Hiding Intrusions: From the Abnormal to the Normal and Beyond

Kymie Tan[1], John McHugh[2], and Kevin Killourhy[1]

[1] Carnegie Mellon University, Department of Computer Science
Pittsburgh, PA 15213 USA
{kmct,ksk}@cs.cmu.edu,
[2] CERT®
Coordination Center and Center for Computer and Communications Security
Carnegie Mellon University, Pittsburgh, PA 15213 USA
jmchugh@cert.org

Abstract. Anomaly based intrusion detection has been held out as the best (perhaps only) hope for detecting previously unknown exploits. We examine two anomaly detectors based on the analysis of sequences of system calls and demonstrate that the general information hiding paradigm applies in this area also. Given even a fairly restrictive definition of normal behavior, we were able to devise versions of several exploits that escape detection. This is done in several ways: by modifying the exploit so that its manifestations match "normal," by making a serious attack have the manifestations of a less serious but similar attack, and by making the attack look like an entirely different attack. We speculate that similar attacks are possible against other anomaly based IDS and that the results have implications for other areas of information hiding.

1 Introduction

For some time a primary dictum of the intrusion detection field has held that anomalous and intrusive activities are necessarily equivalent.[1] Insofar as we have been able to determine, most previous activity in the anomaly based intrusion detection area has concentrated on demonstrating that anomalous manifestations, detectable by whatever detection scheme was being used, often occur at the same time an intrusion is being carried out. This has led many researchers in the intrusion detection field to assume that anomaly detection is the same as intrusion detection. As a consequence, many investigators have failed to examine the underlying causes and characteristics of the anomalous behaviors that

[1] This view is clearly enunciated by Dorothy Denning [1] who said:

> The model is based on the hypothesis that exploitation of a system's vulnerabilities involves abnormal use of the system; therefore, security violations could be detected from abnormal patterns of system usage. ...

Similar, though often less clear, statements appear in many recent papers.

they observe. In particular, they often fail to demonstrate that the anomalous manifestation is a necessary consequence of the intrusive activity.

Recently, we discovered techniques whereby intrusive activities with anomalous manifestations could be modified in such a way as to be indistinguishable from arguably normal activities. We have also discovered techniques that can be used to modify other anomalous, intrusive activities so that, while still anomalous, they fall into the blind spot[2] of one commonly used anomaly detector (stide[3]) and become undetectable. We view both of these transformations as forms of information hiding and are beginning to suspect that the lessons that we are learning may be relevant to other areas of information hiding.

As the paper continues, we will explain just enough about intrusion detection and anomaly based intrusion detection so that the reader has some context into which to place our results. Key to this effort is the notion that a sensor associated with the system being monitored abstracts system activity into a trace of data items on which analysis is performed. In this context, we will discuss the problems involved in establishing "normal" behavior in general, and the classes of sensors used for anomaly based intrusion detection in particular. Recent work [2] has characterized these sensors and has demonstrated that they may suffer from blind spots, that is regions in which they are unable to recognize anomalous data. With this background established, we will provide several examples of anomalous intrusions, concentrating on the characteristics that make them anomalous in the context of our observed "normal." We then show how the intrusions can be transformed so that their traces either appear normal or fall into the blind spots of the anomaly detector. In passing, we also note that similar techniques could be used to produce traces that are anomalous but benign, overloading operators with false alarms that offer a further opportunity for hiding anomalous intrusive activity. At this point, we enter the realm of speculation.

Much of information hiding depends on the unsuspecting observer remaining unsuspecting. Once the observer knows that hidden information is present and understands how the information was hidden, its extraction (or erasure) is relatively simple. Unlike cryptography, information hiding techniques depend on the hider doing a good enough job to operate below the suspicion / detection threshold of observer. In the case of the IDS, we know the detector characteristics and have been able to shape our activities so as to produce traces that avoid them. We speculate that, in general, knowledge of the detection algorithm enables the development of techniques that avoid detection. As a simple example, the Stegdetect package by Niels Provos [4, 5] assumes that the details of the steganographic algorithm are known, but appears not to detect that for which it has not been specifically provisioned.

1.1 Intrusions, Intrusive Activities, Penetrations, and Exploits

From the earliest days of computer security, the possibility that malicious users could defeat protection mechanisms was an area of serious concern. Due to the relatively limited networking of early systems and the prevalence of multiuser batch systems, coupled with the fact that publicly accessible services (such as

	Penetrator Not Authorized to use Data/Program Resource	Penetrator Authorized to use Data/Program Resource
Penetrator Not Authorized Use Of Computer	Case A: External Penetration	
Penetrator Authorized Use Of Computer	Case B: Internal Penetration	Case C: Misfeasance

Fig. 1. General cases of threats (after [6])

present day web servers) were almost unknown, most of the early efforts concentrated on mechanisms that untrusted insiders could use to access sensitive materials on multi-level secure systems[2]. Under this model, the primary threat is from legitimate users of the system who try to gain access to material for which they do not have authorization.

