Toward Realistic and Artifact-Free Insider-Threat Data

Kevin S. Killourhy
ksk@cs.cmu.edu

Roy A. Maxion
maxion@cs.cmu.edu

Dependable Systems Laboratory
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Abstract

Progress in insider-threat detection is currently limited by a lack of realistic, publicly available, real-world data. For reasons of privacy and confidentiality, no one wants to expose their sensitive data to the research community. Data can be sanitized to mitigate privacy and confidentiality concerns, but the mere act of sanitizing the data may introduce artifacts that compromise its utility for research purposes. If sanitization artifacts change the results of insider-threat experiments, then those results could lead to conclusions which are not true in the real world.

The goal of this work is to investigate the consequences of sanitization artifacts on insider-threat detection experiments. We assemble a suite of tools and present a methodology for collecting and sanitizing data. We use these tools and methods in an experimental evaluation of an insider-threat detection system. We compare the results of the evaluation using raw data to the results using each of three types of sanitized data, and we measure the effect of each sanitization strategy.

We establish that two of the three sanitization strategies actually alter the results of the experiment. Since these two sanitization strategies are commonly used in practice, we must be concerned about the consequences of sanitization artifacts on insider-threat research. On the other hand, we demonstrate that the third sanitization strategy addresses these concerns, indicating that realistic, artifact-free data sets can be created with appropriate tools and methods.

1. Introduction

An insider is a person with legitimate access to an organization, and who acts *maliciously* against that organization. The insider threat is a significant and growing concern, especially in fields where espionage and fraud are profitable. A survey of insider incidents in the banking and finance sector found that 30% resulted in losses in excess of $500,000 each [10]. Examples of insider behavior include the unauthorized modification of company data for personal profit, the compromise of other employees' computer accounts, and the installation of "back doors" through which the insider regains access in the event he or she is terminated [4].

For almost two decades, researchers have been proposing systems to detect and prevent insider threat. These systems work by monitoring users, profiling their behavior, and identifying suspicious or anomalous activity. The earliest systems analyzed audit records and built profiles of the commands each user executes [1, 12]. The conjecture was that a legitimate user might be distinguished from an impostor by their distinct usage of commands (e.g., the user used `vi`, the impostor runs `emacs`); also, an insider straying from authorized activity might be detected when he used anomalous commands (e.g., to copy a file he does not normally access). As a result of that early work, many systems have been proposed for detecting insiders using Unix command-line data [5, 8, 11].

These insider-threat detection systems are best evaluated using natural, real-world data to measure, compare, and improve their performance. Researchers have instrumented computer systems to monitor the behavior of participating users [2, 5, 11]. They collect, *sanitize*, and share their data with the research community. Sanitization is the act of replacing sensitive data (such as passwords) with uninformative markers (such as the string `<XXXXX>`). Sharing the data allows other researchers to evaluate and compare the performance of different insider-threat detectors.

However, since a goal of these evaluations is to estimate the "real-world" performance of a system, we must ask how effectively the data serve this goal. When an evaluation uses these data, how confident are we that the outcome of an evaluation carries over to the real world? If these evaluations are being used to determine which insider-threat detector is deployed, then we must be confident that the eval-

1063-9527/07 $25.00 © 2007 IEEE
DOI 10.1109/ACSAC.2007.31

uation is accurate. Deploying an insider-threat system that under-performs is risky, increasing the already high cost of insider-threat.

Unfortunately, because of the way existing data sets were created, we must be wary of generalizing to the real world on the basis of evaluations that use these data sets. First, the data are not *realistic*, by which we mean that the data do not reflect what one would find in a real-world environment. Specifically, in the real world, an insider's choice of commands would stem from his or her underlying malicious intentions. Existing data sets contain no actual insider behavior; it is experimentally induced, typically by designating some normal users as impostors, and using their commands as a substitute for insiders' commands. Those commands might have been observed when the user was checking his mail, writing a program, or any other benign activity. We cannot assume that a system which is good at detecting when a user is checking his mail will perform equally well detecting insider activity.

Second, when the data are sanitized, *artifacts* are introduced into the data set. Artifacts are modifications of the data (due to the sanitization process) that alter the outcome of experiments using the data. For instance, if sanitization replaces all user names with `<XXXXX>`, then commands which were once distinct become indistinguishable. Suppose an evaluation tested whether a system could detect when user `dave` began snooping around user `mary`'s home directory. The benign command "`cd dave`" and the suspicious command "`cd mary`" are easily distinguished in the raw, unsanitized data, but both appear as "`cd <XXXXX>`" in the sanitized data. A detector which would detect `dave`'s snooping in the real world might miss it in the sanitized data because of artifacts. On the basis of this failed evaluation, a detector which would actually work well in practice might never be deployed.

2. Problem and approach

Experiments using existing insider-threat data may not generalize to the real world because: (1) benign commands typed by normal users are injected as a substitute for commands typed by malicious insiders; and (2) unintended sanitization artifacts may be introduced when sensitive data are replaced with uninformative markers. As a result, benign user activity could be flagged as insider activity when it is not, or actual insider activity could go unnoticed.

