Computer Viruses

Theory and Experiments

Fred Cohen
Dept of Computer Science and Electric Engineering, Lehigh University, Bethlehem PA 18215, USA, and The Foundation for Computer Integrity Research, Pittsburgh, PA 15217 USA

This paper introduces "computer viruses" and examines their potential for causing widespread damage to computer systems. Basic theoretical results are presented, and the infeasibility of viral defense in large classes of systems is shown Defensive schemes are presented and several experiments are described.

Keywords: Computer Viruses, System Integrity, Data Integrity

Fred Cohen received a B.S. in Electrical Engineering from Carnegie-Mellon University in 1977, an MS in Information Science from the University of Pittsburgh in 1981 and a Ph.D. in Electrical Engineering from the University of Southern California in 1986
He has worked as a freelance consultant since 1977, and has designed and implemented numerous devices and systems. He is currently a professor of Computer Science and Electrical Engineering at Lehigh University, Chairman and Director of Engineering at the Foundation for Computer Integrity Research, and President of Legal Software Incorporated.
He is a member of the ACM, IEEE, and IACR. His current research interests include computer viruses, information flow model, adaptive systems theory, genetic models of computing, and evolutionary systems

North-Holland
Computers & Security 6 (1987) 22–35

1. Introduction

This paper defines a major computer security problem called a virus. The virus is interesting because of its ability to attach itself to other programs and cause them to become viruses as well. Given the widespread use of sharing in current computer systems, the threat of a virus carrying a Trojan horse [1,20] is significant. Although a considerable amount of work has been done in implementing policies to protect against the illicit dissemination of information [4,7], and many systems have been implemented to provide protection from this sort of attack [12,19,21,22], little work has been done in the area of keeping information entering an area from causing damage [5,18] There are many types of information paths possible in systems, some legitimate and authorized, and others that may be covert [18], the most commonly ignored one being through the user We will ignore covert information paths throughout this paper.

The general facilities exist for providing provably correct protection schemes [9], but they depend on a security policy that is effective against the types of attacks being carried out. Even some quite simple protection systems cannot be proven 'safe' [14]. Protection from denial of services requires the detection of halting programs which is well known to be undecidable [11]. The problem of precisely marking information flow within a system [10] has been shown to be NP-complete. The use of guards for the passing of untrustworthy information [25] between users has been examined, but in general depends on the ability to prove program correctness which is well known to be NP-complete.

The Xerox worm program [23] has demonstrated the ability to propagate through a network, and has even accidentally caused denial of services. In a later variation, the game of 'core wars' [8] was invented to allow two programs to do battle with one another Other variations on this theme have been reported by many unpublished authors, mostly in the context of nighttime games played between programmers. The term virus has also been used in conjunction with an augmentation to

0167-4048/87/$3.50 © 1987, Elsevier Science Publishers B.V. (North-Holland)

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

APL in which the author places a generic call at the beginning of each function which in turn invokes a preprocessor to augment the default APL interpreter [13].

The potential threat of a widespread security problem has been examined [15] and the potential damage to government, financial, business, and academic institutions is extreme. In addition, these institutions tend to use ad hoc protection mechanisms in response to specific threats rather than sound theoretical techniques [16]. Current military protection systems depend to a large degree on isolationism [3]; however, new systems are being developed to allow 'multilevel' usage [17]. None of the published proposed systems defines or implements a policy which could stop a virus.

In this paper, we open the new problem of protection from computer viruses. First we examine the infection property of a virus and show that the transitive closure of shared information could potentially become infected. When used in conjunction with a Trojan horse, it is clear that this could cause widespread denial of services and/or unauthorized manipulation of data. The results of several experiments with computer viruses are used to demonstrate that viruses are a formidable threat in both normal and high security operating systems. The paths of sharing, transitivity of information flow, and generality of information interpretation are identified as the key properties in the protection from computer viruses, and a case by case analysis of these properties is shown. Analysis shows that the only systems with potential for protection from a viral attack are systems with limited transitivity and limited sharing, systems with no sharing, and systems without general interpretation of information (Turing capability). Only the first case appears to be of practical interest to current society. In general, detection of a virus is shown to be undecidable both by a-priori and runtime analysis, and without detection, cure is likely to be difficult or impossible.

Several proposed countermeasures are examined and shown to correspond to special cases of the case by case analysis of viral properties. Limited transitivity systems are considered hopeful, but it is shown that precise implementation is intractable, and imprecise policies are shown in general to lead to less and less usable systems with time. The use of system-wide viral antibodies is examined, and shown to depend in general on the solutions to intractable problems.

It is concluded that the the study of computer viruses is an important research area with potential applications to other fields, that current systems offer little or no protection from viral attack, and that the only provably 'safe' policy as of this time is isolationism.

2. A Computer Virus

We define a computer 'virus' as a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself. With the infection property, a virus can spread throughout a computer system or network using the authorizations of every user using it to infect their programs. Every program that gets infected may also act as a virus and thus the infection grows.

