The **Insider**, Naivety, and Hostility:

Security

The background of the lower half of the cover is a dark, abstract digital image. It features a complex network of glowing blue and white lines, resembling a data network or a digital landscape. There are bright, hazy light sources on the right side, creating a sense of depth and intensity. The overall color palette is dominated by dark blues, greys, and bright whites.

HERBERT H. THOMPSON, SECURITY INNOVATION

RICHARD FORD, FLORIDA INSTITUTE OF TECHNOLOGY

Every year corporations and government installations spend millions of dollars fortifying their network infrastructures. Firewalls, intrusion detection systems, and antivirus products stand guard at network boundaries, and individuals monitor countless logs and sensors for even the subtlest hints of network penetration. Vendors and IT managers have focused on keeping the wily hacker outside the network perimeter, but very few technological measures exist to guard against insiders—those entities that operate *inside* the fortified network boundary. The 2002 CSI/FBI survey estimates that 70 percent of successful attacks come from the inside.¹ Several other estimates place those numbers even higher.²

Perfect Storm?

Keeping nasties out is
only half the battle.

The **Insider**, Naivety, and Hostility: Security Perfect Storm?

Attacks that come from within an organization are not very well understood, and although standard security principles such as discretionary access control and least privilege are simple to understand, their application can be problematic and unsystematic. The issue is trust. Insiders must be trusted to do their jobs; applications must be trusted to perform their tasks. The problem occurs when insiders—be they users or applications—intentionally or unintentionally extend trust inappropriately. There is often a large gap between the rights we believe we are extending to another person, application, or component and the rights that are actually granted. Online trust seems to be an all-or-nothing affair, quite unlike the trust we extend in the nonvirtual world.

A combination of naivety, hostility, architecture, and misunderstood trust relationships makes the issue of the insider threat acute. Although these factors seem unrelated, they all represent different aspects of the same fundamental problem. The confluence of these factors forms the “perfect storm” for information security managers. This article takes a look at the technological challenges involved in enforcing mitigated trust and how this relates to threats from the inside.

ARCHITECTURES BUILT ON TRUST

When considering trust in a software environment, we must first examine the nature of modern software. Systems are becoming highly integrated—development architectures such as COM+ and CORBA (Common Object Request Broker Architecture) encourage architects to distribute systems and leverage unexplored (and often unconsidered) trust relationships. Additionally, the architectural decisions made by many application developers leave the application or its host system vulnerable to attack as a result of inappropriately extended trust in ways that the user may not be aware of.

Through such architectures, trust can be compromised in a number of different ways. For example, malware (malicious code) such as MyDoom relies on user naivety and architectural decisions that can delegate trust to a hostile user who can deliberately act to the detriment of security. This type of worm spreads through e-mail recipi-

ents clicking on executable e-mail attachments with subjects such as “Mail Delivery System,” “Mail Transaction Failed,” or “Server Report.” This problem is exacerbated by the fact that some e-mail readers hide the extensions of attachments. In one variant of MyDoom, the worm displayed the same icon as a Windows Notepad file (a text document).³ Users generally assume text documents to be safe to open; thus, the decision to hide extensions and allow executables to run from inside messages without prompting undoubtedly contributed to the worm’s rate of propagation.

ARCHITECTURAL ASSUMPTIONS

A classic example of architecturally enabled trust is the original architecture of Microsoft Word. By default, Word allowed the automation of repetitive instructions through macros. These macros could be embedded in a document and then executed by a sequence of user keystrokes while the document was being edited. Microsoft included a robust macro language, which later supported Visual Basic scripts. Also of note was the decision to allow certain macros to run automatically when the document was launched, closed, or modified. Most casual users of Word were never aware of the macro feature, and the pervasiveness of Word led to the widespread sharing of documents. Most users thus extended trust to possibly unknown and potentially malicious users by the seemingly innocuous action of launching a document inside the Word editor.