Although there are earlier discussions of the issues associated with malicious users, James P. Anderson's 1980 report, "Computer Security Threat Monitoring and Surveillance"[6] sets up the first coherent framework for a of intrusions and intrusion detection.

Anderson classifies threats as shown in Figure 1. The first task faced by an external penetrator is gaining access to the system in question. Note that the true external penetrator may be either an outsider with no connection to the organization that owns or controls the system being attacked or it may be someone associated with the organization who is not authorized to use the system. In todays world of networked systems, it could also be someone who has legitimate access to systems on the network, but not to the target of the attack.

1.2 Intrusions and Anomalous Behavior

Anderson (and later Denning [1]) assumed that the statistical behavior of users could be characterized with sufficient accuracy so that departures from normal behavior would be indicative of intrusions. After a number of attempts, it was realized that the problem is not that simple, but the notion that some characterization of normal can be found that allows intrusive activity to be recognized persists. In general, anomaly based intrusion detectors comprise a sensor that

[2] A multi-level secure computing system is one that is capable of supporting a mandatory access control policy that bases access decisions on the classifications assigned to the information objects that it stores and clearances given to users on whose behalf processes seek access.

monitors some aspects of system behavior and a decision process that decides if the sensed data is consistent with a predefined notion of normal. The latter is typically defined by observing the data under circumstances where it is certain that intrusive activity is not present. The sensed data may be complex, involving numerous variables and their temporal relationships or it may be fairly simple. The detectors that we examine for this study monitor the system call activity of privileged processes operating on Unix systems. After the system calls have been collected for a sufficiently long period of normal activity (the training data) and characterized as shown below, the system is monitored to look for departures from normal under the assumption that this will indicate an intrusion.

2 Description of the Anomaly Detectors

For our purposes, we choose two relatively simple detectors, stide [3] and t-stide [7]. Both use as input the system calls[3] made by privileged UNIX programs such as `lpr` (the line printer server), `sendmail` (the mail delivery program), etc. These programs typically operate with special privileges because they must be able to read and write files belonging to many users. They are attractive targets for intruders because they can sometimes be abused in such a way that the abuser acquires their privileges. As they operate, these programs may spawn multiple processes. The data used for analysis consists of the lists (or traces) of system calls that each process associated with the program makes from its initial creation to its termination. The system calls may be thought of as unique symbols, each representing a particular system function invoked by the program, in effect, an alphabet consisting of several hundred characters.

Stide makes a binary decision based on whether or not a fixed length subsequence of test data is in its "normal" database. T-stide takes into account the frequency with which such sequences occur in the training data, allowing sequences that occur infrequently in the normal data to be considered as possibly intrusive.

2.1 Description of Stide

Stide acquires a model of normal behavior by segmenting training data into fixed-size sequences [7]. This is done by sliding a detector window of fixed size DW over the training data, one symbol at a time, producing a series of overlapping samples, each a sequence containing DW symbols. Each unique size DW sequence obtained from the data stream is stored in a "normal database." Sequences of size DW are also obtained from the test data using a sliding window and tested to see whether or not they are present in the normal database. If a test sequence is found in the normal database, it is assigned an anomaly score of 0. Sequences that do not exist in the normal database are assigned an anomaly score of 1. In this manner, a sequence of overlapping, fixed length, samples is converted to a sequence of 0s and 1s.

[3] The calls are captured by instrumenting the system call interface to the Unix kernel.

The detector's final response to the test data, the anomaly signal, is the sum of the anomaly scores for the most recent N test sequences. For example, if N, the size of the locality frame is set to 20, then for each sequence of test data the number of mismatches in the last 20 (overlapping) test sequences, including the current one, is calculated. The number of mismatches that occur within a locality frame is referred to as the locality frame count and is used to determine how anomalous the test data is in that region. The size of the locality frame is a user-defined parameter that is independent of the size of the detector-window. See [3, 7] for additional detail.

2.2 Description of T-stide

Warrender et al.[7] observed that some anomaly detection algorithms regarded rare sequences as suspicious events. T-stide ("stide with frequency threshold") was designed to test this premise. T-stide involves a simple modification to the basic stide algorithm. As the normal database is built, counts are maintained of the total number of samples examined and the number of times each sample was seen. This allows the relative frequency of occurrence of each sample to be determined. Rare sequences were defined as those sequences with relative frequencies that fall at or below a user-defined threshold (0.001% in this case). Sequences in the database that are not rare are called "common." In determining the anomaly scores for a sequence of test samples, t-stide treats samples found to be rare in the normal database as though they were not present and returns an anomaly score of 1.

2.3 A Description of an Anomaly-Based Evaluation Strategy

We define a foreign sequence (of any length) as a subsequence of a trace of test data that is not a subsequence of any trace of the normal data. We define a foreign test sequence as a sequence of length DW obtained from the test data that does not appear in the normal database. It is not difficult to see that stide will only detect foreign test sequences, and t-stide will detect *both* foreign and rare test sequences. Testing the detectors involves injecting foreign (or rare) sequences into normal data, a nontrivial process that is discussed in [8] which establishes a evaluation framework that focuses on the structure of anomalous sequences and provides a means to describe the interaction between the anomalous sequences and the sliding window of anomaly detection algorithms like stide.

