Companion Viruses and the Macintosh: Threats and Countermeasures

Jeffrey Horton and Jennifer Seberry

Centre for Computer Security Research
School of Information Technology and Computer Science
University of Wollongong
Northfields Avenue, Wollongong
{jeffh, j.seberry}@cs.uow.edu.au

Abstract. This paper reports on a possible virus attack previously unknown to the authors, which could be implemented under current versions (versions appearing from 1993 – 1998) of the Macintosh operating system and which bears some resemblance to a "Companion Virus" style of attack as seen under MS-DOS. We examine briefly techniques used by other Macintosh viruses, and discuss the implementation of companion viruses under MS-DOS. Companion viruses pose a threat principally to a specific anti-virus technique.
We also propose a simple generic countermeasure that appears very effective against the attack to be outlined.

1 Introduction

The Macintosh virus world is not as active as that of PC and PC-compatible computers. There are only a few dozen Macintosh viruses and variants of those viruses known, not counting macro viruses. To the best of the authors' knowledge, the techniques described here are not employed by any existing virus.

A "companion virus" is a variety of computer virus which avoids modifying the files that it "infects". This paper is the result of work undertaken to explore the possibility of a Macintosh virus capable of exhibiting this variety of behaviour. Implementations of companion viruses under MS-DOS will be briefly explained in Section 1.2. The dangers posed by a companion virus attack are outlined in Section 1.3. This paper was written to raise awareness of the fact that while a method of attack may not be translated directly to another operating system, a similar attack may indeed be possible, and also to supply information on a potential security problem. Macintosh users and anti-virus vendors should be aware of the possibilities we outline.

In the interests of not supplying sufficient information to allow the implementation of a functioning virus, only the basic ideas behind the attack will be discussed. We feel that the security community is better able to respond to a potential security problem of this nature with some degree of foreknowledge of the problem. We present some ideas on how it might be combated.

1.1 What is a Computer Virus?

There is a mathematical definition of a computer virus which is due to Dr. F. Cohen. However, a more accessible English definition that is still thought to describe the essential elements of "real" computer viruses is:

> We define a computer 'virus' as a self-replicating program that can 'infect' other programs by modifying them or their environment such that a call to an 'infected' program implies a call to a possibly evolved, and in most cases, functionally similar copy of the 'virus'. [1]

The term "infect", where used, is used with respect to computer viruses in the sense of the definition above throughout the remainder of this document.

Computer viruses fall into a number of different classes, with some degree of overlap, such as **file infectors**, which modify various executable objects, for example `.EXE` and `.COM` files under MS-DOS, to contain the viral code, or **boot sector infectors**, which replace or modify the machine instructions stored on disk and executed as part of system startup.

Another variety of computer virus is the **companion virus**.

1.2 What is a Companion Virus?

A companion virus is of interest because it does not modify any of the files which it infects. Instead, it creates a separate executable file to hold the virus body. Implementations of such a virus depend on the operating system; two basic types of companion virus which could be created under MS-DOS are [2, 3]:

Regular Companion [2] or Corresponding File Virus [3]:
Creates a file in the same directory as the target of infection but with a filename extension which the operating system chooses to execute before that of the original file when the extension is not explicitly specified (for example, under MS-DOS a `.COM` file with the same name as a `.EXE` file and in the same directory is executed before the `.EXE` file if the file extension is not specified).

PATH Companion:
Create a file with any executable extension in a directory that is searched for executable files before the directory containing the target of infection. Named after the PATH environment variables found in operating systems such as MS-DOS and UNIX.

Magruder [3] also discusses "surrogate file viruses", a type of companion virus which renames the executable file being infected and replaces it with a copy of the virus program.

The Macintosh, however, does not have the concept of a "path" which is followed when searching for executable files, relied upon by the path companion virus, or the notion of filename extensions, relied upon by a regular companion virus. However, various features of the Macintosh operating system enable an attack that appears very similar in conception if not in execution.

1.3 Dangers of Companion Viruses

Bontchev [2] describes companion viruses as one possible attack against an anti-virus measure known as an **integrity checker**. Magruder [3] also discusses the possibility that a companion virus infection may be missed by an integrity checker that is not aware of this type of attack.