The following pseudo-program shows how a virus might be written in a pseudo-computer language. The ':=' symbol is used for definition, the ':' symbol labels a statement, the ';' separates statements, the '=' symbol is used for assignment or comparison, the '~' symbol stands for not, the '{'and'}' symbols group sequences of statements together, and the '...' symbol is used to indicate that an irrelevant portion of code has been left implicit.

This example virus (V) (Fig. 1) searches for an uninfected executable file (E) by looking for executable files without the "1234567" in the beginning, and prepends V to E, turning it into an infected file (I). V then checks to see if some

```
program virus :=
{1234567;

subroutine infect-executable :=
  {loop: file = random-executable;
  if first-line-of-file = 1234567
        then goto loop;
  prepend virus to file;
  }

subroutine do-damage :=
  {whatever damage is desired}

subroutine trigger-pulled :=
  {return true on desired conditions}

main-program :=
  {infect-executable;
  if trigger-pulled then do-damage;
  goto next;
  }

next:}
```

Fig. 1. Simple virus 'V'.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

triggering condition is true, and does damage. Finally, V executes the rest of the program it was prepended [1] to. When the user attempts to execute E, I is executed in its place; it infects another file and then executes as if it were E. With the exception of a slight delay for infection, I appears to be E until the triggering condition causes damage. We note that viruses need not prepend themselves nor must they be restricted to a single infection per use.

A common misconception of a virus relates it to programs that simply propagate through networks. The worm program, 'core wars,' and other similar programs have done this, but none of them actually involve infection. The key property of a virus is its ability to infect other programs, thus reaching the transitive closure of sharing between users. As an example, if V infected one of user A's executables (E), and user B then ran E, V could spread to user B's files as well.

It should be pointed out that a virus need not be used for evil purposes or be a Trojan horse. As an example, a compression virus could be written to find uninfected executables, compress them upon the user's permission, and prepend itself to them. Upon execution, the infected program decompresses itself and executes normally. Since it always asks permission before performing services, it is not a Trojan horse, but since it has the infection property, it is still a virus. Studies indicate that such a virus could save over 50% of the space taken up by executable files in an average system. The performance of infected programs would decrease slightly as they are decompressed, and thus the compression virus implements a particular time space tradeoff. A sample compression virus could be written as in Fig. 2.

```
program compression-virus :=
{01234567;

subroutine infect-executable :=
  {loop: file = random-executable;
  if first-line-of-file = 01234567
        then goto loop;
  compress file;
  prepend compression-virus to file;
  }

main-program :=
  {if ask-permission
        then infect-executable;
  uncompress the-rest-of-this-file
        into tmpfile;
  run tmpfile;
  }
}
```

Fig. 2. Compression virus 'C'.

This program (C) finds an uninfected executable (E), compresses it, and prepends C to form an infected executable (I). It then uncompresses the rest of itself into a temporary file and executes normally. When I is run, it will seek out and compress another executable before decompressing E into a temporary file and executing it. The effect is to spread through the system compressing executable files, decompressing them as they are to be executed. Users will experience significant delays as their executables are decompressed before being run.

As a more threatening example, let us suppose that we modify the program V by specifying trigger-pulled as true after a given date and time, and specifying do-damage as an infinite loop. With the level of sharing in most modern systems, the entire system would likely become unusable as of the specified date and time. A great deal of work might be required to undo the damage of such a virus. This modification is shown in Fig. 3.

As an analogy to a computer virus, consider a biological disease that is 100% infectious, spreads whenever animals communicate, kills all infected animals instantly at a given moment, and has no detectable side effects until that moment. If a delay of even one week were used between the introduction of the disease and its effect, it would be very likely to leave only a few remote villages alive, and would certainly wipe out the vast majority of modern society. If a computer virus of this type could spread through the computers of the world, it would likely stop most computer use for a significant period of time, and wreak havoc on modern government, financial, business, and academic institutions.

```
...
subroutine do-damage :=
  {loop: goto loop;}

subroutine trigger-pulled :=
  {if year > 1984 then return(true)
        otherwise return(false);
...
```

Fig. 3. A denial of services virus.

[1] The term 'prepend' is used in a technical sense in this paper to mean 'attach at the beginning'.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

3. Prevention of Computer Viruses

We have introduced the concept of viruses to the reader, and actual viruses to systems. Having planted the seeds of a potentially devastating attack, it is appropriate to examine protection mechanisms which might help defend against it. We examine here prevention of computer viruses.

3.1 Basic Limitations

In order for users of a system to be able to share information, there must be a path through which information can flow from one user to another. We make no differentiation between a user and a program acting as a surrogate for that user since a program always acts as a surrogate for a user in any computer use and we are ignoring the covert channel through the user. Assuming a Turing machine model for computation, we can prove that if information can be read by a user with Turing capability, then it can be copied, and the copy can then be treated as data on a Turing machine tape.

Given a general purpose system in which users are capable of using information in their possession as they wish, and passing such information as they see fit to others, it should be clear that the ability to share information is transitive. That is, if there is a path from user *A* to user *B*, and there is a path from user *B* to user *C*, then there is a path from user *A* to user *C* with the witting or unwitting cooperation of user *B*.

