A Mantrap-Inspired, User-Centric Data Leakage Prevention (DLP) Approach

Ryan K L Ko, Alan Y S Tan, Ting Gao
Cyber Security Lab, Department of Computer Science
Faculty of Computing and Mathematical Sciences, University of Waikato
Hamilton, New Zealand
Email: {ryan, yst1, tg44}@waikato.ac.nz

***Abstract*—The ease of sharing information through the Internet and Cloud Computing inadvertently introduces a growing problem of data leakages. At the same time, many end-users are unaware that their data was leaked or stolen since most data is leaked by operations running in the background. This paper introduces a novel user-centric, mantrap-inspired data leakage prevention (DLP) approach that can discover, present any sending of data – both authorized and unauthorized – to end-users and subsequently provide them the ability to stop the sending process. We implemented our own kernel module to work together with our user-space program in getting users approval for every sending process – giving the user full control over all outbound data sending process in their devices. With this, the end-user can always decide which data sending process should be allowed or blocked. This overcomes the limitations of current, often inflexible and inaccurate DLP solutions depending on preset rules and content detection. We showcase a proof-of-concept for our new way of detecting data leakages in an end users device. This paves the way for further research covering more complex data stealing techniques, such as the use of covert channels.**

***Index Terms*—Data leakage prevention; user-centric security; cloud computing; kernel module;**

I. Introduction

Data leakage is the act of transmitting data from one device to another without proper authorization [1], and can be unintentional or malicious [2]. According to the Privacy Rights Clearinghouse [3], 253,054,547 out of 633,358,100 data breach records of all causes from 2008 to 2013 were caused by malware. Data leakage malware generally refers to software that is designed to steal data from data storage devices. These malwares often target important data of organizations or individuals [4], [5]. Many of such malware are designed for specific data [6]. For example, the Mt. Gox virtual currency exchange's data archive was found to contain malware [5] which masqueraded as Mt. Gox's database access tool while its real purpose was to steal bitcoin configuration and data.

Current antivirus software can detect only 35% to 70% of malware [7] as they are signature-based. Likewise, data leakage prevention (DLP) software and firewalls are usually either signature or rule based. These software rely on known malware signatures to detect and block malwares. Due to the need for malware signatures, these protection software will not be able to detect and block new or previously unknown malwares. With a unique new malware produced every half a second [8], it is evident that current approaches are ineffective and there is an urgent need for novel and more effective DLP solutions.

We aim to return control of data to users [9], [10]. We developed a user-centric DLP approach with a proof-of-concept implemented in Debian Linux. Our approach adopts a novel kernel-space mantrap approach. Compared to existing data leakage detection solutions, our mantrap approach not only have the ability to detect all data leaving the device, but also empower end-users with the ability to fully decide whether the exiting data should be halted (i.e. preventing the leak) or allowed to proceed into the network (as packets). Our approach also has several application areas, including mobile and cloud computing [11], [12].

II. Current DLP Approaches

A. Pre-defined data approach

Predefined data approaches generally require users to specify to the DLP software, the data or the file type they want to protect. For example, MyDLP [13] supports predefined data types. By doing this, the DLP solution have a higher true positive rate when detecting leakages concerning data that the user deems important. Its advantage is that it responses to certain type of data leakage quicker and more accurately. When the DLP solution detects suspicious behaviors related to the predefined data type, it will raise the alarm without further analysis. The disadvantage is that the predefined data setting needs to be manually updated whenever data changes. When there is too much data or high data change frequency, the update can be troublesome and time consuming.

B. Content detection approach

In order to overcome the disadvantage of predefined data solution, content detection solution was invented. This solution has the ability to identify sensitive data within the system, by itself. This is achieved by using a number of techniques, such as searching keywords, regular expressions, partial document matching and so on. For example EndPoint Protector 4s [14] Content Aware protection uses such technique in their DLP products. The advantage of this type of solution is that it provides some degree of intelligence to the DLP product. Having the ability to discover sensitive data makes this type of solution easier to use by removing the issue of having to update DLP product setting frequently. However, this approach has a higher rate of false positives and false negatives.

978-1-4799-4093-6/14 $31.00 © 2014 IEEE
DOI 10.1109/CloudCom.2014.23

C. Device control solution

This approach focuses on preventing data leakages on portable storage devices. When such DLP solutions detect any sensitive data being transferred to any unidentified or unauthorized portable storage device, an alarm will be raised and the transfer will be stopped. For example GFI EndPointSecurity [15] can be used to prevent copying of data onto portable storage devices on the system that it is deployed on. The advantage is that it prevents data leakage from happening on portable storage devices. This will make an organization safer from inside threat or disgruntled employees. However, this approach is inconvenient and makes adding new portable devices more complicated. In order to use each new device, the administrator has to add those devices to the DLP solution and authorize them.