An integrity checker works by computing some sort of hash or checksum (a "signature") of files believed to be at risk of infection by a virus. These signatures are stored, and at some future date may be recomputed and compared to the originals; if there is a difference, then the file corresponding to that signature has changed, and this change may be the result of a virus infection — many common viruses infect executable code by altering it in some way, and an integrity checker detects the modifications made by the virus. They can be useful in detecting the presence of known and unknown viruses.

However, integrity checkers are not able to detect companion viruses by this approach, as these viruses do not alter any file which is a target of infection. Instead, they alter the environment of the file. So to detect this type of virus, an integrity checker must be modified to also monitor changes in a file's environment. For example, under MS-DOS the appearance of a `.COM` file in the same directory as a `.EXE` file, where none had been observed previously, would be a suspicious occurrence that an integrity checker could detect; a virus-specific anti-virus measure could then be applied to determine if the suspicious occurrence is the result of a virus.

2 How Macintosh Viruses Work

Most existing Macintosh viruses infect executable code intended to run on a Macintosh based on one of the Motorola MC680x0 series of microprocessors. The first Macintosh computers based on PowerPC microprocessors were introduced in 1994, ultimately replacing the 680x0 in new computers. Programs containing code only for 680x0-based Macintoshes continue to function because operating system software running on the new microprocessors provides an emulator for 680x0 code [4, Ch. 1]. Most viruses are written for 680x0-based Macintoshes; they continue to be a threat to 680x0-based applications on PowerPC-based Macintoshes as a consequence of the compatibility provided by this emulator. The following discussion relating to the organization of executable code on a Macintosh applies only to applications containing code intended for a 680x0-based Macintosh.

Every Macintosh file is composed of two forks, a data fork and a resource fork. The data fork contains a file's data — for example, a text file would contain ASCII text in the data fork. The resource fork contains a file's **resources**. An application program's resource fork contains the application's executable code [5, p. 1-4]. Resources within a particular file are described by a resource type (a four-letter code), and an ID number (2-byte signed integer).

The executable code for an application is usually divided up into a number of segments, each segment stored in a "CODE" resource. Only some of these must

be in memory at certain times; code segments will be loaded when needed and may be unloaded when not required. This enables large application programs to be run in limited memory. Segment loading, and references from one segment to a routine located in another segment are coordinated using a table created when the executable code is linked by a compiler. This table, the **jump table**, is stored as the "CODE" resource of ID 0 [6, Ch. 7]. The first entry in the jump table specifies the first code that will be executed when an application is run.

A virus that infects an application program then has two ways in which it may proceed. First, it may add a new code segment to the application's resource fork, and adjust the jump table to refer to the new segment, usually as the first code that is called when the application starts executing. Having performed its startup tasks, the virus can replace the original jump table entry, saved during the infection process, and jump to it. This strategy is used by the nVIR family of viruses [7] and also by the INIT 29 virus [8], among others.

Second, a virus might instead choose to append its code to an existing code resource, perhaps not modifying the jump table at all. Complications can arise due to resource size limitations, however.

Such a virus is presented with a problem in becoming permanently resident in memory, and arranging for calls to virus code at a later time. Patches may be applied to many calls provided by the Macintosh operating system, but will only have global effect when applied at system startup [9, p. 8-9]. So many viruses that infect application programs also infect the operating system, so that virus code is executed when the computer is rebooted. Otherwise, the virus will cease to be effective once an application exits. The nVIR family and INIT 29 are examples of viruses that behave this way.

Patching can also be used by anti-virus programs, to attempt to detect signs of virus activity. The suspicious activity could perhaps be blocked, logged, or reported to the user.

Other viruses work by modifying or overriding code used by the operating system. The WDEF virus worked by overriding the code used by the operating system to draw windows on the screen, and did not involve making any modifications to application programs or making direct modifications to the operating system [10]. Some of these viruses are no longer effective, as operating system features on which they depended were changed or removed [11].

