A First Generation Technical Viral Defense (U)

[redacted] P.L. 86-36

~~This article is classified SECRET in its entirety.~~

Editor's note: This paper was awarded second prize in the Computer Security Category of the 1987 Computer and Information Sciences Institute Essay Competition.

Computer viruses are a form of Trojan horses with a self-propagating property. They can be extremely infectious and virulent when used maliciously in computer systems. Many defenses are available to System Security Officers (SSOs which will limit or detect viruses. Most methods are easy to implement, yet provide the SSO with a high degree of effective viral control. These defenses include "sealing" the program (by encryption techniques), comparing the pre- and post-fix portions of programs, limiting the domains the executable code inhabit, and controlling the flow and access rights of programs. Second generation viral defenses will use heuristics to detect viruses, audit the system looking for specific viral-indicators, or compare the coding style in programs. Standard personnel and procedural techniques will not be discussed.

INTRODUCTION

System Security Officers have a wide variety of options to defend against software sabotage. They can institute technical measures to prevent or detect unauthorized alterations, investigate the backgrounds of their employees, and implement procedures to limit opportunities of introducing malicious code. This paper will discuss the former measure: that of compiling a suite of technical means to limit and detect software sabotage, primarily that sabotage via computer viruses.

Computer viruses are a form of Trojan horse. Their mission is usually malicious and triggered by some event, such as a certain system date or the disappearance of a certain name from the payroll database. They have the additional property of being able to copy themselves from one program to another. When introduced into a system with little or no defenses, they can quickly take over the system (obtain full privileges). [5, 9, 13]

Emphasis on computer viruses as opposed to the general class of Trojan horses was chosen for two reasons: first, because their propagation property makes them potentially more dangerous relative to "ordinary" Trojan horses; [3, 11] and second, because their propagation property makes them potentially easier to detect than ordinary Trojan horses. Technical methods which are germane to this problem will be discussed, while generic personnel and procedural security measures will not.

DEFINITIONS

A computer virus is a form of Trojan horse which has the (additional) property of being able to copy itself to another program, other than the program it inhabits. Both a program "infected" with a virus and a virus-free program are called "hosts."

A virus has three components [2]: the first is the propagation component, that part which causes the virus to propagate to other hosts; second is the mission, which is the ultimate goal of the virus and is usually malicious (delete all files, usurp the system, etc.); and the third is the trigger mechanism. The trigger directs the virus when to execute the other two components.

ASSUMPTIONS

We assume that the programs used to detect viruses are themselves not infected with viruses, and that they contain no other form of malicious logic. If this is not assumed, it is easy to construct scenarios where they fail. A program which ostentatiously checks for viruses could be modified such that it would work *except* when it found a particular virus; in which case the checker would ignore the infection.

DEFENSIVE CLASSES

Defensive measures will be divided into three classes. The first class attempts to save an attribute of a program that is initially "pure." Then it will periodically recompute and compare this attribute to check for contamination. A routine in this class **cannot** determine if a program is initially infected. The class is entitled "attribute monitors."

The second class is called "virus detectors." A routine in this class **can** determine whether a file is initially infected. These routines examine the program by itself or in relation to other programs to determine whether infection has occurred. The previous class established a baseline and then checked to ensure that the baseline was still accurate. Both the first and second class detect viruses in a nondynamic way; that is, they do not rely on the *behavior* of the program during execution to work, rather, they rely on the *appearance*.

The third class is "execution limitations." This class imposes a priori controls on executables to *prevent* virus propagation.

After discussing some examples under each class, three measures that will require much innovative work and engineering will be examined.

ATTRIBUTE MONITORS

Checksum Routine

The first routine in this class is a checksum routine. [1, 11] The checksum routine first computes a checksum on a file to be protected. This initial value is stored and access protected[1] if the system itself cannot provide sufficient control, then the checksum is protected by reducing it to hardcopy or writing the value to write-once media. Synchronously, or on demand, the checksum for the file is recomputed and compared

1. One can not simply access protect the file being checksummed in the same manner. The checksum(s) may be protected with the same constraints the system would use to protect the password file. This level of protection cannot be applied to every users' files. Also, the checksum(s) must be protected from any write access, some files may be written to from authorized programs, but not by others. The system may not provide the needed granularity of control.

against the stored value. If the values differ, the SSO knows that the file has been modified. If an authorized change to the file is made, the "initial" checksum must be updated. It is assumed that updates to operational systems will be infrequent and can be closely controlled.

