symantec.
press

The Art of Computer
Virus Research and Defense

"*Of all the computer-related books I've read recently, this one influenced my thoughts about security the most. There is very little trustworthy information about computer viruses. Peter Szor is one of the best virus analysts in the world and has the perfect credentials to write this book.*"

—Halvar Flake, Reverse Engineer, SABRE Security GmbH

Peter Szor

THE ART OF COMPUTER VIRUS RESEARCH AND DEFENSE

This page intentionally left blank

THE ART OF COMPUTER VIRUS RESEARCH AND DEFENSE

PETER SZOR

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco

New York • Toronto • Montreal • London • Munich • Paris • Madrid

Capetown • Sydney • Tokyo • Singapore • Mexico City

Symantec Press Publisher: Linda McCarthy
Editor in Chief: Karen Gettman
Acquisitions Editor: Jessica Goldstein
Cover Designer: Alan Clements
Managing Editor: Gina Kanouse
Senior Project Editor: Kristy Hart
Copy Editor: Christal Andry
Indexers: Cheryl Lenser and Larry Sweazy
Compositor: Stickman Studio
Manufacturing Buyer: Dan Uhrig

Library of Congress Number: 2004114972

ISBN 0-32-130454-3

Text printed in the United States on recycled paper at Phoenix BookTech in Hagerstown, Maryland.
First printing, February, 2005

to Natalia

This page intentionally left blank

Table of Contents

About the Author

Peter Szor is a world renowned computer virus and security researcher. He has been actively conducting research on computer viruses for more than 15 years, and he focused on the subject of computer viruses and virus protection in his diploma work in 1991. Over the years, Peter has been fortunate to work with the best-known antivirus products, such as AVP, F-PROT, and Symantec Norton AntiVirus. Originally, he built his own antivirus program, Pasteur, from 1990 to 1995, in Hungary. Parallel to his interest in computer antivirus development, Peter also has years of experience in fault-tolerant and secured financial transaction systems development.

He was invited to join the Computer Antivirus Researchers Organization (CARO) in 1997. Peter is on the advisory board of *Virus Bulletin* Magazine and a founding member of the AntiVirus Emergency Discussion (AVED) network. He has been with Symantec for over five years as a chief researcher in Santa Monica, California.

Peter has authored over 70 articles and papers on the subject of computer viruses and security for magazines such as *Virus Bulletin, Chip, Source, Windows NT Magazine,* and *Information Security Bulletin,* among others. He is a frequent speaker at conferences, including Virus Bulletin, EICAR, ICSA, and RSA and has given invited talks at such security conferences as the USENIX Security Symposium. Peter is passionate about sharing his research results and educating others about computer viruses and security issues.

Preface

Who Should Read This Book

Over the last two decades, several publications appeared on the subject of computer viruses, but only a few have been written by professionals ("insiders") of computer virus research. Although many books exist that discuss the computer virus problem, they usually target a novice audience and are simply not too interesting for the technical professionals. There are only a few works that have no worries going into the technical details, necessary to understand, to effectively defend against computer viruses.

Part of the problem is that existing books have little—if any—information about the current complexity of computer viruses. For example, they lack serious technical information on fast-spreading computer worms that exploit vulnerabilities to invade target systems, or they do not discuss recent code evolution techniques such as code metamorphism. If you wanted to get all the information I have in this book, you would need to spend a lot of time reading articles and papers that are often hidden somewhere deep inside computer virus and security conference proceedings, and perhaps you would need to dig into malicious code for years to extract the relevant details.

I believe that this book is most useful for IT and security professionals who fight against computer viruses on a daily basis. Nowadays, system administrators as well as individual home users often need to deal with computer worms and other malicious programs on their networks. Unfortunately, security courses have very little training on computer virus protection, and the general public knows very little about how to analyze and defend their network from such attacks. To make things more difficult, computer virus analysis techniques have not been discussed in any existing works in sufficient length before.

I also think that, for anybody interested in information security, being aware of what the computer virus writers have "achieved" so far is an important thing to know.

For years, computer virus researchers used to be "file" or "infected object" oriented. To the contrary, security professionals were excited about suspicious events only on the network level. In addition, threats such as CodeRed worm appeared to inject their code into the memory of vulnerable processes over the network, but did not "infect" objects on the disk. Today, it is important to understand all of these major perspectives—the file (storage), in-memory, and network views—and correlate the events using malicious code analysis techniques.

During the years, I have trained many computer virus and security analysts to effectively analyze and respond to malicious code threats. In this book, I have included information about anything that I ever had to deal with. For example, I have relevant examples of ancient threats, such as 8-bit viruses on the Commodore 64. You will see that techniques such as stealth technology appeared in the earliest computer viruses, and on a variety of platforms. Thus, you will be able to realize that current rootkits do not represent anything new! You will find sufficient coverage on 32-bit Windows worm threats with in-depth exploit discussions, as well as 64-bit viruses and "pocket monsters" on mobile devices. All along the way, my goal is to illustrate how old techniques "reincarnate" in new threats and demonstrate up-to-date attacks with just enough technical details.

I am sure that many of you are interested in joining the fight against malicious code, and perhaps, just like me, some of you will become inventors of defense techniques. All of you should, however, be aware of the pitfalls and the challenges of this field!

That is what this book is all about.

What I Cover

The purpose of this book is to demonstrate the current state of the art of computer virus and antivirus developments and to teach you the methodology of computer virus analysis and protection. I discuss infection techniques of computer viruses from all possible perspectives: file (on storage), in-memory, and network. I classify and tell you all about the dirty little tricks of computer viruses that bad guys developed over the last two decades and tell you what has been done to deal with complexities such as code polymorphism and exploits.

The easiest way to read this book is, well, to read it from chapter to chapter. However, some of the attack chapters have content that can be more relevant after understanding techniques presented in the defense chapters. If you feel that any of the chapters are not your taste, or are too difficult or lengthy, you can always jump to the next chapter. I am sure that everybody will find some parts of this book very difficult and other parts very simple, depending on individual experience.

I expect my readers to be familiar with technology and some level of programming. There are so many things discussed in this book that it is simply impossible to cover everything in sufficient length. However, you will know exactly what you might need to learn from elsewhere to be absolutely successful against malicious threats. To help you, I have created an extensive reference list for each chapter that leads you to the necessary background information.

Indeed, this book could easily have been over 1,000 pages. However, as you can tell, I am not Shakespeare. My knowledge of computer viruses is great, not my English. Most likely, you would have no benefit of my work if this were the other way around.

What I Do Not Cover

I do not cover Trojan horse programs or backdoors in great length. This book is primarily about self-replicating malicious code. There are plenty of great books available on regular malicious programs, but not on computer viruses.

I do not present any virus code in the book that you could directly use to build another virus. This book is not a "virus writing" class. My understanding, however, is that the bad guys already know about most of the techniques that I discuss in this book. So, the good guys need to learn more and start to think (but not act) like a real attacker to develop their defense!

Interestingly, many universities attempt to teach computer virus research courses by offering classes on writing viruses. Would it really help if a student could write a virus to infect millions of systems around the world? Will such students know more about how to develop defense better? Simply, the answer is no...

Instead, classes should focus on the analysis of existing malicious threats. There are so many threats out there waiting for somebody to understand them—and do something against them.

Of course, the knowledge of computer viruses is like the "Force" in *Star Wars*. Depending on the user of the "Force," the knowledge can turn to good or evil. I cannot force you to stay away from the "Dark Side," but I urge you to do so.

Acknowledgments

First, I would like to thank my wife Natalia for encouraging my work for over 15 years! I also thank her for accepting the lost time on all the weekends that we could have spent together while I was working on this book.

I would like to thank everybody who made this book possible. This book grew out of a series of articles and papers on computer viruses, several of which I have co-authored with other researchers over the years. Therefore, I could never adequately thank Eric Chien, Peter Ferrie, Bruce McCorkendale, and Frederic Perriot for their excellent contributions to Chapter 7 and Chapter 10.

This book could not be written without the help of many friends, great antivirus researchers, and colleagues. First and foremost, I would like to thank Dr. Vesselin Bontchev for educating me in the terminology of malicious programs for many years while we worked together. Vesselin is famous ("infamous?") for his religious accuracy in the subject matter, and he greatly influenced and supported my research.

A big thank you needs to go to the following people who encouraged me to write this book, educated me in the subject, and influenced my research over the years: Oliver Beke, Zoltan Hornak, Frans Veldman, Eugene Kaspersky, Istvan Farmosi, Jim Bates, Dr. Frederick Cohen, Fridrik Skulason, David Ferbrache, Dr. Klaus Brunnstein, Mikko Hypponen, Dr. Steve White, and Dr. Alan Solomon.

I owe a huge thanks to my technical reviewers: Dr. Vesselin Bontchev, Peter Ferrie, Nick FitzGerald, Halvar Flake, Mikko Hypponen, Dr. Jose Nazario, and Jason V. Miller. Your encouragements, criticisms, insights, and reviews of early handbook manuscripts were simply invaluable.

I need to thank Janos Kis and Zsolt Szoboszlay for providing me access to in-the-wild virus code for analysis, in the days when the BBS was the center of the computing universe. I also need to thank Gunter May for the greatest present that an east European kid could get—a C64.

A big thanks to everybody at Symantec, especially to Linda A. McCarthy and Vincent Weafer, who greatly encouraged me to write this book. I would also like to thank Nancy Conner and Chris Andry for their outstanding editorial work. Without their help, this project simply would never have finished. I also owe a huge thanks to Jessica Goldstein, Kristy Hart, and Christy Hackerd for helping me with the publishing process all the way.

A big thanks to all past and present members of the Computer Antivirus Researchers Organization (CARO), VFORUM, and the AntiVirus Emergency Discussion (AVED) List for all the exciting discussions on computer viruses and other malicious programs and defense systems.

I would like to thank everybody at *Virus Bulletin* for publishing my articles and papers internationally for almost a decade and for letting me use that material in this book.

Last but not least, I thank my teacher parents and grandparents for the extra "home education" in math, physics, music, and history.

Contact Information

If you find errors or have suggestions for clarification or material you would like to see in a future edition, I would love to hear from you. I am planning to introduce clarifications, possible corrections, and new information relevant to the content of this work on my Web site. While I think we have found most of the problems (especially in those paragraphs that were written late at night or between virus and security emergencies), I believe that no such work of this complexity and size can exist without some minor nits. Nonetheless, I made all the efforts to provide you with "trustworthy" information according to the best of my research knowledge.

Peter Szor,
Santa Monica, CA
`pszor@acm.org`
`http://www.peterszor.com`

This page intentionally left blank

PART I

STRATEGIES OF THE ATTACKER

This page intentionally left blank

CHAPTER 1

Introduction to the Games of Nature

"To me art is a desire to communicate."
—Endre Szasz

Computer virus research is a fascinating subject to many who are interested in nature, biology, or mathematics. Everyone who uses a computer will likely encounter some form of the increasingly common problem of computer viruses. In fact, some well-known computer virus researchers became interested in the field when, decades ago, their own systems were infected.