One recent virus-like program, the **AutoStart** worm [12], uses techniques not seen previously in Macintosh viruses. It relies on a feature of **QuickTime**, Apple Computers software architecture for creating and viewing digital media. The AutoStart feature allows an application to be designated for automatic execution whenever disks are mounted. When the virus is executed, the operating system is infected. The virus becomes active on the next reboot, searching periodically for other media, like floppy disks, to infect. This virus/worm spreads well, but is easy to remove by hand. The AutoStart feature may be disabled to prevent future infection.

We describe here a technique which could potentially be used by a virus, which does not involve any modification of application programs or the operat-

ing system during the process of infection, unlike most of the techniques outlined above. We feel the style of the attack is very similar to that employed by companion viruses under an operating system such as MS-DOS; hence the name.

3 More Macintosh Basics

This section explains some aspects of the Macintosh file system and operating system which will be required to understand the description of the attack to follow. These details, and the attack description, are valid with respect to versions of the Macintosh operating system from 7.1, which is now several years old, to 8.1, the latest available to the authors when this paper was prepared.

3.1 File Types and Creators

Every file used on a Macintosh computer has both a **file type** and **file creator** associated with it.

The **file creator** is a four-byte code, usually a combination of various ASCII characters, which is identical to that of the application program (the package of executable code which is executed directly by the computer user in a variety of possible ways; henceforth it will be referred to simply as the "application") responsible for creating that particular file. Each application in turn defines a number of **file types**, each of which is another four-byte code, again usually a combination of various ASCII characters, that describes the nature of the data stored in the file. These codes need have meaning only to the application concerned.

Files of certain types may be used by many different applications. For example, files of type `TEXT` are files that consist of plain ASCII text that requires no special handling, so that they may be viewed and altered by applications other than the creating application. A file of type `APPL` is the common variety of executable program file of most interest to a computer user (there are several other types of executable files which are not of interest here).

The application typically defines an **icon**, or small graphical symbol, for each file type which can be created by that application, to help users easily determine which application was responsible for creating the file.

3.2 The Desktop Database

The Desktop Database is a collection of information maintained and used by the operating system. Its format does not appear to have been publicly documented. The function of the Desktop Database which is of interest is that it stores information about the location and creation date of applications. Information about the icons used by files created by particular applications is also stored here.

Not every type of disk has a Desktop Database. For example, floppy disks (storing about 1.4 megabytes) do not have a Desktop Database, but instead have a simpler structure which performs similar functions. Our attention here is

confined to disks which do have Desktop Databases, such as hard disks, or for which one is created, such as AppleShare volumes [13, pp. 9-3–9-4].

The Desktop Database is used when the operating system creates its graphical display of windows and icons — the icon that should be displayed for a file with a certain type and creator may be determined using the Desktop Database.

It is also of use when determining which application program should be started when the user opens a file, if an application with the appropriate creator code is not already being executed. Commonly this is done by double-clicking twice on the file concerned using the mouse, although there are other methods which achieve the same effect. In order to make use of this file, the operating system must start an application which can interpret the contents of the file.

Usually there will be a single application with a certain creator code on a disk. However, it is possible to have several applications with the same creator code on the one disk. Although according to a technical note [14] the application which is the "first choice" is the one with whose information the Desktop Database was last updated, correcting earlier documentation stating that the "first choice" application was the one with the most recent creation date [13, p. 9-5], it appears in most cases that the application selected will be the one with the most recent creation date. Even after rebuilding the Desktop Database, a process that may be initiated by the user and which is sometimes useful in troubleshooting, the application selected is the one with the most recent creation date. It is critical to the attack described in Section 4 that the application selected for execution be the viral application.

3.3 Starting an Application

When the user starts an application by opening a file or files in some manner, the application is notified by the operating system of the files that were selected by the user.

This is accomplished using an **Apple Event**, which is a special type of Macintosh operating system event that may be used for a variety of different purposes. There are many different types of Apple Event, and applications may define their own. In the case being considered here, the operating system sends an "Open Document" (`odoc`) Apple Event to the application when it has started executing to inform the application of the location of the files that the user wishes to open using that application.

Not all application programs support receiving Apple Events; a program that does not support these events is not a candidate for infection by the method we will describe. Applications may provide Apple Event support but don't themselves "own" any files; as will be seen, such applications are not good candidates for infection. However, many common application programs that create files that are "owned" by that application support these events, and would be more difficult for users to work with if they did not.

