Data Paladin – An Application Independent Rights Management System

A Thesis Presented

by

Subhadeep Sinha

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

December 2008

Stony Brook University

The Graduate School

Subhadeep Sinha

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Professor Tzi-cker Chiueh - Thesis Advisor
Department of Computer Science

Professor Scott Stoller – Chairperson of Defense
Department of Computer Science

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

Data Paladin – An Application Independent Rights Management System

by

Subhadeep Sinha

Master of Science

in

Computer Science

Stony Brook University

2008

Data Paladin (DP) is an application independent rights management system. It is a step beyond centralized client-server technologies for resource sharing, towards a more distributed environment, with a goal of making the overall solution scalable and secure. The key idea is to protect shared data stored on a central server. The way Data Paladin is different from current technologies is that it doesn't treat clients using the shared resources as dumb terminals for viewing and editing data. It rather uses their compute power to run applications locally, instead of running them all on the server.

Data Paladin uses technologies like Featherweight Virtual Machine (FVM) technology [1] to ensure that the running environment of applications on clients is a logical extension of the server's runtime. Security of the system is attributed to isolation provided by FVM for applications running locally on clients which access shared data from the server, and to techniques like on-the-fly encryption of data on the network, controlled local printing and clipboard usage, controlled network activity and more.
In this thesis we present the design and implementation details for such a system, along with ways to achieve a totally secure resource sharing environment which guards against data leakage at any level. The implementation talks about setting up such an environment on either Windows, or a mix of Windows and Linux. At the end we try to gauge the overall performance of the system.

To

My Parents, Sister & Girlfriend

Table of Contents

List of Tables

List of Figures

Acknowledgements

First of all, I wish to sincerely thank Dr. Tzi-cker Chiueh for his guidance and support all through the project. I would also like to thank all the present and past colleagues at Rether Networks Inc (RNI). In particular, Dr. Lap-chung Lam for his insightful advice and help throughout my thesis work and Sheng-I Doong, President of Rether Networks, for her kind support.

Chapter 1

1 Introduction

1.1 Data Paladin – An Overview

Data Paladin (DP) as the name suggests, is a solution for data leakage protection. The motive is to design an application independent mechanism for protection of data hosted on a central server. In such a model, once a user checks in data into a secure server, we ensure that the data doesn't leave the server in any insecure way.

Data Paladin is modeled around the client-server architecture, but attempts to solve the bottlenecks associated with traditional centralized implementations. It's rather a shift from centralized design to a more distributed environment, where the clients are much more than just dumb terminals. We use the compute power of the clients to run applications locally, but in the execution environment of the server, essentially carving a piece out of the server onto the clients.

1.2 Motivation and Requirements

One of the main motivations for developing Data Paladin was to achieve scalability over centralized client-server architectures. Display-Only File Viewer [2] (DOFS) is one example of such a centralized design. DOFS uses Windows Terminal Services [6] to stream terminal sessions from the DOFS server back to clients. All applications run on the central DOFS server. The bottleneck here is the lone server, which can only serve up to a fixed number of DOFS clients at a given time. To make the solution scalable by reducing the load on the server and using the clients' local processing power, the applications need to be run locally on the clients. Data Paladin does exactly this.

The challenge however is to make sure that the entire system as such remains secure even after the applications are decoupled from the server. As such the requirements from Data Paladin as a system are:

- Application-Independent: Data Paladin as a rights management system should not be tied to any particular application. In other words, the technology should work seamlessly for all categories of applications e.g. Documents, Streaming Media etc. The problem with technologies like Authentica and Liquid Machines is that they are closely tied to applications. The means of protection offered by these depends on the kind of application one is dealing with e.g. Microsoft Word or Adobe Reader etc. Such technologies would have to constantly keep certifying themselves with new applications as they come out.

  Data Paladin is the first of its kind Enterprise Rights Management System which is totally application independent. The kind of application that would use the data that a DP system protects is the last thing on our mind, and the design doesn't need to be concerned with that at all.

- Immune from network sniffing: Since the DP clients and server exchange data over the network, a man in the middle should not be able to capture sensitive data. This would mean that we should make sure that there is no plain text data on the wire between the clients and the server. DP achieves this through on-the-fly encryption/decryption of traffic going over the network.

- Immune from screen capture: A user viewing a document hosted on the DP server should not be able to capture the screen on the client as a means to extract data. DP blocks screen capture for applications running inside virtual machines on the client host to achieve this goal.

- Immune from system call interception: We need to protect the system from system call interceptions at the client host. More specifically since the client host is in control of the user, he could run a device driver to intercept Windows system calls and try to pull data out through that route. We need to make sure that we are able to detect and thwart such attempts. The current Data Paladin implementation doesn't do this, and its part of the roadmap.

- Integration with products in the same space: Data Paladin needs to be easy enough to integrate with other technologies in the domain of rights management e.g. Sharepoint server. It should also be able to work with Web Servers and Document Management Servers to augment security in the services they already provide. This is another thing that has not been implemented as of now and is part of our future roadmap.

In an enterprise Digital Rights Management (DRM) system, information could leak at multiple places. The next figure enumerates possible leakage points. Data Paladin attempts to block all such illegal exit points for data.

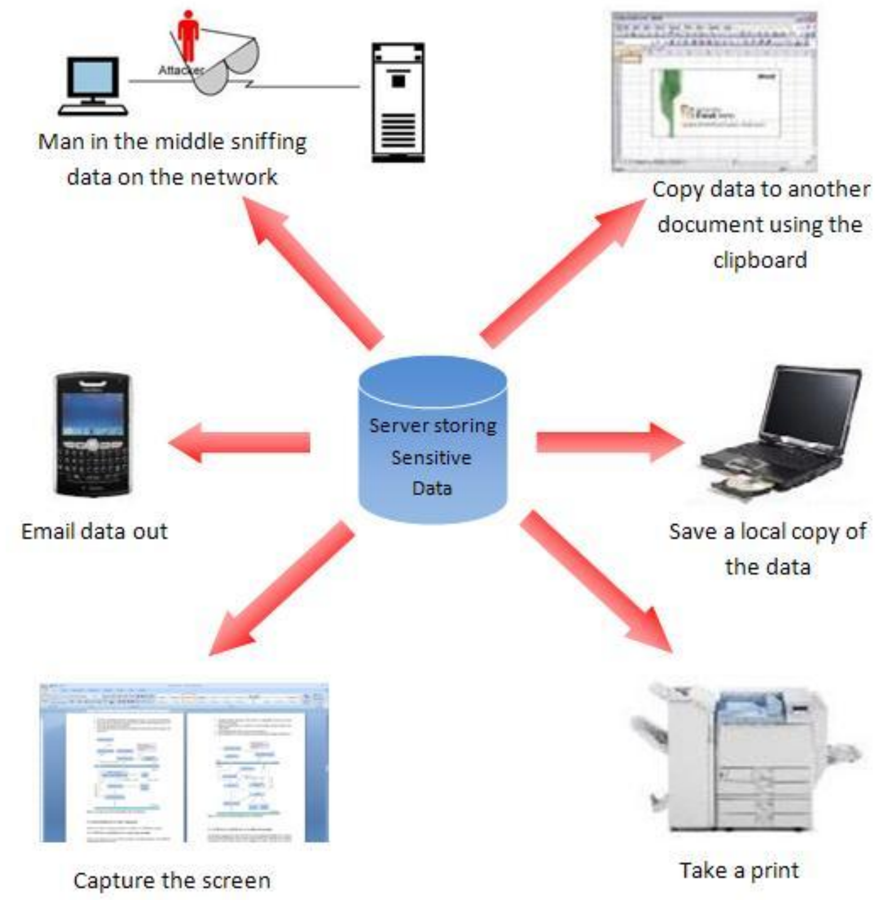
Attacker
Man in the middle sniffing data on the network
Copy data to another document using the clipboard
Server storing Sensitive Data
Email data out
Save a local copy of the data
Capture the screen
Take a print

Figure 1: Possible ways in which sensitive data could leak in an Enterprise DRM system

1.3 Data Paladin and Feather Weight Virtual Machine Technology

As mentioned earlier, running applications locally on the DP clients also means that we now have to worry about having control over how the applications run on the clients – what entities they interact with, what a user on the client host might choose to do to defeat the system etc. Most of these concerns can be addressed by running the applications in an isolated environment on the client host. This isolation is provided by the Featherweight Virtual Machine technology [1].

Featherweight virtual machine (FVM) is an OS level virtualization technique on Microsoft Windows OS. Under FVM architecture, each virtual machine is created using the same state as the host machine. The virtual machines are logically isolated using namespace virtualization, resource copy on write and IPC confinement. The virtualization layer in FVM virtualizes the namespace by renaming the system resources. This renaming takes place at the OS system call interface.

The key idea of FVM is access redirection and copy on write, which enables each VM to read the base environment from the host machine but write into the FVM's private environment (filesystem and registry hierarchy). FVM also identifies various communication interfaces and confines their scope on a per VM basis.

All the VMs share the host OS's kernel-mode component, including the hardware abstraction layer, device drivers, OS kernel, as well as system boot components. The file system image is also shared by default. Each new VM starts with exactly the same operating environment as the current host. We use a copy-on-write mechanism by means of which only when a VM writes into the registry or the filesystem, does it get a local copy of the affected object in its private environment, which it then modifies. Therefore, both the startup delay and the initial resource requirement for a VM are minimized.

The minimal resource requirements of such VMs makes it possible to dynamically create them on client hosts without adding any considerable overhead to the system. Furthermore the virtual filesystems of the VMs created on the DP client host reside on the DP server's filesystem, thus ensuring that any writes from applications running in these VMs on the clients are always re-directed to the server, and a user can't save anything locally on the client in any case.