The title of Donald Knuth's book series[1], *The Art of Computer Programming,* suggests that anything we can explain to a computer is science, but that which we cannot currently explain to a computer is an art. Computer virus research is a rich, complex, multifaceted subject. It is about reverse engineering, developing detection, disinfection, and defense systems with optimized algorithms, so it naturally has scientific aspects; however, many of the analytical methods are an art of their own. This is why outsiders often find this relatively young field so hard to understand. Even after years of research and publications, many new analytical techniques are in the category of art and can only be learned at antivirus and security vendor companies or through the personal associations one must forge to succeed in this field.

This book attempts to provide an insider's view of this fascinating research. In the process, I hope to teach many facts that should interest both students of the art and information technology professionals. My goal is to provide an extended understanding of both the attackers and the systems built to defend against virulent, malicious programs.

Although there are many books about computer viruses, only a few have been written by people experienced enough in computer virus research to discuss the subject for a technically oriented audience.

The following sections discuss historical points in computation that are relevant to computer viruses and arrive at a practical definition of the term *computer virus*.

1.1 Early Models of Self-Replicating Structures

Humans create new models to represent our world from different perspectives. The idea of self-replicating systems that model self-replicating structures has been around since the Hungarian-American, Neumann János (John von Neumann), suggested it in 1948[2, 3, 4].

Von Neumann was a mathematician, an amazing thinker, and one of the greatest computer architects of all time. Today's computers are designed according to his original vision. Neumann's machines introduced memory for storing information and binary (versus analog) operations. According to von Neumann's brother

Nicholas, "Johnny" was very impressed with Bach's "Art of the Fugue" because it was written for several voices, with the instrumentation unspecified. Nicholas von Neumann credits the Bach piece as a source for the idea of the stored-program computer[5].

In the traditional von Neumann machine, there was no basic difference between code and data. Code was differentiated from data only when the operating system transferred control and executed the information stored there.

To create a more secure computing system, we will find that system operations that better control the differentiation of data from code are essential. However, we also will see the weaknesses of such approaches.

Modern computers can simulate nature using a variety of modeling techniques. Many computer simulations of nature manifest themselves as games. Modern computer viruses are somewhat different from these traditional nature-simulation game systems, but students of computer virus research can appreciate the utility of such games for gaining an understanding of self-replicating structures.

1.1.1 John von Neumann: Theory of Self-Reproducing Automata

Replication is an essential part of life. John von Neumann was the first to provide a model to describe nature's self-reproduction with the idea of self-building automata.

In von Neumann's vision, there were three main components in a system:

1. A Universal Machine
2. A Universal Constructor
3. Information on a Tape

A universal machine (Turing Machine) would read the memory tape and, using the information on the tape, it would be able to rebuild itself piece by piece using a universal constructor. The machine would not understand the process—it would simply follow the information (blueprint instructions) on the memory tape. The machine would only be able to select the next proper piece from the set of all the pieces by picking them one by one until the proper piece was found. When it was found, two proper pieces would be put together according to the instructions until the machine reproduced itself completely.

If the information that was necessary to rebuild another system could be found on the tape, then the automata was able to reproduce itself. The original automata would be rebuilt (Figure 1.1), and then the newly built automata was booted, which would start the same process.

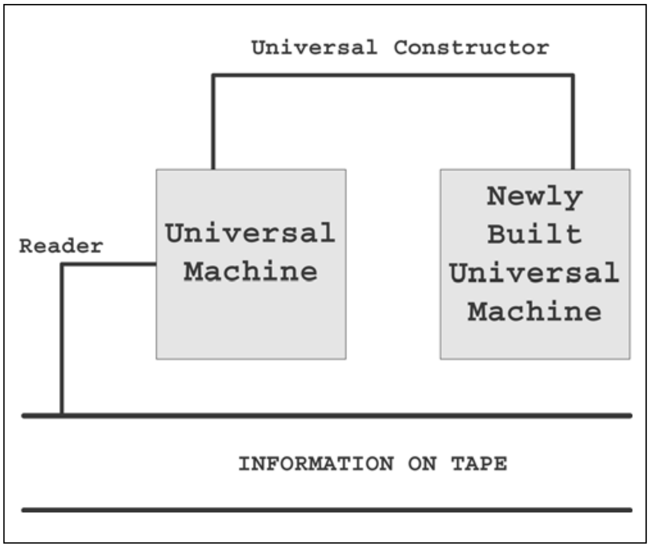


Figure 1.1 The model of a self-building machine.

A few years later, Stanislaw Ulam suggested to von Neumann to use the processes of cellular automation to describe this model. Instead of using "machine parts," states of cells were introduced. Because cells are operated in a robotic fashion according to rules ("code"), the cell is known as an *automaton*. The array of cells comprises the *cellular automata* (CA) computer architecture.

Von Neumann changed the original model using cells that had 29 different states in a two-dimensional, 5-cell environment. To create a self-reproducing structure, he used 200,000 cells. Neumann's model mathematically proved the possibility of self-reproducing structures: Regular non-living parts (molecules) could be combined to create self-reproducing structures (potentially living organisms).

In September 1948, von Neumann presented his vision of self-replicating automata systems. Only five years later, in 1953, Watson and Crick recognized that living organisms use the DNA molecule as a "tape" that provides the information for the reproduction system of living organisms.

Unfortunately, von Neumann could not see a proof of his work in his life, but his work was completed by Arthur Burks. Further work was accomplished by E.F. Codd in 1968. Codd simplified Neumann's model using cells that had eight states, 5-cell environments. Such simplification is the base for "self-replicating loops"[6] developed by artificial life researchers, such as Christopher G. Langton, in 1979. Such replication loops eliminate the complexity of universal machine from the system and focus on the needs of replication.

In 1980 at NASA/ASEE, Robert A. Freitas, Jr. and William B. Zachary[7] conducted research on a self-replicating, growing lunar factory. A lunar manufacturing facility (LMF) was researched, which used the theory of self-reproducing automata and existing automation technology to make a self-replicating, self-growing factory on the moon. Robert A. Freitas, Jr. and Ralph C. Merkle recently authored a book titled *Kinematic Self-Replicating Machines*. This book indicates a renewed scientific interest in the subject. A few years ago, Freitas introduced the term *ecophagy*, the theoretical consumption of the entire ecosystem by out of control, self-replicating nano-robots, and he proposed mitigation recommendations[8].

It is also interesting to note that the theme of self-replicating machines occurs repeatedly in works of science fiction, from movies such as *Terminator* to novels written by such authors as Neal Stephenson and William Gibson. And of course, there are many more examples from beyond the world of science fiction, as nanotech and microelectrical mechanical systems (MEMS) engineering have become real sciences.

1.1.2 Fredkin: Reproducing Structures

Several people attempted to simplify von Neumann's model. For instance, in 1961 Edward Fredkin used a specialized cellular automaton in which all the structures could reproduce themselves and replicate using simple patterns on a grid (see Figure 1.2 for a possible illustration). Fredkin's automata had the following rules[9]:

- On the table, we use the same kind of tokens.
- We either have a token or no token in each possible position.
- Token generations will follow each other in a finite time frame.
- The environment of each token will determine whether we will have a new token in the next generation.
- The environment is represented by the squares above, below, to the left, and to the right of the token (using the 5-cell-based von Neumann environment).
- The state of a square in the next generation will be empty when the token has an even number of tokens in its environment.
- The state of a square in the next generation will be filled with a token if it has an odd number of tokens in its environment.
- It is possible to change the number of states.

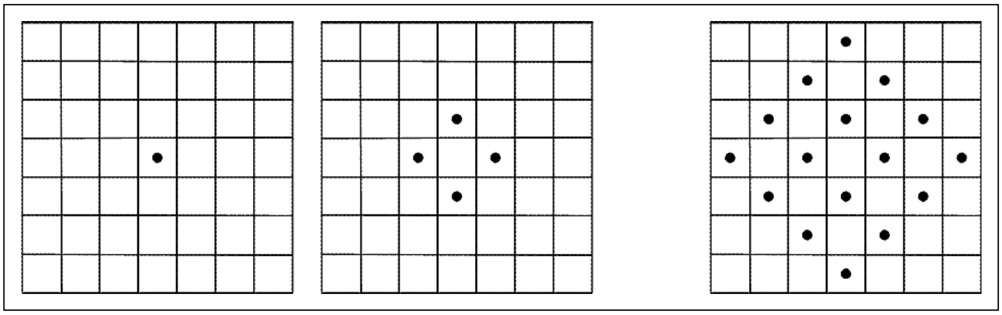


Figure 1.2 Generation 1, Generation 2, and...Generation 4.

Using the rules described previously with this initial layout allows all structures to replicate. Although there are far more interesting layouts to explore, this example is the simplest possible model of self-reproducing cellular automata.

1.1.3 Conway: Game of Life

In 1970, John Horton Conway[10] created one of the most interesting cellular automata systems. Just as the pioneer von Neumann did, Conway researched the interaction of simple elements under a common rule and found that this could lead to surprisingly interesting structures. Conway named his game *Life*. Life is based on the following rules:

- There should be no initial pattern for which there is a single proof that the population can grow without limit.
- There should be an initial pattern that apparently does grow without limit.
- There should be simple initial patterns that work according to simple genetic law: birth, survival, and death.

Figure 1.3 demonstrates a modern representation of the original Conway table game written by Edwin Martin[11].

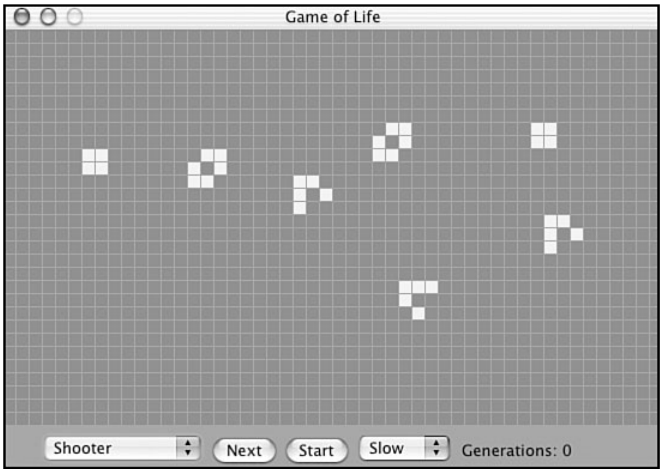


Figure 1.3 Edwin Martin's Game of Life implementation on the Mac using "Shooter" starting structure.

It is especially interesting to see the computer animation as the game develops with the so-called "Shooter" starting structure. In a few generations, two shooter positions that appear to shoot to each other will develop on the sides of the table, as shown in Figure 1.4, and in doing so they appear to produce so-called *gliders* that "fly" away (see Figure 1.5) toward the lower-right corner of the table. This sequence continues endlessly, and new gliders are produced.