D. Hardware-centric DLP solutions

This DLP solution requires a specialised hardware, which can be a computer that only runs DLP or a special server for the whole organization. With the help of an extra hardware, a hardware-centric DLP solution deploy and run a collection of other DLP solutions such as those mentioned above. An example of hardware centric DLP solution is McAfee DLP [16]. An advantage of this type of solution is that it can do much more than other solutions in protecting an organizations data due to the use of different techniques in detecting data leakages. Every solution has its own strength and weakness, developers can overcome their weakness by combining them together. For example predefined data solution may require too much updates on the settings while content detection solution produces false positive and false negative. When using these solutions together, an organization can set up predefined data to help content detection solution detect real threat. However, the hardware-centric approach requires purchasing added investment, e.g. special hardware or an extra employee to manage the hardware.

III. Kernel-Space Mantrap DLP Architecture

A. Our Proposed Mantrap Approach to Data Leakage

Considering the limitations of the approaches above, we would like to propose a novel mantrap approach to data leakage prevention from the kernel space of all systems. Our kernel mantrap concept is inspired from physical mantrap access control techniques deployed in high security locations such as data centers and banks. A physical mantrap typically contains some small space with two doors, such that the first door must close before the second door opens. This allows security guards to verify the identity of the incoming personnel and if the identity of the personnel is suspect, they can 'trap' the intruder until they are able to come onsite to verify the intruder's identity again. In this paper, we will describe how we implemented a similar concept in the Linux kernel, so that all outgoing data can be kept in a mantrap and inspected by the user before it is allowed to exit into the network as packets. This required us to identify the point within the kernel where files get transformed into packets, so that we can build the mantrap at that point.

B. Linux kernel sending process in a nutshell

To understand the design, one has to first understand the inner workings of the Linux kernel sending processes. As this was surprisingly not well-documented in public literature, we conducted several trials to verify the steps, and will provide an overview of the processes involved when sending data out of a system.

1) Linux loadable kernel modules (LKMs): The Linux kernel can run loadable kernel modules (LKMs) that allow developers to expand the functionality of the kernel. The versions after 2.6 of the Linux kernel had a big change compared with previous versions. In newer kernel versions, the loadable kernel modules are handled differently from previous versions. In Linux kernels earlier then version 2.6, users had to manually link the ELF object (.o) file to the running kernel. This is in contrast with Linux kernels that are of versions 2.6 or after; the kernel does the linking. A user space program passes the contents of the ELF object file directly to the kernel. For this to work, the ELF object image must contain additional information. To identify this particular ELF object file, the file is named with suffix ".ko" ("kernel object") instead of ".o".

Hence, a program that was working on kernel version 2.6 will not work on kernel version 2.4. Therefore, we selected a stable and relatively new version of kernel, which is version 3.2.55. Some parts of our approach were implemented in the kernel space, such as pausing of the sending action and capturing the message that a sending action is trying to send. We will explain in more details the difference between a sending action and the process that does the sending in Section IV. On top of the ability to detect kernel-level covert leakages, another reason for implementing in the kernel space is that the privilege granted to user applications in user space is very limited. A program in user space cannot interrupt the sending action, even though it detects another program is sending a message to the network. On the other hand, by modifying the kernel, we are able to interrupt the sending process at the kernel level, before a file is sent out of a system.

2) Linux Kernel Sending Process Decoded: After analyzing the source code of the Linux kernel version 3.2.55, the following list of functions in Figure 1 are identified as being called by the kernel whenever the OS handles a "data send" request.

For example, when a user application wants to send a UDP package to the Internet, it will use the system call *sendto*. The *sendto* system call will in turn call *sys_sendto*, which is the system call handler. The handler will then process part of the sending request and then pass the request to *sock_sendmsg*. Each function in Figure 1 will be called in a sequential order down the hierarchy, from Layer 5 down to Layer 1. We call the whole function calling sequence as a sending action and the user-space process that invokes a sending action as a sending process. Along the way, each function will process data and passes it to the next function. The data is eventually being passed to the hardware for sending out to the network.