4 A Macintosh "Companion Virus"

Recalling that the operating system runs the application with the most recent creation date when there exists more than one application on a given disk with the same creator code, the application to execute not having been explicitly specified by the user, and sends that application an event describing the files which the user desires to open with that application, it suffices to infect an application by:

1. Creating an application program with the same creator code as another application on disk that is the target of infection, but with a more recent creation date, such that the operating system executes the viral application;
2. Intercepting the event sent to the infected application by the operating system; and
3. Running the infected application and sending it the intercepted event; soon after this step, the viral application would exit.

There are, however, a number of other details which must be handled to create an effective virus, such as preserving the original application icon information, which is usually overridden by the icons applicable to the more recently created application. We are reluctant to discuss solutions to such problems here, in the interests of not revealing enough information to easily create an effective virus.

Clearly this attack bears some resemblance to a companion virus style of attack as described in Section 1.2.

It should be noted that it is certainly possible to specify exactly which application program is to be used to manipulate a particular file or files; such a virus would rely on the fact that it is more convenient to permit the operating system to identify and run an application than to perform this task manually.

Some consideration has been given to how such a virus might become resident in memory; that is, how it might place viral code somewhere in memory and arrange for it to be executed at some time in the future, long after the viral application itself has ceased to run. Installing a device driver is one possible option. Device drivers are not required to deal with devices at all; other uses have been found for them. A device driver can elect to receive calls from the operating system to perform periodic tasks — in the case of a companion virus, such a periodic task could be searching the list of currently running application programs, infecting any that are not already infected.

4.1 Detection

The mere presence of several applications with the same creator code on a Macintosh computer system is not something which should cause any alarm, and is not enough to conclude that an application program has been infected by a companion-type virus. This situation commonly arises when, for example, a new version of an application package is installed without removing the previous

version. An integrity checker would need to monitor other information to help it decide how alarming the presence of multiple applications with the same creator code is. For example, as a companion virus would most likely be a much smaller and of a simpler structure than an application that has some more useful functionality, the presence of two application programs with the same creator code but very different sizes or structures might be considered suspicious.

There are a number of system calls which may be of use in implementing such a virus and which may be considered suspicious by some sort of behaviour monitoring program. A behaviour that is a characteristic of such a virus is a call to launch the infected application. As the viral application and infected application have the same creator codes, if a patch were installed before the operating system routine responsible for launching an application, it could check the creator code of the application originating the request against the creator code of the application being launched, and refuse to launch an application having the same creator code as the one making the request; an indication to the user that suspicious activity has been detected would certainly be appropriate. There are various ways that might be used by a virus to circumvent such a check, but employing this check cuts off the simplest and most straightforward way for one application to launch another application. An occasion where one application might legitimately need to launch another with the same creator code seems most unlikely.

As the virus exists as an application separate to the infected application, checks on the creation of applications may also be effective. Some anti-virus applications will likely include such checks, at least as an option, as it may be effective against other varieties of virus. However, this is not as specifically targeted against a companion virus attack as the previous countermeasure, and would not seem to be appropriate to as wide an environment — for example, people working with compilers may find checks on creation of application programs produce many false alarms.

The possible utility that a device driver might have for a companion virus is also discussed in Section 4. There are many legitimate reasons that a program might wish to install a device driver; for example, they are commonly used to implement virtual disk schemes, where the raw disk data resides in a large container file. As a device driver is potentially of use in a virus attack, it would be useful to check drivers for suspicious code that might perform virus-like actions when installed.

Other calls by the virus may also be able to be monitored. For example, the companion virus can use the `PBDTAddAPPL` call to update the Desktop Database with information about the newly created viral application when performing an infection, rather than waiting for the operating system to update the Desktop Database with information about the new application at some time in the future. It should be noted that Apple documentation warns against changing the Desktop Database [13, p. 9-3].

Not using `PBDTAddAPPL` could produce some interesting results from the point of view of the virus writer — a viral application could be created, but would not

become active until added to the Desktop Database by the operating system, producing a crude form of time delay between actual infection and results of the infection.

4.2 Demonstration Program

A non-viral application program demonstrating this attack has been created by the authors. The demonstration program has been found to work appropriately in a variety of environments — single and multiple partition Macintosh hard drives, removable media such as Zip disks, and a simple network consisting of two Macintosh computers.

