January 2014

Insider Threat Detection with Text Libraries and Machine Learning

Bryan Z. Li
Worcester Polytechnic Institute

Patrick Ryan Kane
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation
Insider Threat Detection with
Text Libraries and Machine Learning

A Major Qualifying Project
Submitted to the Faculty of
Worcester Polytechnic Institute
In partial fulfillment of the requirement for the
Degree in Bachelor of Science
in
Computer Science

By

Bryan Z. Li

Patrick R. Kane

Date: 12/19/2013
Project Advisor:

Prof. Craig Shue, Advisor

This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review. For more information about the projects program at WPI, see http://www.wpi.edu/Academics/Projects
Abstract

Company networks are protected by firewalls and other measures to ensure that data remains secure. However, these same companies have limited techniques to protect information from being compromised from inside the closed network. According to CERT, the Computer Emergency Readiness Team, over 700 unique cases are currently documented and growing rapidly [20]. Sensitive company documents are leaked every year by means of email, printed documents, USB flash drives, and other mediums of data storage. Because there are many legitimate reasons for accessing secret data, insider threats are extremely difficult to detect. A company cannot simply create a policy that bans the use of USB drives or printing, because there are many instances where an employee would be required to print or transfer files using a USB flash drive. In addition to the difficulty of detecting insider threats, the consequences of such attacks can be catastrophic, as is the case with the whistleblower Edward Snowden and his release of documents pertaining to the NSA. While flooding the news, Snowden's leaks are some of the most widespread ever recorded, and the documents are continually being released to the press. In an effort to detect insider threats, we researched the possibility of using the text from a user's screen to detect anomalies in user behavior and classify certain text as sensitive. By monitoring the text on a user's screen as well as the different interface protocols used by the computer, we are able to capture every useful action committed on a computer at any given time. This proved an effective approach because by detecting abnormal behavior, we are able to record the incident, where an auditor can then view the scenario where the behavior took place, and take the appropriate actions. The Insider Threat Detection System provides an extra layer of security for preventing unauthorized data from being released outside a company’s secure internal network.

Keywords: insider threat, security, computer systems
# Table of Contents

Abstract.................................................................................................................................2

1   Introduction ........................................................................................................................5
  1.1  Proposal ............................................................................................................................7

2   Background ..........................................................................................................................8
  2.1  Overview of Insider Threat Detection Work .................................................................8
       2.1.1  Computer Emergency Readiness Team ...............................................................9
       2.1.2  SpectorSoft Spector360 .....................................................................................9
       2.1.3  Raytheon Sureview .........................................................................................11

3   Bottom-up approach for text capturing ..............................................................................12
  3.1  X Server .........................................................................................................................12
  3.2  Client-Server communication link ................................................................................13
       3.2.1  Window Manager Approach ...........................................................................13
       3.2.2  XNEE ...............................................................................................................14
  3.3  Libraries Detection ..........................................................................................................18
  3.5  FreeType2 ......................................................................................................................21
  3.6  Pango ............................................................................................................................22
  3.7  GTK+ .............................................................................................................................24

4   Capturing User’s Actions .....................................................................................................26

5   System call monitor with Auditd [15] ................................................................................27

6   Machine Learning for Insider Threat Detection .............................................................29
  6.1  Text preprocessing [16] ...............................................................................................30
  6.2  Latent Dirichlet Allocation (LDA) ...............................................................................30
       6.2.1  Basics of Latent Dirichlet Allocation ...............................................................30
       6.2.2  LDA Application ...............................................................................................31
  6.3  Support Vector Machines (SVMs) ...............................................................................31
       6.3.1  Process SVMs Input ...........................................................................................32
       6.3.2  SVMs kernel function .........................................................................................33
       6.3.3  Training the SVMs Model ...............................................................................37
       6.3.4  SVMs Tools/Implementation .........................................................................37

7   Performance Analysis .........................................................................................................38
7.1 Results........................................................................................................................................40
8 Conclusion and Future Work .......................................................................................................42
References .......................................................................................................................................44

TABLE OF FIGURES

Figure 1 Basic Ubuntu X11 Architecture for text rendering ..........................................................12
Figure 2 Simple X11 App and Text that are captured ..................................................................21
Figure 3 Auditd system architecture .............................................................................................28
Figure 4 decision surface with maximum margin vs. non-optimal decision surface ..............34
Figure 5 Soft margin ......................................................................................................................35
Figure 6 Non-linearly separable data ............................................................................................35
Figure 7 transform the non-linearly separable data to linear .......................................................36
1 Introduction

Every year, reports are published of companies that have data stolen not only from people breaking into the network from the outside, but of leaks coming from inside a company network. One of the most notable and damaging leaks was released by the whistleblower Edward Snowden, who publicized many sensitive documents revealing the inner workings of the National Security Agency (the “NSA”), as reported by Greenwald, MacAskill and Poitras in *The Guardian* [30]. Describing the interception of network traffic from large datacenters and capturing location data from cell phones, the documents leaked by Snowden were previously unknown to the public, and have since raised many questions about personal privacy. Large scale threats like the one in Snowden’s case aren’t common, but they can be catastrophic to the parties involved. Even leaks on a smaller scale can have devastating effects. In a less publicized example, an employee sent 228,000 medical patient records to himself via email [28]. Sent via plaintext, these hospital records contained the personal information for most patients in the hospital, including SSNs and credit card information in addition to addresses and phone numbers.

These inside threats, while harmful to the organization affected, are extremely difficult to detect because in most cases the malicious user has full access to the information. One thing that companies don’t have control over is the intent of the employees. A major concern in computer security is authorization. Security threats largely arise because many companies do not have policies in effect that determine if the recipient is also authorized to access the file. “Authorization defines users’ permissions in terms of access to digital resources and extent of its usage” [31]. When a user is authorized to access a file, it is possible in many cases to print, move/copy, or even email the file to any recipient, but the resulting intent of the employee is what defines the situation as a threat to the data [31]. Not only would such a check be a cumbersome task, checking the authorization of all viewers of the information would prove difficult because of the possible mediums used. For example, if an authorized user prints a sensitive document and hands it to a friend, the company has no feasible way to ensure that the friend has the correct permissions to possess or even view the file.
To address these issues, our approach does not try to determine the authorization of the recipient, but rather determines if the actions of the user are abnormal. Since sending a sensitive document through unencrypted email can be an insider threat, our software will recognize that sensitive text is being sent through email, and that this action is irregular. It will additionally provide an interface for the auditor to view in real time the scenario that triggered the alert, as well as the user actions that were performed during the threat. While our software will not stop the action from taking place, it will detect a possible malicious user from the first offense, and the auditor can then take the necessary steps to ensure that the situation is remediated.

Many actions of a user could be considered malicious depending on the situation. Every attack is different, and there are many ways in which the attacker accomplishes an attack. In an analysis of cyber fraud in financial services by the Computer Emergency Readiness Team (CERT), many patterns of malicious activity were common. One example in CERT’s analysis describes scenarios where a criminal executed a “low and slow” approach to steal privileged data. By laying low and accessing the data infrequently and in small quantities, the criminal was able to stay undetected for longer, and inflict more damage to the company [29]. Because accessing large amounts of data or the often printing of documents would be noticeable to other employees and supervisors, the low and slow approach is effective because the frequency of such malicious actions is so small that there is no need for suspicion. In addition, very few of the subjects served in a technical role (e.g., database administrator) or conducted their fraud by using explicitly technical means [29]. They define technically unsophisticated as printing data, emailing data, or copying it to external media. Privileged users often would not go through the trouble of breaking the encryption of the file or trying to bypass the company firewall to allow others to investigate the company files. Instead, they relied on the simple forms of electronic communication, and were often undetected until a periodic audit, a coworker suspicion, or a customer complaint [29].