To address these issues, we have developed a suite of tools and a methodology for using them. To maximize the realism of injected insider behavior, we developed a library of carefully scripted and vetted insider activities. While the realism of insider injections is a concern, the primary focus of this work is on the effect of sanitization artifacts. To ensure that sanitization does not introduce artifacts, we developed a sanitizing engine which allows users to review their data, mark sensitive data, and export sanitized data sets. This Sanitizer incorporates three different sanitization strategies: Redact-Only, Token-Only, and Word-Token. They differ in how they cover up sensitive data (e.g., Redact-Only uses a "black box" like `<XXXXX>`, Token-Only uses a distinct token like `<TOKEN-12>`, and Word-Token breaks the sensitive data into words and then uses one token for each word). Redact-Only and Token-Only are similar to strategies used to sanitize existing data sets. The Word-Token strategy was designed to be artifact-free.

The present work is framed much like earlier work in insider detection. In particular, we compare two types of insider monitoring, called Enriched (comprising the entire command line typed by a user) and Truncated (comprising only the name of the command executed). The purpose of conducting this experiment is to compare the results obtained using raw, unsanitized data to the results using each of the three sanitization strategies. We intend to establish whether artifacts arise in the two types of monitored data (Truncated and Enriched) as a consequence of sanitization, and what the effects of those artifacts are on the ability to detect insider activity.

3. Related work

Three existing insider-threat data sets are commonly used by researchers. Unfortunately, each contains unrealistic insider injections and sanitization artifacts. Greenberg [2] collected a corpus of Unix command-line data, and Maxion [7] assembled it into an insider-threat data set. However, benign commands from normal users were used as a substitute for insider commands. Further, to protect participating users' privacy, usernames were sanitized by replacing each letter of the username with an "`x`" character (e.g., `dave` and `mary` each contain four characters, and both would appear as `xxxx`). Lane and Brodley [5] also collected users' commands, but again, benign commands were used as a substitute for insider commands. The commands were also sanitized by replacing every sequence of one or more file names with a number indicating how many names were in the sequence. Schonlau et al. [11] collected the names of programs executed (rather than the full command line), but again, benign commands were used in place of insider commands. Further, one could argue that collecting only the names of programs constitutes a form of sanitization (especially since the authors state that full command lines were not collected because of "privacy concerns"). While these data sets have helped researchers develop and refine insider-detection methods, we remain concerned about the effect of unrealistic insider injections and sanitization artifacts on those researchers' experiments.

The problem of data-set artifacts has been demonstrated by Mahoney and Chan [6] in the domain of intrusion detection. They found that an existing data set used to evaluate intrusion-detection systems contained evidence of the artificial procedure used to synthesize the data. They demonstrated that a detector could be built which would perform well in the evaluation (by detecting these artifacts), and yet it would have little chance of working well in practice. Their findings highlight the need for data to be realistic and artifact-free.

The tools we develop in this study are similar to others in the literature. The Honeynet Project offers a data-collection tool called Sebek [3] which is similar to the data collector we develop. Data anonymization algorithms such as those proposed by Sweeney [13], and tools like that developed by Pang and Paxson [9] for Internet traffic are also useful for data sanitization. One might argue that anonymization and sanitization are the same, but the literature is not clear on whether anonymization includes the removal of confidential or sensitive data (as stated by Pang and Paxson) or just the removal of identifying data (as stated by Sweeney). In any event, our tools were not designed to supplant these existing tools but to provide similar capabilities. We created our own suite of tools because they had to be interoperable, not because existing tools provided lesser functionality. As a consequence, our findings should be relevant to users of these similar tools as well.

4. Overview of methodology

In order to examine the consequences of sanitization artifacts on insider-threat experiments, we replicated a typical experiment from the literature. In our replication, the original experiment was conducted first using raw, unsanitized data and then repeated using data treated with each of the three sanitization strategies. We compared the results from these experiments to reveal whether they were altered by sanitization artifacts.

The experiment chosen for this exercise was conducted by Maxion [7] who was studying the effect of two different types of data (called Truncated and Enriched) on the effectiveness of a naive-Bayes insider-threat detector. He compared the performance of a detector given only the (Truncated) program names typed by a Unix user to the performance of the same detector given the full (Enriched) command line. Performance was measured in terms of the cost of error of the detector (calculated as the sum of the miss and false-alarm rates). Maxion found that the cost of using Enriched data was 9% lower than the cost of using Truncated data. However, his experiment used the Greenberg data set [2], which contains sanitization artifacts as discussed above. As a consequence, we cannot be sure that the 9% difference in cost predicted by the experiment would also be seen in a real-world deployment.

In order to see the effects of sanitization on this experiment, we built a data-collection program called Monolog, and deployed it on the workstations of system administrators and operations staff members within the university. The data collector recorded each user's commands during his or her natural daily activities.

We assembled a "library" of realistic insider attacks, scripted the attacks, and launched the scripts against participating users' accounts. Whereas other researchers injected one user's commands into another user's data as a simulated attack, our scripts perform a real-world attack against the user's account (and then recover from the attack). The scripts were designed to impersonate an attacker who gains unauthorized access to an account (e.g., by using the victim's workstation while he or she is away).