Although this scheme (and others below) may be too costly to implement for every executable or file on the system, it may be used to protect a subset of especially critical programs. This subset should include essential operational routines or software development routines such as translators or compilers, as well as whatever security relevant programs exist, such as the login/password responder or auditor.

It is also possible to store the checksum with the file itself and at run-time recompute and compare it. This method has the advantage of catching an infected file before it executes (and potentially infecting others) but the disadvantage of increasing the execution overhead. This system may be modified by allowing the owner to specify an option at invocation time that would cause the checksum to be recomputed and compared.

Encryption

A method which relies on the pairing of a decryption key with the protected file is a routine that uses public key encryption techniques.

Public key (or asymmetric) encryption uses two keys to encrypt/decrypt, where $K_1 <> K_2$. One of the keys is derivable from the other (say K_2 is derivable from K_1); however, given K_2 by itself, it is extremely hard to derive K_1 (see fig.1). K_1 is referred to

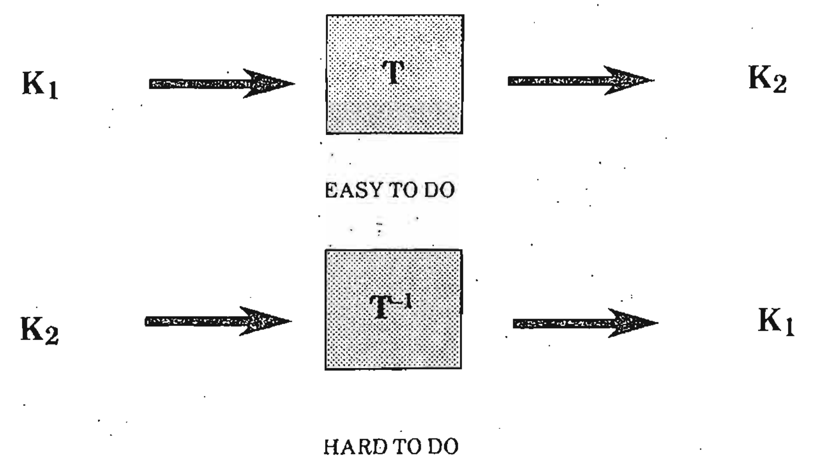


Fig. 1. Public and Secret Keys

as the "secret" key, and K_2 is the "public" key. When the public key is published anyone can use it to encrypt a message which only the holder of the secret key can decrypt (see fig.2), allowing secrecy, or to decrypt a message which purports to be from the holder of the secret key (see fig.3); which, if successful, authenticates the message as being sent from the holder of the secret key. [7, 10]