Our software is able to detect simple user actions such as key presses and mouse clicks as well as more advanced information such as system calls and network traffic. By analyzing the text on the user’s screen in addition to the internal system calls and network traffic, our software can more easily create an overall picture about what the user is doing in order to protect the
company against not only the well-crafted attacks from inside a company network, but also the technologically unsophisticated threats which can be more difficult to detect.

1.1 Proposal

In this project, we research and develop a system that addresses the problem of data being leaked from inside a protected network. We do not attempt to mediate or stop the user from committing a certain action, but we provide an interface for an auditor to monitor in real time everything that a user does. By modifying the graphics libraries in Linux, we are able to output information to a log file every time an object on a screen is created or modified. For example, if a simple application contains a text box, a label, and a button, we are able to record the time that the object was created, as well as the text that appears in each object. So if a user starts typing “Hello World” into the textbox in the sample application, we can record the time that he started typing, the text that was typed, and any subsequent actions after that. By using GTK+ and Pango to intercept the text that is being processed, we can replay the scene in a text only environment to make processing the text in the machine learning algorithm simpler by dealing solely with text and not graphical text or glyphs. By accessing the text handling functions in the inner workings of the graphical libraries, we can avoid the long and tedious process of trying to convert glyphs and anti-aliased text back into text using Object Character Recognition or other similar means. Certain functions in GTK+, mostly the ones that handle text views and buffers, can be used to log the text on the screen. By modifying the source code of GTK+ and including statements to print to a file of our choosing, we can log the time and the actual text being created or modified. From there, a separate program is able to read the log files that are being created and input the text into a hash map. By updating the timestamps of each captured word, we are able to deduplicate the incoming data, and thus reduce storage and processing overhead. Finally, taking this unordered map and analyzing it using machine learning techniques, we can classify sensitive text based on clusters. When sensitive text is detected, a link is added to the interface that allows an auditor to not only view the screen capture of the user’s desktop, but also view the words that were classified as sensitive, and the system calls invoked during the action. With this approach, an auditor can then view all of the relevant information in what has been classified as a possible internal threat, and the auditor can then take the necessary administrative steps to remediate the situation.
2 Background

Because our software touches upon many different facets of computer science, there is a plethora of relevant background information available on the topic. From other systems that attempt to detect and classify insider threats to general company policies that are enforced to try to avoid insider threats, insider threats have been a serious issue in computer security, and thus it is a fairly well documented subject. As previously mentioned, CERT is an organization that records insider threats and techniques used in previous attacks, providing a good overview of the psychology that is involved in the insider threats as well as the various techniques most commonly used in the attack. The organization has researched the different kinds of threats as well as the thought process behind the threats, and this proved tremendously useful in determining which kinds of user actions we want to record in our Insider Threat Detection System.

2.1 Overview of Insider Threat Detection Work

Previous research on insider threat detection done by Bai [1] suggests that using the logging capacity of the Linux system, we can detect insider threats earlier in their attack, rather than after the insider has performed the malicious activity. The advantage for this method is that it requires minimal additional cost and little overhead to employ. However, in order to catch any malicious activities, we will first need to know the pattern of that malicious activity, and then query the logs for that specific malicious pattern. A similar mechanism is also developed for Windows Operating System [2].

Another approach is to log the activities on the physical hardware level. Research by Woolingham [3] presented a method to detect insider threats against Cisco network devices. This method detects insider threat by analyzing network traffics that pass through the network infrastructure devices. The method does not require major change for the existing network devices, which makes it very easy to apply. However, this method only monitors network traffic, so if the insider is performing malicious activity on the local machine, it will not be detected by the system.

According to Crawford and Peterson [4], insider threat detection can also be done with virtual machine introspection. In simple words, virtual machine introspection is a concept that using the host machine to monitor the guest virtual machine. The guest machine is vulnerable to insider threat and malicious software, once the guest host is compromised,
the attacker can modify the guest OS setting so that the administrator cannot detect the malicious activities. By using virtual machine introspection, the host machine continuously monitors the guest’s activities despite the setting of the guest VM. Also the guest OS is unaware of the existence of such monitoring mechanism therefore the malicious insider is not able to compromised the host machine even though he/she has full privileges on the guest VM.

2.1.1 Computer Emergency Readiness Team

According to CERT [29], there is a noticeable correlation between employees involved in insider threats and factors such as working conditions and external factors. Some of the topics that they discussed were the collusion of employees in insider threats with outsiders. Many of the insiders were convinced by outside influences. Criminal organizations, governments, and even single motivated outsiders can influence employees to steal data or try to create holes in a company’s external defenses. Aside from outsiders, employees of trusted business partners can often be the party to blame, because they have full access to the information, and no real motivation to be loyal to the partnering company. In addition, employees feeling stressed or angry during mergers or acquisitions between companies can be the insider through which the data is leaked. Other factors, though less common, include cultural and behavioral differences between employees, and even pay grade differences. Last and certainly not least, international issues, political turmoil, and the desire for the upper edge in an international dispute can lead to insider threats and dire consequences. Published in CSO Magazine in 2011, CERT found that in cases where respondents could identify the perpetrator of an electronic crime, 21% were committed by insiders. Also, 43% of the respondents of the 2011 CyberSecurity Watch Survey experienced at least one malicious, deliberate, insider incident in 2010[29]. Collecting data since 2001, CERT has determined that cyber security threats from inside a company’s closed network are a real and serious issue.

2.1.2 SpectorSoft Spector360

One corporate solution currently used is a software called Spector360. This software is advertised as a complete employee monitoring software that will detect insider threats. They describe insider threats as being data theft, loss, corruption or deletion, employee fraud, the insertion of malicious code, identity theft, or even productivity loss. The goal of this software is
to monitor the actions of a user, classify the actions as malicious or not based on pattern matching, and alerts a supervisor of such threats. Although limited, some of the patterns that Spector360 matches are printing more than 50 documents, copying files to a USB drive, Dropbox, or burned to a DVD, typing confidential project names using Gmail, or accessing key applications after hours. There are multiple strategies that SpectorSoft uses to detect these patterns, including recording specific application information such as IM or web traffic or email. When a malicious pattern is matched, an alert is sent to a supervisor with the relevant information pertaining to the threat [32]. While this approach seems promising, there are several issues with the design of Spector360. First of all, by scanning the input and output of specific applications, the software is limited to those specific applications. If a user downloads and uses another application, there is no guarantee that the software will be able to detect malicious activity within that given application. So, if a user knew that the AIM instant messaging client was monitored by Spector360, he could simply use a separate IM client such as Pidgin, and could effectively say whatever he wanted over IM without being detected as a malicious user. In a more serious example, a user could use another, less common email client, and could send in plaintext or as an attachment any document the user wishes. Because the user is not transferring the document to a storage medium or emailing using the monitored client, he would not be flagged as a malicious user.

What sets our software apart from Spector360 is the design of the overall software. Instead of targeting specific applications and monitoring the traffic associated with them, we capture the text that appears on a user’s screen through the graphical libraries, which includes all applications on the system. Rather than limit ourselves to applications, we assume that the interaction between the user and the computer is through a GUI, so we can capture all of the relevant information about the scenario in which the user is interacting. In addition, our software does not limit the detection of insider threats to pattern matching. Instead, we use machine learning techniques to detect abnormalities, and use initial training data to determine sensitive threats, and classify a threat based on whether the text on the screen at a given time is sensitive or not. We then provide a video interface for a supervisor, where they can view in near real-time the scenario that occurred to flag the text as sensitive. Although much training data is used, and even though the initial trial of the software may yield false positives, the effectiveness of machine learning with a large data set is extremely effective and accurate.
Because of the scalability of our software, it sets us apart from SpectorSoft and their pattern-matching auditing software that targets specific applications.