Finally, there is no fundamental distinction between information that can be used as data, and information that can be used as program. This can be clearly seen in the case of an interpreter that takes information edited as data, and interprets it as a program. In effect, information only has meaning in its interpretation

In a system where information can be interpreted as a program by its recipient, that interpretation can result in infection as shown above. If there is sharing, infection can spread through the interpretation of shared information. If there is no restriction on the transitivity of information flow, then the information can reach the transitive closure of information flow starting at any source. Sharing, transitivity of information flow, and generality of interpretation thus allow a virus to spread to the transitive closure of information flow starting at any given source.

Clearly, if there is no sharing, there can be no dissemination of information across information boundaries, and thus no external information can be interpreted, and a virus cannot spread outside a single partition. This is called 'isolationism.' Just as clearly, a system in which no program can be altered and information cannot be used to make decisions cannot be infected since infection requires the modification of interpreted information. We call this a 'fixed first order functionality' system. We should note that virtually any system with real usefulness in a scientific or development environment will require generality of interpretation, and that isolationism is unacceptable if we wish to benefit from the work of others. Nevertheless, these are solutions to the problem of viruses which may be applicable in limited situations.

3.2 Partition Models

Two limits on the paths of information flow can be distinguished, those that partition users into closed proper subsets under transitivity, and those that do not. Flow restrictions that result in closed subsets can be viewed as partitions of a system into isolated subsystems. These limit each infection to one partition. This is a viable means of preventing complete viral takeover at the expense of limited isolationism, and is equivalent to giving each partition its own computer.

The integrity model [5] is an example of a policy that can be used to partition systems into closed subsets under transitivity. In the Biba model, an integrity level is associated with all information. The strict integrity properties are the dual of the Bell-LaPadula properties; no user at a given integrity level can read an object of lower integrity or write an object of higher integrity. In Biba's original model, a distinction was made between read and execute access, but this cannot be enforced without restricting the generality of information interpretation since a high integrity program can write a low integrity object, make low integrity copies of itself, and then read low integrity input and produce low integrity output

If the integrity model and the Bell-LaPadula model coexist, a form of limited isolationism results which divides the space into closed subsets under transitivity. If the same divisions are used for both mechanisms (higher integrity corresponds to higher security), isolationism results since infor-

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

mation moving up security levels also moves up integrity levels, and this is not permitted. When the Biba model has boundaries within the Bell-LaPadula boundaries, infection can only spread from the higher integrity levels to lower ones within a given security level. Finally, when the Bell-LaPadula boundaries are within the Biba boundaries, infection can only spread from lower security levels to higher security levels within a given integrity level. There are actually nine cases corresponding to all pairings of lower boundaries with upper boundaries, but the three shown graphically in Fig. 4 are sufficient for understanding.

Biba's work also included two other integrity policies, the 'low water mark' policy which makes output the lowest integrity of any input, and the 'ring' policy in which users cannot invoke everything they can read. The former policy tends to move all information towards lower integrity levels, while the latter attempts to make a distinction that cannot be made with generalized information interpretation.

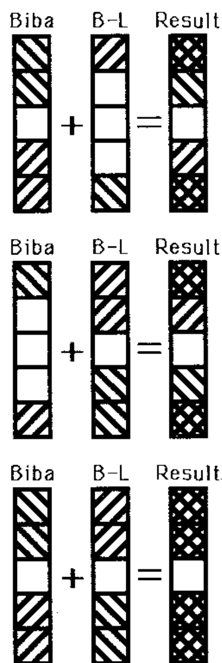

Fig. 4. Pairings of lower boundaries with upper boundaries Top: Biba within B-L; middle: B-L within Biba; bottom: same divisions \\ cannot write; // cannot read; ×× no access; \+/=×

Just as systems based on the Bell-LaPadula model tend to cause all information to move towards higher levels of security by always increasing the level to meet the highest level user, the Biba model tends to move all information towards lower integrity levels by always reducing the integrity of results to that of the lowest incoming integrity. We also know that a precise system for integrity is NP-complete (just as its dual is NP-complete).

The most trusted programmer is (by definition) the programmer that can write programs executable by the most users. In order to maintain the Bell-LaPadula policy, high level users cannot write programs used by lower level users. This means that the most trusted programmers must be those at the lowest security level. This seems contradictory. When we mix the Biba and Bell-LaPadula models, we find that the resulting isolationism secures us from viruses, but does not permit any user to write programs that can be used throughout the system. Somehow, just as we allow encryption or declassification of data to move it from higher security levels to lower ones, we should be able to use program testing and verification to move information from lower integrity levels to higher ones.

Another commonly used policy that partitions systems into closed subsets is the compartment policy used in typical military applications. This policy partitions users into compartments, with each user only able to access information required for their duties. If every user has access to only one compartment at a time, the system is secure from viral attack across compartment boundaries because they are isolated. Unfortunately, in current systems, users may have simultaneous access to multiple compartments. In this case, infection can spread across these boundaries to the transitive closure of information flow.

3.3 Flow Models

In policies that do not partition systems into closed proper subsets under transitivity, it is possible to limit the extent over which a virus can spread. The 'flow distance' policy implements a distance metric by keeping track of the distance (number of sharings) over which data has flowed

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.