In another departure from other researchers' methods, users in the present study sanitized their own data. We felt that the users themselves were in the best position to identify sensitive information, and that their input would help researchers understand what sort of data users are reluctant to share. An application called the Sanitizer was built to allow users to view and to search their own data, to mark records as sensitive, and to export sanitized copies of the data. Data could be exported using any of the three sanitization strategies (Redact-Only, Token-Only, and Word-Token), which are described in detail in Section 5.3. By having users sanitize their own data, researchers avoided having to employ unnecessarily broad, draconian sanitization techniques.

To replicate Maxion's experiment, we converted the users' sanitized data into Truncated and Enriched evaluation data sets, and the performance of the naive-Bayes detector was tested on each one. The miss rate, false alarm rate, and cost of error were calculated as in the earlier experiment. To compare the results to those using raw, unsanitized data, we deployed the naive-Bayes detector on users' workstations, and we walked the users through the process of running it on their own raw data and reporting the results. In this way, we were able to compare the results of the experiment on raw data to the results with each of the three types of sanitization, yet the users' sensitive raw data were never shared with the researchers.

5. Apparatus

We wrote three software programs to enable this investigation: (1) a reliable data-collection package called Monolog; (2) an insider-script library to automate the injection of realistic insider attacks; and (3) a Sanitizer application to enable the review and sanitization of collected data.

5.1. Monolog data collector

Monolog (as in "monitor and logger") was designed to reliably record user behavior within a command shell on the Linux operating system. We instrumented the command shell `bash` to collect data commonly used for user profiling at the command line. Lane and Brodley [5] collected the command lines typed by their users. Greenberg [2] collected not only the command lines, but also the current working directory and the exit status of each command. Schonlau et al. [11] collected the names of the programs executed (which may differ from the commands typed by the user because of shell aliases or scripting). Monolog records each of these types of data used in earlier studies.

Previous researchers found that instrumenting a shell was problematic because of the complexity of the code [2]. We avoid much of the complexity because the shells simply send command-line data to a dedicated logging facility as soon as it becomes available. By keeping little state in the instrumented shells, we avoid data loss if the shell crashes or is unexpectedly terminated. The logging process, called the Monolog daemon, writes data from the shells to restricted-access files, one per monitoring shell process. Access restrictions prevent users from snooping into each others' logs, and also prevents users from corrupting their own logs. The design of the daemon is simple compared to that of a shell. It is under 3K lines of code, and its correctness can be tested independently of the instrumented shells.

5.2. Insider-script library

While Monolog ensures that user commands are reliably collected, another mechanism is needed to inject realistic insider commands. Previous researchers borrowed commands from one benign user to use as insider data for another user [5, 7, 11], but as we have noted, this is unrealistic. We developed a library of scripts designed to impersonate a human attacker who gains access to another user's account. Our scripts were designed to execute the commands of an attack in a shell monitored by Monolog, so we conduct a real attack against a participating user's account. Conducting the attack allows us to verify its realism, whereas adding the commands of the attack to data already collected does not.

Four attack-injection scripts were developed. The attacks were chosen to be consistent with actual insider incidents reported to our research group or documented by Keeney et al. [4]. The commands executed to accomplish the attack were chosen by a researcher with some prior penetration-testing experience.

1. Backdoor. The script downloads the `nc` network-connection utility, compiles it, and installs it in the participating user's home directory. The `nc` program enables a user to set up a listening socket which will execute a program if a remote host connects to it. The `nc` program is configured to listen on a randomly chosen port and, when a remote host connects, to run `/bin/sh`. Consequently, the script sets up a backdoor into the user's account that anyone can use just by connecting to the right port. After performing the injection, the script verifies that the backdoor is running by connecting to it. It recovers from the attack by removing the backdoor and the `nc` program.

2. Portscan. The `nmap` port-scanning program is downloaded, compiled, and installed from the command line. It is used to perform reconnaissance on three other computers (intended to represent future targets for the attacker). After performing the injection, the script verifies the output of the port-scanner. It recovers from the attack by removing the `nmap` program.

3. Exposure. The permissions on key files containing the user's personal data are changed to allow anyone on the system to view them. All "history" files in the user's home directory are changed so that anyone has read permission. This change allows a spy to reconnoiter the files the user accesses. After performing the injection, the script verifies that the permissions on these files have changed. It recovers from the attack by restoring the original permissions.

4. Snoop. The `history` command is run to determine the last several commands typed by the user. Potential filenames are identified and extracted from the list. The `find` command is used to find paths to a selection of those files, and those files which are found are examined with the `head` command. This script examines the first few lines of up to 10 files recently accessed by the user, simulating an attempt to gather information about the user's recent activities. After performing the injection, the script verifies the output of the commands. No recovery from the attack is necessary beyond exiting the shell used for the injection.

We designed each script by first conducting the attack manually under controlled conditions. The attack was recorded by Monolog, and the commands typed by the attacker were extracted, along with the time intervals between them. Then, we wrote a script (using Perl and the Expect package) to replay the commands of the attack, scheduling the timing of each command to match the timing from the recording. Then, we wrote verification functions to parse the output of each command and to ensure that the expected output is printed (e.g., no "file not found" error messages).

We parse and verify the output of each command because we would not want to conduct an attack that fails in practice. Further, attack scripts can incorporate the output of one command as arguments to later commands (e.g., the Snoop attack uses the file names printed by `history` as arguments in subsequent calls to `find`). We end the script

with a final verification and recovery function that restores the system to its pre-attack state.