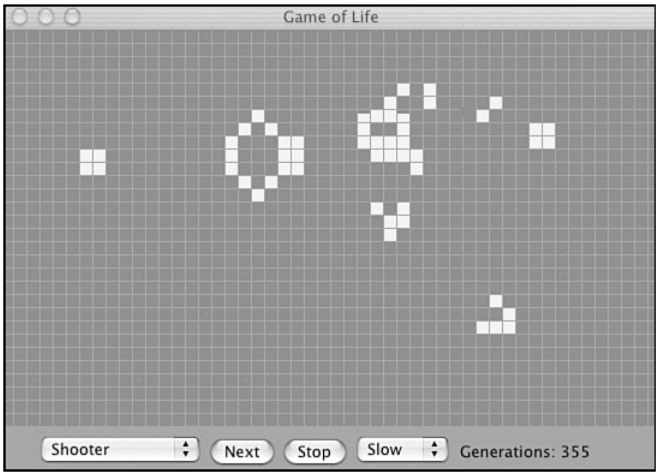


Figure 1.4 "Shooter" in Generation 355.

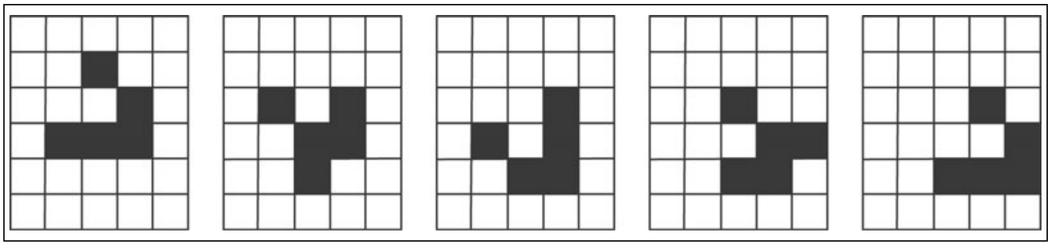


Figure 1.5 The glider moves around without changing shape.

On a two-dimensional table, each cell has two potential states: S=1 if there is one token in the cell, or S=0 if there is no token. Each cell will live according to the rules governed by the cell's environment (see Figure 1.6).

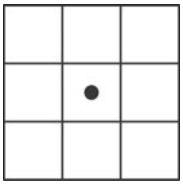


Figure 1.6 The 9-cell-based Moore environment.

The following characteristics/rules define Conway's game, Life:

Birth: If an empty cell has three (K=3) other filled cells in its environment, that particular cell will be filled in a new generation.

Survival: If a filled cell has two or three (K=2 or K=3) other filled cells in its environment, that particular cell will survive in the new generation.

Death: If a filled cell has only one or no other filled cells (K=1 or K=0) in its environment, that particular cell will die because of isolation. Further, if a cell has too many filled cells in its environment—four, five, six, seven, or eight (K=4, 5, 6, 7, or 8), that particular cell will also die in the next generation due to overpopulation.

Conway originally believed that there were no self-replicating structures in Life. He even offered $50 to anyone who could create a starting structure that would lead to self-replication. One such structure was quickly found using computers at the artificial intelligence group of the Massachusetts Institute of Technology (MIT).

MIT students found a structure that was later nicknamed a *glider*. When 13 gliders meet, they create a pulsing structure. Later, in the 100th generation, the pulsing structure suddenly "gives birth" to new gliders, which quickly "fly" away.

After this point, in each 30th subsequent generation, there will be a new glider on the table that flies away. This sequence continues endlessly. This setup is very similar to the "Shooter" structure shown in Figures 1.3 and 1.4.

Games with Computers, written by Antal Csakany and Ferenc Vajda in 1980, contains examples of competitive games. The authors described a table game with rules similar to those of Life. The table game uses cabbage, rabbits, and foxes to demonstrate struggles in nature. An initial cell is filled with cabbage as food for the rabbits, which becomes food for the foxes according to predefined rules. Then the rules control and balance the population of rabbits and foxes.

It is interesting to think about computers, computer viruses, and antiviral programs in terms of this model. Without computers (in particular, an operating system or BIOS of some sort), computer viruses are unable to replicate. Computer viruses infect new computer systems, and as they replicate, the viruses can be thought of as prey for antivirus programs.

In some situations, computer viruses fight back. These are called *retro viruses*. In such a situation, the antiviral application can be thought to "die." When an antiviral program stops an instance of a virus, the virus can be thought to "die." In some cases, the PC will "die" immediately as the virus infects it.

For example, if the virus indiscriminately deletes key operating system files, the system will crash, and the virus can be said to have "killed" its host. If this process happens too quickly, the virus might kill the host before having the opportunity to replicate to other systems. When we imagine millions of computers as a table game of this form, it is fascinating to see how computer virus and antiviral population models parallel those of the cabbage, rabbits, and foxes simulation game.

Rules, side effects, mutations, replication techniques, and degrees of virulence dictate the balance of such programs in a never-ending fight. At the same time, a "co-evolution"[12] exists between computer viruses and antivirus programs. As antivirus systems have become more sophisticated, so have computer viruses. This tendency has continued over the more than 30-year history of computer viruses.

Using models along these lines, we can see how the virus population varies according to the number of computers compatible with them. When it comes to computer viruses and antiviral programs, multiple parallel games occur side by side. Viruses within an environment that consists of a large number of compatible computers will be more virulent; that is, they will spread more rapidly to many more computers. A large number of similar PCs with compatible operating systems create a homogeneous environment–fertile ground for virulence (sound familiar?).

With smaller game boards representing a smaller number of compatible computers, we will obviously see smaller outbreaks, along with relatively small virus populations.

This sort of modeling clearly explains why we find major computer virus infections on operating systems such as Windows, which represents about 95% of the current PC population around us on a huge "grid." Of course this is not to say that 5% of computer systems are not enough to cause a global epidemic of some sort.

Note

If you are fascinated by self-replicating, self-repairing, and evolving structures, visit the BioWall project, `http://lslwww.epfl.ch/biowall/index.html`.

1.1.4 Core War: The Fighting Programs

Around 1966, Robert Morris, Sr., the future National Security Agency (NSA) chief scientist, decided to create a new game environment with two of his friends, Victor Vyssotsky and Dennis Ritchie, who coded the game and called it *Darwin*. (Morris, Jr. was the first infamous worm writer in the history of computer viruses. His mark on computer virus history will be discussed later in the book.)

The original version of Darwin was created for the PDP-1 (programmed data processing) at Bell Labs. Later, Darwin became *Core War*, a computer game that many programmers and mathematicians (as well as hackers) play to this day.

Note

I use the term *hacker* in its original, positive sense. I also believe that all good virus researchers are hackers in the traditional sense. I consider myself a hacker, too, but fundamentally different from malicious hackers who break into other people's computers.

The game is called Core War because the objective of the game is to kill your opponent's programs by overwriting them. The original game is played between two assembly programs written in the Redcode language. The Redcode programs run in the core of a simulated (for example, "virtual") machine named Memory Array Redcode Simulator (MARS). The actual fight between the warrior programs was referred to as Core Wars.

The original instruction set of Redcode consists of 10 simple instructions that allow movement of information from one memory location to another, which provides great flexibility in creating tricky warrior programs. Dewdney wrote several "Computer Recreations" articles in *Scientific American*[13,14] that discussed Core War, beginning with the May 1984 article. Figure 1.7 is a screen shot of a Core War implementation called PMARSV, written by Albert Ma, Na'ndor Sieben, Stefan Strack, and Mintardjo Wangsaw. It is interesting to watch as the little warriors fight each other within the MARS environment.

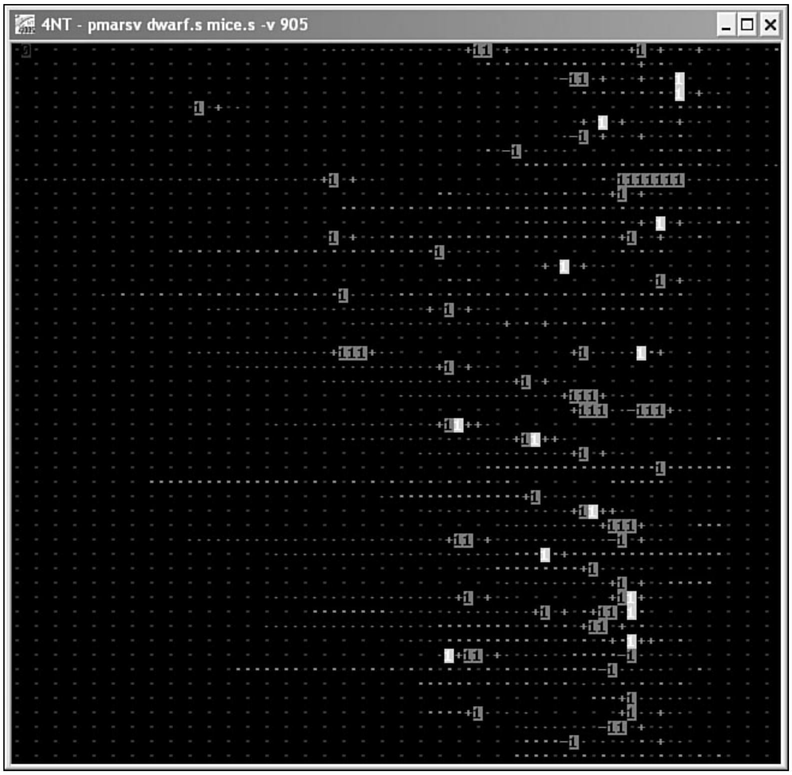


Figure 1.7 Core Wars warrior programs (Dwarf and MICE) in battle.

As programs fight in the annual tournaments, certain warriors might become the King of the Hill (KotH). These are the Redcode programs that outperform their competitors.

The warrior program named MICE won the first tournament. Its author, Chip Wendell, received a trophy that incorporated a core-memory board from an early CDC 6600 computer[14].

The simplest Redcode program consists of only one MOV instruction: MOV 0,1 (in the traditional syntax). This program is named IMP, which causes the contents at relative address 0 (namely the MOV, or move, instruction itself), to be transferred to relative address 1, just one address ahead of itself. After the instruction is copied to the new location, control is given to that address, executing the instruction, which, in turn, makes a new copy of itself at a higher address, and so on. This happens naturally, as instructions are executed following a higher address. The instruction counter will be incremented after each executed instruction.

The basic core consisted of two warrior programs and 8,000 cells for instructions. Newer revisions of the game can run multiple warriors at the same time. Warrior programs are limited to a specific starting size, normally 100 instructions. Each program has a finite number of iterations; by default, this number is 80,000.

The original version of Redcode supported 10 instructions. Later revisions contain more. For example, the following 14 instructions are used in the 1994 revision, shown in Listing 1.1.

Listing 1.1

Core War Instructions in the 1994 Revision

```
DAT  data
MOV  move
ADD  add
SUB  subtract
MUL  multiply
DIV  divide
MOD  modula
JMP  jump
JMZ  jump if zero
JMN  jump if not zero
DJN  decrement, jump if not zero
CMP  compare
SLT  skip if less than
SPL  split execution
```

Let's take a look at Dewdney's Dwarf tutorial (see Listing 1.2).