4.3 Dangers Posed?

Having discussed a method by which a companion virus for the Macintosh might be written, some consideration ought to be given to the dangers posed by an attack of this nature.

We consider this attack to be largely of academic interest. It would be difficult for a virus constructed in this manner to remain undetected or to spread between systems. As this variety of virus is a distinct non-invisible file on the disk, it is noticeable by an observant user. Application files on a Macintosh may not be marked as being "invisible" and still be executed successfully. Furthermore, in the interests of surviving to multiply, such a virus would place itself somewhere not associated with the infected application, and so would be unlikely to spread through distribution of software archives.

Such a virus may be able to spread via a local area network to another Macintosh. Its ability to spread across a network could of course be slowed by proper configuration of user permissions. In particular, users should not have permission to make changes on network volumes unless absolutely necessary.

The attack is rendered considerably more potent if the virus is able to become resident in memory. One way that this might be accomplished is outlined.

The attack is perhaps most dangerous if the virus so constructed is capable of two modes of infection. For example, one time in ten the virus might infect by modifying the target program in the manner of a file-infecting virus; although it would be readily detected by an integrity checker, and perhaps by a behaviour monitor, this would improve its chances of spreading to another computer. The undetected copies of the virus which infect using the "companion" strategy would form a reservoir for future infections.

5 Conclusion

We have considered a possible virus attack that could be implemented under the Macintosh operating system. The attack has a good resemblance to a companion virus style of attack.

The authors know of no Macintosh viruses implementing such an attack, or of discussions of such a method of attack. The attack is not believed to pose as great a danger as other varieties of computer virus, due to limitations of the implementation described. It could, however, avoid detection by an integrity checking program that was not aware of the possibility of this implementation of the companion virus strategy.

We discuss various countermeasures that might be employed against such a virus. We believe that the most effective of these measures is to check the creator code of the application attempting to launch another against that of the application being launched, and to abort the request if the creator codes match. Infection is not prevented, but is readily detected.

References

1. V. Bontchev. Are 'good' computer viruses still a bad idea? In *Proceedings of the EICAR '94 Conference*, pages 25–47, 1994. Available online from ftp://ftp.informatik.uni-hamburg.de/pub/virus/texts/viruses/goodvir.zip.
2. V. Bontchev. Possible virus attacks against integrity programs and how to prevent them. In *Proc. Second International Virus Bulletin Conf.*, pages 131–141, 1992. Available online from ftp://ftp.informatik.uni-hamburg.de/pub/virus/texts/viruses/attacks.zip.
3. S. Magruder. High-level language computer viruses — a new threat? *Computers & Security*, 13(3):263–269, 1994.
4. Apple Computer Inc. *Inside Macintosh: PowerPC System Software.* Addison-Wesley Publishing Company, 1994.
5. Apple Computer Inc. *Inside Macintosh: Files.* Addison-Wesley Publishing Company, 1992.
6. Apple Computer Inc. *Inside Macintosh: Processes.* Addison-Wesley Publishing Company, 1992.
7. David Ferbrache. Virus Analysis: nVIR and its Clones. *Virus Bulletin*, pages 13–14, October 1989.
8. David Ferbrache. Macintosh Viruses: INIT 29 — Infectious, but your data is safe. *Virus Bulletin*, pages 8–9, December 1989.
9. Apple Computer Inc. *Inside Macintosh: Operating System Utilities.* Addison-Wesley Publishing Company, 1994.
10. David Ferbrache. Virus Report: WDEF — The Hidden Virus. *Virus Bulletin*, page 14, January 1990.
11. David Ferbrache. Dirty Macs. *Virus Bulletin*, pages 17–18, February 1992.
12. Craig Jackson. Worms in the ripe apple. *Virus Bulletin*, pages 6–8, July 1998.
13. Apple Computer Inc. *Inside Macintosh: More Macintosh Toolbox.* Addison-Wesley Publishing Company, 1993.
14. TB 19 — How PBDTGetAPPL chooses which copy of an App to launch. Available online from http://developer.apple.com/qa/tb/tb19.html.