2.1.3 Raytheon Sureview
Raytheon, a leading aerospace defense engineering company, developed their own software to detect insider threats, and alert a supervisor of the incident. Much like the SpectorSoft software, Raytheon’s approach involves targeting specific mediums such as individual applications and network traffic to detect malicious activity. Sureview also uses pattern matching on user actions to profile user behavior. They rely on specific combinations of user actions to create an insider threat, which can be misleading. While pattern matching can be effective in small datasets, insider threats are getting more and more creative and well thought out, so relying on a pattern to catch a harmful employee could be ineffective [33]. Our software offers many of the same benefits over Sureview as it did Spector360. We do not rely on pattern matching to reach a conclusion, but rather an output of possible sensitive text that could lead to a misuse of information.
3 Bottom-up approach for text capturing

In Figure 1, we show the architecture of how text data is passed from the X Client to X Server. The reason we decided to work from the bottom-up is that if we manage to capture all text at the very low level such as Xlib and the communication link, we can essentially apply this technique to all UNIX/Linux distributions because they all use X11 as their displaying system. As we move up the chain, there are more and more variations for the rendering libraries. For example, FreeType, TrueType and OpenType libraries are each used by different Linux distributions to do similar things.

3.1 X Server

The X Window System is a bitmap displaying system commonly used in UNIX and Linux operating systems. It uses a basic client-server model. The X Client means the GUI application that is running on the system, the X Server is a low-level application that takes
requests from the clients, and communicates these requests to the hardware drivers. A standard UNIX/Linux operating system such as Ubuntu will have one X Server and multiples X Clients. One thing to note is that the X Client-server notation is usually confusing. For example, a user's machine uses X11 forwarding to retrieve the application GUI from a central server. In this case, the user's machine is the X Server, and the application that is running on the central server is the X Client.

According to Mansfield [5], every hardware interaction (keyboard, mouse and monitor) is handled by the X Server, which means that everything that is displayed on the user's screen must be first processed by the X Server and then is passed down to the hardware drivers. In theory we can basically monitor the X Server and know what exactly is happening on the user screen. Mansfield suggested that a tool call XTrap can be used to intercept and process all X Requests and X Events that passed through the X Server.

XTrap [6] is an X Server extension that is used to intercept all the events (Packets sent from the X Server to the X Clients) and requests (Packets sent from the X Clients to the X Server) for debugging purposes. Having studied this server extension, we discovered several issues with this approach. First, all the interception is done on the X Server side, which is typically the user's local machine, the user will have full control of the system. Secondly, the extension itself is obsolete and no longer supported; therefore the modern X Server (since version X11R6) no longer lists XTrap as its available server extension.

### 3.2 Client-Server communication link

Mansfield [5] also suggests that capturing X communication in the X Client and X Server link has a similar effect to capturing X communication on the server side. One advantage of that change is the capturing mechanism is not hosted on the user machine; therefore the user does not have control of it. This approach is similar to the man-in-the-middle, the monitoring mechanism will sit between the X Client and X Server, every time it sees any X traffic, it will make a copy of the actual content and send the original copy along the way.

#### 3.2.1 Window Manager Approach
The window manager is a perfect example of this man-in-the-middle approach. It appears to the X Server as a regular X Client application and appears to the rest of the clients as the X Server. We can consider the X window architecture as a tree; the root is the main X11 window, which communicates directly with the server. Without the Window Manager, all other application windows are children of this main window. The Window Manager is a client application that is a direct child of the root window, the window manager will re-parent all its siblings so that all of the applications now are the children of the window manager, and the window manager will remain the only child of the root window. Every time an X Client application requests a window right below the root window, the window manager will re-parent it so that the new application will be a child of the window manager. Every communication between the applications and the root window must go through the window manager.

The initial approach is to study different varieties of window managers and see how they handle inter-client communication. We initially downloaded and installed the source of the most simple window manager TinyWM [7], and planned to modify it to fit our need. After working with it for a while, we determined that using the TinyWM will affect the user experience since it is a minimal windows manager implementation; therefore we decided to find a more advance version window manager implementation or other tools to study, and then discovered a tool call xnee.

### 3.2.2 XNEE

Xnee [8] is a GNU open source tool that is used to record and replay user actions under the X11 environment. The xnee project focuses on recording events (traffic from server to client); however, the recorder will intercept all X11 communication, but only process the events. So if we want to use xnee to capture requests, we can simply implement the request handler in the xnee project.

**XNEE Usage**

Xnee is the general name for the project, the project includes a GUI application called gnee and a command line application called cnee. For our purpose, we will only need to use the command line tool cnee. Under the Debian distribution, xnee can be installed by simply running the command "sudo apt-get install xnee". In our case, we need to be able to modify
the source of xnee, therefore we need to run “sudo apt-get source xnee” and install the project from source.

Once the tool is installed, to use it to capture all the requests packets, run the command:

```
cnee --record -reqra 01-119 -o out -e out-e
```

The option -reqra specifies the range of the requests that we want to capture, 01-119 is all the X request types. This command will create two files under the current directory, named `out` and `out-e`. The file `out` contains information about the requests captured during the xnee session, the `out-e` file contain any error message during the xnee session. Below is the format of the output entries for the request:

<table>
<thead>
<tr>
<th>Optcode</th>
<th>request type</th>
<th>request type</th>
<th>request length</th>
<th>request id</th>
<th>server time</th>
</tr>
</thead>
</table>

* Each of this field is represented by a numeric value.

**Properties and Atoms**

Before describing how to add request handler, we need to understand how the X Window System sends and stores its data. X uses properties to describe information; a property is a collection of named, typed data. The window system has a set of predefined properties (for example, the name of a window, size hints), and users can define any other arbitrary information and associate it with windows. Each property has a name, which is an ISO Latin-1 (8-bit single-byte coded graphic character sets) string. For each named property, a unique identifier called “Atom” is associated with it. A property also has a type, for example, a string or integer. These types are also indicated using atoms, so arbitrary new types can be defined. Data of only one type may be associated with a single property name. Clients can store and retrieve properties associated with windows [9]. An atom is essentially a long integer, the relationship between an atom and a property is like the relationship between the key and value in a hash table. For efficiency reasons, an atom is used rather than a character string because atom requires less memory than character string, also integers comparison cost less than strings comparison.

A property is stored in one of several possible formats; the X server can store the information as 8-bit quantities, 16-bit quantities, or 32-bit quantities. This permits the X
server to present the data in the byte order that the client expects. If the specified format is 8, the property data must be a char array. If the specified format is 16, the property data must be a short array. If the specified format is 32, the property data must be a long array. To look for any text that is passed to the X server, we can specifically look for properties request with 8-bits data type because the 8-bit format is used for character strings.

**Unit for XNEE and XLib**

In order to properly parse the X request packets, we will need to understand how the X Window System quantified its data. Every X packet has fields called “length” and “nUnits” which indicates the size of the packet. The length field indicates the total bytes in the packet in 4 byte unit, including the header and data. Therefore:

$$\text{Total packet size in bytes} = \text{length} \times 4$$

The main packet we will capture is the xChangeProperty packet. For the xChangeProperty packet, the header itself contains 6*4 bytes of data. The “nUnits” field indicates how many units of data are following the header; this field depends on the format.

For example, if the xChangeProperty packet contains 32 bits data format, and the data portion of the packet has 2 units of this 4 bytes data format, the length field for this packet will be 8 (6*4-bytes header + 2*4-bytes data) and the “nUnits” field will be 2 (2 unit of data).

**Working with xChangePropertiesReq [10]**

To display content such as text in a window, the client application will call the XLib function xChangeProperty(), which will send an xChangePropertiesReq packet to the X Server, the packet contains detail information about the content and how the content should be displayed. Based on the information in the packet, the X Server will send a command to the hardware driver to update the render the content.