Listing 1.2

Dwarf Bombing Warrior Program

```
;name          Dwarf
;author        A. K. Dewdney
;version       94.1
```

```
;date          April 29, 1993
;strategy      Bombs every fourth instruction.

ORG      1 ; Indicates execution begins with the second
           ; instruction (ORG is not actually loaded, and is
           ; therefore not counted as an instruction).

DAT.F    #0, #0      ; Pointer to target instruction.
ADD.AB   #4, $-1     ; Increments pointer by 4.
MOV.AB   #0, @-2     ; Bombs target instruction.
JMP.A    $-2, #0     ; Loops back two instructions.
```

Dwarf follows a so-called bombing strategy. The first few lines are comments indicating the name of the warrior program and its Redcode 1994 standard. Dwarf attempts to destroy its opponents by "dropping" DAT bombs into their operation paths. Because any warrior process that attempts to execute a DAT statement dies in the MARS, Dwarf will be a likely winner when it hits its opponents.

The MOV instruction is used to move information into MARS cells. (The IMP warrior explains this very clearly.) The general format of a Redcode command is of the Opcode A, B form. Thus, the command MOV.AB #0, @-2 will point to the DAT statement in Dwarf's code as a source.

The A field points to the DAT statement, as each instruction has an equivalent size of 1, and at 0, we find DAT #0, #0. Thus, MOV will copy the DAT instruction to where B points. So where does B point to now?

The B field points to DAT.F #0, #0 statement in it. Ordinarily, this would mean that the bomb would be put on top of this statement, but the @ symbol makes this an indirect pointer. In effect, the @ symbol says to use the contents of the location to where the B field points as a new pointer (destination). In this case, the B field appears to point to a value of 0 (location 0, where the DAT.F instruction is placed).

The first instruction to execute before the MOV, however, is an ADD instruction. When this ADD #4, $-1 is executed, the DAT's offset field will be incremented by four each time it is executed—the first time, it will be changed from 0 to 4, the next time from 4 to 8, and so on.

This is why, when the MOV command copies a DAT bomb, it will land four lines (locations) above the DAT statement (see Listing 1.3).

Listing 1.3

Dwarf's Code When the First Bomb Is Dropped

```
0       DAT.F #0, #8
1 ->    ADD.AB 4, $-1
2       MOV.AB #0, @-2 ; launcher
```

continues

Listing 1.3 continued

Dwarf's Code When the First Bomb Is Dropped

```
3       JMP.A $-2, #0
4       DAT ; Bomb 1
5       .
6       .
7       .
8       DAT ; Bomb 2
9       .
```

The JMP.A $-2 instruction transfers control back relative to the current offset, that is, back to the ADD instruction to run the Dwarf program "endlessly." Dwarf will continue to bomb into the core at every four locations until the pointers wrap around the core and return. (After the highest number possible for the DAT location has been reached, it will "wrap" back around past 0. For example, if the highest possible value were 10, 10+1 would be 0, and 10+4 would be 3.)

At that point, Dwarf begins to bomb over its own bombs, until the end of 80,000 cycles/iterations or until another warrior acts upon it. At any time, another warrior program might easily kill Dwarf because Dwarf stays at a constant location—so that it can avoid hitting itself with friendly fire. But in doing so, it exposes itself to attackers.

There are several common strategies in Core War, including scanning, replicating, bombing, IMP-spiral (those using the SPL instruction), and the interesting bomber variation named the *vampire*.

Dewdney also pointed out that programs can even steal their enemy warrior's very soul by hijacking a warrior execution flow. These are the so-called vampire warriors, which bomb JMP (JUMP) instructions into the core. By bombing with jumps, the enemy program's control can be hijacked to point to a new, predefined location where the hijacked warrior will typically execute useless code. Useless code will "burn" the cycles of the enemy warrior's execution threads, thus giving the vampire warrior an advantage.

Instead of writing computer viruses, I strongly recommend playing this harmless and interesting game. In fact, if worms fascinate you, a new version of Core War can be created to link battles in different networks and allow warrior programs to jump from one battle to another to fight new enemies on those machines. Evolving the game to be more networked allows for simulating worm-like warrior programs.

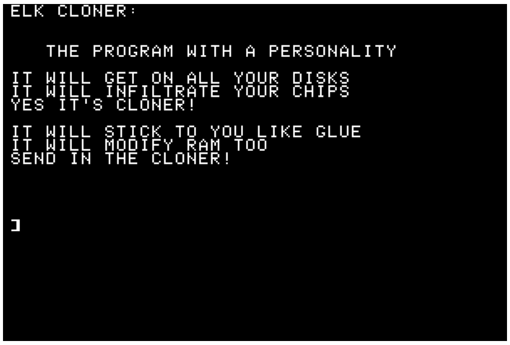
1.2 Genesis of Computer Viruses

Virus-like programs appeared on microcomputers in the 1980s. However, two fairly recounted precursors deserve mention here: Creeper from 1971-72 and John Walker's "infective" version of the popular ANIMAL game for UNIVAC[15] in 1975.

Creeper and its nemesis, Reaper, the first "antivirus" for networked TENEX running on PDP-10s at BBN, was born while they were doing the early development of what became "the Internet."

Even more interestingly, ANIMAL was created on a UNIVAC 1100/42 mainframe computer running under the Univac 1100 series operating system, Exec-8. In January of 1975, John Walker (later founder of Autodesk, Inc. and co-author of AutoCAD) created a general subroutine called PERVADE[16], which could be called by any program. When PERVADE was called by ANIMAL, it looked around for all accessible directories and made a copy of its caller program, ANIMAL in this case, to each directory to which the user had access. Programs used to be exchanged relatively slowly, on tapes at the time, but still, within a month, ANIMAL appeared at a number of places.

The first viruses on microcomputers were written on the Apple-II, circa 1982. Rich Skrenta[17], who was a ninth-grade student at the time in Pittsburgh, Pennsylvania, wrote "Elk Cloner." He did not think the program would work well, but he coded it nonetheless. His friends found the program quite entertaining—unlike his math teacher, whose computer became infected with it. Elk Cloner had a payload that displayed Skrenta's poem after every 50th use of the infected disk when reset was pressed (see Figure 1.8). On every 50th boot, Elk Cloner hooked the reset handler; thus, only pressing reset triggered the payload of the virus.



```
ELK CLONER:

   THE PROGRAM WITH A PERSONALITY

IT WILL GET ON ALL YOUR DISKS
IT WILL INFILTRATE YOUR CHIPS
YES IT'S CLONER!

IT WILL STICK TO YOU LIKE GLUE
IT WILL MODIFY RAM TOO
SEND IN THE CLONER!


]
```

Figure 1.8 Elk Cloner activates.

Not surprisingly, the friendship of the two ended shortly after the incident. Skrenta also wrote computer games and many useful programs at the time, and he still finds it amazing that he is best known for the "stupidest hack" he ever coded.

In 1982, two researchers at Xerox PARC[18] performed other early studies with computer worms. At that time, the term *computer virus* was not used to describe these programs. In 1984, mathematician Dr. Frederick Cohen[19] introduced this term, thereby becoming the "father" of computer viruses with his early studies of them. Cohen introduced *computer virus* based on the recommendation of his advisor, Professor Leonard Adleman[20], who picked the name from science fiction novels.

1.3 Automated Replicating Code: The Theory and Definition of Computer Viruses

Cohen provided a formal mathematical model for computer viruses in 1984. This model used a Turing machine. In fact, Cohen's formal mathematical model for a computer virus is similar to Neumann's self-replicating cellular automata model. We could say, that in the Neumann sense, a computer virus is a self-reproducing cellular automata. The mathematical model does not have much practical use for today's researcher. It is a rather general description of what a computer virus is. However, the mathematical model provides significant theoretical foundation to the computer virus problem.

Here is Cohen's informal definition of a computer virus: *"A virus is a program that is able to infect other programs by modifying them to include a possibly evolved copy of itself."*

This definition provides the important properties of a computer virus, such as the possibility of evolution (the capability to make a modified copy of the same code with mutations). However, it might also be a bit misleading if applied in its strictest sense.

This is, by no means, to criticize Cohen's groundbreaking model. It is difficult to provide a precise definition because there are so many different kinds of computer viruses nowadays. For instance, some forms of computer viruses, called *companion viruses*, do not necessarily modify the code of other programs. They do not strictly follow Cohen's definition because they do not need to include a copy of themselves within other programs. Instead, they make devious use of the program's environment—properties of the operating system—by placing themselves with the same name ahead of their victim programs on the execution path. This can create a problem for behavior-blocking programs that attempt to block malicious actions of other programs—if the authors of such blockers strictly apply Cohen's informal definition. In other words, if such blocking programs are looking only for viruses that make unwanted changes to the code of another program, they will miss companion viruses.

Note

Cohen's mathematical formulation properly encompasses companion viruses; it is only the literal interpretation of the single-sentence human language definition that is problematic. A single-sentence linguistic definition of viruses is difficult to come up with.

Integrity checker programs also rely on the fact that one program's code remains unchanged over time. Such programs rely on a database (created at some initial point in time) assumed to represent a "clean" state of the programs on a machine. Integrity checker programs were Cohen's favorite defense method and my own in the early '90s. However, it is easy to see that the integrity checker would be challenged by companion viruses unless the integrity checker also alerted the user about any new application on the system. Cohen's own system properly performed this. Unfortunately, the general public does not like to be bothered each time a new program is introduced on their systems, but Cohen's approach is definitely the safest technique to use.

Dr. Cohen's definition does not differentiate between programs explicitly designed to copy themselves (the "real viruses" as we call them) from the programs that can copy themselves as a side effect of the fact that they are general-purpose copying programs (compilers and so on).

Indeed, in the real world, behavior-blocking defense systems often alarm in such a situation. For instance, Norton Commander, the popular command shell, might be used to copy the commander's own code to another hard drive or network resource. This action might be confused with self-replicating code, especially if the folder in which the copy is made has a previous version of the program that we overwrite to upgrade it. Though such "false alarms" are easily dealt with, they will undoubtedly annoy end users.

Taking these points into consideration, a more accurate definition of a computer virus would be the following: *"A computer virus is a program that recursively and explicitly copies a possibly evolved version of itself."*

There is no need to specify how the copy is made, and there is no strict need to "infect" or otherwise modify another application or host program. However, most computer viruses do indeed modify another program's code to take control. Blocking such an action, then, considerably reduces the possibility for viruses to spread on the system.

As a result, there is always a host, an operating system, or another kind of execution environment, such as an interpreter, in which a particular sequence of symbols behaves as a computer virus and replicates itself recursively.

Computer viruses are self-automated programs that, against the user's wishes, make copies of themselves to spread themselves to new targets. Although particular computer viruses ask the user with prompts before they infect a machine, such as, "Do you want to infect another program? (Y/N?)," this does not make them non-viruses. Often, novice researchers in computer virus labs believe otherwise, and they actually argue that such programs are not viruses. Obviously, they are wrong!

When attempting to classify a particular program as a virus, we need to ask the important question of whether a program is able to replicate itself recursively and explicitly. A program cannot be considered a computer virus if it needs any help to make a copy of itself. This help might include modifying the environment of such a program (for example, manually changing bytes in memory or on a disk) or—heaven forbid—applying a hot fix to the intended virus code itself using a debugger! Instead, nonworking viruses should be classified as *intended viruses*.