The attack scripts are run from a shell that is not monitored by the data collector, so that the name of the script itself does not appear in the logs as an artifact. For the same reason, the functions to verify the success of the attack and recover from it do not themselves execute shell commands.

5.3. Sanitizer application

The data collector and injection library described above ensure that realistic user behavior is collected, and that realistic attacks are injected. The Sanitizer ensures that users can look through the collected data to find and sanitize information that users judge to be sensitive. It provides a graphical interface with which to review the data collected by the Monolog data collector.

The Sanitizer contains three panels, shown in Figure 1. The large panel on the right contains a marked-up copy of a Monolog session log. Text that may be potentially sensitive is shown on a gray background, and text that has been marked as sensitive is shown on a black background. A user can mark text as sensitive by selecting it and pushing a button. The top-left panel contains a list of all the sessions not yet sanitized (i.e., a "to-do" list). The lower-left panel contains a list of sessions that have been sanitized already.

When marking text as sensitive, users choose between two marking options, called Redact and Tokenize. They represent different sanitization preferences, and are intended to solicit the user's judgment about how sensitive the text is.

1. Tokenize: A user selects Tokenize if he or she prefers that the selected text be replaced with tokens in the sanitized copy of the data. If the same span of text has been marked for tokenization in two separate places, the same token will be used to cover it up. This might be a problem if, for instance, a user's password is "God" and he tokenizes it. Suppose the password is covered up with the string `<TOKEN-12>` and elsewhere in the sanitized data appears the phrase "In `<TOKEN-12>` we trust." Someone looking at the sanitized data could infer the user's password.

2. Redact: The fact that a token will be reused leaks some information about what was sanitized. The Redact button can be used in situations where this leakage is problematic. A user selects redaction if he or she prefers that the selected text is completely blacked out (as is done to cover up classified information in declassified documents). Substituting the fixed-length redaction string `<XXXXX>` makes it impossible to tell whether one redacted string is the same as another, making it appropriate for extremely sensitive data such as passwords.

It is important to note that these marks register a preference. Whether the data are either tokenized or redacted depends on the sanitization strategy selected when a sanitized copy of the data is actually exported.

After sanitizing all the sessions that he or she wishes to export, the user is given a choice between three sanitization strategies called Redact-Only, Token-Only, and Word-Token. The strategies differ in how they cover up data marked as sensitive. Intuitively, they represent different balance points in the trade-off between a user concerned with maintaining as much privacy as possible and a researcher trying to preserve useful information. Of the three, Redact-Only favors privacy the most, Word-Token favors the preservation of useful information, and Token-Only is in the middle..

1. Redact-Only. All spans of text marked for sanitization (either redaction or tokenization) are redacted. All sensitive spans of text are replaced by a string of five `X`'s (i.e., `<XXXXX>`).

2. Token-Only. All spans of text marked for either redaction or tokenization are replaced by numbered token strings (e.g., `<TOKEN-1>` or `<TOKEN-12>`). If the exact same span of text appears in two different locations, both will be replaced by a token with the same number.

3. Word-Token. Spans of text marked for redaction or for tokenization are first divided into words (with whitespace-delimited boundaries). Then, each word is tokenized individually. Whenever a word is tokenized, a search is performed, and every instance of that word is replaced with the same token without regard for whether it was marked for sanitization or not (e.g., if the username `mary` is tokenized as `<TOKEN-12>`, then every whitespace-delimited occurrence of `mary` will be replaced with `<TOKEN-12>`).

The Redact-Only and Token-Only strategies mimic strategies that others have used in practice [2, 9]. The Word-Token strategy was designed specifically to avoid introducing artifacts into experimental evaluations of anomaly-based insider-threat detectors. In essence, Word-Token sanitization makes a list of every distinct word that appears in any span of text marked sensitive, associates a distinct token with each word, and replaces every instance of the word with the corresponding token.

Anomaly-based detection systems (e.g., naive Bayes [8], Lane and Brodley's detector [5], and Schonlau et al.'s detectors [11]) are built to recognize patterns between the words on a command line (e.g., relative frequencies or common repeating sequences), and to detect violations of these patterns. We theorize that Word-Token sanitization will not introduce artifacts because the performance of an anomaly-based detector should not change if every instance of a word in a data set is replaced with a new word. This study explores whether our theory holds true.