Chapter 2

2 Related Work

There are a couple of solutions available for data protection and Enterprise Rights Management (ERM) today. In today's world sensitive data can't be confined to just one environment to ensure security. The need of the hour is to be able to share such data with multiple users at multiple places in order to achieve productivity. Enterprise Rights Management is becoming a hot area in the industrial marketplace, more so with biggies like Microsoft, Adobe, Oracle and EMC joining the race. In this section we look at various available solutions to ERM, and how Data Paladin fares to the challenge.

Every ERM solution aims to provide security to shared sensitive data which is used by a multitude of users. One of the biggest challenges facing ERM systems is protection against insiders who have authorized access to data. The way Data Paladin attempts to address this is by forbidding protected data to reside outside protected servers at any time, so that the data never leaves the servers. Authorized users use the data as if it was stored locally.

Display-Only File Server [2] (DOFS) is a centralized system for storing sensitive data on a server. Clients can connect to the server to view/edit information in files stored on it. At any given time all applications servicing clients run on the central server, and the clients are served through Windows Terminal Service sessions. So when a new client connects to the DOFS server, a terminal server session is automatically launched and the client just gets the display exported to it. This approach is inherently secure since all the client can do is to view and edit files stored on the server. The DOFS client software also implements screen capture, print blocking etc. so that the end user has no means of extracting the data out of the system. Data Paladin is a step ahead since its distributed in nature. We don't overload a server by running applications on it. Rather the DP clients run all applications locally. So in addition to providing all the features that a DOFS system does, Data Paladin adds scalability to the system.

Microsoft Windows Rights Management System (RMS) [11], like Liquid Machine's ERM Suite associates usage policies at the file level. It's a solution for protecting E-mails and Documents in an Enterprise setup. However it does suffer from insider attacks. In spite of all the features that it provides it can't prevent an authorized user from stealing information and leaking it out. In addition one can only create or use protected documents using applications that incorporate Windows RMS technology. Any new application needs to use the RMS Software Development Kit (SDK) to make it RMS enabled. As mentioned earlier, Data Paladin comes over these limitations by not preserving state across DP sessions, restricting data to be only present on the DP server at any time, and using virtualization and encryption to guarantee security.

EMC's Information Rights Management Software [9] (EMC Documentum IRM family), formerly Authentica, addresses data security concerns for various applications like Adobe

Acrobat, Microsoft Office etc. However it uses Windows Digital Rights Management Client APIs and thus has application specific solutions. For each application that it attempts to add security features to, it needs to have a different approach and an installable package for the end user. Data Paladin on the other hand provides an end to end rights management system which is application independent, thus being a big step ahead.

The Liquid Machines ERM suite [10] provides quite a number of options to users and administrators pertaining to Data Protection. However it works at document level granularity, meaning that administrators set explicit permissions at the file level to allow/inhibit users from accessing documents in certain ways. The state information that the ERM has to maintain in such cases is pretty significant. However since it also allows data to go out of the protected servers (carried around on laptops, flash drives etc.) the data is still subject to compromise in spite of being encrypted (brute force on offline data for example). It also needs extra work through ERM Policy Management and Content Protection APIs for new applications that users want to include in the ERM framework. A Data Paladin system doesn't need to bother about the kind of application that a user might choose to use, since the isolation and containment of applications running on DP clients is through virtualization using FVM technology. The state maintained in a DP session lasts only as long as the session lasts, after which it is destroyed automatically for security. The state here is the footprint of the virtual machines which are created on the DP clients. In addition DP provides most of the services that are provided by Liquid Machine's ERM suite including automatic encryption of data, virtualizing the Windows Clipboard, preventing local printing etc.

Adobe's LiveCycle Rights Management ES system [13] offers almost the same kind of security features as the others mentioned here. However its integrated tightly with applications that it supports. LiveCycle only supports a fixed list of file formats. They integrate with applications, so that the end user can use the application interfaces (e.g. Adobe Reader, Pro/ENGINEER) to enable document protection. Again dependence on applications limits the types of data that LiveCycle can protect, which is clearly an area where Data Paladin wins.

Oracle's Information Rights Management (IRM) [12], formerly SealedMedia, is another player in this area. It relies heavily on encrypting or "sealing" data that is stored outside of the IRM servers on clients. Sealing or encrypting tools need to be installed on the client desktops for integration with authoring applications, email clients, content management and collaborative repositories. The decryption keys are stored on the IRM servers. User rights for documents can be assigned either at the time of sealing them, or later on. These rights are stored for each document on the IRM server. Like in all the other technologies mentioned above, Oracle IRM too is not application independent, which means that only a limited set of applications and environments are supported.

To sum it all up, Data Paladin was designed keeping in mind the overall requirements for Enterprise Rights Management Systems, and aims to plug the gaps in existing solutions.

Chapter 3

3 Data Paladin Architecture

3.1 System Design

The Data Paladin architecture has a central server which exports a filesystem over Samba. The client hosts which run Windows access the share using the Common Internet Filesystem (CIFS) protocol. On the server side, the setup could either have a:

- Windows machine running CIFS and exporting its filesystem to the clients, or

- A Linux CIFS proxy Server sitting in front of multiple File Servers which implement Samba protocol and export their filesystems.

When a DP client attempts to open a file from the DP server, a DP session is initiated. One of the things that happen during the session setup is a key exchange between the DP client and server. Once the keys are exchanged, a virtual machine (VM) gets created on the client dynamically, and the file is then opened with the associated application inside the virtual machine. Like was mentioned earlier, the virtual machine has its virtual filesystem residing on the filesystem of the DP server. For each server that the client connects to, there is a VM that is created on the client.

Furthermore to make sure that the communication between the DP server and the client stays secure, we encrypt all traffic on the network on the fly. The key used for encryption is the one which was exchanged between the client and the server when the DP session got initiated. Any communication attempted by the applications running inside the VMs to talk to an external network entity other than the DP server is blocked at the network layer.

In order to ensure data consistency and security of the overall system, there needs to be a heartbeat mechanism to ensure that both sides know of each other’s well being. If either side notices a network isolation of the other party, a cleanup is triggered.

When the user on the client host chooses to close the DP session, he closes all the applications that he had opened for that DP server, and then terminates the DP client. This causes the VM that was created for that DP session to get destroyed automatically. Thus there is no state that is preserved between DP sessions. Whatever a user does in a session is either written back to the DP server and preserved there, or is wiped out once the session ends to protect any chances of information leakage.

3.1.1 Data Paladin Client Side Components

A Data Paladin client host has the following components running on it:

- FVM for confinement and for extending the server's execution environment
- Network Driver Interface Specification [5] (NDIS) Intermediate Driver for network encryption and blocking
- Data Paladin Client for heartbeat, dynamic key generation and exchange with server etc.

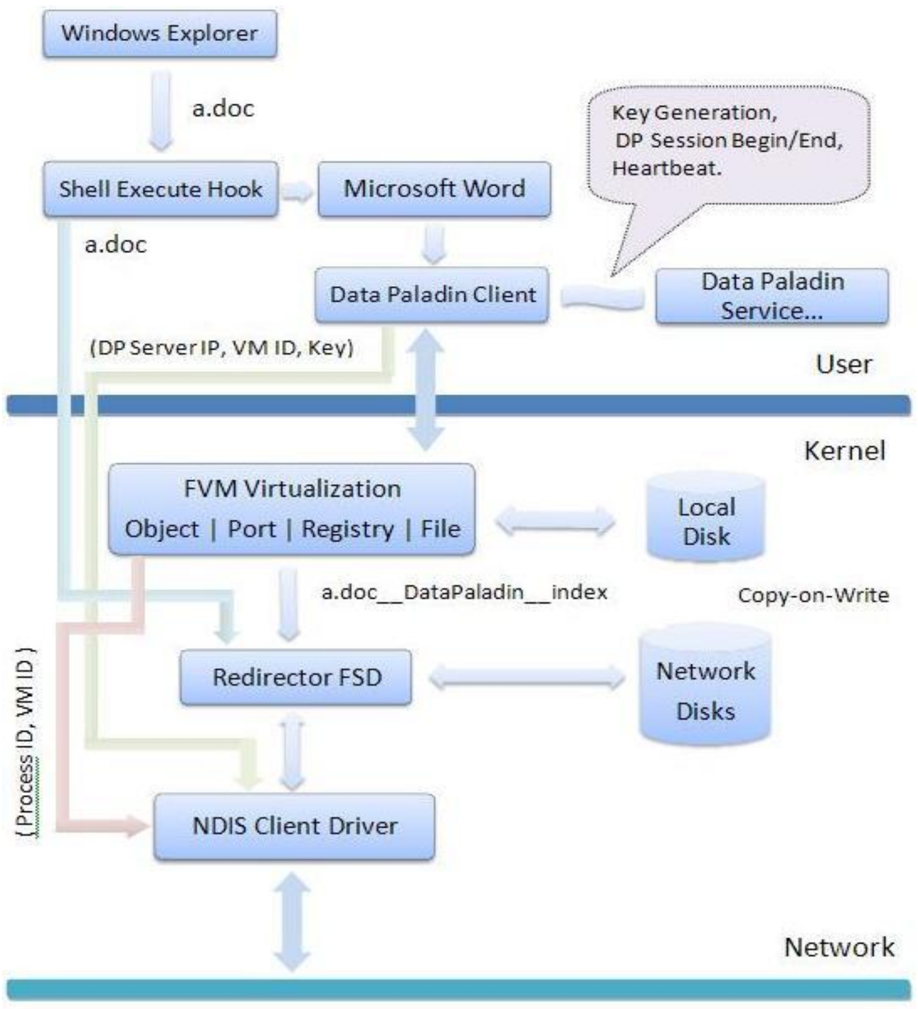

Figure 2: Components of the Data Paladin Client on Windows

3.1.2 Data Paladin Server Side Components

On the server side, as mentioned earlier we could have two different scenarios:

3.1.2.1 DP Server and File Server on the same machine

In this case both the server and client machines are running Windows. The different components on the server are:

- Transport Driver Interface (TDI) Driver to distinguish between local and remote CIFS clients
- Filesystem Filter Driver to selectively send back file contents or links to the CIFS client.
- NDIS Intermediate Driver for network encryption
- Data Paladin service for dynamic key generation and exchange, heartbeat etc.