The copy in question does not have to be an exact clone of the initial instance. Modern computer viruses, especially so-called metamorphic viruses (further discussed in Chapter 7, "Advanced Code Evolution Techniques and Computer Virus Generator Kits"), can rewrite their own code in such a way that the starting sequence of bytes responsible for the copy of such code will look completely different in subsequent generations but will perform the equivalent or similar functionality.

References

1. Donald E. Knuth, *The Art of Computer Programming*, 2nd Edition, Addison-Wesley, Reading, MA, 1973, 1968, ISBN: 0-201-03809-9 (Hardcover).

2. John von Neumann, "The General and Logical Theory of Automata," *Hixon Symposium*, 1948.

3. John von Neumann, "Theory and Organization of Complicated Automata," *Lectures at the University of Illinois*, 1949.

4. John von Neumann, "The Theory of Automata: Contruction, Reproduction, Homogenity," *Unfinished manuscript*, 1953.

5. William Poundstone, *Prisoner's Dilemma*, Doubleday, New York, ISBN: 0-385-41580-X (Paperback), 1992.

6. Eli Bachmutsky, "Self-Replication Loops in Cellular Space," `http://necsi.org:16080/postdocs/sayama/sdrs/java`.

7. Robert A. Freitas, Jr. and William B. Zachary, "A Self-Replicating, Growing Lunar Factory," *Fifth Princeton/AIAA Conference*, May 1981.

8. Robert A. Freitas, Jr., "Some Limits to Global Ecophagy by Biovorous Nanoreplicators, with Public Policy Recommendations," `http://www.foresight.org/nanorev/ecophagy.html`.

9. György Marx, *A Természet Játékai*, Ifjúsági Lap és Könyvterjesztô Vállalat, Hungary, 1982, ISBN: 963-422-674-4 (Hardcover).

10. Martin Gardner, "Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'Life,'" *Scientific American*, October 1970, pp. 120–123.

11. Edwin Martin, "John Conway's Game of Life," `http://www.bitstorm.org/gameoflife` (Java version is available).

12. Carey Nachenberg, "Computer Virus-Antivirus Coevolution," *Communications of the ACM*, January 1997, Vol. 40, No. 1., pp. 46-51.

13. Dewdney, A. K., *The Armchair Universe: An Exploration of Computer Worlds*, New York: W. H. Freeman (c), 1988, ISBN: 0-7167-1939-8 (Paperback).

14. Dewdney, A. K., *The Magic Machine: A Handbook of Computer Sorcery*, New York: W. H. Freeman (c), 1990, ISBN: 0-7167-2125-2 (Hardcover), 0-7167-2144-9 (Paperback).

15. John Walker, "ANIMAL," `http://fourmilab.ch/documents/univac/animal.html`.

16. John Walker, "PERVADE," `http://fourmillab.ch/documents/univac/pervade.html`.

17. Rich Skrenta, `http://www.skrenta.com`.

18. John Shock and Jon Hepps, "The Worm Programs, Early Experience with a Distributed Computation," ACM, Volume 25, 1982, pp. 172–180.

19. Dr. Frederick B. Cohen, *A Short Course on Computer Viruses*, Wiley Professonal Computing, New York, 2nd edition, 1994, ISBN: 0471007684 (Paperback).

20. Vesselin Vladimirov Bontchev, "Methodology of Computer Anti-Virus Research," *University of Hamburg Dissertation*, 1998.

This page intentionally left blank

CHAPTER 2

The Fascination of Malicious Code Analysis

"*The Lion looked at Alice wearily. 'Are you animal—or vegetable—or mineral?' he said, yawning at every other word.*"

—Lewis Carroll (1832–1898), *Through the Looking-Glass and What Alice Found There* (1871).

For people who are interested in nature, it is difficult to find a subject more fascinating than computer viruses. Computer virus analysis can be extremely difficult for most people at first glance. However, the difficulty depends on the actual virus code in question. Binary forms of viruses, those compiled to object code, must be reverse-engineered to understand them in detail. This process can be challenging for an individual, but it provides a great deal of knowledge about computer systems.

My own interest in computer viruses began in September of 1990, when my new PC clone displayed a bizarre message, followed by two beeps. The message read

"Your PC is now Stoned!"

I had heard about computer viruses before, but this was my first experience with one of these incredible nuisances. Considering that my PC was two weeks old at the time, I was fascinated by how quickly I encountered a virus on it. I had introduced the Stoned boot virus with an infected diskette, which contained a copy of a popular game named Jbird. A friend had given me the game. Obviously he did not know about the hidden "extras" stored on the diskette.

I did not have antivirus software at the time, of course, and because this incident happened on a Saturday, help was not readily available. The PC clone had cost me five months' worth of my summer salary, so you can imagine my disappointment!

I was worried that I was going to lose all the data on my system. I remembered an incident that had happened to a friend in 1988: His PC was infected with a virus, causing characters to fall randomly down his computer screen; after a while, he could not do anything with the machine. He had told me that he needed to format the drive and reinstall all the programs.

Later, we learned that a strain of the Cascade virus had infected his computer. Cascade could have been removed from his system without formatting the hard drive, but he did not know that at the time. Unfortunately, as a result, he lost all his data. Of course I wanted to do the exact opposite on my machine—remove the virus without losing my data.

To find the Stoned virus, I first searched the files on the infected diskette for the text that was displayed on the screen. I was not lucky enough to find any files that contained it. If I had had more experience in hunting viruses at the time, I might have considered the possibility that the virus was encrypted in a file. But this virus was not encrypted, and my instinct about a non–file system hiding place was heading in the right direction.

This gave me the idea that the virus was not stored in the files but instead was located somewhere else on the diskette. I had Peter Norton's book, *Programmer's Guide to the IBM PC*, on-hand. Up to this point, I had only read a few pages of it, but luckily the book described how the boot sector of diskettes could be accessed using a standard DOS tool called DEBUG.

After some hesitation, I finally executed the DEBUG command for the first time to try to look into the boot sector of the diskette, which was inserted in drive A. The command was the following:

```
DEBUG
-L 100 0 0 1
```

This command instructs DEBUG to load the first sector (the boot sector) from drive A: to memory at offset 100 hexadecimal. When I used the dump (D) command of DEBUG to display the loaded sector's content, I saw the virus's message, as well as some other text.

```
-d280
1437:0280 03 33 DB FE C1 CD 13 EB-C5 07 59 6F 75 72 20 50 .3........Your P
1437:0290 43 20 69 73 20 6E 6F 77-20 53 74 6F 6E 65 64 21 C is now Stoned!
1437:02A0 07 0D 0A 0A 00 4C 45 47-41 4C 49 53 45 20 4D 41 .....LEGALISE MA
1437:02B0 52 49 4A 55 41 4E 41 21-00 00 00 00 00 00 00 00 RIJUANA!........
```

You can imagine how excited I was to find the virus. Finally, it was right there in front of me! I spent the weekend reading more of the Norton book because I did not understand the virus's code at all. I simply did not know IBM PC Assembly language at the time, which was required to understand the code. There were so many things to learn!

The Norton book introduced me to a substantial amount of the information I needed to begin. For example, it provided detailed and superb descriptions of the boot process, disk structures, and various interrupts of the DOS and basic input-output system (BIOS) routines.

I spent a few days analyzing Stoned on paper and commenting every single Assembly instruction until I understood everything. It took me almost a full week to absorb all the information, but, sadly, my computer was still infected with the virus.

After a few more days of work, I created a detection program, then a disinfection program for the virus, which I wrote in Turbo Pascal. The disinfection program was able to remove the virus from all over: from the system memory as well as from the boot and Master boot sectors in which the virus was stored.

A couple of days later, I visited the university with my virus detector and found that the virus had infected more than half of the PC labs' machines. I was amazed at how successfully this simple virus code could invade machines around the world. I could not fathom how the virus had traveled all the way from New Zealand where, I learned later, it had been released in early 1988, to Hungary to infect my system.

The Stoned virus was in the wild. (IBM researcher, Dave Chess, coined the term *in the wild* to describe computer viruses that were encountered on production systems. Not all viruses are in the wild. The viruses that only collectors or researchers have seen are named *zoo* viruses.)

People welcomed the help, and I was happy because I wanted to assist them and learn more about virus hunting. I started to collect viruses from friends and wrote disinfection programs for them. Viruses such as Cascade, Vacsina, Yankee_Doodle, Vienna, Invader, Tequila, and Dark_Avenger were among the first set that I analyzed in detail, and I wrote detection and disinfection code for them one by one.

Eventually, my work culminated in a diploma, and my antivirus program became a popular shareware in Hungary. I named my program Pasteur after the French microbiologist Louis Pasteur.

All my efforts and experiences opened up a career for me in antivirus research and development. This book is designed to share my knowledge of computer virus research.

2.1 Common Patterns of Virus Research

Computer virus analysis has some common patterns that can be learned easily, lending efficiency to the analysis process. There are several techniques that computer virus researchers use to reach their ultimate goal, which is to acquire a precise understanding of viral programs in a timely manner to provide appropriate prevention and to respond so that computer virus outbreaks can be controlled.

Virus researchers also need to identify and understand particular vulnerabilities and malicious code that exploits them. Vulnerability and exploit research has its own common patterns and techniques. Some of these are similar to the methods of computer virus research, but many key differences exist.

This book will introduce these useful techniques to teach you how to deal with viral programs more efficiently. Along the way, you will learn how to analyze a computer virus more effectively and safely by using disassemblers, debuggers, emulators, virtual machines, file dumpers, goat files, dedicated virus replication machines and systems, virus test networks, decryption tools, unpackers, and many other useful tools. You can use this information to deal with computer virus problems more effectively on a daily basis.

You also will learn how computer viruses are classified and named, as well as a great deal about state-of-the-art computer virus tricks.

Computer virus source code is not discussed in this book. Discussions on this topic are unethical and in some countries, illegal[1]. More importantly, writing even a dozen viruses would not make you an expert on this subject.

Some virus writers[2] believe that they are experts because they created a single piece of code that replicates itself. This assumption could not be further from the truth. Although some virus writers might be very knowledgeable individuals, most of them are not experts on the subject of computer viruses. The masterminds who arguably at various times represented the state of the art in computer virus writing go (or went) by aliases such as Dark Avenger[3], Vecna, Jacky Qwerty, Murkry, Sandman, Quantum, Spanska, GriYo, Zombie, roy g biv, and Mental Driller.

2.2 Antivirus Defense Development

Initially, developing antivirus software programs was not difficult. In the late '80s and early '90s, many individuals were able to create some sort of antivirus program against a particular form of a computer virus.

Frederick Cohen proved that antivirus programs cannot solve the computer virus problem because there is no way to create a single program that can detect all future computer viruses in finite time. Regardless of this proven fact, antivirus programs have been quite successful in dealing with the problem for a while. At the same time, other solutions have been researched and developed, but computer antivirus programs are still the most widely used defenses against computer viruses at present, regardless of their many drawbacks, including the inability to contend with and solve the aforementioned problem.