Arguably the first widespread malicious macro appeared in 1995: the Concept macro virus infected the Normal.dot template file so that all future documents created by the user would also contain the virus. The original strain of Concept did little damage to end users, but successors such as the Wazzu macro virus changed the contents of documents. The most infamous and costly macro virus to date was Melissa, which was much more ambitious in its propagation. Instead of waiting for the user to send an infected document to someone else, Melissa sent infected documents to the first 50 entries in every address book stored in Microsoft Outlook. Melissa cost users several hundred million dollars.⁴

It is interesting to note that none of these viruses exploited a mistake made in the way the software was coded. Instead, they relied upon trust extended on behalf of the user, and thus no simple bug fix would have been effective at protecting the organization. Essentially, a trust relationship exists between users and the application: when users open an application, they *trust* that the document will not cause any permanent change to their machines.

The discrepancy between *perceived* and *actual* extended trust can have its roots in the architecture of an application or a system. Technology enables us to extend trust in ways that are not readily discernable or desirable. To continue the Word example, by design in Word 95 and 97 the action of opening a document was tantamount to giving that document's author the ability to delete files, steal data, or perform any other task that the user viewing the document was capable of. Beyond trusting that document's author, the user would extend trust to anyone trusted by the document's author. If any one of those individuals had opened an infected document, the result

would be an infection. Many users do not understand the implications of opening documents with executable content and the chain of trust involved (see figure 1).

By opening an e-mail attachment, an insider is tacitly extending trust to the sender, everyone trusted by the sender, everyone trusted by people trusted by the sender, and so forth.

Microsoft's response to the macro-virus problem was interesting—and highly illustrative. The developers presumably determined that simply removing all macro functionality was not an acceptable solution, so an increasingly complex system for managing trust was built into the Office suite of programs. This process was not without problems; several bugs were found in the system as “trust control” was bolted on to a framework for which it was never designed.⁵

Trust can also be extended through much subtler architectural decisions. In 2000, Andre Dos Santos of the Georgia Institute of Technology was engaged to test the security of the online banking service of a large international bank.⁶ Online users were required to enter their account numbers and PINs to access their account information. The bank realized that it was open to the threat of an attacker taking a valid account number and trying to brute-force the password. In response it implemented a common architectural control that locks an account out after a small number of log-in failures within a given period. Such controls can greatly reduce the risk of an attacker brute-forcing the PIN for a given account, as a single account has 10,000 possible PIN numbers. With the lockout mechanism in place, it would take just over nine years to try all possible PIN combinations, and, thus, *prima facie*, this appears to be a sound architectural decision.

This mechanism, however, makes a flawed assumption about the attacker. The first is that an attacker's goal is to break into a specific account and not just any account at the bank. An attacker might choose a common PIN such as “1234,” then iterate through a list of sequential account numbers. The accounts that use a different PIN would then show only one failed log-in attempt and the lockout would never be triggered. Dos Santos's study found that by using this technique—combined with the issuance of sequential account numbers—his team was able to access three percent of all accounts just with the simple PIN “1234.” Dos Santos's team had marked success by iterating through other common PINs, all without triggering bank alarms.

An even subtler extension of trust is made by the lock-out control itself. For example, what if an attacker were

The **Insider**, Naivety, and Hostility: Security Perfect Storm?

not after financial gain but instead wanted to deny legitimate account holders online access to their accounts? This could easily be accomplished by exploiting the security mechanism itself and writing automation to purposely make three failed log-ins to every account on the system—effectively shutting out all legitimate users for 24 hours. This attack could be repeated daily causing even further harm for the bank customers and accruing expense for the bank's IT department and support staff (not to mention loss of reputation by the bank itself).

At the network level, architectural decisions can extend trust in more overt ways—through extranets, which blindly trust users based upon credentials, not action. Existing technology is not easily amenable to implementing layers of trust: in general, any code run by a user executes with the full privileges of that user. This extension of trust usually occurs in unexpected ways. For example, extranets may be made very secure by fortifying the network boundary. Once a user with an account is authenticated, however, all of the firewalls, routers, and antivirus products may not be configured to block a malicious action from *within* the trusted extranet.

A classic example of this is the architecture of Web applications that are assumed to be for internal and extranet use only. Many of these intranet and extranet applications thus inappropriately extend trust through their lack of testing and add-on security mechanisms, which are usually reserved for sites assumed to be directly Internet-facing. It is interesting, therefore, to consider the extension of trust as a *failure* to do something as opposed to a deliberate action.