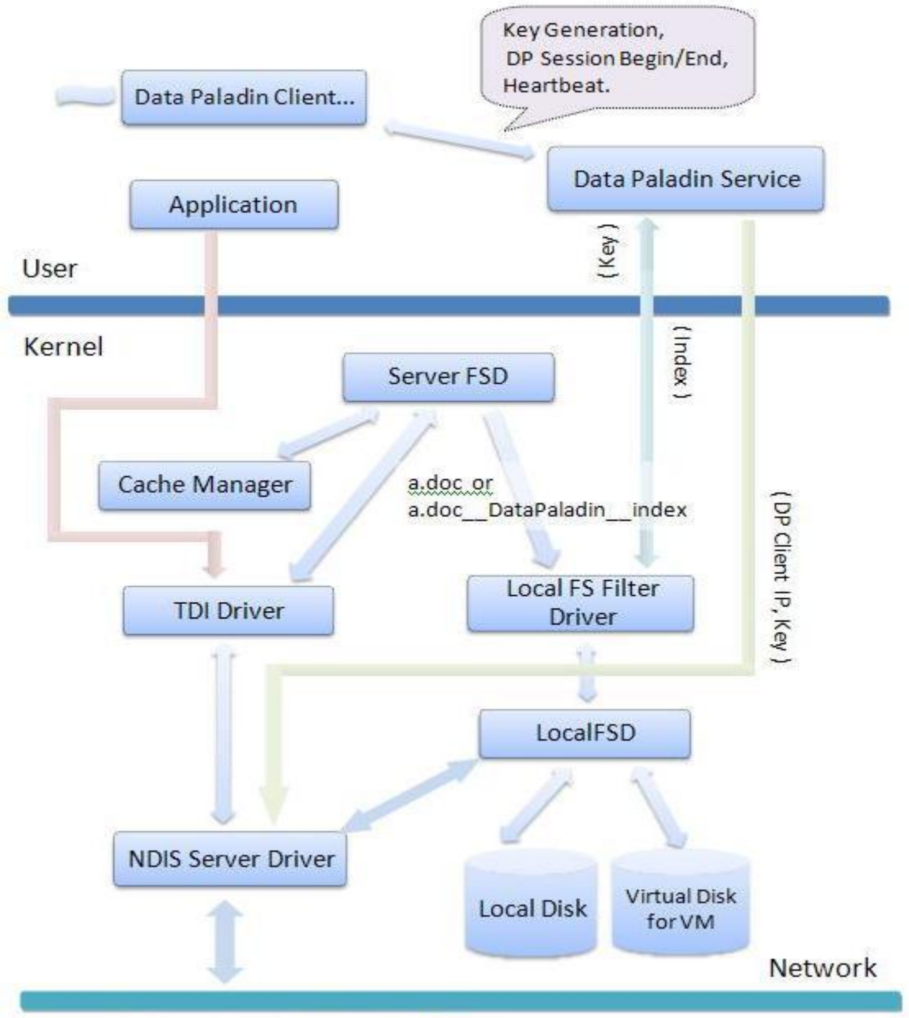

Figure 3: Components of Data Paladin Server on Windows

3.1.2.2 DP Server and File Server on different machines

An alternative approach is to have the DP server decoupled from the File Server. This is done so that the architecture can be storage vendor neutral. The actual file server could be anything that talks Samba protocol e.g. NetApp NAS filers, Storage Area Network (SAN) Arrays etc. The DP server sits on a Linux machine which acts as a CIFS proxy. The proxy provides a single SMB namespace to the clients on one side, and interacts with multiple different File Servers over Samba at the backend.

In such a setup it is assumed that the traffic between the DP server (CIFS proxy) and the File server(s) is secure, and the Data Paladin architecture is then only concerned about securing the channel between the DP server and the clients.

In this case the DP server needs to run the following components:

- CIFS proxy to re-direct the CIFS requests to appropriate backend SMB servers. The proxy also needs to have the discretion as to when to send a link back as opposed to the file contents for a read request.
- Netfilter Xtables addon for packet level encryption. This would be the counterpart for the client side encryption at the NDIS layer.
- Data Paladin service for key generation, heartbeat, identification as a DP server.

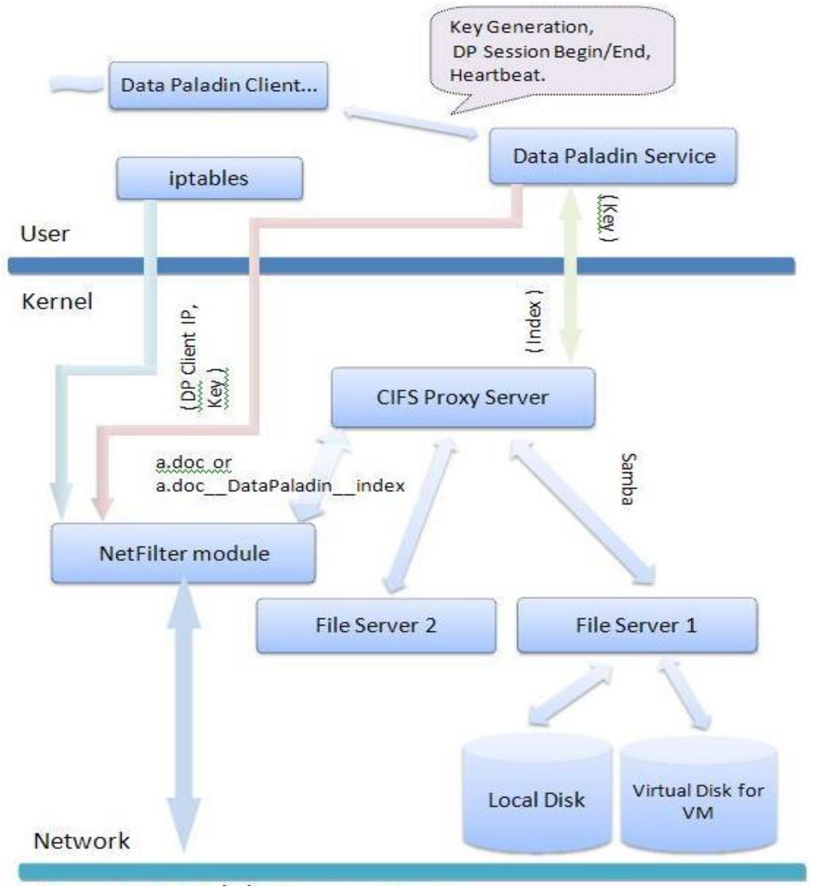

Figure 4: Components of Data Paladin Server on Linux

3.2 Extending the Server's Execution Environment with FVM

As mentioned earlier, we leverage light weight inexpensive Featherweight virtual machines to achieve containment of the applications on the DP client, so that we can run them in the execution environment of the DP server.

The FVM virtualization layer is implemented by intercepting Windows system calls, which are exposed to the user-mode applications through a set of user mode dynamic linked libraries (DLL"s). The intercepting mechanism is implemented mostly in the kernel (since it is difficult to be subverted or bypassed than the user mode interceptions). However, some of the interceptions are implemented in the user space too. The reason for user mode interception being, some of the system calls like those managing daemon service, GUI window and network interface either do not have kernel mode interface or have a kernel mode interface with no clear documentation. The files, registries, kernel objects are virtualized inside the kernel. The kernel mode interceptions are implemented using a driver, which modifies the system call entry point in the System Service Dispatch Table (SSDT) within the kernel. The user mode component is a set of DLLs that modify library function entry points using the Detours library [7].

For Data Paladin the user level interception is used to implement features like virtualizing the Windows clipboard, and blocking printing from applications running inside the DP client.

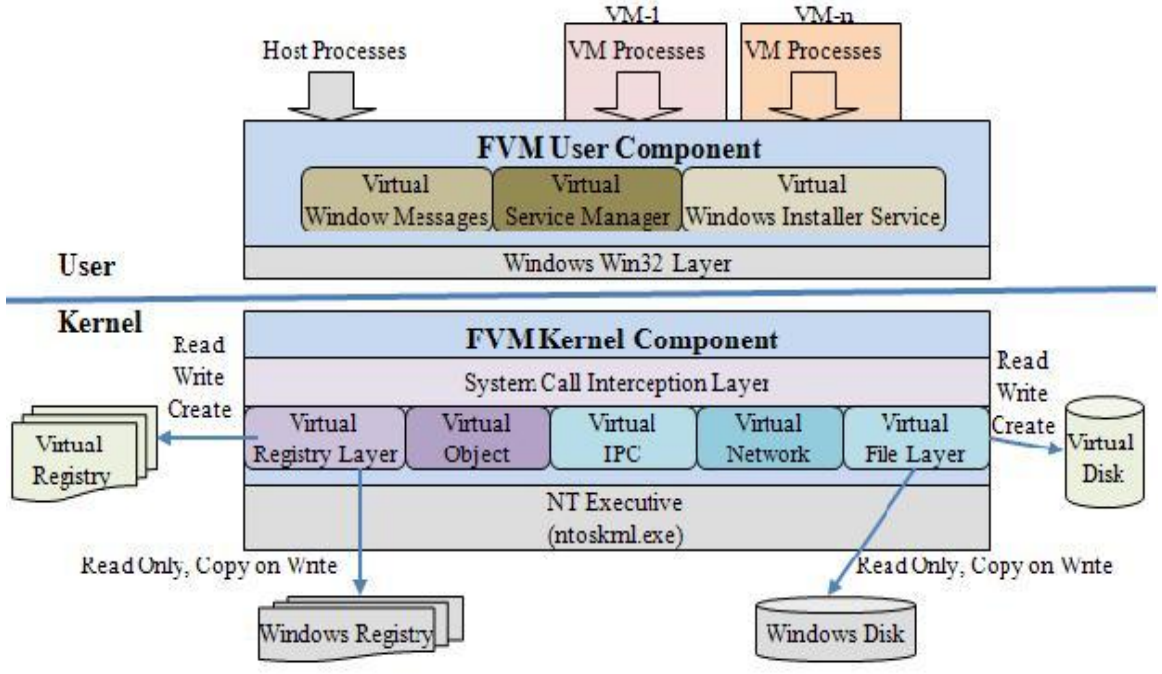

Figure 5: Design of Feather Weight Virtual Machine Technology

3.2.1 Clipboard Virtualization

We need to virtualize the Windows clipboard so that the user can't use the clipboard to copy information from applications running inside the VMs on the client host, and paste it in some other application running either on the client host or inside a different VM. Essentially we make sure that each VM on the client gets it own clipboard buffer, which is isolated from the clipboard buffer of the host OS and that of other VMs running on the same machine. This makes sure that the information inside a VM stays within it and can't be taken out via the clipboard.

