rootkits. Following this success, we will focus on implementations within user profile which will result in a system that continually monitors against intrusions and impersonations of accounts.

References:

- [1] Sandra Ring and Eric Cole, Detecting and Dealing with New Rootkits. *Sys Admin Magazine*. September 2003.
- [2] T0rn. User space rootkit. j0hnn7/zho-d0h. <http://www.packetstormsecurity.org/UNIX/penetration/rootkits/tk.tgz>
- [3] Linux Rootkit 5. User space rootkit. Lord Somer. <http://www.packetstormsecurity.org/UNIX/penetration/rootkits/lrk5.src.tar.gz>
- [4] SuckIT. Kernel rootkit. Sd. <http://sd.q-art.nl/sk>
- [5] Adore. Kernel rootkit. Stealth <http://www.team-teso.net/releases/adore-0.42.tgz>
- [6] ddb-ste. A callback TCP shell backdoor channel initialized by an ICMP trigger packet. Recidjvo. <http://www.packetstormsecurity.org/UNIX/penetration/rootkits/ddb-ste.tar.gz>
- [7] Anil Somayaji, Steven Hofmeyr, and Stephanie Forrest, Principles of a Computer Immune System, *ACM New Security Paradigms Workshop*, Charlottesville, VA, 1998, 75-82.
- [8] Fernando Esponda, Stephanie Forrest, and Paul Helman, A Formal Framework for Positive and Negative Detection Schemes. *IEEE Transactions on System, Man, and Cybernetics*, 34(1), 2004, 357-373.

- [9] Chkrootkit. Software to check for signs of a rootkit.
<http://www.chkrootkit.org>
- [10] Tripwire. Change identification software.
<http://www.tripwire.com>
- [11] Anil Somayaji and Stephanie Forrest, Automated Response Using System-Call Delays, In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, 2000, 185-198.
- [12] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schröter, and D. Verworner, *Linux kernel programming, third edition* (Addison-Wesley, 2002).
- [1] Polemi, D. *Biometric techniques: review and evaluation of biometric techniques for identification and authentication, including an appraisal of the areas where they are most applicable*. Report prepared for the European Commission DG XIII-C.4 on the Information Society Technologies (IST) (Key action 2: New Methods of Work and Electronic Commerce). 2000.
<http://www.cordis.lu/infosec/src/stud5fr.htm>.
- [2] R. Gaines, W. Lisowski, S. Press, and N. Shapiro, *Authentication by Keystroke Timing: Some Preliminary Results*, Rand Report R-256-NSF, Rand Corporation, Santa Monica, CA, 1980.
- [3] Leggett, J. and Williams G. *Verifying identity via keyboard characteristics*. *Int. J. Man-Machine Studies* 23, 1 (Jan. 1988), 67-76.
- [4] Leggett, J., Williams, G., and Umphress, D. *Verification of user identity via keyboard characteristics*. *Human Factors in Management Information Systems*, J.M. Carey, Ed., Ablex Publishing, Norwood, NJ.
- [5] Umphress, D., and Williams, G. *Identify Verification Through Keyboard Characteristics*. *Int. J. Man-Machine Studies* 23, 3 (Sept 1985), 263-273.
- [6] Garcia, J. Personal identification apparatus. Patent Number 4,621,334. U.S. Patent and Trademark Office, Washington, D.C. 1986.
- [7] James R. Young and Robert W. Hammon. Method and apparatus for verifying an individual's identity. Patent Number 4,805,222. U.S. Patent and Trademark Office, Washington, D.C. 1989.
- [8] Rick Joyce and Gopal Gupta. *Identify Authorization Based on Keystroke Latencies*. *Communications of the ACM*, 33(2):168-176, February 1990.
- [9] David Primeaux, Doraiswamy Sundar, , and Willard L. Robinson, Jr. Usage pattern based user authenticator. Patent Number 6,334,121. U.S. Patent and Trademark Office, Washington, D.C. 2001.
- [10] Saleh Bleha, Bassam Hussien, and Robert McLaren. An Application of fuzzy algorithms in a computer access security system. *Pattern Recognition Letters*, 9:39-43, 1989.
- [11] Marcus Brown and Samuel Joe Rogers. User Identification via Keystroke Characteristics of Typed Names using Neural Networks. *International Journal of Man-Machine Studies*, 39(6):999-1014, 1993.
- [12] Thomas J. Alexandre. Biometrics on Spartcards: An approach to keyboard behavioral signature. *Second Smart Card Research & Advanced Applications Conference*, 1996.
- [13] J. Stader. Applying Neural Networks. AIAI-IR-11, 1992.
<http://www.aiai.ed.ac.uk/project/ftp/documents/1992/92-nnai.ps>