2.4 Stide's Performance

The most significant result provided by the evaluation of stide was that the detector is completely blind to a particular kind of foreign sequence, a minimal foreign sequence, that was found to exist (in abundance) in real-world intrusion data [2]. A minimal foreign sequence is foreign sequence whose proper subsequences all exist in the normal data, i.e. it contains no smaller foreign sequences.

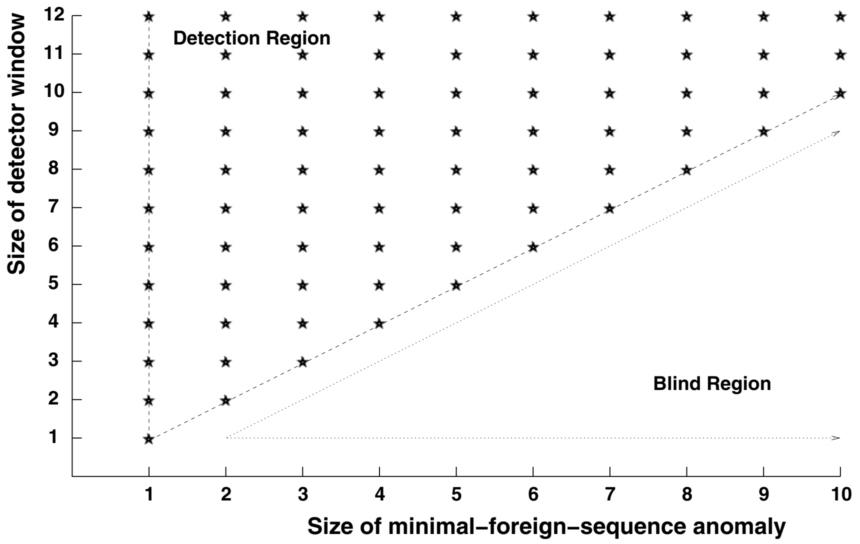

Fig. 2. The detector coverage (detection map) for stide; A comparison of the size of the detector window (rows) with the ability to detect different sizes of minimal foreign sequence (columns). A star indicates detection

For stide to detect[4] a minimal foreign sequence, its detector window size, DW must be larger than the size of the minimal foreign sequence. This phenomenon can be seen in Figure 2. The graph in the figure plots the size of the minimal foreign sequence on the x-axis and the size of the detector window on the y-axis. Each star marks the size of the detector window that successfully detected a minimal foreign sequence whose corresponding size is marked on the x-axis.

The diagonal line shows the relationship between the detector window size and size of the minimal foreign sequence, a relationship that can be described by the function, $y = x$. Figure 2 also shows an area of blindness in the detection capabilities of stide with respect to the minimal foreign sequence. This means that it is possible for a foreign sequence to exist in the data in such a way as to be completely invisible to stide. This weakness will be shown to be exploitable by an attacker in the subsequent sections.

2.5 T-stide's Performance Results from the Anomaly-Based Evaluation Strategy

The most significant result provided by the anomaly-based evaluation of t-stide was that there were conditions that caused the detector to be completely blind to

[4] The term detect for stide means that the minimal foreign sequence must have caused as at least one sequence mismatch as it passed through the detector window.

both the minimal foreign and rare sequences. Like the minimal foreign sequence, a minimal rare sequence is rare sequence whose proper subsequences are all common sequences in the normal data.

Although t-stide will be able to detect a minimal foreign sequence when the detector window is equal to or larger than the size of the minimal foreign sequence, it is possible for t-stide to detect a minimal foreign sequence when the detector window is smaller than the size of the minimal foreign sequence if the minimal foreign sequence contains at least one rare subsequence the size of the detector window. However, if the minimal foreign sequence is composed entirely of common subsequences, then t-stide exhibits the same behavior as stide, i.e. an area of detection blindness, identical to the one displayed in Figure 2, exists in the performance map. It is the t-stide's blindness to minimal foreign or rare sequences composed entirely of common subsequences that can be exploited to hide the presence of attacks from the detector.

3 The Victim Programs, Normal Data, and Attacks

The attacks selected for this study are typical of those that stide is intended to detect, i.e., attacks that exploit privileged UNIX system programs. UNIX system programs typically run with elevated privileges in order perform tasks that require the authority of the system administrator, privileges that ordinary users are not usually afforded. Exploiting vulnerabilities in privileged system programs can result in the attacker acquiring its privileges[7].

We chose the three attacks examined in this study because they can be used to illustrate the types of information hiding with which we are concerned. The three attacks are called `restore`, `tmpwatch`, and `kernel` after the programs which are exploited during the attacks. Each of the attacks successfully allows an attacker with an unprivileged local account on a system to elevate his privileges to those of the system administrator, giving total control over the victimized host.

For each of these programs, it is necessary to establish a baseline of normal behavior against which the attacks can be compared. Typically, this would be done by the administrator of the system being protected and would reflect the typical usage of that installation. An attacker wishing to hide an attack needs to know a reasonable approximation of the normal behavior as well.