3.2.1.1 Implementation of Clipboard Virtualization

For the purposes of virtualizing the clipboard, there are four user level APIs that are intercepted:

- GetClipboardData() – This is called when an applications wants to paste something from the clipboard.
- SetClipboardData() – This is called when an application does a copy into the clipboard.
- EmptyClipboard() – This API is called to empty the clipboard before SetClipboardData() is called.
- GetFormatAvailable() – Since the Windows clipboard supports multiple formats (text, images, rich text etc.), this API is used to check what formats are available in the system clipboard pertaining to a given application. So while an application like Notepad might work with only a few clipboard formats (plain text etc.), an application like Microsoft Word would work with much more formats (normal text, rich text, images etc.).

  When an application calls GetClipboardData(), if the Windows clipboard buffer doesn't already have the format that the application is asking for, then a conversion happens on the fly (done by Windows) from the format currently in the clipboard, to what the application is asking for, if such a conversion is possible. For example, one might see a conversion from normal text to rich text happening on the fly, if the copy was done from Notepad and the paste is happening into Wordpad.

  The Windows system clipboard stores pairs of the format ID, and the associated data in that format inside the clipboard buffer.

As mentioned earlier, we introduce new clipboard buffers for the host and each of the virtual machines on the host. This is in addition to the standard Windows clipboard buffer. The purpose is to keep the clipboard for the host and that for the individual VMs separate from each other, and from the system clipboard buffer.

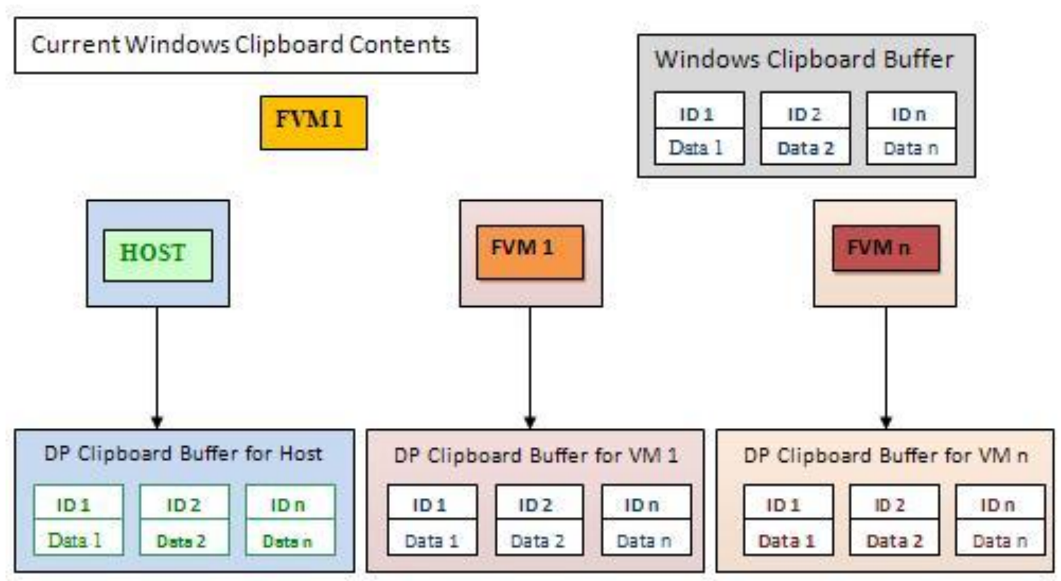

Figure 6: Clipboard Virtualization using FVM

In our intercepted APIs this is what is done:

- EmptyClipboard() – Whenever an application calls EmptyClipboard(), we need to empty the system clipboard, as well as the clipboard specific to the host or a VM, depending on where the application is currently running.

- GetFormatAvailable() – When an application calls GetFormatAvailable(), depending on where the application is running, we consult either the host or the VM specific clipboard buffer, to see if the format is available, returning TRUE or FALSE accordingly.

- SetClipboardData() – When an user does a copy operation, and a SetClipboardData() is called, depending on if the application is running on the host or inside a VM, we need to copy the data into the System Clipboard buffer, and to the host or VM specific buffer that we maintain. In addition, we also need to store the information if the Windows system buffer currently has clipboard contents for either the host or one of the VMs.

- GetClipboardData() – When a user does a paste, GetClipboardData() is called. At that point, we need to find out if the related application runs on the host or inside a VM. We then check if the Windows clipboard buffer stores the clipboard buffer contents that we are interested in. If yes, then we need not really do anything and the contents of the Windows clipboard buffer is passed on to the application.

  If however the System clipboard buffer doesn't have the relevant clipboard buffer in it, we need to copy the contents from the host or VM specific buffer that we maintain to the System buffer, after which the data is sent to the application. As a result of this, a paste operation inside a given VM can never get the contents of either the host's or another VM's clipboard buffer.

The purpose of bringing in the System clipboard buffer every time is to keep us independent of the format conversion that Windows does automatically for applications.

3.2.2 Blocking Local Printing from Inside VMs

To make sure that a user can't print locally from an application running inside a VM on the client host, we block local printing for applications inside virtual machines.

The way this is done is by hooking a user level API OpenPrinter(). Inside our interception for this API, we check if the application calling this API is running inside a VM. If it is, then we return a FALSE from the API, which essentially blocks the application from opening the printer(s).

Since we rely on FVM to provide an application independent way of providing data protection, the FVM layer's functionality is critical to Data Paladin. Due to the multitude of ways in which applications interact with the registry and the file system on a machine, ensuring high performance of the Registry and the File Virtualization layers in FVM is very important, and is an on-going area of improvement. COM virtualization is another area where work is on to handle all different ways in which applications use COM for their functionality. As we evolve using different kinds of applications with the Data Paladin system, there are things about COM usages which come up, which FVM has to accommodate. Since FVM also virtualizes IPC on a Windows machine, we need to be careful that we do not block any kind of IPC that an application requires to function.

3.3 Run-time Key Exchange Using Diffie-Hellman Technique

The DP client and the server exchange keys when a new DP client connects to a DP server. We use the standard Diffie-Hellman [8] key agreement method to generate keys on the client and the server. The method itself is secure as the actual key is never transferred over the wire. The key is used for the following purposes:

- Encrypting network traffic between the client and the server during the lifetime of the DP session. Once the connection ceases to exist due to normal or abnormal conditions, the key is no longer relevant.

- The key is also used at the client to encrypt the filename which is to be opened. After key exchange, the DP server stores the key for the client in question, and sends back an index or an identifier to the DP client. The client then encrypts the filename it wants to open with the exchanged key, and appends the index to the encrypted string, using a marker (something like "__DataPaladin__"). This encrypted filename is then passed in the file open/read request to the server.

When the request reaches the server, it needs to extract the index out of the filename (with the help of the marker in the filename as mentioned above), map the index to a key which it would have stored in its internal data structures, and then use the key to decrypt the filename sent by the client, to get the original filename back. This is done so that the server is sure that the request it got for a file open/read is from an authentic DP client and not someone who is trying to fake identity. If the server can't trust the client, it sends back just the link instead of the actual file say in case of a read. Only a valid DP client can decrypt the encrypted link contents and proceed further.

3.4 Detecting Remote Samba Clients Using a TDI Driver

This is relevant in case of the DP server and the file server running on the same Windows machine. We use a TDI Driver on the DP server machine. The Transport Driver Interface (TDI) layer is just above the TCP layer in the Networking stack. It's a bridge between applications and network-layer protocols. So a filter driver at this layer sits immediately above the kernel-mode TCP/IP driver, and by intercepting at the TDI layer using a Filter Driver, we get to see and change how applications talk to the network stack.