EVOLVING ARCHITECTURES

A large part of the challenge of digital security is that there is little correspondence between typical real-world trust relationships and cyber-trust relationships. While the real world has numerous examples of partial trust, many architectures rely on complete system trust to function.

As a design example, we cite early implementations of Word, which allowed malicious mobile code to be executed with the full permissions of the local user. This

serves as an excellent example of total trust, as opposed to mitigated trust. When users open Word documents, they tacitly accept an unstated contract that opening the document will not make any permanent changes to the system. Implementations of Visual Basic for Applications, however, have no context for supporting such a contract—it is left entirely to chance and the intent and ability of the document's macro writer to ensure that this is the case.

Another classic example of trust models is the underlying difference between Java applets and ActiveX. Under Java, the design paradigm is one of mitigated trust, in that applets downloaded from untrusted sources are highly limited in their ability to make "important" changes to the local machine operation. Here, "important" would mean, essentially, actions that require "undoing," such as modifying arbitrary files or sending e-mail on behalf of the user. The out-of-the-box configuration of the applet Java Virtual Machine fairly efficiently protects the user from the problem of global and permanent changes, while still providing for limited inter-session persistence. As such, Java applets are considered a fairly "safe" form of mobile code. This consideration is held out in practice, as illustrated by the small number of malicious Java applets one encounters.

Compare this with the state of ActiveX, which works on the principle of complete trust. Essentially, ActiveX components are not sandboxed in any fashion, and once invoked have an all-or-nothing relationship to the local machine, usually running with the privileges of the local user. This architecture may be considered to be more highly functional than the corresponding applet framework, as an ActiveX control can carry out any computing function whatsoever. This framework is also highly dangerous; once again, hostile ActiveX is encountered "in the wild," and corporate entities are rightly more concerned about the security implications of ActiveX than Java.

Architectures have also been forced to evolve through regulation. Microsoft, for example, made several changes in its Windows operating system based on U.S. government requirements laid out in the Department of Defense Trusted Computer System Evaluation Criteria, a.k.a.

the Orange Book. The process, however, of setting such standards can be somewhat arbitrary, and the requirements themselves are often outdated by changes to the threat profile or base technology. The implementation of these requirements has also been of questionable benefit in many cases.

Consider, for example, the Orange Book requirements that dictate that when a file is created, the operating system must zero the file before allowing access to it. This was intended to deny users access to other data that may have previously been on the disk. While the Windows NT file system was made to comply with the requirement of zeroing out new files before they are used, users are still able to recover data from the disk after a file is deleted but not reclaimed. The result is that “deleted” data can remain on disk for a significant period of time.

NAIVETY AND HOSTILITY

Applications are not the only entities that are cause for trust issues in modern computing. Employees, partners, and contractors must all be extended privilege in some sense to be able to do their jobs. One of the most basic privileges is physical access to facilities, equipment, and coworkers. Arguably, such physical access is a trust concept that we are familiar with, and the implications of access are somewhat readily discernable. Mechanisms

such as door locks, sealed cabinets, and safes, along with the people who are given the mechanisms to bypass these physical controls, are familiar concepts.

Even here, however, architectural issues come into play. Consider, for example, the ubiquitous VPN (virtual private network) tunnel. Often, such a tunnel virtually relocates the user’s PC inside the corporate firewall. All too frequently, however, the only functionality actually required is e-mail access or access to stored files. Similarly, intra-office VPNs often completely join networks even when limited trust is more practical. Thus, network architects often trade simplicity of configuration for security, even though architectural solutions exist to mitigate trust. “All or nothing” once again raises its ugly head.