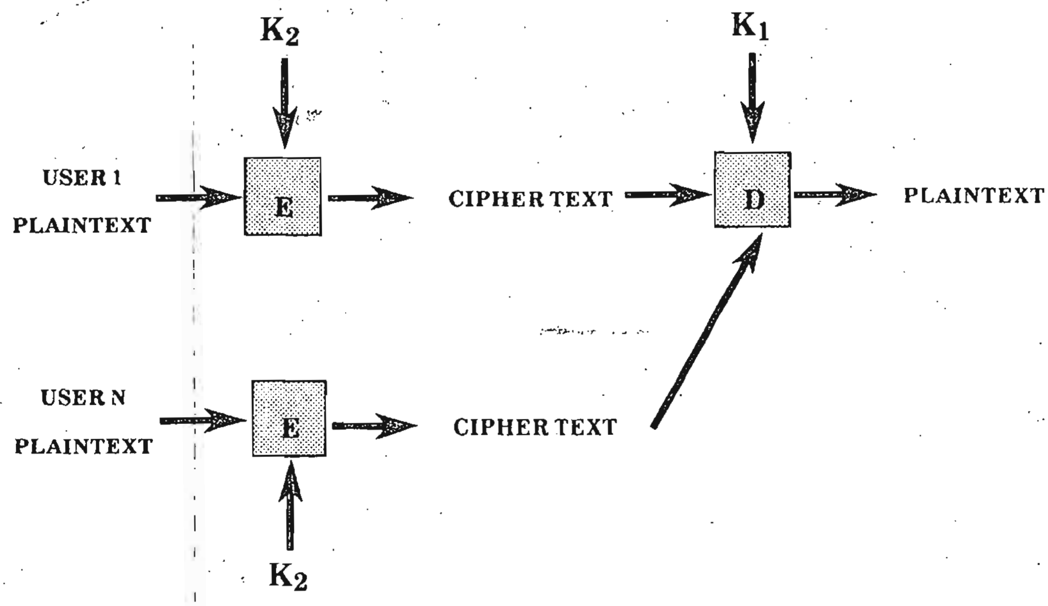


Fig. 2. Private Communication – Anybody Encrypts with K_2; Only Holder of K_1 Can Decrypt

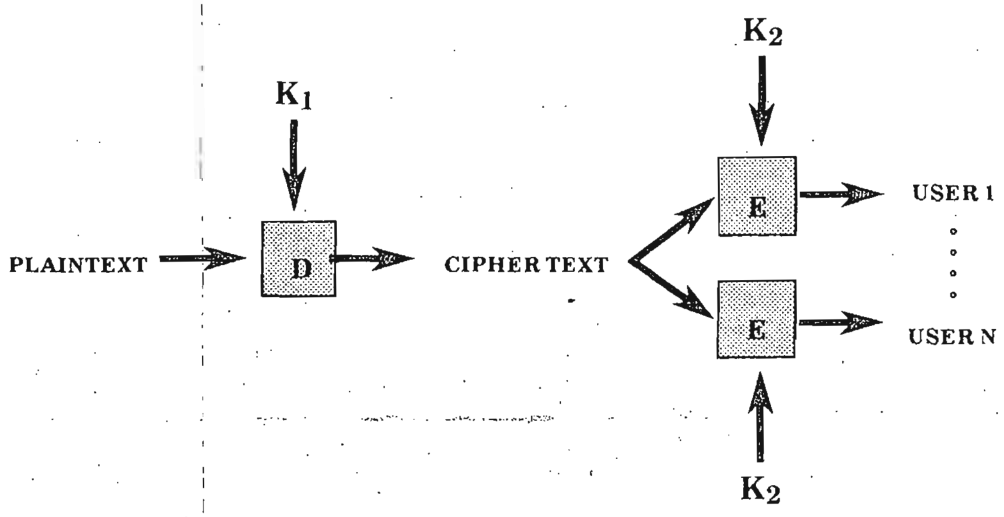
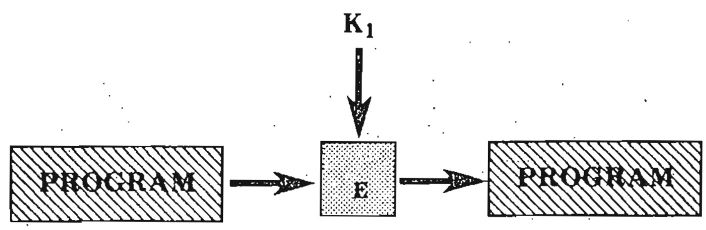
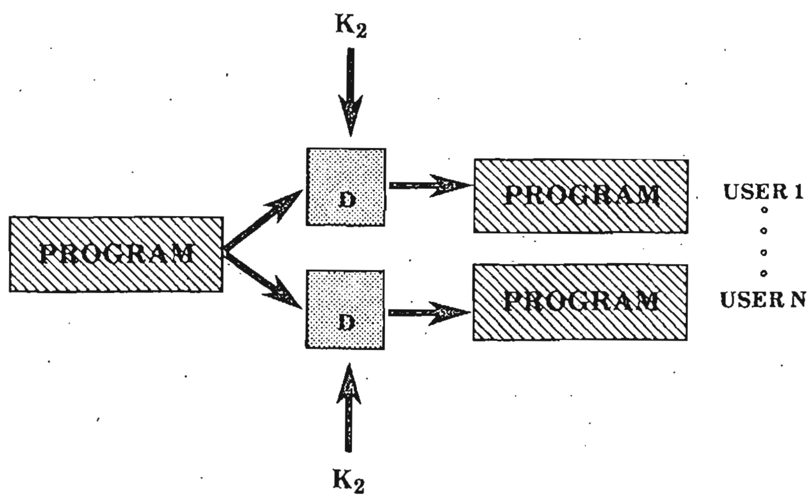