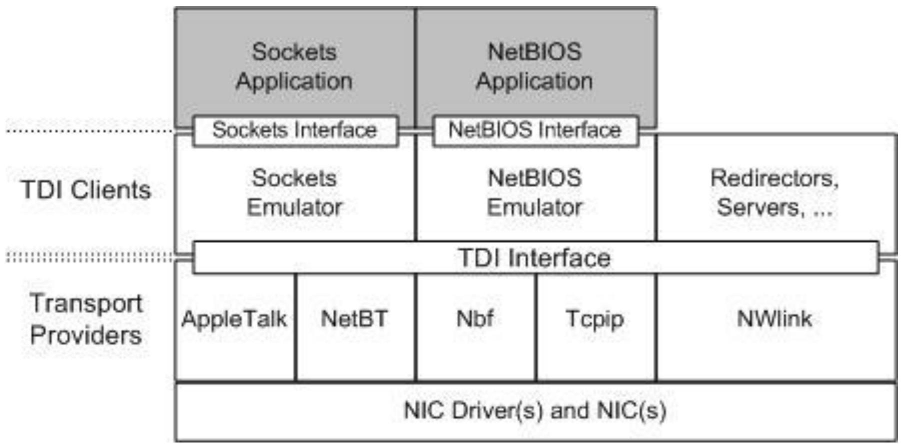

Figure 7: Transport Driver Interface Stack

The main purpose of the TDI driver in case of Data Paladin is for detecting remote Samba clients. This obviously can't be done at the Filesystem layer where one can't figure out if the client requesting a file open/read is a remote one. We need to know this to decide if to send actual back file contents on a file read, or a link. This becomes necessary as a Samba server could act as a samba client for its own share, in which case we don't need to worry about security and the file contents can be sent back directly from the file read operation.

The TDI driver reads into the packets and deciphers Samba headers to identify packets which carry Samba file read requests (Samba operation code 0x2e). It then encodes characters in the filenames which are part of the SMB payload in a way that can be decoded by the Filesystem Filter Driver running on the DP server. On Windows filenames can't have characters whose integer representations are in the range from zero through 31. There are a few additional reserved characters too. We use these characters to encode the filename and to keep the encoding unique.

If the Filesystem Filter driver during a read operation (typically when it's servicing the IRP_MJ_READ request from the I/O Manager) sees that the filename in question has encoded characters (those mentioned above), it can be sure that file read request originated from a remote Samba client and it then sends a link back instead of the actual file contents.

The link contains the Samba server's domain name, IP address, the share name and the filename. This information is then used by the client to open the file using a UNC path.

In conjunction with what was mentioned earlier about the DP client encrypting the filename with the DP session specific key, it is only when the filename is not already encrypted by the DP client at the FVM layer, that the TDI layer encodes it. Here is the how the file open proceeds:

1. The DP client first opens the file using the actual filename. The TDI layer encodes the filename.
2. The DP server decodes it and in the process knows that the request came in from a remote Samba Client. In then sends a link back to the client.
3. The client then creates a VM on the client host, and sends the file open request from inside the VM. Inside the VM we encrypt the filename and append the marker ("__DataPaladin__") and the index that was sent by the DP server to it as part of the initial key generation and session setup. In this case, the TDI driver on the server sees the "__DataPaladin__" marker in the filename and doesn't encode the characters of the filename. The Filesystem Filter Driver on the server decrypts the filename, using the key corresponding to the index that it sees in the filename that it got. Once the filter driver has the unencrypted filename, it opens the file.

3.5 Using a Linux CIFS Proxy to Abstract File Servers

This is the case where the Data Paladin Server and the File Server are not on the same machine. More specifically we want to decouple the DP server logic from the actual File Server. The Data Paladin server runs on a Linux machine which runs a CIFS proxy server. We use Samba v4 for our purposes. The reasons for wanting to decouple are:

- Be independent of the File Servers: Users of the DP system might not want to or might not have the capability to install the Data Paladin Server bits on the File Server directly. A typical example of such a case would be the use of NetApp Filers to host the Samba share. One can't possibly install Data Paladin Server bits on a NetApp filer. The way out then would be to use a proxy which sits in front of the File Server and provides a layer of abstraction for the Samba clients. The proxy redirects all Samba requests to the backend file server. In such a scenario we install the Data Paladin Server bits on the CIFS proxy server running on Linux.

- Providing a uniform namespace: Since the proxy server sits in front of multiple backend File Servers, it provides a uniform namespace comprising of exported filesystems from all the backend file servers, to the clients.

3.5.1 CIFS Proxy Support in Samba v4

In Samba v4 all the file-system related operations go through an abstraction layer for the virtual file system (VFS). This VFS layer is modeled after both POSIX and NTFS semantics. The latter is called the NTVFS interface.

NTVFS was created in Samba v4 to ensure that multiple, '*rich*' virtual file-systems could be created and plugged into Samba. One example of such a file-system is 'CIFS on CIFS' back end for samba, also called as the *CIFS Proxy*. It provides a NTVFS backend that talks to one or many CIFS servers at the back-end, thereby providing a unified namespace to all the clients in front of it.

3.5.1.1 Namespace Management in CIFS Proxy

Since, the CIFS Proxy provides a unified namespace for all the shares existing across all the backend file servers in a domain, it needs to map each of the backend server's shares to a virtual share and expose it to the CIFS/Samba clients. To achieve this mapping operation, the Samba configuration file "smb.conf" located usually in /etc could look like this:

```
[myshare]
        ntvfs handler = cifs
        cifs:server = myserver
        cifs:share = test
```

In the sample shown above, backend server named – "myserver" has a CIFS share called 'test'. The CIFS Proxy Server is exposing it as *myshare*. Suppose the proxy server's netbios name is 'cifsproxy', then the client can access 'test' shares data as '*\\cifsproxy\myshare'.*

3.5.1.2 Authentication in CIFS Proxy

CIFS Proxy provides two methods of authentication:

- Password Specified: In this the samba config file (/etc/smb.conf) has the username/password hardcoded in the file. This method is usually not preferred due to security reasons. E.g.

```
[myshare]
    ntvfs handler = cifs
    cifs:server = myserver
    cifs:user = tridge
    cifs:password = mypass
```

```
cifs:domain = TESTDOM
cifs:share = test
```

- Delegated credentials for domain level authentication: If the incoming user is authenticated with Kerberos, and the machine account for this Samba v4 proxy server is 'trusted for delegation', then the Samba v4 proxy can forward the client's credentials to the target Samba Server. This method works only if all the clients, proxy server and the Linux Samba server (the file server here in this case) all belong to one domain.

  The Samba server can join the domain using the command – ‘net join <domain> member’.

  ```
  [myshare]
          ntvfs handler = cifs
          cifs:server = myserver
          cifs:share = test
  ```

  Since, we do not provide any credential information, in the “smb.conf” file, the Samba server needs to forward this request to the domain controller for that domain.

As is done in the Filesystem Filter Driver running on a Windows DP server, the CIFS Proxy Server, detects from the filename in a read operation, if the request came from a DP client. If not, then it sends back a link as is done by the Filesystem Filter Driver on Windows.

3.6 On the Fly Encryption/Decryption

To keep the communication channel between the server and the client secure, there ought to be no plaintext on the wire between the two. Essentially we need to make sure that data is not transmitted in plain text over the network between the DP client and server. In addition it is preferable to have the encryption layer in the kernel, so that it can’t be bypassed as easily as it could be if it were at user level.

A couple of options were weighed in here as to how and where to encrypt and decrypt the data on the client and the server hosts:

- **Encryption at the Filesystem Filter Driver Layer**: One could encrypt and decrypt at a filesystem filter driver on the client and server. This would mean that after a read request is serviced, the filter driver in the read completion routine (completion routine for an IRP with major code IRP_MJ_READ) would encrypt the buffer which would have the bytes that were read from the underlying filesystem.

  On the other hand just before a write is made to the underlying filesystem, the filter driver would decrypt the contents of the IRP buffer, and then let the I/O go through.

This ensures that data on the disk is always plaintext, but whenever it goes out of the filesystem layer it is encrypted, to be decrypted only when it comes back to be written on to the disk. This definitely sounds plausible logically. The problem however arises out of encrypting the contents of the read buffer at the filter driver layer. This buffer is memory mapped and is shared by the filter driver and the Windows Cache Manager. Due to this, the parts of files which were read and which are in cache get encrypted. If one then opens these files locally on the machine, he would see partly encrypted contents. It is not trivial to disassociate this buffer from the Cache Manager, which makes this approach not practical.

- **Encryption at the TDI Layer**: The TDI layer was another possible place where we could encrypt contents before they went out on the network. Both the DP client and server would then be running TDI drivers. We would have to read Samba headers in the packets that the TDI layer sees, and look for "read responses" and "write requests" as places to encrypt and decrypt contents respectively. To be more specific, a Samba read request/response can be identified by looking at the Samba command code (0x2e) in the Samba header in a packet. In case of a write, the Samba "write request" and "write response" command code (0x2f) can be searched for.

  We could encrypt in case of a "read response" originating at the server, and a "write request" originating at the client, so that the data on the network is not plain text. The TDI driver on the client would decrypt in the "read response" to get back plain text. The TDI driver on the server would decrypt it at "write request" to make sure that the data on the disk is unencrypted.

  The TDI driver is sent IRPs (I/O Request Packets) by the I/O Manager in Windows for every packet that goes through the TDI filter driver. The packet payload is stored in what is called Memory Descriptor Lists (MDLs) which are associated with every IRP. These MDLs are linked lists of small buffers which store data. However the problem arises due to the fact that these buffers point to memory mapped files which a Samba server opens and reads data from, to send over the network to clients. So if we attempt to encrypt the packet payload in these MDLs, parts of the original files on the DP server turn out getting encrypted. It is not trivial either to get out of the way of the Windows Virtual Memory Manager, and to create one's own MDL and IRP to send down to the lower layer in the network stack to get around this problem.

  Essentially with both the approaches outlined above, we were too close to the application layer (Samba in our case). So we had to constantly be on the lookout for ways to not come in the way of how these applications get affected by the Windows Cache Manager, the Virtual Memory Manager etc. Since what we needed was to just make sure that contents over the wire don't go as plain text, it made more sense to move as much away from these applications as we could, and encrypt data lower down the stack in the network. This is how we came up with implementing on the fly encryption/decryption at the NDIS layer on Windows and the Netfilter layer on Linux, both of which are much lower in the stack and are independent of the Samba layer.