Tampering with software is another interesting threat to consider. Software developers usually have a degree of autonomy in coding a particular component or feature. Typically, a developer works from a detailed specification that describes how components should behave and what their interfaces are. Good software engineering practice is that this code is then reviewed by another developer or group of developers before it is included in the final application. In theory, this process not only improves code quality, but also prevents the insertion of unspecified code by a rogue developer. In practice, however, unspecified code often finds its way into released prod-

TABLE 1 Top 10 viruses for 2003 as reported by Sophos

Rank	Virus	Percent of Reports	Infection Vector
1	W32/Sobig-F	19.9 percent	Relies on user action
2	W32/Blaster-A	15.1 percent	Uses software vulnerability
3	W32/Nachi-A	8.4 percent	Uses software vulnerability
4	W32/Gibe-F	7.2 percent	Exploits vulnerability but relies on user action if vulnerable software not present
5	W32/Dumaru-A	6.1 percent	Relies on user action
6	W32/Sober-A	5.8 percent	Relies on user action
7	W32/Mimail-A	4.8 percent	Exploits vulnerability but relies on user action if vulnerable software not present
8	W32/Bugbear-B	3.1 percent	Exploits vulnerability but relies on user action if vulnerable software not present
9	W32/Sobig-E	2.9 percent	Relies on user action
10	W32/Klez-H	1.6 percent	Exploits vulnerability but relies on user action if vulnerable software not present

The Insider, Naivety, and Hostility: Security Perfect Storm?

ucts. This extra functionality is usually referred to as an “Easter egg,” and for most software vendors, including it is an offense that can lead to dismissal of the party who wrote/included it.

Trust issues are equally complex when considering the many (and thorny) issues of digital rights management. In such a scenario, content is placed in an environment where the very administrator or controller of that content is the entity that may not be trustable. This aspect of the problem is crucial to understanding the complex solutions that serve to protect mobile information. In this worldview, the ultimate insider is the user of the data, who controls the environment in which the data is stored and manipulated.

User and administrator naivety contribute significantly to the insider threat. The all-or-nothing trust models that pervade both software and network architectures make it likely that a user will inadvertently extend privilege to a malicious user or application. Table 1 shows a listing of the 10 most common viruses of 2003, according to anti-virus company Sophos.⁷ Six of the 10 relied on software flaws to infect a machine. The remaining four—including the number 1 virus—exploited no software flaws: they relied on an overt action of trust by the victim (such as opening an executable e-mail attachment). In those four instances, users made an explicit—albeit naive—extension of trust to an application.

IT managers and users also make uninformed deployment and execution decisions as a matter of course. Whenever we choose one application over another to purchase, install, or deploy, we open ourselves and our organizations to the unknown security flaws contained within that software. Two similar applications can be compared on performance, cost, compatibility, and a variety of other factors that are perceived to impact the total cost of owning that application over its lifetime. All of these characteristics factor into purchasing decisions. One of the biggest costs in “owning” an application, however, is the impact of a virus or worm that exploits unknown-at-the-time-of-purchase flaws in that application and the cost of deploying and maintaining patches.

There are no generally accepted comparative software

security metrics to help buyers make discriminating purchasing decisions with respect to security. Typically, decision makers are forced to rely on product marketing literature, vendor reputation, and personal experiences to evaluate the security of COTS (consumer off-the-shelf) software. None of these factors is likely to be a good indicator of latent application security flaws.

WORKING TOWARD MITIGATION

At this point, it is worth briefly considering the total cost of ownership and risk *mitigation*. Despite our purist security training, we recognize that the security process is not—and should not be—one of risk annihilation. Rather, risks are mitigated to provide for the optimum balance between functionality/ease of use and security. The medicine should not be worse than the disease! There is a clear trade-off between the simple enforcement of limited trust and usability. For example, some personal firewalls plague the user with incessant pop-ups asking if a particular action is desired. Such interruptions are deleterious to productivity and can actually reduce security, as users are trained to click through the warnings indiscriminately. Clearly a more sophisticated model of mitigated trust is required, such that benign activities do not generate a go/no-go decision on the part of the user.

In terms of mitigation, refocusing the problem in terms of changes in behavior or information flow outside particular boundaries may be of use. Thus, the problem is moved toward a question of permissions. While we are under no illusions as to the difficulty of this task, we do believe that even a fairly simplistic model of levels of trust will significantly improve the security of our systems. Furthermore, by refocusing the discussion on the idea of permanent system changes as opposed to atomic and contained modifications within a certain sphere of influence, we can create a fairly simple false-positive reduction mechanism.

Finally, we argue that in extending trust, decisions are rarely made within a well-considered risk-reward matrix. Rather, trust is blindly doled out without thought, as the common paradigm is simply binary as opposed to continuously variable. This tends to act to the detriment of

both security *and* total cost of ownership. A more holistic view of the issue is required.