After identifying the functions involved, the next step is to analyze the state of the data as it passes through the list

Layers	UDP function name	TCP function name
Layer 5	sendto	send
		sendto
	sys_sendto	sys_sendto
	sock_sendmsg	sock_sendmsg
	__sock_sendmsg	__sock_sendmsg
	inet_sendmsg	inet_sendmsg
Layer 4	udp_sendmsg	tcp_sendmsg
	udp_push_pending_frames	tcp_send_skb
	ip_append_data	tcp_transmit_skb
	ip_push_pending_frames	ip_append_data
		ip_push_pending_frames
Layer 3	ip_queue_xmit	ip_queue_xmit
	ip_queue_xmit2	ip_queue_xmit2
	ip_output	ip_output
	ip_finish_output	ip_options_build
	ip_finish_output2	ip_finish_output
	dev_queue_xmit	ip_finish_output2
		dev_queue_xmit
Layer 1 2	dev->qdisc->enqueue	timer_handler
	qdisc_run	dev->qdisc->enqueue
	qdisc_restart	qdisc_run
	dev->qdisc->dequeue	qdisc_restart
	dev->hard_start_xmit	dev->qdisc->dequeue
		dev->hard_start_xmit

Fig. 1. System Call Hierarchy of Linux kernel version 3.2.55

of functions. Jprobe [17] is a tool used for debugging and analyzing the Linux kernel environment. Jprobe provides the ability to intercept function calls and direct it to our own handler functions. By replacing the original function calls with our own, we can intercept and retrieve the parameters passed to the original sending function and the message to be sent, at the kernel level. Using Jprobe, we managed to analyze the state of the data message as it passes through the list of functions shown in Figure 1. This is achieved by inserting Jprobe into every function in the list above to view the data as it passes through them.

Through the use of Jprobe, it was found that the functions before *udp_sendmsg* and *tcp_sendmsg* could be used for communication between local processes. For example, the *sock_sendmsg* can be used by some applications to ask for information from the kernel. In contrast, the *udp_sendmsg* and *tcp_sendmsg* functions are only used when the data has to be sent to the network. The functions after *udp_sendmsg* and *tcp_sendmsg* handle the job of fragmenting the data to fit into the size of one network package and adding the header to a network package. If the message has been fragmented, it is harder to collect the whole message. Although in this paper, we only analyze the message for its file type, an entire message will be required for deeper future analysis work. Hence *udp_sendmsg* and *tcp_sendmsg* are better choices as the mantrap point to stop the exiting data for inspection.

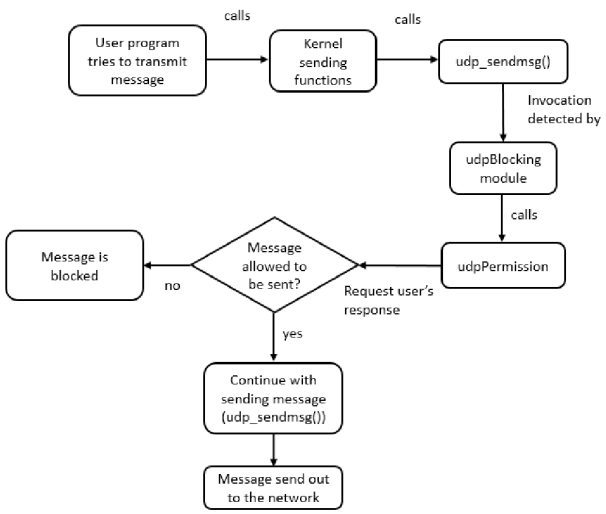

Fig. 2. User-centric mantrap DLP Components and Data Flow

C. Overall structure of our design

Figure 2 explains the overall flow of our design. There are **4 main components in our mantrap DLP design**: (1) capturing messages being sent, (2) pausing the sending action, (3) presentation of information to users and (4) handling user decisions. We will now describe the key components enabling these steps. We use 'sending an UDP package' as an example in describing our implementation. The same process will apply to TCP packages, except the prefix of the function names will be "tcp" instead of "udp".

1) Modifying udp_sendmsg kernel function: The original udp_sendmsg kernel function does not have features for (1) pausing a sending action, (2) waiting for user decision and (3) resuming a data sending action. These three functionalities cannot be achieved with standard system calls or C language function library. Hence we modifed the kernel function call to our requirements. In this case, *udp_sendmsg* was modified to as shown in Figure 3. A waiting function was added at the beginning of *udp_sendmsg*. It will wait for one second after each time it checks for the arrival of a user decision. If the duration between each check time is too short, there is a risk that the kernel will be too busy when there are too much paused sending actions. If the duration is too long, the user might feel a delay on executing their decision. The blocking and resuming of the send process is handled by the next component, our *udpBlocking* module.