3.6.1 NDIS Intermediate Driver Layer Encryption on Windows

Windows based operating systems support several types of kernel-mode network drivers. Microsoft Windows 2000 and later versions support the following kernel-mode network drivers (NDIS drivers):

- Miniport Drivers: A miniport driver directly manages a network interface card (NIC), providing an interface to the higher-level drivers.

- Protocol Drivers: An upper-level protocol driver implements either a TDI interface, or possibly an application-specific interface, at its upper edge to provide services to users of the networking layer. At its lower edge, a protocol driver provides a protocol interface to pass packets to and receive incoming packets from the next-lower driver.

- Intermediate Drivers: An intermediate driver interfaces between upper-level protocol drivers (e.g. a legacy transport driver), and a miniport driver below.

For the purpose of Data Paladin, we use a NDIS Intermediate Driver to encrypt and decrypt traffic between the DP client and server. Because of its intermediate position in the driver hierarchy, an intermediate driver must communicate with both overlying protocol drivers and underlying miniport drivers in order to expose the following:

- Protocol entry points: At its lower edge, NDIS calls the ***Protocolxxx*** functions to communicate requests from underlying miniport drivers upwards. The intermediate driver thus looks like a protocol driver to an underlying miniport driver.

- Miniport driver entry points: At its upper edge, NDIS calls the *Miniportxxx* functions to communicate the requests of one or more overlying protocol drivers downwards. The intermediate driver here looks like a miniport driver to an overlying protocol driver.

Although it exports a subset of the *MiniportXxx* functions at its upper edge, an intermediate driver does not actually manage a physical NIC. Instead, it exports one or more virtual adapters, to which overlying protocols can bind. To a protocol driver, a virtual adapter exported by an intermediate driver appears to be a physical NIC. When a protocol driver sends packets or requests to a virtual adapter, the intermediate driver propagates these packets and requests to the underlying miniport driver. When the underlying miniport driver indicates received packets, responds to a protocol's requests for information, or indicates status, the intermediate driver propagates such packets, responses, and status up to the protocol drivers that are bound to the virtual adapter.

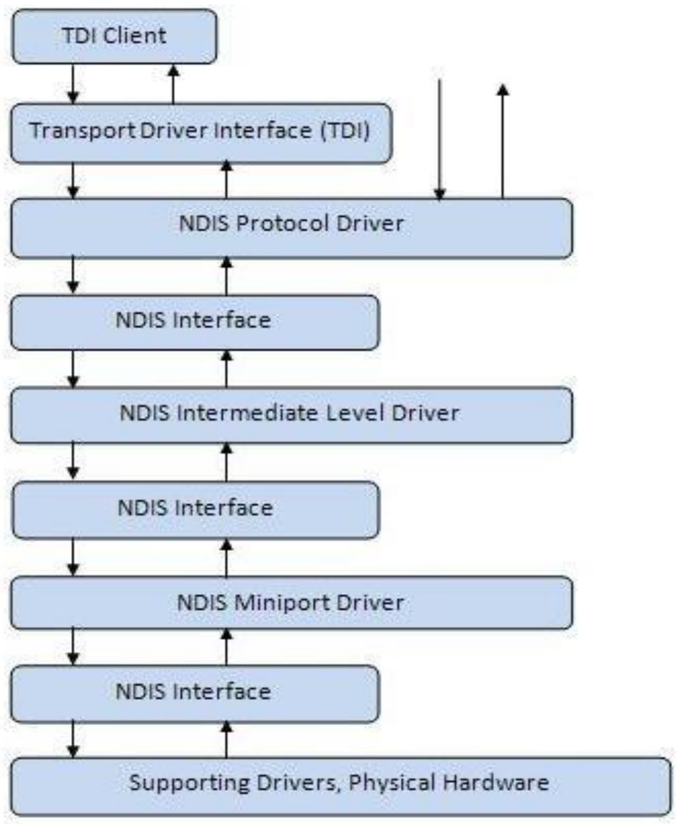
TDI Client
Transport Driver Interface (TDI)
NDIS Protocol Driver
NDIS Interface
NDIS Intermediate Level Driver
NDIS Interface
NDIS Miniport Driver
NDIS Interface
Supporting Drivers, Physical Hardware

Figure 8: NDIS Driver Stack

The NDIS layers on the DP client and the server selectively encrypt outgoing packets based on the destination IPs on the packets. We also set the 6th and 7th bits of the TOS (Type of Service) field in the IP header to mark packets which we encrypt for the other end to identify encrypted packets and decrypt them. These fields are reserved for future use in the IP header and are always 0 otherwise.

Also since we are concerned about securing the Samba traffic between the nodes, we only care to encrypt traffic that is coming from or going to the TCP Samba port (445). The DP server is always configured to accept Samba connections over TCP at port 445, rather than over UDP. We of course also need to re-compute the IP and TCP checksums after we encrypt packets and change IP headers.

3.6.1.1 Client-side NDIS Encryption

At the client side the following components interact with the NDIS driver:

- Data Paladin Client: The Data Paladin client inserts the key for the encryption into the NDIS layer, along with the IP of the DP server with which it exchanged the key, and the VM ID for the VM which was created on the client for talking to the particular DP server. So if the client is connected to ‘n’ different DP servers, there would be ‘n’ VMs that would be created on the client host. So the DP client would insert ‘n’ different keys with the corresponding DP server IPs, andVM IDs into the NDIS driver.

- FVM Driver: Since all processes in the DP session are running inside VMs created on the client, the FVM driver knows the Process IDs of the processes for a particular DP session. It inserts the pid and the VM ID pairs for all processes running inside VMs on the client host, into the NDIS driver.

So the NDIS driver has the following information:
- DP Server IP, Key, VM ID triplet
- Process ID, VM ID pair

When a packet has to be sent down the stack, the NDIS driver checks the destination IP on the packet. If it happens to be one of those DP server IPs which was inserted into the NDIS driver by the DP client, that packet might be a target for encryption. It then checks if the application which owns the current packet corresponds to one that is running inside a VM (from the process id and by matching it with the VM ID that the NDIS driver has), and if it is, the associated key that it needs to use for encryption from the information it already has. This key is then used to encrypt the TCP payload of the packet.

Decrypting incoming packets is easier. The driver has to first check if the 6th and 7th bits of the TOS field are set. If they are, then the driver can get the key to decrypt the TCP payload by looking up the key for the source IP of the current packet.

3.6.1.2 Server-side NDIS Encryption

This is relevant in the case of the DP server running on Windows. The server side components which interact with the NDIS driver are:

- Data Paladin Service: Once the DP service has generated the keys for a new DP session by talking to the client, it inserts the client IP and key pair into the NDIS driver.

When the NDIS driver receives a packet to send down to the lower layer, it checks the destination IP on the packet. If the IP happens to one of those IPs which was inserted by the DP service into the NDIS driver, then it knows that the packet is destined to the DP client, and it encrypts the TCP payload of the packet with the key it had stored for the client IP.

Decryption of packets is done exactly the same way as it done on the client.

3.6.2 Netfilter Xtables Addon on Linux

This approach is used in the case where the DP server and the File Server are different machines. So the DP server is a Linux machine which is acting as a CIFS proxy. As mentioned earlier, since we assume that the network between the DP server and the file server is trusted, the encryption needs to be done only on the network between the DP client and the Linux machine running the DP server.

3.6.2.1 Overview of Netfilter and Xtables

To have a secure communication channel between the Windows CIFS clients and the CIFS proxy Server running on Linux, the Netfilter framework was used as a counterpart of the NDIS layer on Windows for encrypting at the packet level. The framework uses hooks at various places in the networking stack, which get invoked at various stages of a packet's lifetime in and out of a machine.

Each protocol (IPv4, IPv6 etc.) defines such "hooks", which are well-defined points in a packet's traversal of that protocol stack. At each of these hooks, the protocol calls Netfilter framework with the packet.

Kernel modules can also register to listen to different hooks for each protocol. In such cases, when a packet is passed to the Netfilter framework, it checks to see if any such module has registered for the particular protocol and the hook. In case there are such modules, then Netfilter passes the packet to these modules, at which point the module can decide to either examine or alter the packet, following which, it can do one of the following:

- Discard or drop the packet
- Allow it to pass
- Direct Netfilter to ignore the packet
- Ask Netfilter to queue the packet for user-space, for further asynchronous handling

In this context, Xtables refers to the kernel module that provides the generic, protocol-independent firewalling mechanism on Linux. ip_tables, ip6_tables etc. are the kernel modules providing family-specific tables for iptables, ip6tables tools. When compiling the kernel, one could choose to have these built as part of the kernel itself, instead of being loadable modules.

Xtables deals with four tables - filter, nat, mangle and raw. The names "filter" (filtering packets) and "nat" (network address translation) explain pretty much what the tables are used for. The "mangle" table is used for packet alteration, and the "raw" table is used for connection tracking. The purpose of these tables is to hold chains, both system and user-defined. The chains in turn contain rules which can be terminating or non-terminating in nature. The rules specify targets for packets which match the rules. A packet enters a chain in a table based on its type (incoming, outgoing or forwarded), and then traverses rules till it encounters a terminating rule.

For the purpose of Data Paladin, we need to change the packet payload and hence we use the "mangle" table, with the “INPUT” and the “OUTPUT” chains. In essence keeping the ethernet, IP and TCP headers intact, we have to encrypt/decrypt the TCP payload, and re-computer IP and TCP checksums.