CLOSING THOUGHTS

The issue of insider attacks is complex and often misunderstood. In this discussion, we have argued that significant insight can be gained by a reexamination of the problem based on the concept of *trust* extended to multiple entities. Typically, insider attacks are thought of in terms of users; we have argued that human insiders are only one example of an insider. Applications, viruses, and malware in general all can operate as insiders to a system, as can hardware.

Who or what is the insider? To whom or what do we extend trust, and what does that trust allow? Who or what has the *ability* to carry our hostile actions on our systems? By incompletely answering these questions, we believe that the ongoing combination of naivety on the part of users, architects, and security professionals provides for the perfect storm: the age of the insider security threat is upon us.

Architectural understanding of trust is crucial in building systems that promote trust mitigation and thereby limit the size of the universe of insiders. Even where secure architectures exist for such an approach, proper implementation is lacking.

There are signs that the industry is beginning to take the insider threat seriously. The U.S. government is starting to invest in solutions to mitigate the threat. Our company and university are funded by the Office of Naval Research to create innovative protections that mitigate the insider threat. Vendors such as Microsoft and IBM are making changes in the way that software is administered. Companies such as Verdasys are being founded specifically to help companies protect themselves. Antivirus companies are working on proactive antivirus solutions to protect naive users. While these efforts are certainly a step in the right direction, the insider threat still looms large. Our goal here is to help define the problem; a viable and comprehensive solution remains on the distant horizon. Q

REFERENCES

1. Power, R. 2002 CSI/FBI computer crime and security survey. *Computer Security Issues and Trends VIII*, 1 (Spring 2002).
2. Hayden, M. V. The Insider Threat to U.S. Government Information Systems. Report from NSTISSAM INFOSEC /1-99, July 1999.
3. Ferrie, P., and Lee, T. Analysis of W32.Mydoom.A@mm;

<http://securityresponse.symantec.com/avcenter/venc/data/w32.novarg.a@mm.html>.

4. Bridwell, L., and Tippet, P. ICSA Labs 7th Annual Computer Virus Prevalence Survey 2001. ICSA Labs, 2001.
5. See, for example, Microsoft Security Bulletin MS03-050, Vulnerability in Microsoft Word and Microsoft Excel Could Allow Arbitrary Code To Run: <http://www.microsoft.com/technet/security/bulletin/MS03-050.msp>; or MS03-035, Flaws in Microsoft Word Could Enable Macros To Run Automatically: <http://www.microsoft.com/technet/security/bulletin/MS03-035.msp>.
6. Dos Santos, A., Vigna, G., and Kemmerer, R. Security testing of the online banking service of a large international bank. *Proceedings of the First Workshop on Security and Privacy in E-Commerce* (Nov. 2000).
7. Sophos Corporation. Top ten viruses reported to Sophos in 2003; <http://www.sophos.com/virusinfo/topten/200312summary.html>.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

DR. HERBERT H. THOMPSON is director of security technology at Security Innovation (www.sisecure.com). He is coauthor of *How to Break Software Security: Effective Techniques for Security Testing* (Addison-Wesley, 2003) and is the author of more than 40 academic and industrial papers on software security in publications such as *Dr. Dobbs Journal*, *IEEE Security and Privacy*, *Journal of Information and Software Technology* and *ACM Queue*. Thompson earned his Ph.D. in applied mathematics from the Florida Institute of Technology. At Security Innovation, he leads contract penetration-testing teams for some of the world's largest software companies and is principal investigator on several grants from the U.S. Department of Defense.

DR. RICHARD FORD is research professor at the Center for Information Assurance at Florida Institute of Technology. He graduated from the University of Oxford in 1992 with a Ph.D. in quantum physics. Since that time, he has worked extensively in computer security and malicious mobile code. Previous projects include work on the computer virus immune system at IBM Research and development of the largest Web-hosting system in the world while director of engineering for Verio. Ongoing projects include Gatekeeper, a proactive antivirus solution with undo capabilities. Ford is executive editor of *Reed-Elsevier's Computers & Security and Virus Bulletin*.

© 2004 ACM 1542-7730/04/0600 \$5.00