Perhaps under the delusion that they are experts on computer viruses, some security analysts state that any sort of antivirus program is useless if it cannot find all the new viruses. However, the reality is that without antivirus programs, the Internet would be brought to a standstill because of the traffic undetected computer viruses would generate.

Often we do not completely understand how to protect ourselves against viruses, but neither do we know how to reduce the risk of becoming infected by them by adopting proper hygiene habits. Unfortunately, negligence is one of the biggest contributors to the spread of computer viruses. The sociological aspects of computer security appear to be more relevant than technology. Carelessly neglecting the most minimal level of computer maintenance, network security configuration, and failing to clean an infected computer opens up a Pandora's box that allows more problems to spread to other computers.

In the early phases of virus detection and removal, computer viruses were easily managed because very few viruses existed (there were fewer than 100 known strains in 1990). Computer virus researchers could spend weeks analyzing a single virus alone. To make life even easier, computer viruses spread slowly, compared to the rapid proliferation of today's viruses. For example, many successful boot viruses were 512 bytes long (the size of the boot sector on the IBM PC), and they often took a year or longer to travel from one country to another. Consider this: The spread time at which a computer virus traveled in the past compared to today's virus spread time is analogous to comparing the speed of message transfer in ancient times, when messengers walked or ran from city to city to deliver parcels, with today's instant message transfer, via e-mail, with or without attachments.

Finding a virus in the boot sector was easy for those who knew what a boot sector was; writing a program to recognize the infection was tricky. Manually disinfecting an infected system was a true challenge in and of itself, so creating a program that automatically removed viruses from computers was considered a tremendous achievement. Currently, the development of antivirus and security defense systems is deemed an art form, which lends itself to cultivating and developing a plethora of useful skills. However, natural curiosity, dedication, hard work, and the continuous desire to learn often supersede mere hobbyist curiosity and are thus essential to becoming a master of this artistic and creative vocation.

2.3 Terminology of Malicious Programs

The need to define a unified nomenclature for malicious programs is almost as old as computer viruses themselves[4]. Obviously, each classification has a common pitfall because classes will always appear to overlap, and classes often represent closely related subclasses of each other.

2.3.1 Viruses

As defined in Chapter 1, "Introduction to the Games of Nature," a computer virus is code[5] that recursively replicates a possibly evolved copy of itself. Viruses infect a host file or system area, or they simply modify a reference to such objects to take control and then multiply again to form new generations.

2.3.2 Worms

Worms are network viruses, primarily replicating on networks. Usually a worm will execute itself automatically on a remote machine without any extra help from a user. However, there are worms, such as mailer or mass-mailer worms, that will not always automatically execute themselves without the help of a user.

Worms are typically standalone applications without a host program. However, some worms, like W32/Nimda.A@mm, also spread as a file-infector virus and infect host programs, which is precisely why the easiest way to approach and contain worms is to consider them a special subclass of virus. If the primary vector of the virus is the network, it should be classified as a worm.

2.3.2.1 Mailers and Mass-Mailer Worms

Mailers and mass-mailer worms comprise a special class of computer worms, which send themselves in an e-mail. Mass-mailers, often referred to as "@mm" worms such as VBS/Loveletter.A@mm, send multiple e-mails including a copy of themselves once the virus is invoked.

Mailers will send themselves less frequently. For instance, a mailer such as W32/SKA.A@m (also known as the Happy99 worm) sends a copy of itself every time the user sends a new message.

2.3.2.2 Octopus

An octopus is a sophisticated kind of computer worm that exists as a set of programs on more than one computer on a network.

For example, head and tail copies are installed on individual computers that communicate with each other to perform a function. An octopus is not currently a common type of computer worm but will likely become more prevalent in the future. (Interestingly, the idea of the octopus comes from the science fiction novel *Shockwave Rider* by John Brunner. In the story, the main character, Nickie, is on the run and uses various identities. Nickie is a phone phreak, and he uses a "tapeworm," similar to an octopus, to erase his previous identities.)

2.3.2.3 Rabbits

A rabbit is a special computer worm that exists as a single copy of itself at any point in time as it "jumps around" on networked hosts. Other researchers use the term rabbit to describe crafty, malicious applications that usually run themselves recursively to fill memory with their own copies and to slow down processing time by consuming CPU time. Such malicious code uses too much memory and thus can cause serious side effects on a machine within other applications that are not prepared to work under low-memory conditions and that unexpectedly cease functioning.

2.3.3 Logic Bombs

A logic bomb is a programmed malfunction of a legitimate application. An application, for example, might delete itself from the disk after a couple of runs as a copy protection scheme; a programmer might want to include some extra code to perform a malicious action on certain systems when the application is used. These scenarios are realistic when dealing with large projects driven by limited code-reviews.

An example of a logic bomb can be found in the original version of the popular Mosquitos game on Nokia Series 60 phones. This game has a built-in function to send a message using the Short Message Service (SMS) to premium rate lines. The functionality was built into the first version of the game as a software distribution and piracy protection scheme, but it backfired[6]. When legitimate users complained to the software vendor, the routine was eliminated from the code of the game. The premium lines have been "disconnected" as well. However, the pirated versions of the game are still in circulation, which have the logic bomb inside and send regular SMS messages. The game used four premium SMS phone numbers such as 4636, 9222, 33333, and 87140, which corresponded to four countries. For example, the number 87140 corresponded to the UK. When the game used this number, it sent the text "king.001151183" as short message. In turn, the user of the game was charged a hefty £1.5 per message.

Often extra functionality is hidden as resources in the application—and remains hidden. In fact, the way in which these functions are built into an application is similar to the way so-called Easter eggs are making headway into large projects. Programmers create Easter eggs to hide some extra credit pages for team members who have worked on a project.

Applications such as those in the Microsoft Office suite have many Easter eggs hidden within them, and other major software vendors have had similar credit pages embedded within their programs as well. Although Easter eggs are not malicious and do not threaten end users (even though they might consume extra space on the hard drive), logic bombs are always malicious.

2.3.4 Trojan Horses

Perhaps the simplest kind of malicious program is a Trojan horse. Trojan horses try to appeal to and interest the user with some useful functionality to entice the user to run the program. In other cases, malicious hackers leave behind Trojanized versions of real tools to camouflage their activities on a computer, so they can retrace their steps to the compromised system and perform malicious activities later.

For example, on UNIX-based systems, hackers often leave a modified version of "ps" (a tool to display a process list) to hide a particular process ID (PID), which can relate to another backdoor Trojan's process. Later on, it might be difficult to find such changes on a compromised system. These kinds of Trojans are often called user mode rootkits.

The attacker can easily manipulate the tool by modifying the source code of the original tool at a certain location. At first glance, this minor modification is extremely difficult to locate.

Probably the most famous Trojan horse is the AIDS TROJAN DISK[7] that was sent to about 7,000 research organizations on a diskette. When the Trojan was introduced on the system, it scrambled the name of all files (except a few) and filled the empty areas of the disk completely. The program offered a recovery solution in exchange of a bounty. Thus, malicious cryptography was born. The author of the Trojan horse was captured shortly after the incident. Dr. Joseph Popp, 39 at the time, a zoologist from Cleveland, Ohio was prosecuted in the UK[8].

The filename scrambling function of AIDS TROJAN DISK was based on two substitution tables[9]. One was used to encrypt the filenames and another to encrypt the file extensions. At some point in the history of cryptography[10], such an algorithm was considered unbreakable[11]. However, it is easy to see that substitution ciphers can be easily attacked based on the use of statistical methods (the distribution of common words). In addition, if given enough time, the defender can disassemble the Trojan's code and pick the tables from its code.

There are two kinds of Trojans:

- One hundred percent Trojan code, which is easy to analyze.
- A careful modification of an original application with some extra functionality, some of which belong to backdoor or rootkit subclasses. This kind of Trojan is more common on open source systems because the attacker can easily insert backdoor functionality to existing code.

Note

The source code of Windows NT and Windows 2000 got into circulation in early 2004. It is expected that backdoor and rootkit programs will be created using these sources.

2.3.4.1 Backdoors (Trapdoors)

A backdoor is the malicious hacker's tool of choice that allows remote connections to systems. A typical backdoor opens a network port (UDP/TCP) on the host when it is executed. Then, the listening backdoor waits for a remote connection from the attacker and allows the attacker to connect to the system. This is the most common type of backdoor functionality, which is often mixed with other Trojan-like features.

Another kind of backdoor relates to a program design flaw. Some applications, such as the early implementation of SMTP (simple mail transfer protocol) allowed features to run a command (for example, for debugging purposes). The Morris Internet worm uses such a command to execute itself remotely, with the command placed as the recipient of the message on such vulnerable installations. Fortunately, this command was quickly removed once the Morris worm exploited it. However, there can be many applications, especially newer ones, that allow for similar insecure features.

2.3.4.2 Password-Stealing Trojans

Password-stealing Trojans are a special subclass of Trojans. This class of malicious program is used to capture and send a password to an attacker. As a result, an attacker can return to the vulnerable system and take whatever he or she wants. Password stealers are often combined with keyloggers to capture keystrokes when the password is typed at logon.

2.3.5 Germs

Germs are first-generation viruses in a form that the virus cannot generate to its usual infection processes. Usually, when the virus is compiled for the first time, it exists in a special form and normally does not have a host program attached to it. Germs will not have the usual marks that most viruses use in second-generation form to flag infected files to avoid reinfecting an already infected object.

A germ of an encrypted or polymorphic virus is usually not encrypted but is plain, readable code. Detecting germs might need to be done differently from detecting second, and later, -generation infections.

2.3.6 Exploits

Exploit code is specific to a single vulnerability or set of vulnerabilities. Its goal is to run a program on a (possibly remote, networked) system automatically or provide some other form of more highly privileged access to the target system. Often, a single attacker builds exploit code and shares it with others. "White hat" hackers create a form of exploit code for penetration (or "pen") testing. Therefore, depending on the actual use of the exploit, the exploitation might be malicious in some cases but harmless in others—the severity of the threat depends on the intention of the attacker.

2.3.7 Downloaders

A downloader is yet another malicious program that installs a set of other items on a machine that is under attack. Usually, a downloader is sent in e-mail, and when it is executed (sometimes aided with the help of an exploit), it downloads malicious content from a Web site or other location and then extracts and runs its content.

2.3.8 Dialers

Dialers got their relatively early start during the heyday of dial-up connections to bulletin board systems (BBSs) in homes. The concept driving a dialer is to make money for the people behind the dialer by having its users (often unwitting victims) call via premium-rate phone numbers. Thus, the person who runs the dialer might know the intent of the application, but the user is not aware of the charges. A common form of dialer is the so-called porn dialer.

Similar approaches exist on the World Wide Web using links to Web pages that connect to paid services.

2.3.9 Droppers

The original term refers to an "installer" for first-generation virus code. For example, boot viruses that first exist as compiled files in binary form are often installed in the boot sector of a floppy using a dropper. The dropper writes the germ code to the boot sector of the diskette. Then the virus can replicate on its own without ever generating the dropper form again.

When the virus regenerates the dropper form, the intermediate form is part of an infection cycle, which is not to be confused with a dedicated (or pure) dropper.