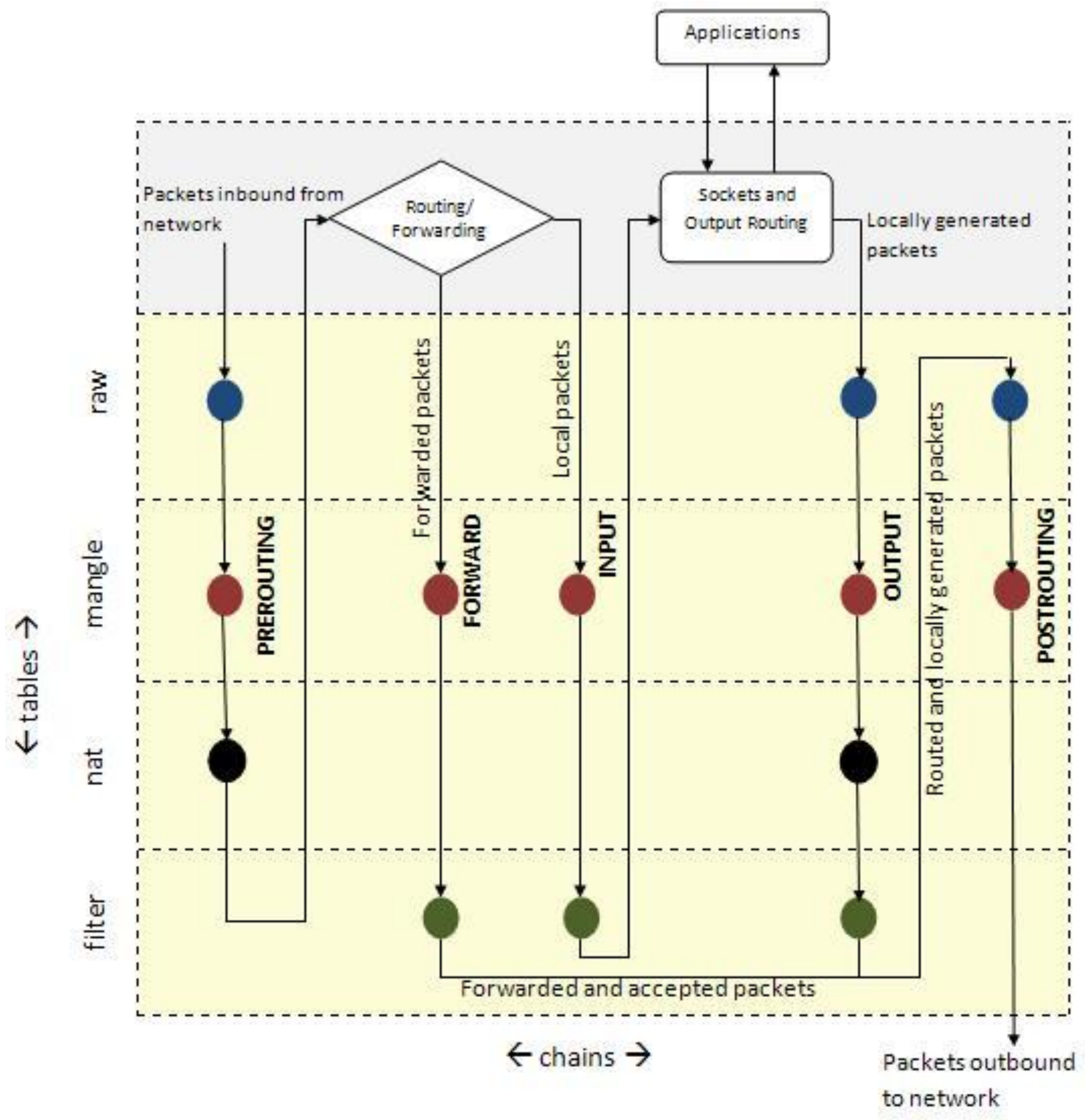
Applications
Packets inbound from network
Routing/ Forwarding
Sockets and Output Routing
Locally generated packets
raw
mangle
nat
filter
← tables →
PREROUTING
Forwarded packets
FORWARD
Local packets
INPUT
OUTPUT
Routed and locally generated packets
POSTROUTING
Forwarded and accepted packets
← chains →
Packets outbound to network

Figure 9: Netfilter tables and chains

3.6.2.2 Implementation of the Xtables Addon Module

The logic for the packet level encryption/decryption is implemented as an Xtable addon. Such an addon has 2 parts - a kernel part (a loadable module) which implements a new target ("DPEncDec") for encrypting and decrypting packets, and a user-land part (add on to the "iptables" user-land utility via a shared library).

The kernel module registers itself with Netfilter to be notified of packets which match a certain criteria:

```
static struct xt_target DPEncDec_tg_reg __read_mostly = {
     .name       = "DPEncDec",
     .revision   = 0,
     .family     = PF_INET,
     .proto      = IPPROTO_TCP,
     .table      = "mangle",
     .target     = DPEncDec_tg,
     .targetsize = XT_ALIGN(sizeof(struct
                      xt_DPEncDec_info)),
     .me         = THIS_MODULE,
};
```

Here "DPEncDec" is the new target added by this module. The module deals with IPv4 and specifically TCP packets. It also associates itself with the mangle table, which means that iptable rules only for the mangle table would cause packets to reach this module.

Let’s now look at a rule added to the "mangle" table’s "OUTPUT" chain, with protocol as TCP, to encrypt all packets which are destined to say a node with IP 192.168.2.7. The key for encryption/decryption is passed on as an option’s (“DataPaladinKey”) value.

```
iptables -t mangle -A OUTPUT -j DPEncDec -p tcp  --DataPaladinIP
192.168.2.7 --DataPaladinKey RETHER_DP
```

Similarly to add a rule to decrypt all packets coming from a given IP, here is the iptables command:

```
iptables -t mangle -A INPUT -j DPEncDec -p tcp --DataPaladinIP
192.168.2.7 --DataPaladinKey RETHER_DP
```

Note the "INPUT" chain in the above rule.

The packet mangling continues till the time we have the rules added in iptables. Once they are deleted, the module ceases to be notified on any incoming or outgoing packets.

We use an in-place encryption algorithm. The idea being that we do want to introduce fragmentation by changing the payload length in the original packet, as part of encryption. RC4 is one such algorithm which keeps the payload length intact.

3.7 Heartbeat between the DP Server and Client

In any secure environment, one needs to be able to handle failures of participating entities, and make sure that in the face of failures, the data that is being protected stays safe. For DP, the failure could be due to multiple reasons:

- The DP client crashing – Could either be an unlikely DP client application crash, or the client host machine crashing, or experiencing temporary network problems.

- The DP server crashing – Again this could be the DP server side service crashing or the server host crashing, or experiencing temporary network problems.

- The network between the client and the server going down.

In any of the above scenarios, we need to make sure that the data stays protected. To achieve this, there is a dedicated heartbeat thread running in both the DP server and the DP client. At regular intervals, both sides exchange heartbeat messages (request and replies). There are two time intervals involved here:

- Heartbeat Quantum ‘q’ secs – A Timer Routine runs every ‘q’ seconds on the DP server and client, and exchanges data with the other side. On a successful exchange of a request and a reply message, the routine signals an event.

- Heartbeat Timeout ‘t’ secs – The per-connection Heartbeat Thread waits in a loop for the event mentioned above to be set by a timer routine indicating that all is well. If the event is not set within a time period of ‘t’ seconds, then it is assumed that the other side is unreachable. The value of the heartbeat timeout is a multiple of the heartbeat quantum value, so that in case of network latencies, at least one of the timer routines which get invoked every heartbeat quantum seconds, gets to talk to the other side and confirm well being.

Once a Heartbeat Timeout is detected, cleanup is to follow. In case of either the client or the server host crashing, the cleanup happens on the party which didn’t have a host machine crash. The keys in the networking subsystem (the NDIS Intermediate Driver in case of a Windows DP client/server, or the Netfilter module in case of a Linux DP server) have to be removed.

One important thing to keep in mind with this kind of a design is synchronization between threads running the Timer Routines and the per-connection Heartbeat Thread. Since the Timer Routines are launched asynchronously every “Heartbeat Quantum” seconds, one needs to make sure that the data they share with the main Heartbeat Thread is valid when they actually access it. This kind of a scenario arises specifically when a heartbeat timeout is detected. The Heartbeat Thread waits for “Heartbeat Timeout” seconds for an event to get signaled by any of the threads running the Timer Routine, failing which it assumes that the other side is unreachable. However before it can do the cleanup and exit, it needs to make sure that all the threads running the Timer Routines for that connection have exited. This ensures that there are no threads which attempt to run the Timer Routine for the dead connection, and end up accessing invalid data.

3.7.1 Client-side Cleanup

The cleanup on the client side involves:

- Removal of the encryption keys from the NDIS layer: This needs to be done so that when the client tries to connect to the server again, the traffic is not encrypted with the same keys which were used in the last DP session. The server side would also remove the keys from the NDIS layer when it detects that the DP client is unreachable.

- Terminating DP applications that were open on the client machine: This is to ensure that in the face of a broken DP connection, the information being protected is not compromised. So all DP applications that were running inside the VMs are explicitly terminated upon a heartbeat timeout.

- Terminating and cleaning up the VM itself: Since the scope of the client side VM is the lifetime of the DP session between the DP client and the server, the VM should be destroyed when the session doesn't exist anymore. So after all the applications inside the VM are terminated, the VM itself is terminated and destroyed.

3.7.2 Server-side Cleanup

The cleanup on the server involves:

- Removal of the encryption keys from the networking subsystem: In case of the DP server on a Windows machine, the keys for traffic encryption between the DP server and the unreachable client need to be removed from the NDIS layer. It is worthwhile to mention that the NDIS layer on the server might be using multiple keys, one per DP client that it talks to. So only the key pertaining to the client for which the server noticed a timeout is removed, and the other keys remain intact. So is the case when the server is on Linux. As part of cleanup in such a case, we need to delete the iptables rules for that particular DP client (rules from the INPUT and the OUTPUT chains).

- Removal of the key pertaining to the client which the server had stored as part of the key generation and session setup. As mentioned earlier, this key is used to decrypt filenames encrypted by the client in a file open/read.