2) udpBlocking module: Our design of the *udpBlocking* module is shown in Figure 4. This module handles the task of capturing the message transmitted to *udp_sendmsg*, transforming the data from kernel space to user space (i.e. *udpPermission*), maintaining the sequence of the sending actions through the use of a queue and resume or block the sending process, based on the users decision. Communication between user space and kernel space is not straightforward. The data stored in user space should not be accessed in kernel space and vice versa. Doing so can cause the kernel

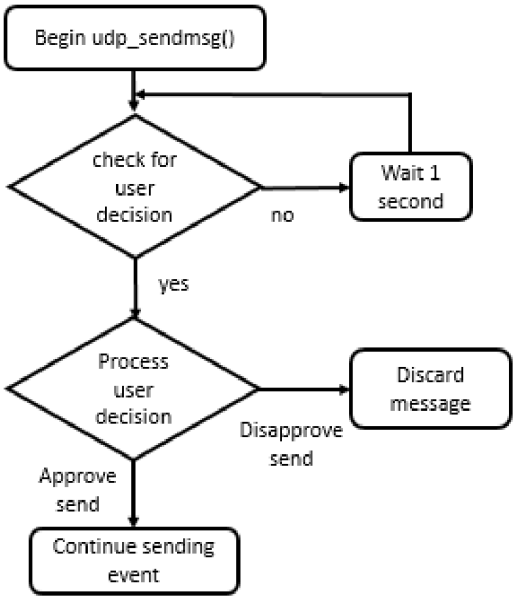

Fig. 3. Modified *udp_sendmsg*

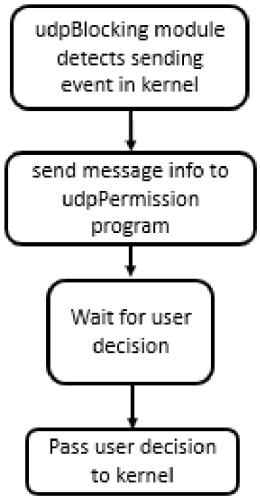

Fig. 4. The *udpBlocking* module

to crash. The only safe way to communicate between user space and kernel space is to send the data through a channel. To overcome this communication issue, Netlink [18] was used. Netlink is a special socket used for transferring information between kernel and user space processes. It provides a full-duplex communication link between the kernel and user space by using standard socket application programming interfaces (APIs) for user-space processes and a special kernel API for kernel modules. With the help of Netlink, we can achieve the communication between kernel and user space.

3) udpPermission program: Our *udpPermission* program uses two concurrent threads: a Main thread for listening to incoming data from the *udpBlocking* module and a Worker thread. The Main thread listens to kernel space and add the received message to a queue. The Worker thread waits for a job and uses the **File** [19] program to identify format of the message. Everything in a computer is stored in bits, file format specifies how these bits are encoded in the digital storage. With the help of file format, the operating system can easily find the correct programs to handle any tasks related to the file. *File* is a standard Unix program for identifying the file format of a file. However, files do not need to have an extension to be identified – this suits the captured message of our design. *File* has been installed on every known Linux distributions, executable by command line. After identifying the file format using *File*, we present the result to users for asking their permission. The last task is sending the users decision back to *udpBlocking* module.

```
udpBlocking module or tcpBlocking module
Jprobe interrupt handler function(same parameters with interrupted function)
    send captured message to udpPermission or tcpPermission program
init_module function
    my_jprobe.kp.addr==kallsyms_lookup_name("udp_sendmsg")
                        or  kallsyms_lookup_name("tcp_sendmsg")
    register_jprobe(&my_jprobe)
```

Fig. 5. Skeleton code of the udpBlocking and tcpBlocking module

IV. Implementation

A. Capturing sent messages

After the confirmation of the system call hierarchy explained in Section III-B, we need to find out what kind of data was transmitted by the function chain and inform the user of the transmission. The process of informing users of the transmission would require analyzing the captured message so as to be able to present meaningful information about the message. To achieve this, the first step is to capture the messages being sent at the kernel space. To intercept function calls in the kernel, we need to register Jprobe to interrupt kernel functions *udp_sendmsg* and *tcp_sendmsg*.

Firstly, we need to look up the function addresses of *udp_sendmsg* and *tcp_sendmsg* from a system function call table. This can be achieved by using the *kallsyms_lookup_name* function. Secondly, a Jprobe *struct* type variable is created and used for setting the function addresses. To inform Jprobe about functions which need to be interrupted, we register the Jprobe variable to the system using *register_jprobe* function. After registering the Jprobe variable, we can then create a handler function. This function will be called whenever the interrupted function is called. The handler function also needs to have the same input parameters as the function that Jprobe is interrupting. This way, the function parameters will be copied from kernel function to the Jprobe handler function. In our implementation, we created *udpBlocking* and *tcpBlocking* module with this Jprobe functionality for capturing the message being sent, at the kernel level. A skeleton code of this part is shown as Figure 5.