For anomaly detectors that monitor system programs, training data can easily be approximated because system programs typically have limited behaviors and it is possible to make reasonable assumptions about their typical usage. Program documentation and experience supplement these assumptions. It is important to note, however, that the success of this method for undermining stide and t-stide is reliant on the attacker being able to estimate the normal usage of the system program, however the attacker does not need to obtain every possible example of normal behavior for a given system program in order to undermine stide.

When an exploit is performed, we need to determine whether the actions associated with the attack are actually manifested in the sensor data, and whether the manifestation is an anomalous event detectable by stide. The former is done by inspecting the system call trace captured and examining the source code of the attacked program to determine the execution path that produced it in response to the attack command. The later requires identifying the minimal foreign sequence(s) in the trace that associated with stide detection. In our evaluation, we used window sizes from 1 through 15 with both stide and t-stide.

3.1 The `restore` Attack

`restore` is a program used to restore corrupted or deleted files from a backup copy. In order to allow normal users to access tape devices used to perform backups and restores, the program must run with administrative privileges. In addition, the `restore` program allows the retrieval of backups from a remote, the backup server. In this case, the user of the `restore` program may be required to authenticate to the backup server. To support this authentication as well as the network connection required, `restore` executes a helper program, typically a "remote shell" on behalf of the user. A vulnerability exists in `restore` that passes its privileges to the helper program. The helper program can be specified by the user, allowing an attacker to create an arbitrary program and then use `restore` to execute it with root privileges. One example of such an attack program creates a "suid root" shell which the attacker can use to regain root access even if the vulnerability in `restore` is fixed.

For the `restore` system program, normal data was obtained by monitoring a regular user executing the `restore` system program to retrieve backup data from a remote backup server. A second computer is set up to act as this backup server and maintain regular backups of the files on the target computer system which the user could access using `ssh` as detailed in [9].

The `restore` attack was simply downloaded from [10] and run. The successful execution of the exploit was confirmed by noting the elevated privileges given the command shell created and run during the attack.

The manifestation of the `restore` attack was determined manually. An inspection of the source code for the `restore` program and its exploit script identified the system calls which are attributable to the attack. This sequence of system calls is

`dup2`, `close`, `close`, `dup2`, `getpid`, `setpgid`, `execve`.

In addition to the system calls of the child process, the sequence of system calls made by the `restore` process after it forks the child consists of failed attempts to interact with the attacker's program. These system calls,

`fork`, `close`, `close`, `fstat`, `mmap`, `ioctl`, `write`, `getuid`,
`setuid`, `write`, `read`, `write`, `munmap`, `exit`,

are also considered part of the manifestation of the attack. The attack was detectable by stide and t-stide at all detector window sizes greater than one because the pair `write`, `munmap` is a minimal foreign sequence of size 2.

3.2 The `tmpwatch` Attack

`tmpwatch` is a program which is intended to periodically clean up the temporary files left in the `tmp` file system by both users and system programs. The `tmpwatch` program must be run by the administrator in order to remove files created by arbitrary users. Since removing a file which is currently open by another processes might put that process in an unstable state, `tmpwatch` uses another program, `fuser`, to determine whether the file is currently open by another process. If it is, the file is not removed. The manner by which `tmpwatch` invokes `fuser` is unsafe from a security standpoint. `tmpwatch` program assembles a shell command from the `fuser` program name and the name of the file to be tested.t `tmpwatch` does not check the filename for special characters that will be interpreted by the shell as something other than part of the file name. For example, an attacker can create a filename containing a semicolon. When the filename is passed to the shell, the semicolon will be interpreted as the end of one command and the beginning of another and the rest of the filename will be treated as a command and executed with the administrative privileges inherited from `tmpwatch`. In our example, the attacker forces `tmpwatch` to run a sequence of commands which creates a "setuid root" shell.

For the `tmpwatch`[11] system program, normal data was obtained by populating the `/tmp` file system with a small directory hierarchy containing five directories and thirteen files, the access times of five of which are set to be more than five days old. Then the system calls of the `tmpwatch` program are logged while it is invoked by the system administrator to clean the `/tmp` directory of all files older than five days, using the `fuser` program to protect the files which are currently open by any other processes.

The `tmpwatch` exploit was created based on the description [12]. The attack script creates a file in the `tmp` directory which will cause a root compromise the next time `tmpwatch` is run as root. The success of the attack is confirmed with the creation of a shell with administrative privileges by the `tmpwatch`. The sequence of system calls which constitutes the manifestation of the attack is:

`lstat`, `access`, `rt_sigaction`, `rt_sigaction`, `rt_sigprocmask`, `vfork`, `wait4`, `rt_sigaction`, `rt_sigaction`, `rt_sigprocmask`, `unlink`.

The `vfork` in this sequence is the creation of the child process which in turn executes the `fuser` program. Since it is in that execution which leads to the creation of the "root" shell, the sequence of system calls made by that process constitute the rest of the manifestation of the attack:

`rt_sigaction`, `rt_sigaction`, `rt_sigprocmask`, `execve`.