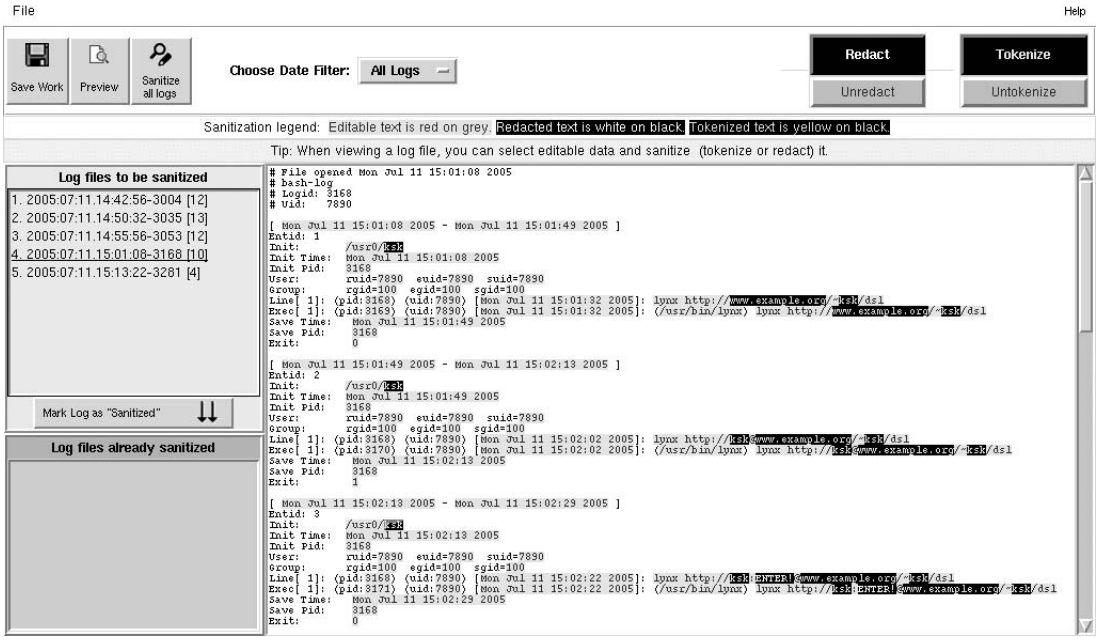

Figure 1. The Sanitizer application's graphical user interface shows a list of logs to be sanitized (top-left panel), a list of logs already sanitized (bottom-left panel), and the contents of a log file currently under review (right panel). Potentially sensitive content is displayed with a gray background. Content that has been marked by the user as sensitive is displayed on a black background.

6. Experimental methodology

We used these three tools (the Monolog data collector, the insider-script library, and the Sanitizer application) to investigate the consequences of sanitization artifacts on Maxion's insider-threat experiment [7]. He compared the performance of the naive-Bayes detector on two types of data (Truncated and Enriched). Our goal was to establish whether one particular sanitization strategy (Word-Token) was artifact-free for this experiment, and what the consequences are when we use the other two strategies (Redact-Only and Token-Only).

6.1. Deploy the data collector

We deployed the Monolog data-collection software on workstations used by twelve system administrators and operations staff members in our organization. These users were shown that the monitoring shell would appear identical to the one they normally used even though it was instrumented to collect data. They were instructed as to what data were being collected (e.g., no passwords unless they are typed on the command line), and they were assured that only sanitized data which they vetted would leave their workstation and never raw, unsanitized data.

6.2. Data collection and subject selection

In order to collect a large sample of legitimate user behavior, we left the staff for four to six months to use their workstations as they normally would. Once a week, a researcher would record the number of shell sessions collected and the number of commands in each session. On the basis of these records, four subjects were selected to continue with the study. Seven of the twelve staff members were deselected because they were part-time operators or primarily used a Windows workstation, and so only a few sessions were recorded for them. One full-time operator was not selected because he had night and weekend shifts, and there was no convenient opportunity for a researcher to meet with both him and his supervisor. The four subjects were all full-time operators or system administrators and each accumulated over 100 recorded sessions.

A sample of four subjects may seem small. However, the subjects all come from a population that is small but valuable (i.e., experienced system administrators and operators). A survey has shown that such positions are at high risk of insiders [4], and every full-time operator in our organization except one (as noted above) was included in the pool. Further, since we gather many sessions of data from each subject, and sessions are the true "element" being studied,

we judged the size of the subject pool to be adequate for this work. A larger, more heterogeneous pool would be desirable in future studies.

6.3. Perform insider-attack injections

We presented the four attack scripts to the subjects and to their supervisors (who administer their workstations). We described the details of each attack, and we demonstrated the post-attack recovery procedure. Permission to run each attack script was obtained from all parties, and the scripts were deployed on the subjects' workstations.

The subjects ran the four attack scripts against their own accounts. They typed the script names into shells that were not monitored by the Monolog data collector (to avoid introducing injection artifacts). With the subject, we observed the progress of the attack, verified that the attack succeeded, and confirmed that the recovery mechanism repaired any damage.

6.4. Sanitize the data files

We asked the subjects to sanitize their own sessions. They were given a demonstration of the Sanitizer and instructed on the difference between redacting and tokenizing sensitive data. A researcher navigated through the interface on an illustrative data set. He explained that he would mark usernames and hostnames with the Tokenize button, but he would use the Redact button for a password and a potentially embarrassing URL because of their sensitivity. The subjects were instructed that they should use their own judgment in deciding what to tokenization or redaction.

We instructed the subjects to sanitize at least 60 sessions worth of data because 50 sessions would be used by an insider-threat detector to build a profile of their usage and the remaining 10 would be used to test that profile. We also asked the subjects to review the four sessions that contain the attack injections and to sanitize those as they would sanitize their other sessions. In this manner, each subject sanitized at least 64 sessions of his or her own data.

The subjects took between 10 and 30 minutes to sanitize their data, and they sanitized between 111 and 219 sessions each. One or more words were tokenized or redacted from 40% of the command lines. One subject sanitized 58% of the command lines, while another sanitized only the subject's own username and home directory. Only one subject used the Redact button, but all subjects used the Tokenize button.