B. Transmitting captured messages to user space

After successfully capturing the exiting message, the next step is to present the captured message to the user. The source code of udpBlocking and tcpBlocking module can be found on [20]. The Netlink sender at user space needs to send a message to the kernel space first, so that the kernel space Netlink receiver can obtain the PID of the Netlink sender in

the user space. Using this PID the Netlink sender at kernel space also can send message to that Netlink sender program in the user space. Then, no matter when the message is put to the Netlink, it will be transmitted between the user space and kernel space like sending a package.

User space programs, *udpPermission* and *tcpPermission*, require the captured message to perform further analysis, in order to provide useful information to the user. Therefore Netlink sender needs to be added to *udpBlocking* and *tcpBlocking* modules. This Netlink sender is used to send the message that *udpBlocking* or *tcpBlocking* modules captured to *udpPermission* or *tcpPermission* programs. When the user makes a decision, the decision needs to be sent back to *udpBlocking* and *tcpBlocking* modules in kernel space. Hence a Netlink receiver should also be created in *udpBlocking* and *tcpBlocking* modules to receive users decision.

C. Pausing the kernel function

When the message is presented to the users, udp_sendmsg and tcp_sendmsg need to wait for users decision. Therefore-After obtaining the message, *udpBlocking* and *tcpBlocking* modules will transmit the message to the user space using Netlink sender. After which, the respective *udp_sendmsg* and *tcp_sendmsg* will wait for users' decision.

Therefore, pausing these two functions is necessary. We achieve this by modifying *udp_sendmsg* and *tcp_sendmsg*. A while loop is added at the beginning of *udp_sendmsg* and *tcp_sendmsg*. This while loop will keep checking a Boolean type variable which indicating whether the user has made a decision or not. If the decision is not received, then the *msleep* function is called to let *udp_sendmsg* or *tcp_sendmsg* function pause for 1 second. After 1 second, the Boolean type variable will be checked again.

The kernel built-in linked list was used to build a queue in kernel space. This queue is used to store the order of paused sending actions. Before *udp_sendmsg* and *tcp_sendmsg* enter the loop, they add themselves into the queue. This queue can be further used when handling users decisions. It keeps a record of the order of the paused sending actions. The users decisions will be executed as the order as this queue. The source code of the modified udp.c and tcp.c file can be found on [20]. When the user-space program receives the message, the captured message is analyzed and presented to the user in a useful way.

D. Presenting useful information to users

After *udpPermission* and *tcpPermission* receive the captured message from the kernel space, the user needs to know some information about the message in order for them to make an informed decision. The *File* program is used to identify the messages file format. *udpPermission* and *tcpPermission* need to have the ability to **both** receive new message from kernel space and present file format to users to make decision concurrently. A single-threaded program cannot achieve this. Therefore, these two programs have to use more than one thread. The default C language library has thread functionality, but not easy to use. *Pthread* [21] is a version of threads developed under POSIX standard. It has a lot of advantages compared with other threads, such as lightweight, efficient communications/data exchange, etc. The main reason that we use *Pthread* to achieve multi-thread functionality is that it makes multi-thread programming much easier. One thread is used to listen for new captured messages from the kernel space while the worker thread is used to check whether a captured message needs to be analyzed and presented to the user.

Both *udpPermission* and *tcpPermission* have queues storing messages that have not been presented to the user. If the worker thread finds a message in the queue, it will use the *File* program to identify the file format and print the format to the terminal. This is achieved by using *popen* function, which is design to execute an OS command inside a user-space program. In our prototype, we created an OS command that will use *File* on the captured message. Then present the file format to the user, telling them *a file with such file format is leaving the computer, do you want to allow this to happen or block this sending action? yes or no?* A user can then type 'Y' for "yes, allow the sending action" and 'N' for "no, block the sending action". The source code of *udpPermission* and *tcpPermission* program can be found on [20].

E. Handling user decisions

The user decision needs to be transmitted back to kernel-space for the *udpBlocking* and *tcpBlocking* module to take the necessary actions via a Netlink receiver. The receiver receives users decision from the Netlink sender in the user space program. We created *letItResumeUdp* and *letItResumeTcp* functions in the kernel. These two functions can notify *udp_sendmsg* and *tcp_sendmsg* that user has already made a decision. Then *udp_sendmsg* and *tcp_sendmsg* will quit waiting and users decision will be executed.