The attack was detectable by both stide and t-stide at all detector window sizes greater than thirteen because the first part of the manifestation, listed above, formed a minimal foreign sequence of size thirteen when combined with the preceeding two (which are not part of the manifestation).

3.3 The `kernel` Attack

The Linux kernel enforces the security and access control policies of the system. One aspect of this is ensuring that the kernel support for debugger programs cannot be misused. Debuggers are programs which allow a developer to interactively control the execution of another program in order to analyze its behavior. The kernel mediates this interaction and must restrict these capabilities to authorized processes. For example, were an unprivileged process able to attach to and control a privileged program, the unprivileged program would be able subvert the privileged program and force it to take arbitrary actions. Unfortunately, a serialization error in the kernel creates an interval in which any process is able to attach to and control any other process before an authorization check is made. In our example, an attacker takes control of `traceroute`, a network diagnostic tool, and redirects it to create a “setuid root” shell.

For the `traceroute` system program, normal data was obtained by executing it to acquire diagnostic information regarding the network connectivity between the local host and the Internet site `nist.nsf.org`[5].

The `kernel` exploit was downloaded from [14]. Since the exploitation of the vulnerability in the kernel requires the execution of a privileged system program, the exploit was configured to take advantage of the `traceroute` system program. The sequence of system calls that comprise the manifestation of the attack embodied by the `kernel` exploit is:

`setuid`, `setgid`, `execve`.

It was found that the attack was detectable at all detector window sizes by both stide and t-stide. More precisely, the attack was detectable by stide and t-stide because `setgid` and `execve` are foreign symbols (i.e. they do not appear in the normal data at all).

4 Hiding Attacks by Modifying Exploits

Thus far, we have established some limitations on the detection abilities of stide and t-stide but have shown that they easily detect our example exploits. Knowing that stide is running on the host system and *will* alarm if any of the exploits are used, we wonder if they can be modified to hide their manifestations.

The detection map shown in figure 2 is the key to this process. Effectively, we have two choices: 1) ensure that the attack appears to be normal or 2) ensure that the attack falls into the detector's blind region. This means ensuring that the attack either manifests no foreign sequences at all or manifests only minimal foreign sequences longer that the detector window, DW. If this is not possible, the consequences of detection may be reduced by making the attack appear to

[5] Chosen because it is the simplest example in the documentation[13].

be either a less devastating attack; or another random attack altogether. We give examples for each of the approaches and each of the later alternatives.

4.1 Hiding in Normal

The `restore` attack can be hidden by making it appear normal. We do this by comparing the evidence left by the attack with what is left by a normal usage of the program and modifying the attack so that the evidence left is no different than what would have appeared normally. A comparison of the system call sequence made during the attack with that made during the normal restoration of files shows that the attack data can be distinguished from the normal data because the attack does not set up a communication channel to a remote host and the `restore` program fails. The system calls made when the `restore` program fails contain the foreign sequence that allows detection of the attack. In order to make the attack look normal, the helper program used must retrieve the backup file from the remote host, and at the same time perform its malicious activity. We modified the attack so that the program run by `restore` serves the dual purpose of giving the attacker elevated privileges and making the `ssh` connection to the backup server. Since the `restore` program receives the backup file and completes successfully, it never enters the error state and the evidence of the attack is hidden in what appears to be a normal run of `restore`.

The sequence of system calls observed when the modified attack is made exactly match the sequence of system calls observed when the `restore` program is run normally as described above. Note that the helper program is specified by the user, so that there is no way to include its activities in the definition of normal for `restore`.

4.2 Hiding in the Blind Spot

In addition to the exploits described above, we have discovered another exploit against `traceroute` that can be modified to produce arbitrarily long minimal foreign sequences[15]. `traceroute` must have unrestricted access to the network interface, a resource only provided to privileged system programs. A logic error in `traceroute` allows an attacker to corrupt the process' memory by specifying multiple network gateways on the command line[15]. The attack uses this memory corruption to redirect the process to instructions that execute a command shell with the elevated privileges of the `traceroute` system program. The attack can be confirmed from the system call trace of `traceroute` and observing that the attack has caused the process to launched a shell with administrative privileges.

The result of this modification is illustrated graphically in Figure 3. The x-axis for Figure 3, represents the size of the minimal foreign sequence anomaly, and the y-axis represents the size of the detector window. Each star marks the size of the detector window that successfully detected a minimal foreign sequence whose corresponding size is marked on the x-axis.