Below is the definition of the xChangePropertyReq structure:

```c
typedef struct {
    CARD8 reqType;    //indicate the format of the data, 8-bit, 16-bit or 32-bit
```
CARD8 mode;
CARD16 length B16;
Window window B32;  // specify the window id
Atom property B32, type B32;
CARD8 format;       // similar to reqType
BYTE pad[3];        // pad byte, make the header 6 units = 24 bytes
CARD32 nUnits B32;  // length of stuff following, depends on format
} xChangePropertyReq;

Every captured request will be first cast to a generic structure, xResourceReq, which is used to determine the request type. To properly parse the data into the correct structure, first we will need to examine the type of request base on the xResourceReq, and then cast the data into different request struct based on the type field in the xResourceReq. For the xChangeProperty request, we will check if the packet type is 18 (integer representation of change properties request), and then cast the XReq field in the XRecordDatum into xChangePropertyReq (instead of the standard xResourceReq). Having done that allows us to access the actual data that is passed to the X Server, we can then print out this data for analysis.

**XNEE Text Capture Experiment**

To display the Atom name to determine the data format, use the Xlib function call

```c
char *XGetAtomName(Display *display, Atom atom);
```

For the *display, use the xnee data structure xnee_data->control, for the atom, use the xChangePropertyReq -> type to determine the format of the data.

For every xChangeProperty with STRING or UTF8_STRING format, we examine the “nUnits’ field to determine how many characters are following the packet header. And then we try to print out the exact number of characters indicates by the “nUnit” field. However, the output does not look like a complete sentence; many of them are displayed as nonprintable
Based on the experiment, we conclude that the actual strings of characters might be turned into bitmap before they reach the link; we will need to work upward to see where we can get the actual raw text. Even though XNEE did not help us capture the text displayed on the screen, we will continue to use it as part of our system; this will be discussed later in the “Capturing user actions” session.

### 3.3 Libraries Detection

Before we start investigating each individual library, we need to make sure that the application we want to monitor is actually using the library to render its graphics. For Linux operating system, there is a command call “ldd”, which will list all the shared libraries (libraries that are installed in the operating system and used by other applications) an application is using. To use the command, follow the syntax:

```
ldd /Path/to/bin/file
```

For example, if we want to know the shared libraries that Emacs uses, we will run the command: "ldd /usr/bin/emacs". We run the “ldd” command for some of the application we want to monitor. Below, we list the libraries related to text rendering that these application used:

<table>
<thead>
<tr>
<th>Application</th>
<th>Related Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emacs</td>
<td>Xft, XLib, Pango, Cairo, FreeType, GTK, libm17n, libgdk-X11, libfontconfig</td>
</tr>
<tr>
<td>Gnome-terminal</td>
<td>Xft, XLib, Pango, GTK, Cairo, FreeType, libtinfo</td>
</tr>
<tr>
<td>Evince (PDF viewer)</td>
<td>Xft, Xlib, FreeType, Pango, Cairo, GTK</td>
</tr>
<tr>
<td>vi / vim</td>
<td>libtinfo, Xft</td>
</tr>
<tr>
<td>Nautilus</td>
<td>Xft, XLib, Pango, Cairo, FreeType, GTK, libm17n, libgdk-X11</td>
</tr>
<tr>
<td>Gedit</td>
<td>Xft, XLib, Pango, Cairo, FreeType, GTK</td>
</tr>
<tr>
<td>Firefox</td>
<td>Not a dynamic executable, which means that they don't use the shared libraries in the operating system; they include a copy of their own libraries in their source code.</td>
</tr>
<tr>
<td>OpenLibre Office</td>
<td></td>
</tr>
<tr>
<td>Chrome</td>
<td></td>
</tr>
</tbody>
</table>

The result shows that these applications use pretty much all of the text rendering libraries we know, so we will need to study each library. We note that Firefox, OpenLibre Office,
Chrome are not dynamic executables, which means that they will have their own copy of the text libraries they use.

### 3.4 XLib [11]

Xlib is the basic X system library written in C. It provides functions for connecting to a particular X server, creating window, drawing graphic and responses to events. Xlib calls are translated to protocol request that are passed either to local or remote X Server. For example, an application uses the Xlib function call CreateWindow() in CrWindow.c to send an xCreateWindowReq to the X Server. Once the X Server receives this packet, it will interprets the information and create a window for the application. Once the window is created, the application will call the Xlib function calls ChangeProperties() in Chprop.c to send packets to the X server to request a window update.

Xlib provided multiple text rendering functions for different situations. These functions are [12]:

<table>
<thead>
<tr>
<th>Function</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDrawString()</td>
<td>Text.c</td>
</tr>
<tr>
<td>XDrawString16()</td>
<td>Text16.c</td>
</tr>
<tr>
<td>XDrawImageString()</td>
<td>ImText.c</td>
</tr>
<tr>
<td>XDrawImageString16()</td>
<td>ImText16.c</td>
</tr>
<tr>
<td>XDrawText()</td>
<td>PolyTxt.c</td>
</tr>
<tr>
<td>XDrawText16()</td>
<td>Polytxt16.c</td>
</tr>
<tr>
<td>StrToText()</td>
<td>StrToText.c</td>
</tr>
</tbody>
</table>

XDrawString() and XDrawString16() draw a string into a drawable (e.g. a text panel, window title). They required only the character string to be drawn, the length of the string and the position of the string.

XDrawImageString() and XDrawImageString16() act just like XDrawString(), except that the bounding box of the string is filled with “background” that is defined in the graphic context.

XDrawText() and XDrawText16() can draw one or more string to the screen using the Xlib structure XTextItem. Each structure contains the string to be drawn, the font it uses and the location and offset of the text.
The function StrToText() will convert a character string into the XtextItem.

**Experiment:**

For each of the function described above, we added a fprintf() statement at the beginning of the function call, so every time the function call is executed, it will first prints a line into the file located in “/tmp/x11_out”, along with the text content that is passed into the function call, for XDrawString and XDrawImageString, this text will just be the character string that is passed to the function call, for XDrawText, we need to extract the character string from the XTetxItem structure.

After these changes, we compiled and install the source code with the standard procedures:

- `sudo ./configure`
- `sudo make`
- `sudo make install`

the above commands generated the .so and .a library files in the directory “/usr/local/lib”. And then we override the original .so and .a file of the Xlib libraries located in “/usr/lib/i382*” directory with the new Xlib libraries. After the installation we ran multiple applications such as xterm, firefox, emacs, gnome-terminal and nautilus. However, we only saw CreateWindow() and ChangeProperties() function call. None of the text rendering function are called, which suggests that text rendering can be done by upper level libraries without using the Xlib function calls.

To prove that our code can in fact capture text from the Xlib text rendering function call, we followed the example from [13] to build a simple X11 app that will create a window and display text on the screen. This app use the function XDrawImageString() to display text, after we install and run the app, the XLib can capture the text that are displayed by this simple X11 app.
Experiment Summary:

This experiment shows that many of the modern applications (at least all of the applications we tested on our Ubuntu VM) do not use the X11 draw text function calls to display text. However, all the windows that are displayed on the screen are still created by X11, and their contents are changed by X11 function call XChangeProperties. The actual contents of the window are provided by higher level libraries.

Based on this experiment, we also now know that there is not a centralized library that puts all the text onto the screen. The text is displayed on the screen is dependent on what library the application is using. For example, the simple X11 app can put the text “Hi” onto the screen by calling the Xlib function call, whereas other applications, such as Emacs and gnome-terminal, use other higher level libraries to achieve the same effect.

Given these results, we now going to explore the FreeType2 library, which is one level higher than the Xlib in the X11 architecture.

3.5 FreeType2
FreeType is a development library used to render text onto bitmap and provide support for other text related operation. Before text can be displayed with the FreeType library, they will be converted into character index by the function call.

FT_Get_Char_Index() in /src/base/ftobjs.c

To see what we can capture with the FreeType library, we added a print statement in the FT_Get_Char_Index() function call so that it will log every character that are converted into character index.

The updated library can capture small portion of text that is displayed on the screen, and most of the captured text is system information and random repeating strings; the amount of useful text that is actually being captured is minimal. Later we found that the text that is captured by this library can be captured by Pango as well, and the result is much cleaner and easier to process. Therefore we conclude that further work on FreeType will not yield a better result than working with Pango.