In order to use *letItResumeUdp* and *letItResumeTcp* functions in *udpBlocking* and *tcpBlocking* modules, *EXPORT_SYMBOL* function was used. This function will make *letItResumeUdp* and *letItResumeTcp* available inside kernel space and *udpBlocking* and *tcpBlocking* module are inside kernel space. Otherwise only the kernel can use these two functions. Functions *letItResumeUdp* and *letItResumeTcp* will pass the users decision as a Boolean variable, to indicate the users decision to block or to allow the message to be sent, to *udp_sendmsg* or *tcp_sendmsg*. These two functions will execute the users decision by resuming or terminating the sending process. If the sending action was approved, *udp_sendmsg* and *tcp_sendmsg* will call the original kernel sending functions and send the message out of the system. On the other hand, if the user wants to block the message, then these two functions will use return '-1' to stop the system call hierarchy. In this case, the sending action is stopped and the kernel knows the sending was not successful. Figure 6 is the overall structure of what has been modified inside kernel, using *udp_sendmsg* and *letItResumeUdp* as an example.

V. Limitations and Ongoing Work

While our prototype proves that a mantrap approach to data leakage is possible, we are currently working on addressing

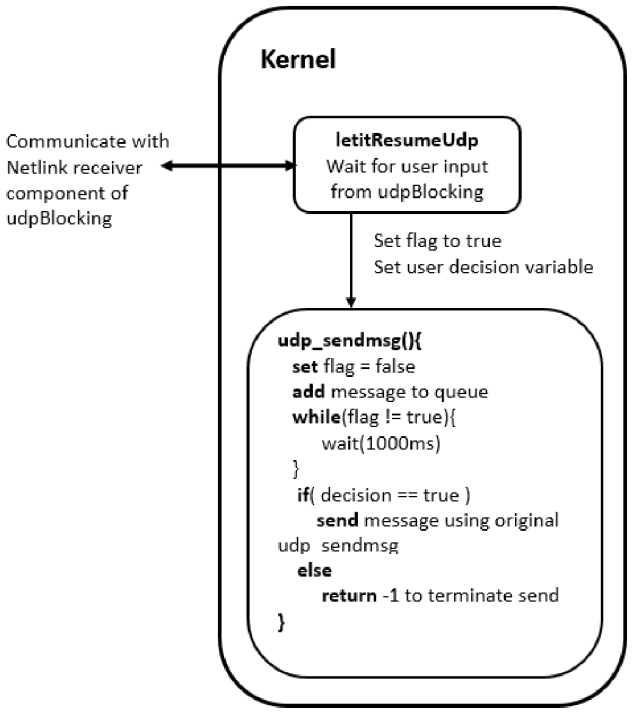

Fig. 6. Modified *udp_sendmsg* and *letItResumeUdp*

the following enhancements:

- A graphical user interface for the users.
- Richer analysis of messages presented to end-users. In the future, content analysis techniques such as keyword matching, regular expression and partial document matching can be used for analyzing the content of the message.
- A white list feature. A white list function can allow users to predefine programs that the user knows it is harmless. That way, the software skips asking for permission to send from the users for the white listed programs.
- Dealing with encrypted files being sent out. The reason is that the *File* program that our implementation used for analyzing the sending message will fail to identify the file format. This will result in the software telling the user that an unknown file format message is leaving the device. The users would not be able to make a proper decision based on such information. Content detection techniques could be applied on the analysis of the message being sent out. The users can make better decision when they are provided with more information.
- Addressing other more sophisticated data stealing techniques. One of them is covert channel [22]. It is a type of attack that transfers information objects not using usual data transfer mechanism like read and write. It is not using the legitimate data transfer mechanisms, hence it cannot be discovered by normal data leakage prevention solutions. Other data steal techniques can also avoid the detection, such as Steganography [23]. In order to overcome these data steal techniques, more researches and experiments should be done.

VI. Challenges and Lessons Learned

A. Challenge 1: "Pausing" a live process

Trying to pause a 'live' sending process in the kernel is not a trivial endeavor. We share the challenges and lessons learned in this section, with the hope that peers can benefit from them:

1) "Kill" command cannot meet requirements: One of the earliest approach was trying to stop the sending of message out from the system with the *Kill* command. However, this initial approach of using Kill to stop the sending process failed. Further investigations and debugging revealed that the root cause of the failure is that a kernel sending action cannot technically 'pause'. A process performs many actions to achieve its goal. The result after using Kill command is that the process is paused right after the sending action finished.