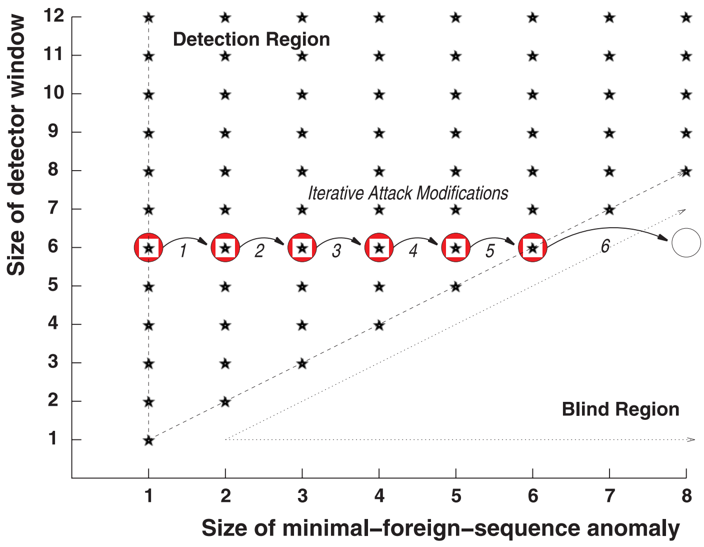

Fig. 3. Modifying `traceroute` exploit for an arbitrarily large Minimal Foreign Sequence

As expected the graph mirrors the detection map for stide, showing that the larger the minimal foreign sequence that is the manifestation of an exploit, the larger is the detector window required to detect that exploit. The circles and arrows illustrate the following scenario. If stide were deployed with a detector window of size 6, then it is possible to modify the `traceroute` exploit so that it manifests as a minimal foreign sequence of successively larger sizes until size 7 is reached where the exploit falls into the detector's blind spot. This shows that it is very possible to exert control over a common exploit so that its manifestation is moved from an anomaly detector's detection region, to its region of complete blindness. Such movement of an exploit's manifestation effectively hides the exploit from the detector's view.

Achieving an attacker's objectives is not affected by the modification to the exploit programs, and neither is the training data tampered with in order to render an anomaly detector blind to the attacks. While it would not be in the attacker's interests to modify an attack to make it more easily visible, this may also be possible. These results have implications for both detector design and for detector evaluation.

4.3 Hiding in a Less Serious Attack

An example of an attack being made to look like a less devastating attack is the `tmpwatch` attack. We consider an alternative attack using `tmpwatch` which performs a denial of service on the host. This is arguably less damaging that al-

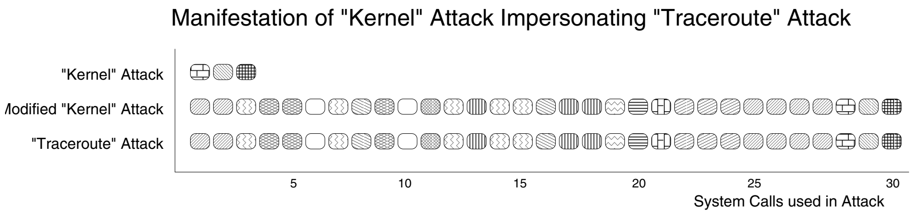

Fig. 4. A comparison of the manifestations of the `kernel` attack, the modified `kernel` attack, and the attack it was modified to impersonate

lowing a root compromise. The evidence left by this lesser attack will be collected and analyzed and the original `tmpwatch` attack modified so that the evidence it leaves is made to look like the evidence left by the alternative attack.

In order to recursively clean every subdirectory of the `/tmp` file system, the `tmpwatch` process forks a new process, copying itself. If an attacker can create a large number of subdirectories, the `tmpwatch` program will have to create a new process for each subdirectory. It is possible for the number of `tmpwatch` processes to grow until it reaches a system-wide limit on the number of running processes, typically crashing the system.

An exploit based around this approach was created, also based on the description of the vulnerability available at [12]. When the `tmpwatch` process is run, the success of the attack is confirmed when `tmpwatch` reports that the process table is full. The manifestation of this denial of service attack is thousands of sequences which look exactly like the manifestation of the root compromise attack.

If both attacks are launched against a system, the elevation of privileges attack may go unnoticed by the intrusion detection system. Evaluating the evidence left by just the denial-of-service attack and then the evidence left by both the denial-of-service and the elevation of privilege attacks launched in parallel, it can be confirmed that the detection system reports the same number and type of anomalous sequences in both attacks. Hence, the more devastating attack has been hidden within the less devastating one.

4.4 Hiding as Another Attack

The `kernel` attack can be made to look like a very different attack. The evidence from an attack that exploits a vulnerability in the `traceroute` program is described. The `kernel` attack is then modified so that the evidence it leaves exactly matches that left by the `traceroute` attack which is described in Section 4.2 above.

As described above, the `kernel` attack does not require any particular system program to be present on the system, most every system program which runs with administrative privileges works equally well. In this experiment, the `kernel` attack uses the `traceroute` program.

Figure 4 shows that the system calls issued by the impersonating (`kernel`) and impersonated (`traceroute`) attacks are identical. Each patterned block indicates a system call in the sequence of system calls that will be logged during the attack. This is possible because the `traceroute` attack contains the sequence of calls used in the `kernal` attack. It is possible to pad the `kernal` attack so that it takes on the appearance of the `traceroute` attack while preserving its own semantics.