While the subjects sanitized their data, a researcher was present to offer assistance, but he did not look at the unsanitized data on their screen unless given explicit permission. After sanitizing their data, the subjects were instructed to export their data using each of the three sanitization strategies (Redact-Only, Token-Only, and Word-Token).

6.5. Create evaluation data sets

Since we intended to replicate the experiment conducted by Maxion [7], we had to derive data sets analogous to the ones he used in his experiment. Maxion compared the performance of an insider-threat detector when monitoring "Truncated" command-line data to its performance when monitoring "Enriched" command-line data. Truncated command-line data consists only of the names of the programs executed, while Enriched command-line data includes the whole command line. Maxion's hypothesis was that using Enriched data lowered the cost of error (calculated as the sum of the false-alarm rate and the miss rate), and he found a 9% reduction in cost.

In order to replicate Maxion's experiment, we derived two evaluation data sets (Truncated and Enriched) from each of the three sanitized data sets (Redact-Only, Token-Only, and Word-Token). To create the Redact-Only Truncated evaluation data set, we extracted the commands typed by the subject in each session of the Redact-Only sanitized data. We "truncated" the commands so that only the command name was left (i.e., discarding flags, filenames, and other command arguments). We labeled the first 50 sessions of commands as training data. We labeled the next 10 sessions and the 4 attack-injected sessions as test data. These data, labeled for training and test, constitute the Redact-Only Truncated evaluation data set. To create the Redact-Only Enriched evaluation data set, we extracted commands typed by the subject in each session of the Redact-Only sanitized data, but we did not truncate the commands. We label sessions of un-truncated (or "enriched") commands as training and test data in the same way as in the Truncated data set. We derived Truncated and Enriched evaluation data sets from each of the Token-Only and Word-Token sanitized data using the same procedure.

To compare the results of Maxion's experiment on sanitized data to the equivalent experiment on raw, unsanitized data, we created Truncated and Enriched evaluation data sets using each subject's raw data as well. The procedure described above for creating Truncated and Enriched data sets was scripted. We deployed the script to each subject's workstation, and we asked the subjects to create Truncated and Enriched evaluation data sets from the raw data using this script. In this way, the raw data never left the subjects' workstations. For each subject, a total of eight evaluation data sets were created: two evaluation data sets (Truncated and Enriched) from the raw data, and two from each of the three types of sanitized data (Redact-Only, Token-Only, and Word-Token).

6.6. Evaluate the naive-Bayes detector

The insider-threat detector evaluated in this study is based on that used previously to evaluate the effect of using Enriched rather than Truncated command lines [7]. The insider-threat detector builds a profile of command usage from the user's training sessions. For each test session, the detector calculates an *anomaly score* that represents how much the session deviates from the profile. Consequently, the detector attempts to separate the user's own sessions (which should fit the profile) from the attack-injected sessions (which should not).

The details of this detector are described fully by Maxion and Townsend [8] who refer to this particular detector as *naive Bayes with one-class training input*. Basically, the profile of the user is built by counting the number of occurrences of each command in the training sessions, and calculating the relative frequency of each command. More formally, let n be the number of commands in the training data, and k be the number of unique commands. If the commands are named $c_1, c_2, \ldots, c_k$, then let n_1 be the number of times c_1 appears in the training sessions, n_2 be the number of times c_2 appears, and so on. The profile consists of probability estimates for each command that are calculated as follows.

$$P(c_1) = \frac{n_1 + \alpha}{n + \alpha k}, P(c_2) = \frac{n_2 + \alpha}{n + \alpha k}, \cdots, P(c_k) = \frac{n_k + \alpha}{n + \alpha k}$$

The addition of α to the numerator and αk to the denominator prevents any command from having an estimated probability of zero. The term α is a configurable parameter of the detector called the pseudocount. In accordance with Maxion [7], we set the pseudocount to 0.01.

The detector is called naive Bayes because it (naively) assumes that the probability of a session of commands is simply the product of the probabilities of the individual commands (as though each command's probability were independent). The anomaly score for a test session is the negative logarithm of the probability of the session given the profile. To account for sessions of different length, the anomaly score is normalized by dividing by the number of commands in the session. Formally, let n' be the number of commands in the test session, and let c'_1 be the name of the first command, c'_2 be the name of the second command, and so on. The anomaly score is calculated using the probability estimates for each command from the training-data profile as follows.

$$\text{anomaly score} = -\frac{1}{n'} \sum_{i=1}^{n'} \log P(c'_i)$$

Using the anomaly score, the detector makes a determination as to whether or not an alarm should be raised. The determination is made by comparing the anomaly score to a threshold. If the score exceeds the threshold, an alarm is raised. The threshold is calculated using five-fold cross-validation on the training data. The training sessions are divided into five groups called *folds*. The first fold is set aside and the remaining four-fifths are used to build a profile. The anomaly scores are then calculated for the first fold using that profile. Anomaly scores for each of the other folds are calculated in the same way (using a profile built on the four-fifths of the training data not in the fold). The maximum score in each fold is calculated, and the threshold is calculated as the average of these maximum scores.