2) Hijacking the kernel – insecure and not future-proof: Realising this, we moved into trying to pause the sending action by hijacking the kernel. The design involved replacing the addresses of *udp_sendmsg* and *tcp_sendmsg* in the system call table to point to the new functions that were created by us. In the new functions, we can provide the pausing feature and then call the original *udp_sendmsg* and *tcp_sendmsg*. However, attempts to implement the idea into our software failed. It was discovered, after some research, that hijacking of kernel works only for Linux kernels that are before version 2.6. The developers of Linux kernel considered hijacking of kernel a security issue and as such fixed it on version 2.6. As such we had to reconsider the design. The reason is that even if we successfully achieve hijacking the kernel, our solution will not be relevant with all future Linux versions.

3) Modifying the kernel code – memory exhaustion risks: Subsequently, we came up with the idea of modifying the kernel code. We aimed to continue with the idea of replacing original *udp_sendmsg* and *tcp_sendmsg*. A problem was encountered when trying to save the information of a sending action. Unlike programming in user space, one cannot simply copy a *struct* variable type to another when programming in kernel space. We have to copy every element inside a struct variable – which is going to exhaust memory.

B. Challenge 2: Extracting useful information from binary

Extracting useful information from the captured message and presenting it to users was another challenge. In our current implementation, we focus on extracting the file name and suffix from the captured message. We believe that these two pieces of information would be able to help users easily identify the data leaving their devices. The first method we used in attempt to obtain the full file name and suffix, is to read the information from the kernel structure variables that were passed into the kernel function *udp_sendmsg* and *tcp_sendmsg*. However, at this point, the structure variable that contains the metadata, contains information such as file descriptor value of the data file; information that is used by the kernel rather than information that describes the actual data file. Information describing the actual data file is not passed into the kernel. As the message itself is in binary, reading the message content is

not straightforward as we would not be able to tell whether the message content is of ASCII or other data formats or encoding. The second method that was used was to track down the file name and file suffix from the PID of the sending process. However, we found out that the process structure does not keep a record of which file it has accessed. Also the file system does not keep a record of which file was accessed by any process. The last method tried was to pass the whole chunk of data to the user space and perform some analysis on the data. At this point, extracting the file name is very difficult. Since the only way to obtain this is by comparing the data with every file in the users device. This will cause a long delay if there are too many files on the device. Hence, searching for the file using content matching is not feasible. That said, file suffixes can be extracted easily. We use the program, *File*, to identify the file format and successfully obtained the file format. File format is the equivalent of file suffix. However, we did not further analyze the file. The reason is that we have to find out the related application to successfully open the file. Doing so will require matching each different file format to a program that is known to be able to open the respective file format. This can be an issue if those programs do not exist on the device.

VII. Concluding Remarks

We proposed and implemented a novel DLP approach inspired by the physical access control method of 2-door 'mantraps'. While our implementation is still an initial proof-of-concept stage, our approach can already detect all kinds of data leaving the system without *a priori* information of which files to protect. This contrasts existing solutions where you have to predefine rules on what information to protect.

To achieve the mantrap concept, we first modified the *udp_sendmsg* kernel function to allow pausing of any data sending action at the kernel space – allowing user time to decide whether to allow the data to leave the system. Then, we built two LKM, *udpBlocking* and *tcpBlocking*. These two modules are used for capturing the message that is being sent out, and empowers further user analysis. Once captured, the message is passed to the user space programs, *udpPermission* and *tcpPermission*, using Netlink. The file format is then extracted from the message using the File program and presented to the user. The user space programs would then proceed to ask users on whether to allow or block the sending of the data.

By notifying the users on what data is leaving their system and asking for a decision on whether to allow or block the data sending, we raised their awareness of what data is leaving their system. We also empowered users with control over what data can leave their system.

Our source code can be downloaded from GitHub at [20].

References

[1] P. Gordon, "Data Leakage - Threats and Mitigation," Retrieved: 06/09/2014 from SANS Institute InfoSec Reading Room, 2007. [Online]. Available: http://www.sans.org/reading-room/whitepapers/awareness/data-leakage-threats-mitigation-1931

[2] R. K. L. Ko, "Data Accountability in Cloud Systems," in *Security, Privacy and Trust in Cloud Systems*. Springer-Verlag Berlin Heidelberg, 2013.