4.5 Attack Hiding Results

The procedures described in the previous sections, designed to modify attacks to hide them, were all successful. For each of the attacks and for each of the two intrusion detection systems, stide and t-stide, the detection system is used to detect the attack at all window sizes from 1 through 15 and the number of alerts is recorded. The number of alerts produced by the attack is compared with the number of alerts produced by the target event which the attack is being modified to impersonate i.e. normal, a less devastating attack, or a totally different attack. A typical comparison is shown in Figure 5 in which it is shown that the `kernel` attack can be modified to successfully impersonate the `traceroute` attack to the t-stide anomaly-based intrusion detector. The top graph shows the alerts produced by the original `kernel` attack. The middle graph shows the alerts produced by the `traceroute` attack. And, the bottom graph shows the alerts produced by the `kernel` attack modified to look like the `traceroute` attack. Since the middle and bottom graphs match, T-stide is unable to distinguish between the two attacks, at window sizes from 1 through 15, with rarity threshold 0.005.

The comparison between the `restore` attack modified to look like a normal run of `restore` and such a normal run, and the comparison between the `tmpwatch` attack modified to look like a denial-of-service attack and the denial-of-service attack itself produce results similar to Figure 5.

5 Related Work

While this paper was under review, we became aware of similar work being performed by David Wagner [16] at Berkeley. Wagner's approach is linguistically based, using a mechanical search to embed an attack, padded with system calls that have been effectively converted to "no-ops" if necessary, into strings that can be composed from the normal stide database for the program being attacked. This approach seems to be primarily applicable to attacks based on storage overflows where the attacker controls the execution sequence of the attacked program from the point of the overflow.

Preliminary results showing that intrusions can be hidden in the blind spot of stide also appear in a paper by Tan, *et. al.* [17].

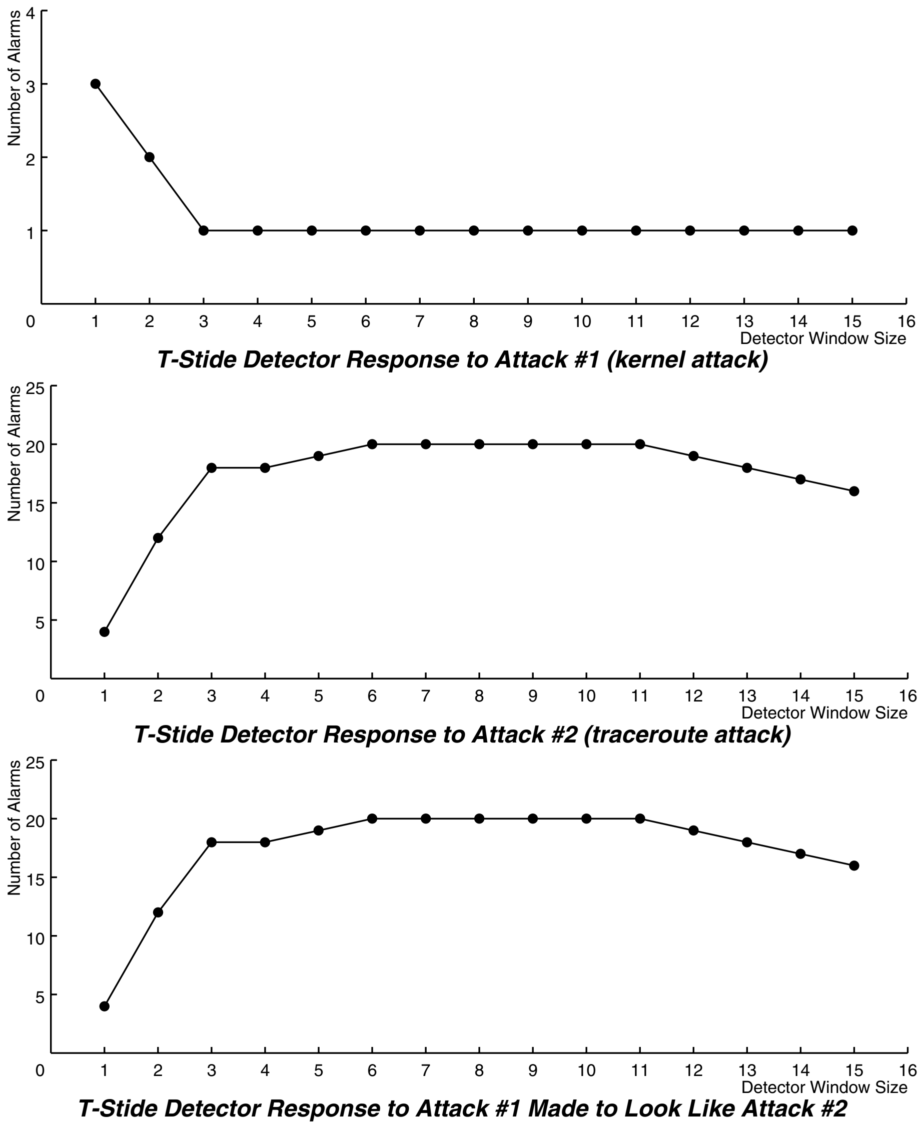

Fig. 5. A comparison of the alerts produced by different attacks. The top graph shows the number of alerts that are produced when t-stide is used with window sizes 1 through 15 to detect attack #1, the `kernel` attack. The middle graph shows the number of alerts that are produced when t-stide is used to detect attack #2. The bottom graph shows the number of alerts that are produced when t-stide is used to detect attack #1 modified to look like attack #2. Since the graphs match, the modification is successful in getting attack #1 to look like attack #2 to t-stide