For each of the eight evaluation data sets, we trained the naive-Bayes detector on the training sessions and tested it on the test sessions. Each evaluation data set contained 4 attack-injected sessions for each of the 4 subjects, leading to a total of 16 attack-injected sessions. Each data set contained 10 non-attack sessions of test data for each of the 4 subjects, leading to a total of 40 sessions of non-attack test data overall. The anomaly score and detector response (i.e., alarm or no alarm) were recorded for each test session.

To obtain the responses of the detector on the Truncated and Enriched evaluation data sets derived from the subject's raw data, we deployed the naive-Bayes detector on each subject's workstation. The subjects ran the detector on each of their two raw evaluation data sets, inspected the log containing the detector's responses to confirm that it contained no sensitive information, and released it to the researchers.

7. Results and analysis

The consequences of each of the three sanitization strategies on the experiment are shown in Table 1. The performance of naive Bayes is shown for each of the eight evaluation data sets in terms of misses, false alarms, and cost of error. For consistency with Maxion's analysis [7], the cost of error was calculated as the sum of the miss rate and the false-alarm rate. Since he compared the cost of Enriched data to Truncated data, the cost ratio of the two has been calculated for the raw data and for each of the three sanitization types. Unlike the earlier experiment which found a cost reduction of 9% by using Enriched data, we found Enriched data to have a 79% higher cost than Truncated data when using no sanitization (i.e., the raw data). We discuss possible explanations for this discrepancy in Section 8.

Regarding the consequences of sanitization, if there were no sanitization artifacts, all the results should match the raw-data results. However, only Word-Token sanitization produced results that match the raw-data results. Token-Only sanitization increased the cost of using Truncated data, while Redact-Only sanitization reduced the cost of using Enriched data.

It may seem that Token-Only sanitization has no effect

Sanitization Strategy	Data Type: Truncated Misses	Truncated False Alarms	Truncated Cost	Enriched Misses	Enriched False Alarms	Enriched Cost	Cost Ratio
Raw / None	25% (4/16)	10% (4/40)	0.350	50% (8/16)	12.5% (5/40)	0.625	1.79
Redact-Only	25% (4/16)	10% (4/40)	0.350	37.5% (6/16)	10% (4/40)	0.475	1.36
Token-Only	37.5% (6/16)	7.5% (3/40)	0.450	50% (8/16)	12.5% (5/40)	0.625	1.39
Word-Token	25% (4/16)	10% (4/40)	0.350	50% (8/16)	12.5% (5/40)	0.625	1.79

Table 1. The detector's hit and false alarm statistics are shown for each of the eight evaluation data sets. The cost of error is calculated as the sum of the miss and false alarm rates, and the final column indicates the ratio of the Enriched cost to the Truncated cost. Only the Word-Token sanitization results exactly match the raw-data results for both Truncated and Enriched data.

on Enriched data, or that Redact-Only sanitization has no effect on Truncated data. Further, it may seem that the consequences of using either sanitization strategy instead of raw data are small. For instance, Redact-Only sanitization merely changes the detector's response to three Enriched sessions (i.e., two fewer misses and one fewer false alarm). However, misses and false alarms are the coarsest summary of the experimental outcome. Sanitization also affects the anomaly scores and alarm thresholds used by naive Bayes, and in these terms, the effects of Redact-Only and Token-Only sanitization are much more pronounced, and seen across Truncated and Enriched data sets.

The anomaly scores computed by naive Bayes for each session describe the results of the experiment in finer detail. The score gives valuable insight into the decision procedure used by the detector, and it can help explain why errors occurred (e.g., why an attack-injected session was missed). Without accurate anomaly scores, it is harder to identify what a detector did wrong and how to improve it. The effects of Redact-Only and Token-Only sanitization are much greater on the anomaly scores than on the hit and false-alarm statistics. Table 2 shows that Redact-Only and Token-Only affect many Truncated and Enriched anomaly scores while Word-Token does not. Redact-Only has the greatest effect on the anomaly scores, with all the Enriched anomaly scores differing from the raw data scores. Redactions in the training data caused all the probability estimates to change; in turn, the probabilities altered the anomaly scores.

The alarm threshold is used by naive Bayes to decide whether an anomaly score should trigger an alarm. It depends on the anomaly scores calculated during cross validation, and when the anomaly scores change as a result of sanitization, so does the threshold. Redact-Only sanitization tended to lower the threshold (partially explaining the lower miss rate on the Enriched data), while Token-Only sanitization tended to raise it (partially explaining the higher miss rate on the Truncated data). Again, Word-Token sanitization has no effect on the alarm threshold.

Sanitization Strategy	Data Type: Truncated	Enriched	Overall
Redact-Only	29%	100%	64%
Token-Only	50%	50%	50%
Word-Tokens	0%	0%	0%

Table 2. The percentage of anomaly scores altered as a consequence of each of the three sanitization strategies is broken down by data type. Only the Word-Token sanitization strategy did not alter the score.