[3] Privacy Rights Clearinghouse, "Chronology of Data Breaches," Retrieved: 06/09/2014 from Privacy Rights Clearinghouse, 2014. [Online]. Available: http://www.privacyrights.org/data-breach/new

[4] T. Olavsrud, "Study: Data Loss Affects Nearly One-Third of Enterprises," Retrieved: 06/09/2014 from eSecurity Planet, September 2010. [Online]. Available: http://www.esecurityplanet.com/trends/article.php/3904636/Study-Data-Loss-Affects-Nearly-OneThird-of-Enterprises.htm

[5] L. Constantin, "Bitcoin-stealing malware hidden in Mt.Gox data dump, resresearch says," Retrieved: 06/09/2014 from ComputerWorld, March 2014. [Online]. Available: http://www.computerworld.com/article/2488704/malware-vulnerabilities/bitcoin-stealing-malware-hidden-in-mt–gox-data-dump–researcher-says.html

[6] P. Ducklin, "Digitally signed data-stealing malware targets Mac uuser in "undelivered courier item" attack," Retrieved: 06/09/2014 from naked security, January 2014. [Online]. Available: http://nakedsecurity.sophos.com/2014/01/21/data-stealing-malware-targets-mac-users-in-undelivered-courier-item-attack/

[7] M. A. Rajab, L. Ballard, N. Lutz, P. Mavrommatis, and N. Provos, "CAMP: Content-Agnostic Malware Protection," in *20th Annual Network & Distributed System Security Symposium (NDSS13)*, 2013.

[8] F. Adkins, L. Jones, M. Carlisle, and J. Upchurch, "Heuristic malware detection via basic block comparison," in *8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, Oct 2013, pp. 11–18.

[9] Cyber Security Lab - University of Waikato, "CROW - Cybersecurity Researcher Of Waikato," Retrieved: 07/09/2014 from CROW, 2013. [Online]. Available: https://crow.org.nz

[10] R. K. L. Ko, M. Kirchberg, and B. S. Lee, "From System-centric to Data-centric logging - Accountability, Trust and Security in Cloud Computing," in *Defense Science Research Conference and Expo (DSR)*, 2011.

[11] R. K. L. Ko, "Cloud computing in plain english," *ACM Crossroads*, vol. 16, no. 3, pp. 5–6, 2010.

[12] R. K. L. Ko, S. S. G. Lee, and V. Rajan, "Understanding cloud failures," *IEEE Spectrum*, vol. 49, no. 12, pp. 84–84, 2012.

[13] MyDLP, "Features: MyDLP Enterprise Edition," Retrieved: 06/09/2014 from MyDlp, June 2013. [Online]. Available: http://www.mydlp.com/features/

[14] CoSoSys, "Endpoint Protector 4," Retrieved: 06/09/2014 from Endpoint Protector, 2014. [Online]. Available: http://www.endpointprotector.com/products/endpoint_protector

[15] GFI.com, "Control User Access to Endpoint Connections," Retrieved: 06/09/2014 from GFI EndPoint Security, 2012. [Online]. Available: http://landdewa.gfi.com/control-user-access-sm?adv=28104&loc=4&kwd=5&gclid=CP7PlIf35L4CFcWSvQod_0YA6A

[16] McAfee, "McAfee Total Protection for Data Loss Prevention," Retrieved: 06/09/2014 from McAfee, 2014. [Online]. Available: http://www.mcafee.com/us/products/total-protection-for-data-loss-prevention.aspx

[17] W. Cohen, "Gaining Insight into the Linux kernel with Kprobes," Retrieved: 06/09/2014 from RedHat, March 2005. [Online]. Available: http://www.redhat.com/magazine/005mar05/features/kprobes/

[18] K. K. He, "Kernel Korner - Why and How to use Netlink Socket," Retrieved: 06/09/2014 from Linux Journal, Janurary 2005. [Online]. Available: http://www.linuxjournal.com/article/7356

[19] Juergen Haas, "Linux/Unix Command: file," Retrieved: 06/09/2014 from About.com Linux, 2014. [Online]. Available: http://linux.about.com/library/cmd/blcmdl1_file.htm

[20] CROW, "CROWLaboratory - User-Centric-Mantrap git repository," Retrieved: 07/09/2014 from github, 2014. [Online]. Available: https://github.com/CROWLaboratory/user-centric-mantrap

[21] B. Barney, "POSIX Threads Programming," Retrieved: 06/09/2014 from Lawrence Livermore National Laboratory, 2014. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads

[22] E. Pennington, W.Oblitey, S. Ezekiel, and J.Wolfe, "An OverOver of Covert Channels." [Online]. Available: http://twitdoc.com/upload/onaccel/an-overview-of-covert-channels.pdf

[23] S. D. Dickman, "An OverOver of Steganography," James Madison University Department of Computer Science, Tech. Rep., 2007. [Online]. Available: http://www.infosec.jmu.edu/documents/jmu-infosec-tr-2007-002.pdf