3.6 Pango

Pango is another graphical library that is solely dedicated to text processing and rendering. Pango can be used anywhere that text layout is needed, though most of the work on Pango so far has been done in the context of the GTK+ widget toolkit. Pango forms the core of text and font handling for GTK+-2.x [14].

Pango itself has built-in function that can be used to display text on the screen, or it can use other front end like Cairo, FreeType or Xft.

The Pango text rendering function is pango_layout_set_text() located in Pango_layout.c.

Experiment

We added printf statements in the pango_layout_set_text() function call and let it log the text it displays into a text file in “/tmp/pango_out”. After installation, the library captures part of the text that is displayed on the screen. Captured text is mainly from:
- Text that is displayed with gedit text editor
- File names (From the file browser Nautilus)
- Title bars
- Menu items
- System popup windows
- Gnome widgets
- File Extractor
- System information (login username, current directory)
- Random strings (random window dimension, blank space, “The quick brown fox jumps over the lazy dog”)

**Pango text rendering with other front ends**

Pango can use Xft, Cairo and FreeType as its front-end and use them to render text. Pango communicates with these front ends with glyphs rather than actual strings of characters. The problem is that once text is converted into glyphs, it is difficult to convert back into characters because glyphs are fonts dependent. So the only way to efficiently capture these strings is to capture it right before they are turned into glyphs.

In Pango, the function calls that turn the character string into glyphs are:

```c
pango_shape() in shape.c
pango_itemize() in pango-context.c -> this call pang_itemize_with_base_dir()
```

**Experiments**

We added print statements into the characters conversion functions described above so that they will log the text they converted into a text file in “/tmp/pango_out”. The result we obtained are similar to the result from just the pango_layout_set_text() function (without using any front end). However, this capturing technique can capture certain user action, for example it will logs “file.txt is selected” when the user selects the file “file.txt”. The disadvantage is that this will capture large amount of irrelevant system information over and over again, which severely impact the system performance. For example, the library logs 25,000 lines when it has been running for only less than 5 minutes. When browsing the
log file briefly, we notice that most of extractions are not useful such as the login user name, and “ubuntu ****”, and some random window dimensions and blank lines.

Even though Pango can capture some very useful information, we want our monitoring system to have larger coverage. Therefore we are going to investigate GTK+, the highest level rendering library.

3.7 GTK+

The GIMP Toolkit, or GTK+, is the main graphical library for many different distributions of Linux. While there are many libraries involved in creating the graphical user interface for applications in Linux, GTK+ is the most commonly used one. It provides objects that can represent different elements in an application, such as buttons and text boxes. For example, if an application contains typical buttons, labels, and text boxes, it can be created solely using GTK+. While somewhat simplistic by design, GTK+ applications allow a certain amount of customization, including changing text and background colors, placement of individual elements, and handling different actions in the application. Even if the application contains objects that cannot be created by GTK+, such as an unconventional font or a picture, GTK+ is still used to place the elements, even though different libraries such as Pango or Cairo are used to design the element [34]. For example, if a developer creates an application that uses a label with a vertical orientation and a colored gradient for the text, it would be impossible to create using solely GTK+, but he could pass the GTK+ label object to a Cairo function where he could apply the properties that he needed. Finally, the text can be displayed in the application in the designated layout. GTK+ would then handle the actions associated with interacting with the label, such as highlighting or changing the text. For our application, it was imperative that we capture as much of the text on a user’s screen as possible, and much of the text that is handled in the entire Linux operating system deals with GTK+. Because of the portability of GTK+ and the number of popular distributions that use GTK+, we decided to focus on text extraction in GTK+ instead of trying to navigate the less documented and less used individual libraries. By using GTK+, we are able to extract the necessary information from running applications to be able to effectively detect insider threats.
Despite the complexity of the applications that can be created using GTK+, there are only a limited number of objects that can be used in the graphical user interfaces in the application. And surprisingly enough, there are even fewer objects within GTK+ that deal with textual processing. By analyzing the code for these objects, we were able to find a select number of functions that mainly deal with the creation and updating of the text in an application.

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gtk_label_set_text</td>
</tr>
<tr>
<td>Gtk_clipboard_set_text</td>
</tr>
<tr>
<td>Gtk_entry_set_text</td>
</tr>
<tr>
<td>Gtk_text_buffer_set_text</td>
</tr>
<tr>
<td>Pango_layout_set_text</td>
</tr>
<tr>
<td>Gtk_entry_buffer_set_text</td>
</tr>
<tr>
<td>Gtk_progress_bar_set_text</td>
</tr>
<tr>
<td>Gtk_selection_data_set_text</td>
</tr>
<tr>
<td>Gtk_tooltip_set_text</td>
</tr>
<tr>
<td>Gtk_label_set_text</td>
</tr>
<tr>
<td>Gtk_entry_completion_set_text_column</td>
</tr>
</tbody>
</table>

Figure 7 Useful GTK+ Functions

We were able to log the text that is created and modified in an application, added code to the functions in that collect information about the window that the text belongs to, as well as logging the timestamp and the actual text that was being displayed. The general strategy was to examine the functions that either initialize text or set the text after the object was created, and we did this by checking the length of the text as it was created, and if there was actual text contained in the pointer, we would write it to a file and include the process id and the time that it occurred. The functions in GTK+ that deal with textual handling is mainly contained in text views, text buffers, labels, and entries, but there are also some less common uses in menu bars and dialog boxes. These functions allowed us to collect as much text as possible from any given application that utilizes GTK+.

Because of the volume and the repetition by which text get created and updated in a GTK+ application, we developed a system of creating many different files based on the time when they were created, and logged the text to that specific file. In attempt to prevent the rapid
expansion of the data storage, we developed this system so that we could parse the data as quickly as possible, and delete the files of the text that has already been parsed. In this way, we could handle the amount of data coming in, because our detection program could then take the input of each file and read it into the parser. By using a clever deduplication technique, we can keep track of only the text that we want to analyze, and not hundreds of copies of the same string. The way we did this is by updating an unordered map with all of the current data being parsed. When we add data to the map, we check to see if the text is already contained in the map. If it is, we then look at the timestamp. If the timestamp of the found element is within 3 seconds of the timestamp of the text that we are looking to insert, we simply update the timestamp of the older object and keep the text in the hash map. If, however, the timestamp of the text already entered into the map is older than 3 seconds, we discard the text, because it is no longer visible on the screen. By updating the timestamp instead of inserting duplicate data, we were able to minimize the amount of storage needed for the text, as well as efficiently deal with the flood of incoming data from our code in GTK+. This technique proved to be very effective and efficient at trimming the amount of data to be processed, and the overhead associated with the data analysis was greatly reduced.

4 Capturing User’s Actions

The system will need to capture user actions such as keyboard input, mouse position and mouse click. The best location to do this is at the X Server since these user actions will all go through the X Server before it reaches any client applications. However, since we don’t want any of the monitoring tools be hosted on the user’s machine, we decided to use XNEE to capture all the event packets between the server-client channels.

To capture all the keyboard input, use the xnee command:

```
cnee --record --keyboard -o /tmp/xnee_out -e /tmp/xnee_error_out
```

Below are the output file after a user presses the keys ‘a’, ‘s’, ‘d’, ‘f’ when xnee is running in the background:

```
7,2,0,0,0,38,0,963102583,9,AT Translated Set 2 keyboard
6,2,0,0,0,38,0,963102584,3,Virtual core keyboard
7,3,0,0,0,38,0,963102695,9,AT Translated Set 2 keyboard
6,3,0,0,0,38,0,963102695,3,Virtual core keyboard
7,2,0,0,0,39,0,963103224,9,AT Translated Set 2 keyboard
```
As we can see from the highlighted part of the output, each key on the keyboard is assigned with a unique code. Also the eighth field is the system time when the key is pressed. We can simply write a parser to turn the key code into actual character.