8. Discussion and future work

This research confirms that Word-Token sanitization is artifact-free for an experiment that evaluates the naive-Bayes anomaly detector. In theory, Word-Token sanitization should be artifact-free for any experiment where tokens can be substituted for words. For instance, it should not introduce artifacts in the evaluation of other anomaly detectors in the same family (e.g., Lane and Brodley's and Schonlau et al.'s detectors). Whereas Word-Token sanitization replaces every occurrence of a word with a symbol, a more lenient sanitization strategy might only replace occurrences that are in the same context (e.g., "God" will only be replaced with `<TOKEN-12>` when it appears as a password). This strategy might provide more privacy while continuing to avoid introducing artifacts. Experiments with a variety of detectors and different sanitization strategies are needed to further develop this theory.

When tokens cannot be substituted for words (e.g., in evaluations of signature-based systems), Word-Token sanitization may introduce artifacts. However, just as Word-Token sanitization was designed to be artifact-free for a certain family of detectors, a different artifact-free sanitization technique might be designed to accommodate different detectors. The lesson remains that sanitization can have unde-

sirable consequences on the outcome of an experiment, and it must be accommodated by researchers.

Since sanitization artifacts alter the results of experiments, it stands to reason that the unrealistic use of benign commands as a substitute for insider commands will cause similar problems. In fact, our use of more realistic insider-type behavior may explain the discrepancy between our findings and Maxion's about the cost of using Truncated and Enriched commands. (Note that this explanation is conjecture, since the goal of this study was not to refute that earlier work, and the discrepancy might be explained by other means, such as our small subject-pool size.) However, we want to reiterate that unrealistic use of benign commands in insider-injections may have the same dangerous consequences as sanitization artifacts. An investigation of the consequences of unrealistic insider injections remains for future work.

9. Summary and conclusion

Sanitization artifacts did not affect the experiment when Word-Token sanitization was used, but they did alter the results when two other strategies were used (Redact-Only and Token-Only). Since existing insider-threat data sets [2, 5, 11] all used some form of redaction or tokenization, these findings cause concern. Conclusions based these data sets may not generalize to the real world. Insider-threat data sets must be validated before we place real-world reliance on them. If an insider-threat detector is chosen for deployment on the basis of precarious conclusions, and that detector under-performs, the consequences carry considerable risk. On the other hand, we demonstrate that appropriate tools and methods do make it possible to collect realistic data and perform sanitization without introducing artifacts.

10. Acknowledgments

The authors are grateful for helpful comments from Kymie Tan and Dan Siewiorek, as well as from anonymous reviewers. Many thanks to Fahd Arshad for his implementation of the sanitizer application. Thanks also to the CMU/CS Operations Group for help in the evaluation portion of the project. This work was supported by National Science Foundation grant number CNS-0430474, and by the Army Research Office through grant number DAAD19-02-1-0389 (Perpetually Available and Secure Information Systems) to Carnegie Mellon University's CyLab. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

References

[1] D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2), February, 1987.

[2] S. Greenberg. Using Unix: Collected traces of 168 users. Technical Report 88/333/45, Department of Computer Science, University of Calgary, Calgary, Canada, 1988.

[3] The Honeynet Project. *Know Your Enemy: Sebek*, November 2003. `http://www.honeynet.org/papers/sebek.pdf`.

[4] M. Keeney, E. Kowalski, D. Cappelli, A. Moore, T. Shimeall, and S. Rogers. Insider threat study: Computer system sabotage in critical infrastructure sectors. Technical report, U.S. Secret Service and CERT Coordination Center/SEI, 2005.

[5] T. Lane and C. E. Brodley. An application of machine learning to anomaly detection. In *Proceedings of the 20th Annual National Information Systems Security Conference*, pages 366–380. Held on 7–10, October, 1997, Baltimore, MD, NIST, 1997.

[6] M. V. Mahoney and P. K. Chan. An analysis of the 1999 DARPA/Lincoln Laboratory evaluation data for network anomaly detection. In *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID-2003)*, pages 220–237. Held on 8–10 September, 2003, Pittsburgh, PA, Springer-Verlag, Berlin, 2003.

[7] R. A. Maxion. Masquerade detection using enriched command lines. In *International Conference on Dependable Systems and Networks (DSN-03)*, pages 5–14. Held on 22–25 June, 2003, San Francisco, CA, IEEE Computer Society Press, Los Alamitos, CA, 2003.

[8] R. A. Maxion and T. N. Townsend. Masquerade detection augmented with error analysis. *IEEE Transactions on Reliability, Special Section on Quality/Reliability of Engineering of Information Systems*, 53(1):124–147, March 2004.

[9] R. Pang and V. Paxson. A high-level programming environment for packet trace anonymization and transformation. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'03)*, pages 339–351. Held on 25–29 August, 2003, Karlsruhe, Germany, ACM Press, 2003.

[10] M. R. Randazzo, M. Keeney, E. Kowalski, D. Cappelli, and A. Moore. Insider threat study: Illicit cyber activity in the banking and finance sector. Technical report, U.S. Secret Service and CERT Coordination Center/SEI, 2004.

[11] M. Schonlau, W. DuMouchel, W.-H. Ju, A. F. Karr, M. Theus, and Y. Vardi. Computer intrusion: Detecting masquerades. *Statistical Science*, 16(1):58–74, 2001.

[12] S. E. Smaha. Haystack: An intrusion detection system. In *Proceedings of the Fourth Aerospace Computer Security Applications Conference*, pages 37–44. Held on 12–16 December, 1988, Orlando, FL, IEEE Press, 1989.

[13] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):557–570, 2002.