6 Conclusions and Implications

At the very least, our work has demonstrated, through the application of a novel information hiding paradigm, that malicious acts and anomalous manifestations from intrusion detection sensors are not necessarily synonymous. We have demonstrated hiding serious attacks in such a way that they either appear completely normal or are likely to be confused with other attacks. We believe that we can apply the approach with other attacks and with other sensors. A skilled attacker who understands the detector and the environment in which it is deployed may be able to devise undetectable attacks. If this result holds for a wide variery of anomaly based intrusion detection systems, it may undermine the effectiveness of the anomaly detection approach to intrusion detection as a vehicle for detecting unknown attacks.

At the same time, the "cat and mouse" game implied by this approach has an unsatisfying aspect to it. The anomaly detector evaluation that led to our work has a sound scientific basis, but the application of these detectors to intrusion detection is very much *ad hoc* as has been noted by Lee and Xiang [18]. In this respect, the work seems to have much in common with the rest of the information hiding field and with other areas involving "slights of hand" in general. This can be summed up as "It isn't a trick once you understand it." Much of the work in information hiding has this flavor, depending on obscurity for protection. We were hoping to gain insights that might move us toward a more theoretical basis for understanding intrusions. Instead, we seem to have discovered an interesting approach for serious intruders.

References

[1] Denning, D. E.: An intrusion detection model. IEEE Transactions on Software Engineering **SE-13** (1987) 222–232 1, 3

[2] Tan, K. M. C., Maxion, R. A.: "Why 6?" Defining the operational limits of stide, an anomaly-based intrusion detector. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, Oakland, CA (2002) 2, 5

[3] Forrest, S., Hofmeyr, S. A., Somayaji, A., Longstaff, T. A.: A sense of self for unix processes. In: Proceedings 1996 IEEE Symposium on Security and Privacy, Los Alamitos, CA, IEEE Computer Society Press (1996) 2, 4, 5

[4] Provos, N.: Steganography press information. On line report of work performed at the University of Michigan Center for Information Technology Integration (2002) Observed at `http://www.citi.umich.edu/projects/steganography/faq.html` as of 4 february 2002 2

[5] Provos, N., Honeyman, P.: Detecting steganographic content on the internet. In: ISOC NDSS'02, San Diego, CA (2002) 2

[6] Anderson, J. P.: Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., Fort Washington, PA (1980) Available online at `http://seclab.cs.ucdavis.edu/projects/history/CD/ande80.pdf` 3

[7] Warrender, C., Forrest, S., Pearlmutter, B.: Detecting intrusions using system calls: Alternative data models. In: Proceedings of the 1999 IEEE Symposium on Security and Privacy, Oakland, CA (1999) 133–145 4, 5, 7

[8] Maxion, R. A., Tan, K. M. C.: Anomaly detection in embedded systems. IEEE Transactions on Computers **51** (2002) 108–120 5

[9] Pop, S., Card, R.: Restore(8) system manager's manual. Included in `dump` version 0.4b13 software package (2000) 8

[10] fish stiqz: Redhat linux `restore` insecure environment variable vulnerability. Internet – `http://www.securityfocus.com/bid/1914` (2000) bugtraq id 1914 8

[11] Troan, E., Brows, P.: Tmpwatch(8). Included in `tmpwatch` version 2.2 software package (2000) 9

[12] Yurchenko, A. Y.: Tmpwatch arbitrary command execution vulnerability. Internet – `http://www.securityfocus.com/bid/1785` (2000) bugtraq id 1785 9, 13

[13] Jaconson, V.: Traceroute(8). Included in `traceroute` version 1.4a5 software package (1997) 10

[14] Anonymous: Linux ptrace/execve race condition vulnerability. Internet – `http://www.securityfocus.com/bid/2529` (2001) bugtraq id 2529 10

[15] Kaempf, M.: Lbnl traceroute heap corruption vulnerability (2000) bugtraq id 1739 11

[16] Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: 9th ACM Conference on Computer and Communications Security. (2002) To Appear 14

[17] Tan, K. M., Killourhy, K. S., Maxion, R. A.: Undermining an anomaly-based intrusion detection system using common exploits. In Wespi, A., Vigna, G., Deri, L., eds.: 5th International Symposium, RAID 2002. Number 2516 in LNCS, Zurich, Switzerland, Springer (2002) 54–73 14

[18] Lee, W., Xiang, D.: Information-theoretic measures for anomaly detection'. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, Oakland, CA, IEEE Computer Society Press, Los Alamitos, CA (2001) 130–143 16