However, for every key press and release, we got a total of 4 entries in the log file, and the output for key press and key release is identical. Because XNEE is open source software, we can modify the source file so that the output will be more suitable for our needs.

These output entries is generated by the function call `xnee_handle_xinput_event()` in `xnee_input.c`. We change the code such that for each key press, the output entry will only contain the server time and the key code, which looks like this:

```
7ServerTime:1012567610, 24
6ServerTime:1012567610, 24
```

Using a simple Shell script and a Python script, we can easily sanitize the output file and parse the key code into actual character.

5 System call monitor with Auditd [15]

The Auditd daemon can be used to monitor the activities related to a subset of user (admin) defined files, or to monitor a subset of user (admin) defined system call. All of the activities that meet the monitoring criteria will be logged in `/var/log/audit/audit.log` which can only be access by root. The Auditd system architecture is described in the below diagram:
Auditd can be configured with the command line utility `auditctl`, or by changing the file in `/etc/audit/audit.rules`. Below is the basic syntax for the audit.rules:

```
-w /FileOrDirectory/Path
```

The `-w` option tells auditd to add a watch for a file or directory specified, and all system call request access permission to those files are analyzed and logged to the `audit.log` which can be review later with command `aureport` and `ausearch`.

Suppose all the sensitive organization assets are stored in the directory `/Secrets`, we can configure Auditd so that it will record every action that is performed in that directory. If a user enters the directory and copies a file from that directory, the Auditd daemon will record these actions, our system can based on these actions to determine whether these action is a threat or not, and notifies the auditor.

Suppose we want to monitor system calls rather than some specific file, Auditd is able to do that. The configure file syntax for monitoring system call is:

```
-a exit,always -S syscall
```

This syntax adds a watch to the system call specified by the `syscall` variable. Every time the `syscall` is invoked, auditd will log the detail of the incident into the `audit.log` file.

However, monitoring system calls tend to generate very large log file because of the chain of events. Suppose we want to monitor the system call “open”. When I use Emacs to open a regular text file, Auditd will put a few dozen entries into the log file. Because when I open the text file, I will first open Emacs, and then Emacs will open its libraries and dependencies,
and at the end I will finally open the text file. Based on this reason, we think that configuring Auditd to monitor specific directories will be more efficient; however, we will need to organized sensitive files into fix directories.

To demonstrate this functionality, we configure Auditd to monitor the directory "/tmp/testDir/". Below is the output when I remove a file "hi.txt" inside the directory "/tmp/testDir/":

```
type=SYSCALL msg=audit(1384238830.726:10178): arch=40000003 syscall=301 success=yes exit=0 a0=ffffff9c a1=97b28d8 a2=0 a3=97b3a90 items=2 ppid=10671 pid=16635 auid=4294967295 uid=1000 gid=1000 euid=1000 suid=1000 fsuid=1000 egid=1000 sgid=1000 fsgid=1000 tty=pts0 ses=4294967295 comm="rm" exe="/bin/rm" key=(null)
type=CWD msg=audit(1384238830.726:10178): cwd="/tmp/testDir"
type=PATH msg=audit(1384238830.726:10178): item=0 name="/tmp/testDir" inode=1063063 dev=08:01 mode=040775 ouid=1000 ogid=1000 rdev=00:00
type=PATH msg=audit(1384238830.726:10178): item=1 name="hi.txt" inode=1088026 dev=08:01 mode=0100664 ouid=1000 ogid=1000 rdev=00:00
```

The important information is labeled orange, as we can see Auditd captured detail information about the target and the actual action performed.

An advantage of Auditd is that it came with 2 log parsers; one is called aureport, which is used to report general information about the entire session. The other is call ausearch, which is used to query information about a specific action, file activity or a system call activity. All of these query tools will require root privileges.

## 6 Machine Learning for Insider Threat Detection

The system will need to use a machine learning approach to handle the large amount of data captured. It needs to determine sensitive frames automatically and notify the auditor. The machine learning approach for text classification includes three steps, which are:

1. Text preprocessing
2. Training, building the model
3. Automatic classification

The text preprocessing step is similar for many of the machine learning algorithms, the second and third step will vary depending on what machine learning approach is applied.
6.1 Text preprocessing [16]

Captured text will need to be preprocessed to reduce its complexity and to match the format for the machine learning algorithm. The preprocessing includes multiple steps.

First is tokenization, we need to break down the large document into individual words, and process this list of words individually. The second step is lowercase conversion; a lowercase conversion can reduce the number of distinct words in the document. The third step is stop word removal, which is to remove words that are usually not useful for text classification in order to reduce the complexity of the data set. Examples of these words are: “a”, “the”, “he”, “she” and “are”. The last step is stemming, which is to normalize words derived from the same root. For example, the word “teaching” and “teacher” can be normalized into the word “teach”.

Having done the text preprocessing, we now have a less complex data set for the algorithm and can apply the different machine learning algorithms to this new data set.

6.2 Latent Dirichlet Allocation (LDA)

LDA is a way of automatically discovers topics that a set of documents contain. A document is represented as a mixture of topic that contains words with certain probabilities.

6.2.1 Basics of Latent Dirichlet Allocation

Using LDA to learn the topic representation of a set of documents and the words associated with each topic, we can use the technique of collapsed Gibbs Sampling, noted that the below method is from Chen [17]:

1. Go through each document, and randomly assign each word in the document to one of the K topics.
2. This random assignment gives the initial topic representations of all the documents and word distributions of all the topics; we will need to improve this representation since they are randomly assigned for now.
3. To improve them, for each document d,
   a. go through each word w in d
      i. And for each topic t, compute two things: 1) \( p(\text{topic } t / \text{document } d) = \) the proportion of words in document d that are currently assigned to
topic \( t \), and 2) \( p(\text{word } w / \text{topic } t) \) = the proportion of assignments to topic \( t \) over all documents that come from this word \( w \). Reassign \( w \) a new topic, where we choose topic \( t \) with probability \( p(\text{topic } t / \text{document } d) \times p(\text{word } w / \text{topic } t) \) (according to the generative model, this is essentially the probability that topic \( t \) generated word \( w \), so it makes sense to resample the current word’s topic with this probability).

ii. In other words, in this step, we are assuming that all topic assignments except for the current word in question are correct, and then updating the assignment of the current word using our model of how documents are generated.

4. After repeating these steps a large amount of time, it will eventually reach a steady state. We now have a pretty good LDA model for the relationship between the documents and the topics.

6.2.2 LDA Application

To use LDA in our system, we could break down the collected data by time interval (e.g. 30s or 1min interval). Text that is collected during these intervals can be considered as a document. A basic LDA model will be built prior to the actual monitoring by sample data, after the LDA model picked the topics, we can then label each of these topics as “sensitive” or “non-sensitive”. During the monitoring, the LDA algorithm will determine how much the “document” is related to the topics that we labeled as “sensitive”, if the percentage exceeds the threshold, the system will alarm the auditor by highlighting the time interval of that “document” on the auditor’s UI.

There is a C command line tool for LDA [18] developed by Princeton University and a supervised-LDA implementation developed by Carnegie Mellon University [19].

6.3 Support Vector Machines (SVMs)

Support Vector Machines are a generally applicable tool for machine learning; it has been proven to be one of the most powerful learning algorithms for text categorization [20].
To use SVMs, the documents need to be in attribute-value representation; this means that they need to be transformed into vector space. According to Pilászy [21], for the preprocessed document, each distinct word will correspond to one dimension (identical words have same dimension). Let the word $w_i$ corresponds to the $i^{th}$ dimension of the vector space. The most commonly used method is the so-called TF-IDF term-weighting method [22]:

$$TFIDF(i,j) = TF(i,j) \times IDF(i)$$

$$IDF(i) = \log \left( \frac{N}{DF(i)} \right)$$

TF stands for term frequency, $TF(i,j)$ represents the percentage of the $i^{th}$ word occur in the $j^{th}$ document. The variable $N$ is the number of documents. DF stands for Document Frequency, and $DF(i)$ counts the documents containing the $i^{th}$ word at least once. IDF stands for Inverse Document Frequency; the log computation is to dampen the effect of IDF since we do not want the IDF value dominates the TF value.