2.3.10 Injectors

Injectors are special kinds of droppers that usually install virus code in memory. An injector can be used to inject virus code in an active form on a disk interrupt handler. Then, the first time a user accesses a diskette, the virus begins to replicate itself normally.

A special kind of injector is the network injector. Attackers also can use legitimate utilities, such as NetCat (NC), to inject code into the network. Usually, a remote target is specified, and the datagram is sent to the machine that will be attacked using the injector. An attacker initially introduced the CodeRed worm using an injector; subsequently, the worm replicated as data on the network without ever hitting the disk again as a file.

Injectors are often used in a process called *seeding*. Seeding is a process that is used to inject virus code to several remote systems to cause an initial outbreak that is large enough to cause a quick epidemic. For example, there is supporting digital evidence that W32/Witty worm[12] was seeded to several systems by its author.

2.3.11 Auto-Rooters

Auto-rooters are usually malicious hacker tools used to break into new machines remotely. Auto-rooters typically use a collection of exploits that they execute against a specified target to "gain root" on the machine. As a result, a malicious hacker (typically a so-called script-kiddie) gains administrative privileges to the remote machine.

2.3.12 Kits (Virus Generators)

Virus writers developed kits, such as the Virus Creation Laboratory (VCL) or PSMPC generators, to generate new computer viruses automatically, using a menu-based application. With such tools, even novice users were able to develop harmful computer viruses without too much background knowledge. Some virus generators exist to create DOS, macro, script, or even Win32 viruses and mass-mailing worms. As discussed in Chapter 7 "Advanced Code Evolution Techniques and Computer Virus Generator Kits," the so-called "Anna Kournikova" virus (technically VBS/VBSWG.J) was created by a Dutch teenager, Jan de Wit, from the VBSWG kit—sadly, de Wit got lucky and the kit, infamous for churning out mainly broken, intended code produced a working virus. De Wit was subsequently arrested, convicted, and sentenced for his role in this.

2.3.13 Spammer Programs

> Vikings: *Spam spam spam spam*
>
> Waitress: *...spam spam spam egg and spam; spam spam spam spam spam baked beans spam spam spam...*
>
> Vikings: *Spam! Lovely spam! Lovely spam!*
>
> —Monty Python Spam Song

Spammer programs are used to send unsolicited messages to Instant Messaging groups, newsgroups, or any other kind of mobile device in forms of e-mail or cell phone SMS messages.

Two lawyers helped to make spam an international, albeit notorious, superstar of the worldwide Internet virus scene. Their main objective was to send advertisements to Internet newsgroups. Spam mail has become the number one Internet nuisance for the global community. Many e-mail users complain that their inbox is littered with more than 70% spam each day. This ratio has been on the rise for the last couple of years.

The primary motivation of spammers is to make money by generating traffic to Web sites. In addition, spam messages are often used to implement phishing attacks. For example, you might receive an e-mail message asking you to visit your bank's Web site and telling you that if you don't, they will disable your account. There is a link in the e-mail, however, that forwards you to the fraudster. If you fall victim to the attack, you might disclose personal information to the attacker on a silver plate. The fraudster wants to get your credit card number, account number, password, PIN (personal identification number), and other personal information to make money. In addition, you might become the prime subject of an identity theft as well.

2.3.14 Flooders

Malicious hackers use flooders to attack networked computer systems with an extra load of network traffic to carry out a denial of service (DoS) attack. When the DoS attack is performed simultaneously from many compromised systems (so-called zombie machines), the attack is called a distributed denial of service (DDoS) attack. Of course, there are much more sophisticated DoS attacks including SYN floods, packet fragmentation attacks, and other (mis-)sequencing attacks, traffic amplification, or traffic deflection, just to name the most common types.

2.3.15 Keyloggers

A keylogger captures keystrokes on a compromised system, collecting sensitive information for the attacker. Such sensitive information might include names, passwords, PINs, birthdays, Social Security numbers, or credit card numbers. The keylogger is installed on the system. Unbeknownst to the user, a computer could be compromised for weeks before the attack is ever noticed. Attackers often use keyloggers to commit identity theft.

2.3.16 Rootkits

Rootkits are a special set of hacker tools that are used after the attacker has broken into a computer system and gained root-level access. Usually, hackers break into a system with exploits and install modified versions of common tools. Such rootkits are called *user-mode rootkits* because the Trojanized application runs in user mode.

Some more sophisticated rootkits, such as Adore[13], have kernel-mode module components. These rootkits are more dangerous because they change the behavior of the kernel. Thus, they can hide objects from even kernel-level defense software. For example, they can hide processes, files in the file system, registry keys, and values under Windows, and implement stealth capabilities for other malicious components. In contrast, user-mode rootkits cannot typically hide themselves effectively from kernel-level defense software. User-mode rootkits only manipulate with user-mode objects; therefore, defense systems relying on kernel objects have chance to reveal the truth.

2.4 Other Categories

Some other categories of commonly encountered Internet pests are not necessarily malicious in their primary intent. However, they can be a nuisance to end users; therefore, antivirus and antispam products have been created to detect and remove such annoying burdens from computers.

2.4.1 Joke Programs

Joke programs are not malicious; however, as Alan Solomon (author of one of the most widely used scanning engines today) once mentioned, "Whether a program should be classified as a joke program or as a Trojan largely depends on the sense of humor of the victim." Joke programs change or interrupt the normal behavior of your computer, creating a general distraction or nuisance. Colleagues often make fun of each other by installing a joke program or by tricking others to run one on their systems. A typical example of a joke program is a screen saver that randomly locks the system.

However, such programs can be considered harmful in some cases. Consider, for example, a joke program that locks the system but never unlocks it. Thus, computers cannot be stopped safely. As a result, important data could be lost because it was never saved to the disk. Or worse, the file allocation table could get corrupted, and the machine would become unbootable.

2.4.2 Hoaxes: Chain Letters

On computers, hoaxes typically spread information about computer virus infections and ask the recipient of the message to forward it to others. One of the most infamous hoaxes was the Good Times hoax. Good Times appeared in 1994 and warned users about a potential new kind of virus that would arrive in e-mail. The hoax claimed that reading a message with "Good Times" in the subject line would erase data from the hard disk. Although many believed at the time that such an e-mail based virus was a hoax, the reality is that such a payload might be possible. Hoaxes typically mix some reality with lies. Good Times claimed that a particular virus existed, which was simply not true.

End users then spread the e-mail hoax to new people, "replicating" the message on the Internet by themselves and overloading e-mail systems with the hoax. At larger corporations, policies must be implemented to avoid the spread of hoaxes on local systems.

In the past, a typical hoax circulating at large corporations tried to deceive people into believing an untrue story about a very sick child, attempting to collect money for the child's medical procedure. Most people were sympathetic and did not recognize the danger of forwarding the e-mail message in this case; they trusted the source and believed the fabricated story.

With company policies intact, the problems that such hoaxes create can be effectively eliminated. However, hoaxes are considered one of the most successful Internet threats every year; take for example, the new chain letters that surface and rapidly spread around the world.

2.4.3 Other Pests: Adware and Spyware

A new type of application has appeared recently as a direct result of increased residential Internet access. Many companies are interested in what people look for or research on the Web, especially what kinds of products consumers might buy. Therefore, some consumer retail businesses install little applications to collect information and display customized advertisements in pop-up messages.

The most obvious problem with this type of application is that such applications were not written with malicious intent. In fact, many programmers make a living out of writing such tools. However, many of these Internet pests get installed on a system without the user's permission or knowledge, raising questions about privacy. Not surprisingly, corporations as well as home users dislike this type of program, referred to as *spyware*, which collects various information of user activity and then sends these data to a company via the Internet. Home users are undoubtedly disturbed by this invasive activity, not to mention the frustration that users feel in response to pop-ups.

In addition, these programs are often very poorly written and are resource hogs, particularly when two or more become installed on the same machine. Many also have the highly undesirable habit of lowering Internet Explorer's already deplorable security settings to unconscionable levels, opening the (usually unwitting) "victim" up to even worse exploits and infections[14].

Because these applications are often a major source of business for organizations driven by consumer revenue, such businesses prefer that antivirus products not detect such programs at all, or at least not by default. Often such companies bring lawsuits against vendors who produce software to detect and remove their "applications." Such litigation makes the fight against this kind of pest much more difficult.

It is expected, however, that such programs will be illegal to create in several countries in the future. To make things even more interesting, some corporations prefer to remove "unwanted" spyware but want to keep the few "tools" that they use to monitor their employees on a regular basis.

2.5 Computer Malware Naming Scheme

Back in 1991, founding members of CARO (Computer Antivirus Researchers Organization) designed a computer virus naming scheme[15] for use in antivirus (AV) products. Today, the CARO naming scheme is slightly outdated compared to daily practice, but it remains the only standard that most antivirus companies ever attempted to adopt. An up-to-date version of the document is in the works and is expected to be published by CARO soon at `www.caro.org`. In this short section, I can only show you a 10,000-foot view of malware naming. I strongly recommend Nick FitzGerald's *AVAR 2002* conference paper[16], which greatly expands on further naming considerations. Furthermore, credit must be given to all the respected antivirus researchers of CARO.

Note

The original naming scheme was designed by Dr. Alan Solomon, Fridrik Skulason, and Dr. Vesselin Bontchev.

Virus naming is a challenging task. Unfortunately, there has been a major increase in widespread, fast-running computer virus outbreaks. Nowadays, antivirus researchers must add detections of 500, 1000, 1500, or even more threats to their products each month. Thus, the problem of naming computer viruses, even by the same common name, is getting to be a hard, if not impossible, task to manage. Nonetheless, representatives of antivirus companies still try to reduce the confusion by using a common name for at least the in-the-wild computer malware. However, computer virus outbreaks are on the rise, and researchers do not have the time to agree on a common name for each in-the-wild virus in advance of deploying response definitions. Even more commonly, it is very difficult to predict which viruses will be seen in the wild and which will remain zoo viruses.

Most people remember textual family names better than the naked IDs that many other naming schemes have adopted in the security space. Let's take a look at malware naming in its most complex form:

```
<malware_type>://<platform>/<family_name>.<group_name>.<infective_length>.
➥<variant><devolution><modifiers>
```

In practice, very little, if any, malware requires all name components. Practically anything other than the family name is an optional field:

```
[<malware_type>://][<platform>/]<family_name>[.<group_name>]
➥[.<infective_length>][.<variant>[<devolution>]][<modifiers>]
```

The following sections give a short description of each naming component.

2.5.1 `<family_name>`

This is the key component of any malware name. The basic rule set for the family name follows:

- Do not use company names, brand names, or the names of living people.
- Do not use an existing family name unless the virus belongs to the same family.
- Do not use obscene or offensive names.
- Do not use another name if a name already exists for the family. Use a tool, such as VGrep, to check name cross-references for older malware.
- Do not use numeric family names.
- Avoid the malware writer's suggested or intended name.
- Avoid naming malware after a file that traditionally or conventionally contains the malware.
- Avoid family names such as Friday_13th, particularly if the dates represent payload triggers.
- Avoid geographic names that are based on the discovery site.
- If multiple acceptable names exist, select the original one, the one used by the majority of existing antivirus programs, or the most descriptive one.