3.8 Network Blocking

To make sure that applications running on the DP client inside VMs, which are reading or writing files protected by the DP server, are not able to extract the information and pass them outside (through emails, or socket connections to servers running on untrusted hosts), we need to make sure that we thwart such attempts. This involves identifying what kind of communication is allowed for applications running inside VMs on the client host.

The allowed list might contain network traffic for purposes like DNS queries, ARP requests/replies etc. These can be identified on either a source/destination IP combination or a source/destination port combination. It could also be based on some kind of a policy which is specific to an environment, and is laid down by an administrator. Once we know what traffic to allow, all other traffic has to be blocked.

One way to thwart network connections is to identify packets which are not in the allow list, and to encrypt the payload and set the RST bit on them. This way a successful network connection would never be possible between the DP client and the external host that it is trying to talk to.

Chapter 4

4 Evaluation

4.1 Evaluation of On-the-Fly Encryption/Decryption

Since we capture every packet at either the NDIS layer on Windows, or at the Netfilter layer on Linux, the encryption logic will add some overhead to data transfer rates between the server and the client. In this section we try to measure the penalty for on-the-fly encryption decryption. One thing to note here is that the in a Data Paladin setup, the data transferred between the client and the server is not bulky in general. For example, if a user opens a Microsoft Word document from the DP server, the data that will be transferred over the network would include the document itself, and any data that Microsoft Word writes on to the disk (temporary files etc). Typically one wouldn't see megabytes of data being exchanged between the client and the server.

There are two kinds of scenarios in which we need to measure performance of the encryption layers:

- All Windows setup: In this case both the DP client and server are on Windows machines, and so we have encryption/decryption at both ends happening at the NDIS layer.

- Windows client and Linux server: In this kind of a setup the DP client is on Windows (encryption at the NDIS layer) and the DP server is on Linux (encryption at the Netfilter layer).

To get an estimate of the overhead of just the encryption layer in the DP system, we do our experiments on Windows and Linux machines which have only the encryption layer running. Of the two machines in the setup, one machine exports a filesystem over Samba. We try to copy a file from the share over to the other machine (all Samba packets are encrypted), measuring the time taken to do so, with and without encryption enabled. We wrote a small Java program to do this. To get the best transfer rates, we read data from the file in chunks as large as the native page size on the test machine (4K bytes). For ensuring the correctness of the encryption/decryption logic, we match the checksums of the original file and the copied file. Experiments were done on files of various sizes, averaging the transfer rates over multiple transfers.

4.1.1 Encryption at NDIS Layer at Both Ends

File Size	Transfer time Without Encryption (ms)	Transfer time With Encryption (ms)
67 kb	18.6	28
251 kb	59.4	77.2
742 kb	222.4	272.2
1565 kb	412.6	674.8
4695 kb	1202.2	1977.8
7723 kb	1828.4	3615.2

Table 1: Encryption overhead for DP Client and Server on Windows

4.1.2 Encryption at NDIS Layer on Client and Netfilter at Server

File Size	Transfer time Without Encryption (ms)	Transfer time With Encryption (ms)
67 kb	15.6	18.6
251 kb	56.4	62.8
742 kb	159.2	169
1565 kb	315.4	375
4695 kb	865.6	1075
7723 kb	1427.8	1866

Table 2: Encryption overhead for DP Client on Windows and Server on Linux

Overall, for the end user, the encryption layer in a Data Paladin setting doesn't cause any noticeable delays. Also encrypting at the Netfilter layer seems to be faster than the NDIS layer, and overall performance numbers in the case where Netfilter runs on the Linux based DP server look slightly better.

4.2 Evaluation of Data Paladin as a System

In this section we calculate the overhead of running applications inside Data Paladin by measuring the time applications take to start on the client in a DP setup. This time is then compared to the time they need to start when we have a normal CIFS client-server setup without Data Paladin. The overhead in case of DP comes mostly from the FVM layer and the encryption layer.

We measure application startup times by using Windows High Resolution Timers. Windows provides the APIs QueryPerformanceCounter() and

QueryPerformanceFrequency() for using the high resolution timers to compute time taken by programs. In case of the DP setup, we keep track of the timestamps just before the client connects to the DP server, and just after the application has been launched on the client host inside a VM. The way to detect when an application has started completely is by using the WaitForInputIdle() API. This API returns after a process has started and is ready to accept user input.

For startup times in a standard CIFS client server setup without any Data Paladin, we open a file using the associated application using the CreateProcess() API and then call WaitForInputIdle() for the application to start up and be ready for user input.

One thing to note here is that the startup times for applications which are launched for the first time in a Data Paladin session is slightly more than the subsequent startup times. This is due to the virtualization that we do in the FVM layer. The first time an application starts inside a new VM (which is created automatically for every new DP Server a client is connecting to), the Registry and the File System virtualization logic in FVM has to setup the environment for use. This is what bumps up the first startup time for applications in a DP environment. This is seen more for applications which do a lot of registry and file system accesses when they start, e.g. Web Browsers which access a multitude of preference related information on the disk (either in the File System or in the Registry), Microsoft Office Suite of Applications etc.

Once an application has been previously run in a DP environment, the startup times for subsequent runs of the applications are almost as good as running them outside of Data Paladin. The overhead that DP adds in such a case is very minimal. Hence the average overhead for applications running inside Data Paladin is only a very small figure. Listed below are the calculated startup times for some common applications inside DP (both first time application startups and average startup times) and outside DP.

Application	Startup Time without DP (ms)	First Startup Time with DP (ms)	Average Startup Time with DP (ms)
Microsoft Word	1296	3237	1544
Microsoft Excel	1274	2216	1363
Microsoft PowerPoint	1143	1959	1371
Internet Explorer	1358	3530	1860
Mozilla Firefox	1322	3404	1736
Notepad	155	432	225

Table 3: Application Startup Times with and without Data Paladin

Chapter 5

5 Conclusion and Future Work

In this thesis, we describe how to implement an application independent end-to-end rights management system. We highlight how we make the system scalable by extending a server's execution environment onto the client using FVM technology, and by running applications locally on the client using the client's compute power.

We also highlight how we can decouple ourselves from the actual storage vendors who make different file servers. We do not need to run any of our binaries on these file servers by choosing to have a CIFS proxy server sitting in front of the actual file servers. This also helps us to have a uniform namespace view of the shared filesystems coming from different file servers, through the Proxy Server.

The network encryption logic developed as part of this solution can be used outside of Data Paladin too for purposes of securing a communication channel. Both in case of the Windows NDIS layer intermediate driver, and the Linux Netfilter Xtables addon, the encryption logic is pretty generic in the sense that it can:

- Be adjusted to use different encryption algorithms at different times.

- Have different criteria for encryption based on IPs or ports. One could dynamically direct the networking subsystem to start/stop encrypting/decrypting traffic to or from a given IP/port.

- Can handle multiple keys at the same time, using each encryption key for a given remote network entity that it is talking to.

- Be used to block or thwart network connections to/from certain hosts.

We could extend the encryption framework to support out-of-place encryption. In other words, right now we use encryption algorithms which do in-place encryption and do not change the length of the target buffer which is to be encrypted. To support algorithms which change the target buffer length, we need to extend the encryption subsystem to handle packet fragmentation/re-assembly correctly. In other words, changing the length of the packet payload as a result of encryption might trigger packet fragmentation and subsequent re-assembly. We need to be able to account for and absorb such kind of changes.

Future work in this area of data protection could include:

- Integration with other DRM systems: We should be able to integrate with other Rights Management Systems in the marketplace. E.g. SharePoint by adding features like preventing users from downloading shared documents which they don't own onto their local machines, thus restricting them to just viewing and editing them online, if they have permissions. A Data Paladin session could automatically be created when the user tries to view or edit a document online, thus adding all the security features present in DP to the Sharepoint session.

We should also be able to integrate with Web Servers to augment security features in web environments, and with Document Management Servers to add to usability and functionality of the combined setup.

- As mentioned earlier, we need to be able to thwart users from running kernel drivers on DP clients to subvert security by intercepting system calls.

Bibliography

[1] Ph.D. dissertation on “Feather-Weight Virtual Machine” by Yang Yu.
http://www.ecsl.cs.sunysb.edu/tr/TR223.pdf

[2] Yang Yu, Hariharan Kolam Govindarajan, Lap-Chung Lam and Tzi-cker Chiueh, “Applications of a Feather-weight Virtual Machine”, to appear in Proceedings of the 2008 International Conference on Virtual Execution Environments (VEE‟08), March 2008.

[3] Feather-Weight Virtual Machines on sourceforge.net
http://sourceforge.net/projects/fvm-rni/

[4] Firewalling, NAT, and packet mangling for Linux
http://netfilter.org

[5] Introduction to Intermediate Drivers
http://msdn.microsoft.com/en-us/library/ms800889.aspx

NDIS Developer’s Reference
http://ndis.com

[6] Microsoft Corporation: Technical overview of windows server 2003
terminal services.
http://download.microsoft.com/download/2/8/1/281f4d94-ee89-4b21-9f9e-9accef44a743/TerminalServerOverview.doc, January 2005.

[7] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In Proceedings of the 3rd USENIX Windows NT Symposium, July 1999.

[8] Diffie-Hellman Key Agreement Method
http://www.ietf.org/rfc/rfc2631.txt

[9] EMC Documentum
http://www.emc.com/domains/authentica/index.htm

[10] Liquid Machines ERM Suite
http://www.liquidmachines.com

[11] Microsoft Windows Rights Management System – Technical Overview
http://www.microsoft.com/windowsserver2003/techinfo/overview/rmenterprisewp.mspx

[12] Oracle Information Rights Management (IRM) – Technical Whitepaper
http://www.oracle.com/technology/products/content-management/irm/IRM-technical-whitepaper.pdf

[13] Adobe LiveCycle Rights Management ES
http://www.adobe.com/products/livecycle/rightsmanagement