Suppose we have a document “doc1” contains 1000 words and the word “secret” appears 3 times in “doc1”, and we have 10,000 documents in total, and the word “secret” appears in 100 of them.

Then the term frequency:  
$$TF("secret","doc1") = \frac{3}{1000} = 0.003$$

The Inverse Document Frequency:  
$$IDF("secret") = \log(\frac{10000}{100}) = 2$$

Therefore the:  
$$TFIDF("secret","doc1") = 0.003 \times 2 = 0.006$$

If the word “secret” appears very frequently in the document, then the TF value will be large, this will lead to a large TF-IDF value. On the other hand, if the word is very common across all the documents, then the DF value will be large, in the TF-IDF value we use the inverse of the DF value multiply the TF value; therefore a large DF value will decrease the TF-IDF value. If the TF-IDF value of a particular word in a sample document is very large, then we will know that this document is very unusual, if the word happen to be a sensitive keyword, then we will know than this document contain sensitive information.
According to Pilaszy, “the transformed documents together form the term-document matrix. It is desirable that documents of different length have the same length in the vector space’ because we will want to compare them against each others, to achieved this we will use the method called “document normalization” [21]:

\[ TFIDF'(i,j) = \frac{TFIDF(i,j)}{\sqrt{\sum_i TFIDF(i,j)^\beta}} \]

The denominator is the scalar form of the entire vector space, The \( \beta \) value will usually be 2.

6.3.2 SVMs kernel function

SVM uses kernel functions (or kernel methods, similarity functions) to compare the documents to the training sets. Both the training set and the documents must have the same format, which means that if the training set is set to strings, the documents will be seen as text strings. If the training set is just array of bytes, then the documents will be composed of an array of bytes.

Depending on how the data can be separated, different kernel function should be used. The simplest kernel function is the linear function; it is used to separate data that is linearly separable. Below is an example of SVM with 2 dimensions (features) and a linear kernel function.

Suppose we are given with training examples \( x_i \), and the target values \( y_i = \{-1,1\} \), with 1 representing a sensitive frame and -1 representing a normal frame. SVM searches for a separating line (a hyperplane if the SVM contain more than 2 dimensions), which separates positive and negative examples from each other with maximal margin, similar to the figure below [21]:
The equation of the separating line is:

\[ w^T x + b = 0 \]

The classification of an unseen test example \( x \) is based on the sign of \( w^T x_i + b \). The separator property can be formalized as [21]:

\[
W^T x_i + b \geq 1 \quad \text{iff} \quad y_i = 1 \\
W^T x_i + b \leq 1 \quad \text{iff} \quad y_i = -1
\]

This equation is the linear kernel function, the term \( W \) transpose of \( x_i \) in a simpler term, is just the sum of each component in the \( W \) vector, multiply by \( x_i \) which looks like:

\[ W^T x_i = W_1 x_i + W_2 x_i \ldots \ldots \ldots W_n x_i \]

The above equation determines where the new data will be located in the SVM graph, we could say that if \( W^T x_i + b \geq 1 \), then the document \( i \) belongs to the positive class \( (y = 1, \) which will defined as sensitive class\), otherwise it belongs to the negative class.

For this linearly separable data, the linear kernel function for the above example has the form:

\[ K(W, x) = W^T x_i + b \]
The vector $\mathbf{W}$ is the document broken into multiple words, $x_i$ is the training data, and the constant $b$ determines the behavior of the margin. If $b$ is large, then the margin is called a “soft margin”, if $b$ is small, the margin is called a “hard margin”. Soft margins allow data to be slightly across the margin, as show in the below figure.

![Figure 5 Soft margin](image)

To better explain SVM, we will also introduce the case when the data is not linearly separable. Consider the below data distribution.

![Figure 6 Non-linearly separable data](image)

For now, this is still a 2 dimensional problem. As we can see, we cannot draw any lines in this plane to clearly separate the 2 sets of data. This is call non-linearly separable. To handle this data distribution, SVM requires a radial kernel function rather than a linear kernel function. The Gaussian kernel function is the most popular radial kernel function in use for SVM.
The Gaussian kernel function introduces a new dimension to reduce the non-linear problem to a linear problem. Suppose the original axis is x and y, we introduce a new dimension z, which is the distance between the data point and the origin:

\[ z = \sqrt{x^2 + y^2} \]

Based on this new dimension, we can re-plot the graph with only the z-axis. The new graph will became linearly separable, as shown in the below graph. The blue stars are close to the origin of the x-y plane, so they are close to zero in the z-axis. On the other hand, the red lines are farther away from the origin of the x-y plane. Therefore they are farther away from the z-axis origin.

![Graph showing linear separability](image)

**Figure 7: Transform the non-linearly separable data to linear**

This is basically what the Gaussian kernel does; the Gaussian kernel function has the form

\[ K(x, y) = \exp \left(-\frac{\|x - y\|^2}{2\sigma}\right) \]

The \(\sigma\) value represents the standard deviation in the Gaussian probability density function; it determines the width of the Gaussian kernel [23].

For our purpose, we need to categorize documents (the text data we obtain from our auditing system) into 2 categories: sensitive and non-sensitive. Based on this requirement, a linear kernel function will be sufficient; however, there are kernel functions specifically used to compare strings in the documents. These are called the String kernels.

The String kernels [24] are used to compare the Substrings contained in a string rather than compare strings word by word. Depended on how the substring is matched, the string kernels used a “decay” factor \(\lambda\) which is between 0 and 1 to represent how similar are the substrings.

For example, if we compare the word “car” and “can” using the String kernel and only comparing 2 characters substrings, we will have the map as follow:
And then we multiply the columns and sum them to get the Kernel function value, for this example, \( K("car", "can") = \lambda^4 \) since they only have the "ca" substring matched. This is the non-normalized value. To normalize it, we will need to compute the K value for the longest string, in this case either \( K("cat","cat") \) or \( K("can","can") \). They both equal to \( 2\lambda^4 + \lambda^6 \), and then we will have the normalized value

\[
K("car","can") = \frac{\lambda^4}{2\lambda^4 + \lambda^6} = \frac{1}{2 + \lambda^2}
\]

This value will show how similar the two strings are.

### 6.3.3 Training the SVMs Model

In order for the SVMs algorithm to work at all, we will need to train the SVM model with a set of training data. There are tool for a simple SMVs implementation, but we will need to generate the training data manually and pass it to the SVMs tool.

The larger the training data is, the better the SVMs model will perform. To generate this large training data, we will first generate a large set of sample screen and the associated text output. An auditor will examine the content on the screens and determine the sensitivity level of each of them. A program will process the associated text output and generate the training set based on the score assigned by the auditor. The auditors will only need to work with the actual screen shot; they do not need to interpret the actual captured data directly.

### 6.3.4 SVMs Tools/Implementation

There is an existing C implementation of SVMs called “SVM Light” developed by Thorsten Joachims in Cornell University [25]. Our system can use this implementation to analyze the captured text and determine whether the captured text is related to a potential threat or not.
“SVM Light” is a command line tool for SVM classifications, it has simple build-in kernel functions like linear kernel and Gaussian kernel, it also support customized kernel, which we will need to provide to the “SVM Light” tool.

The training data for SVM Light will need to be in a single file, with each line for each document. Each line required a specific format, which looks like:

```
<target> <feature>:<value> <feature>:<value> ...... <feature>:<value>
```

Target: The target has only 3 different values: -1, 0 or 1. Target value of 1 means that this document belong to the positive sample, which we will used for sensitive information, the value of -1 represent normal document.