2.5.2 `<malware_type>://`

This part of the name indicates whether a malware type is a virus, Trojan, dropper, intended, kit, or garbage type (Virus://, Trojan://, .. ,Garbage://). Several products have extended this set slightly, and these are expected to become part of the standard malware naming in the future.

2.5.3 `<platform>/`

The platform prefix indicates the minimum native environment for the malware type that is required for it to function correctly. An annotated list of officially recognized platform names is listed in the next section.

Note

Multiple platform names can be defined for the same threat, for example, `virus://{W32,W97M}/Beast.41472.A`[17]. This name indicates a file-infecting virus called Beast that can infect on Win32 platforms and also is able to infect Word 97 documents.

2.5.4 `.<group_name>`

The group name represents a major family of computer viruses that are similar to each other. The group name is rarely used nowadays. It was mostly used to group DOS viruses.

2.5.5 `<infective_length>`

The infective length is used to distinguish parasitic viruses within a family or group based on their typical infective length in bytes.

2.5.6 `<variant>`

The subvariant represents minor variants of the same virus family with the same infective length.

2.5.7 `[<devolution>]`

The devolution identifier is used most commonly with the subvariant name in the case of macro viruses. Some macro viruses have a common ability (mostly related to programming mistakes) to create a subset of their original macro set during their natural replication cycle. Thus, the subset of macros cannot regenerate the original, complete macro set but is still able to recursively replicate from the partial set.

2.5.8 `<modifiers>`

The original intent of the modifier was to identify the polymorphic engine of a computer virus. However, most antivirus developers never used this modifier in practice. Nowadays, modifiers include the following optional components:

```
[[:<locale_specifier>][#<packer>][@'m'¦'mm'][!<vendor-specific_comment>]]
```

2.5.9 `:<locale_specifier>`

This specifier is used mostly for macro viruses that depend on a particular language version of their environment, such as Word. For example, virus://WM/Concept.B:Fr is a virus that affects only the French version of Microsoft Word.

2.5.10 `#<packer>`

The packer modifier is rarely used in practice. It can indicate that a computer malware was packed with a particular "on-the-fly" extractor unpacker, such as UPX.

2.5.11 `@m or @mm`

These symbols indicate self-mailer or mass-mailer computer viruses. Suggested by Bontchev, this is probably the most widely recognized modifier. This modifier highlights computer viruses that are more likely to be encountered by the general public because of the way the viruses use e-mail to propagate themselves.

2.5.12 `!<vendor-specific_comment>`

The vendor-specific modifier is a recent addition to the set of modifiers. Vendors are allowed to postfix any malware name with such a modifier. For example, a vendor might want to indicate that a virus is multipartite by using !mp in the name.

2.6 Annotated List of Officially Recognized Platform Names

The platform names shown in Table 2.1 are the only officially recognized identifiers following the proposed naming standard. A platform name that does not appear on this list cannot be used as a platform identifier in a malware name following this standard. The Comments column helps to explain some of the finer points of platform name selection. This is intended to be an authoritative list at this book's publication date. The platform list will need to be extended in the future.

Table 2.1

Officially Recognized Platform Names

Short Form	Long Form	Comments
ABAP	ABAP	Malware for the SAP /R3 Advanced Business Application Programming environment.
ALS	ACADLispScript	Malware that requires AutoCAD Lisp Interpreter.
BAT	BAT	Malware that requires a DOS or Windows command shell interpreter or close clone.
BeOS	BeOS	Requires BeOS.
Boot	Boot	Requires MBR and/or system boot sector of IBM PC–compatible hard drive and/or floppy. (Rarely used in practice.)
DOS	DOS	Infects DOS COM and/or EXE (MZ) and/or SYS format files and requires some version of MS-DOS or a closely compatible OS. (Rarely used in practice.)
EPOC	EPOC	Requires the EPOC OS up to version 5.
SymbOS	SymbianOS	Requires Symbian OS (EPOC version 6 and later).
Java	Java	Requires a Java run-time environment (standalone or browser-embedded).
MacOS	MacOS	Requires a Macintosh OS prior to OS X.
MeOS	MenuetOS	Requires MenuetOS.
MSIL	MSIL	Requires the Microsoft Intermediate Language runtime.
Mul	Multi	This is a pseudo-platform, and its use is reserved for a few very special cases.
PalmOS	PalmOS	Requires a version of PalmOS.
OS2	OS2	Requires OS/2.
OSX	OSX	Requires Macintosh OS X or a subsequent, essentially similar version.
W16	Win16	Requires one of the 16-bit Windows x86 OSes. (**Note:** Several products use the Win prefix.)
W95	Win95	Requires Windows 9x VxD services.
W32	Win32	Requires a 32-bit Windows (Windows 9x, Me, NT, 2000, XP on x86).
W64	Win64	Requires Windows 64.
WinCE	WinCE	Requires WinCE.
WM	WordMacro	Macro malware for WordBasic as included in WinWord 6.0, Word 95, and Word for Mac 5.x.

continues

Table 2.1 continued

Officially Recognized Platform Names

Short Form	Long Form	Comments
W2M	Word2Macro	Macro malware for WordBasic as included in WinWord 2.0.
W97M	Word97Macro	Macro malware for Visual Basic for Applications (VBA) v5.0 for Word (that shipped in Word 97) or later. Changes in VBA between Word 97 and 2003 versions (inclusive) are sufficiently slight that we do not distinguish platforms even if the malware makes a version check or uses one of the few VBA features added in versions subsequent to VBA v5.0.
AM	AccessMacro	Macro malware for AccessBasic.
A97M	Access97Macro	Macro malware for Visual Basic for Applications (VBA) v5.0 for Access that shipped in Access 97 and later. As for W97M, changes in VBA versions between Access 97 and 2003 (inclusive) are insufficient to justify distinguishing the platforms.
P98M	Project98Macro	Macro malware for Visual Basic for Applications (VBA) v5.0 for Project that shipped in Project 98 and later. As for W97M, changes in VBA versions between Project 98 and 2003 (inclusive) are insufficient to justify distinguishing the platforms.
PP97M	PowerPoint97Macro	Macro malware for Visual Basic for Applications (VBA) v5.0 for Project, which shipped in Project 97 and later. As for W97M, changes in VBA between Project 97 and 2002 inclusive are insufficient to justify distinguishing the platforms.
V5M	Visio5Macro	Macro malware for Visual Basic for Applications (VBA) v5.0 for Visio that shipped in Visio 5.0 and later. As for W97M, changes in VBA versions between Visio 5.0 and 2002 inclusive are insufficient to justify distinguishing the platforms.
XF	ExcelFormula	Malware based on Excel Formula language that has shipped in Excel since the very early days.
XM	ExcelMacro	Macro malware for Visual Basic for Applications (VBA) v3.0 that shipped in Excel for Windows 5.0 and Excel for Mac 5.x.
X97M	Excel97Macro	Macro malware for Visual Basic for Applications (VBA) v5.0 for Excel that shipped in Excel 97 and later. As for W97M, changes in VBA versions between Excel 97 and 2002 (inclusive) are insufficient to justify distinguishing the platforms.

Short Form	**Long Form**	**Comments**
O97M	Office97Macro	This is a pseudo-platform name reserved for macro malware that infects across at least two applications within the Office 97 and later suites. Cross-infectors between Office applications and related products, such as Project or Visio, can also be labeled thus.
AC14M	AutoCAD14Macro	VBA v5.0 macro viruses for AutoCAD r14 and later. As with W97M malware, minor differences in later versions of VBA are insufficient to justify new plat form names.
ActnS	ActionScript	Requires the Macromedia ActionScript interpreter found in some ShockWave Flash (and possibly other) animation players.
AplS	AppleScript	Requires AppleScript interpreter.
APM	AmiProMacro	Macro malware for AmiPro.
CSC	CorelScript	Malware that requires the CorelScript interpreter shipped in many Corel products.
HLP	WinHelpScript	Requires the script interpreter of the WinHelp display engine.
INF	INFScript	Requires one of the Windows INF (installer) script interpreters.
JS	JScript, JavaScript	Requires a JScript and/or JavaScript interpreter. Hosting does not affect the platform designator—standalone JS malware that requires MS JS under WSH, HTML-embedded JS malware, and JS malware embedded in Windows-compiled HTML help files (.CHM) all fall under this platform type.
MIRC	mIRCScript	Requires the mIRC script interpreter.
MPB	MapBasic	Requires MapBasic of MapInfo product.
Perl	Perl	Requires a Perl interpreter. Hosting does not affect the platform designator—standalone Perl infectors under UNIX(-like) shells, ones that require Perl under WSH and HTML-embedded Perl malware all fall under this platform type.
PHP	PHPScript	Requires a PHP script interpreter.
Pirch	PirchScript	Requires the Pirch script interpreter.
PS	PostScript	Requires a PostScript interpreter.
REG	Registry	Requires a Windows Registry file (.REG) interpreter. (We do not distinguish .REG versions or ASCII versus Unicode.)

continues

Table 2.1 continued

Officially Recognized Platform Names		
Short Form	**Long Form**	**Comments**
SH	ShellScript	Requires a UNIX(-like) shell interpreter. Hosting does not affect the platform name—shell malware specific to Linux, Solaris, HP-UX, or other systems, or specific to csh, ksh, bash, or other interpreters currently all fall under this platform type.
VBS	VBScript, VisualBasicScript	Requires a VBS interpreter. Hosting does not affect the platform designator—standalone VBS infectors that require VBS under WSH, HTML-embedded VBS malware, and malware embedded in Windows-compiled HTML help files (.CHM) all fall under this platform type.
UNIX	UNIX	This is a common name for binary viruses on UNIX platforms. (More specific platform names are available.)
BSD	BSD	Used for malware specific to BSD (-derived) platforms.
Linux	Linux	Used for malware specific to Linux platforms and others closely based on it.
Solaris	Solaris	Used for Solaris-specific malware.

References

1. Joe Hirst, “Virus Research and Social Responsibility,” *Virus Bulletin*, October 1989, page 3.
2. Sarah Gordon, “The Generic Virus Writer,” *Virus Bulletin Conference*, 1994.
3. Vesselin Bontchev, “The Bulgarian and Soviet Virus Writing Factories,” *Virus Bulletin Conference*, 1991, pp. 11-25.
4. Dr. Keith Jackson, “Nomenclature for Malicious Programs,” *Virus Bulletin*, March, 1990, page 13.
5. Vesselin Bontchev, “Are ‘Good’ Computer Viruses Still a Bad Idea?,” *EICAR*, 1994, pp. 25-47.
6. Jamo Niemela, “Mquito,” `http://www.f-secure.com/v-descs/mquito.shtml`.
7. Jim Bates, “Trojan Horse: AIDS Information Introductory Diskette Version 2.0,” *Virus Bulletin*, January 1990, page 3.
8. Mark Hamilton, “U.S. Judge Rules In Favour Of Extradition,” *Virus Bulletin*, January, 1991.