Feature: The feature can be seen as the dimension of the SVM graph, in a text classification case, each word can be seen as a dimension.

Value: The value of the feature, can be assigned in multiple ways. One example is use the TF-IDF value.

Once we have this training set, we can use the command

```
> svm-learn  train_file  model_file
```

to build the SVM model, the model will be return as a file at the model_file location, the train_file is where we store the file contains the training sets.

Once we have the SVM model, we can use it to process new data by using the command:

```
> svm_classify  new_data_file  model_file  predictions_file
```

This command will make prediction about the new_data_file based on the model_file, and put the results into the predictions_file.

---

7 Performance Analysis
We decided to use SVM’s learning algorithm to do the threat detection because it was proven to be one of the most powerful learning algorithms for text categorization [20]. However, SVMs requires large training set in order to perform properly. Therefore, the first step for the performance analysis is to generate a training set based on sample data.

But first, we define what qualifies as sensitive data. Suppose we have a satellite company that has developed a special kind of satellite that can harvest solar power on a geostationary orbit and transmit the generated power to the earth station via microwave. This is a top secret project and regular employees should not access this data. The auditor was provided with 10 frames of sample screen shots to train the SVM model. The text that is displaying on those screens is logged in separate files. The auditor will decide if the frames contain information about the solar power satellite, and then he will mark the frames as “sensitive” or “normal.”

Within those 10 frames, 5 of those are normal frames and 5 of those are sensitive frames that contain information about the solar power satellite. Base on this, we will take the text contained in the sensitive frames and mark them as positive samples; the rest of the text is marked as a negative sample.

To process the text, we first used the Python module “NLTK” to remove the stop words and tokenized the words in the documents. “NLTK” stands for natural language toolkit. It is a Python program used for natural language processing. The “NLTK” module itself contains large amount of corpus data. The data we want from the NLTK is the list of stop words. Once we have the stop words list, we can pass sentences into the NLTK API and it will return tokenized values with the stop words removed. Once we have the tokenized documents, we will use the “Apache Lucene” to calculate the TF-IDF value for each token. Combining the class value (+1 or -1), the tokenized word and the TF-IDF value, we can generate a training set for the “SVM light” model.

After the auditor had trained the SVMs model, the auditor uses this model to analysis the activities of employee A. The job of employee A is to maintain the company’s repositories; therefore the employee has full access to any of those repositories. Suppose the text that appeared on employee A’s screen is logged into 5 different log files, within these 5 files, 2 of them actually contain information about the solar power satellite. We now want to use the
SVM model to classify these files and see if the SVMs model can detect the 2 files that contain the sensitive information.

### 7.1 Results

To analyze the performance of the SVMs Model, we will consider its efficiency, accuracy, precision and recall.

Efficiency – How long does the SVMs model takes to learn the training set, and how long it takes to classify the new documents.

Accuracy – Number of correct predictions divided by the total number of new documents to be classified.

Precision – Number of positive predictions that are correct, divided by number of positive predictions.

Recall – Number of positive prediction that is correct divided by the true number of positive case.

Below is the result we obtained from this experiment:

<table>
<thead>
<tr>
<th>Document</th>
<th>1 (sensitive)</th>
<th>2 (sensitive)</th>
<th>3 (normal)</th>
<th>4 (normal)</th>
<th>5 (normal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>Negative</td>
<td>Negative</td>
<td>Negative</td>
<td>Negative</td>
<td>Negative</td>
</tr>
<tr>
<td>Score</td>
<td>-0.1704</td>
<td>-0.1348</td>
<td>-0.3024</td>
<td>-0.2039</td>
<td>-0.3920</td>
</tr>
</tbody>
</table>

Efficiency – The SVMLight program required less than 0.01 second to process 10 set of training data and generate the SVMs model using a default linear kernel function. It also used less than 0.01 second to make a prediction on the 5 new data sets.

Accuracy – Unfortunately, the SVM model cannot detect any sensitive frames from these 5 frames. The SVMLight tools provided the confidence level for each of the prediction it made, these number will range from -4 to 4 (but they will usually be in between -3 and 3 unless the new data is almost identical to one of the training set), the higher the absolute value of this number is, the more confident the SVMs model is. After we analyze the confidence level for each of the cases, we noticed that they are all negative number and their absolute values are very small (negative value means the SVMs model labeled them as negative class, and
the absolute value of these number represent how confident the SVM model is on these predictions). This means that the SVM s model is not confident about the prediction that it made. However, when we compare the score of each of these documents, we notice that the 2 sensitive documents have a score that is closer to positive number than the other 3 normal documents.

Precision – The precision is 0% since we do not have any positive classes identified.

Recall – The recall is 0% since we do not have any positive classes identified.

The result from this experiment is not very precise because our training set is very small; given a larger training set we can expect that the prediction will be more accurate. To demonstrate that, we obtain a large training set for the SVMLight from the author's website [25]. These training sets are generated based on Reuters articles [26]. The goal for this training set is to train the SVMs model so that it can classify documents that are related to "corporate acquisitions." In the training set there are 1000 positive examples and 1000 negative examples.

For the actual data to be classified, we have 600 documents, 300 of them are related to “corporate acquisitions” and they are marked as positive class, another 300 of them are some other articles and they are marked as negative class. We will pass this test data into the SVMs model that we trained and see how accurate the predictions are.

Result:

Efficiency – Based on the output time stamp, the time required to learn 2000 samples and generate the model is about 0.08 seconds with the default linear kernel function, and the time to classified 600 test data is less than 0.01 second. This is surprisingly fast.

Accuracy – The accuracy on this test set is 97.67%. Based on the program's output, within the 600 testing data, 586 of them are correctly classified, and 14 of them are incorrectly classified.

Precision – The precision on this data set is 96.43%. Based on the programs output, there are totally 307 cases are marked positive by the SVMs model, and 297 of them are actually positive case.
Recall – The recall on this data set is 99%. We have 300 true positive cases. In the 307 positive cases that our SVMs model predicted, 297 are actually positive cases.

This test result shows that with a large enough training set, the text classification problem with SVMs machine learning algorithm can achieve very high accuracy.

8 Conclusion and Future Work

In this project, we explored a new approach to identify insider threat in a Linux environment by capturing and analyzing text that is displayed on the user screen. We capture the text by logging the program inputs that are going to the text rendering libraries such as GTK+ and Pango. And we analyze the text data with several machine learning algorithms such Support Vector machines (SVMs) and Latent Dirichlet Allocation (LDA).

By using graphical libraries, we can efficiently capture data in the user’s session without interrupting the user’s experience. Furthermore, this approach is all based on existing architectures; therefore making it very easy to apply in practice.

In our approach, we also monitor user input by logging key strokes and mouse actions using a man-in-the-middle technique on the link between the X Server and X Clients. We also monitor the system calls behavior in the kernel. Combining the text, user actions and system call behavior, we have extensive information on the system behavior.

In our auditing system, we use the SVMs machine learning algorithm to detect insider threat based on the data we obtained from the monitoring interface. By using the SVMs machine learning algorithm, we can detect insider threat more accurately and more efficiently. Based on the experiment we perform (see session 7.1), with 2000 training data sets, SVMs can positively identify a related topic with more than 97% accuracy. The time required for the SVMs model to process all these data is less than 0.1 second.

Future Work

For our monitoring interface, we still not able to get 100% text coverage for some application such as Firefox and OpenLibre Office because they do not use the shared libraries in the operating system.
For the SVMs machine learning algorithm, we are using the default linear kernel function for the text categorization. For future work, we think it would be beneficial to study the different results with different kernel functions. Also applying other machine learning algorithm such as LDA and analyze the result it produces will also be beneficial to our research.
References


[13] "http://www.cs.colorado.edu/~mcbryan/5229.03/mail/35.htm".


 allocation/,” 2011.


[34] "What is GTK+, and how can I use it?" The GTK Project. 2007.