Fig. 3. Authenticated Communication – Cipher Text Decryptable by Anyone with the Public Key

The method involves encrypting an executable using K_1 (the secret key). Then K_1 is destroyed. K_2 (the public key) is published for everyone to use to decrypt the executable file for use.[2] As long as no one has K_1, it is impossible for a virus to infect the executable (see fig.4). The virus cannot write directly to the executable without being decoded to



PROGRAM IS ENCRYPTED WITH THE SECRET KEY



PROGRAM IS DECRYPTED WITH THE PUBLIC KEY
AND PROVIDED TO THE USER

Fig. 4. Programs Encrypted with Secret and Public Keys

gibberish (see fig. 5), because the executable is encrypted and will be decrypted to run. If the virus decrypts the file and then attaches itself and writes the corrupt version back out, the OS will decrypt it into meaningless bits whenever anyone attempts execution[3] (see

2. There could be an operating system service such that whenever someone requests an encrypted program be executed, the operating system would first decrypt the executable with the matching public key.
3. Obviously, the operating system must not have a Trojan horse which allows the decrypting of protected executables to be bypassed. Otherwise, the virus would decrypt the executable, insert itself, and write the executable back to memory, flagging the OS not to decrypt it to execute.

fig.6). The virus cannot use K_2 for encryption purposes, and it cannot derive K_1 to reencrypt the executable properly.

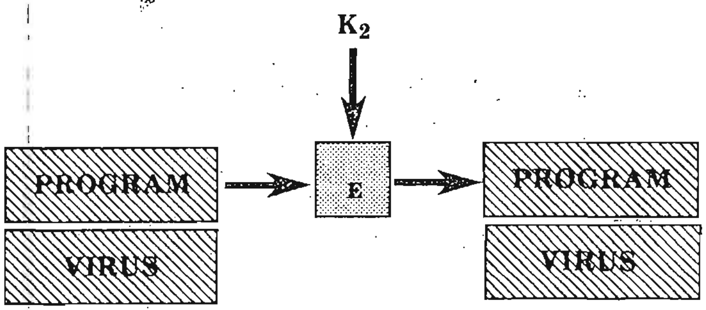


Fig. 5. Plain Text Virus is Decoded into Random Bits when Program is Decrypted to Execute

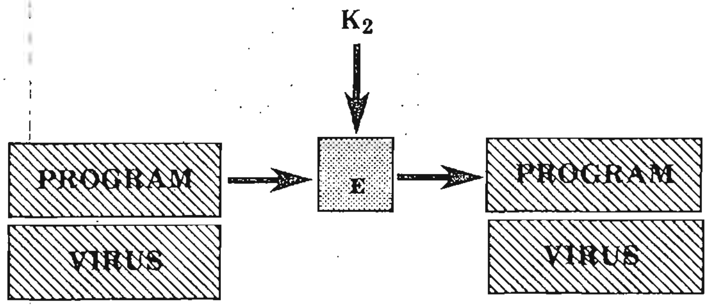


Fig. 6. Plain Text Program/Virus Pair Fares No Better

A key-per-executable or one key for all executables are two alternative methods to use (see fig.7). If key-per-executable is chosen, installation of the encrypted executable and the list of public keys must be protected like the checksums were protected. Otherwise, a virus would decrypt the executable, insert itself, obtain a public/secret key-pair, encrypt the infected version, then write out the new "good" public key into its spot on the public key list. Of course if the other method is used, a new executable or a change to any of the protected ones will necessitate decrypting all executables and then reencrypting them with the new secret key. The new executable cannot just be encrypted and added because K_1 was destroyed. If the key was not destroyed, sufficient precautions must be taken to guard an unauthorized user from obtaining it to undetectably insert viruses.

A compromise method would be to group files and have a key per **n** files. Files which are almost guaranteed not to change could have their own key. This guarantees that no more than **n** files must be decrypted/reencrypted to add a file or change an existing one.

Other routines in this class may focus on saving file characteristics such as length (in bytes), samplings from known positions, or date-and-time-of-last-change. Although a clever virus can "optimize" a program so that the length does not change, such an attack would be detected through the checksum protection method.

DOCID: 3896732

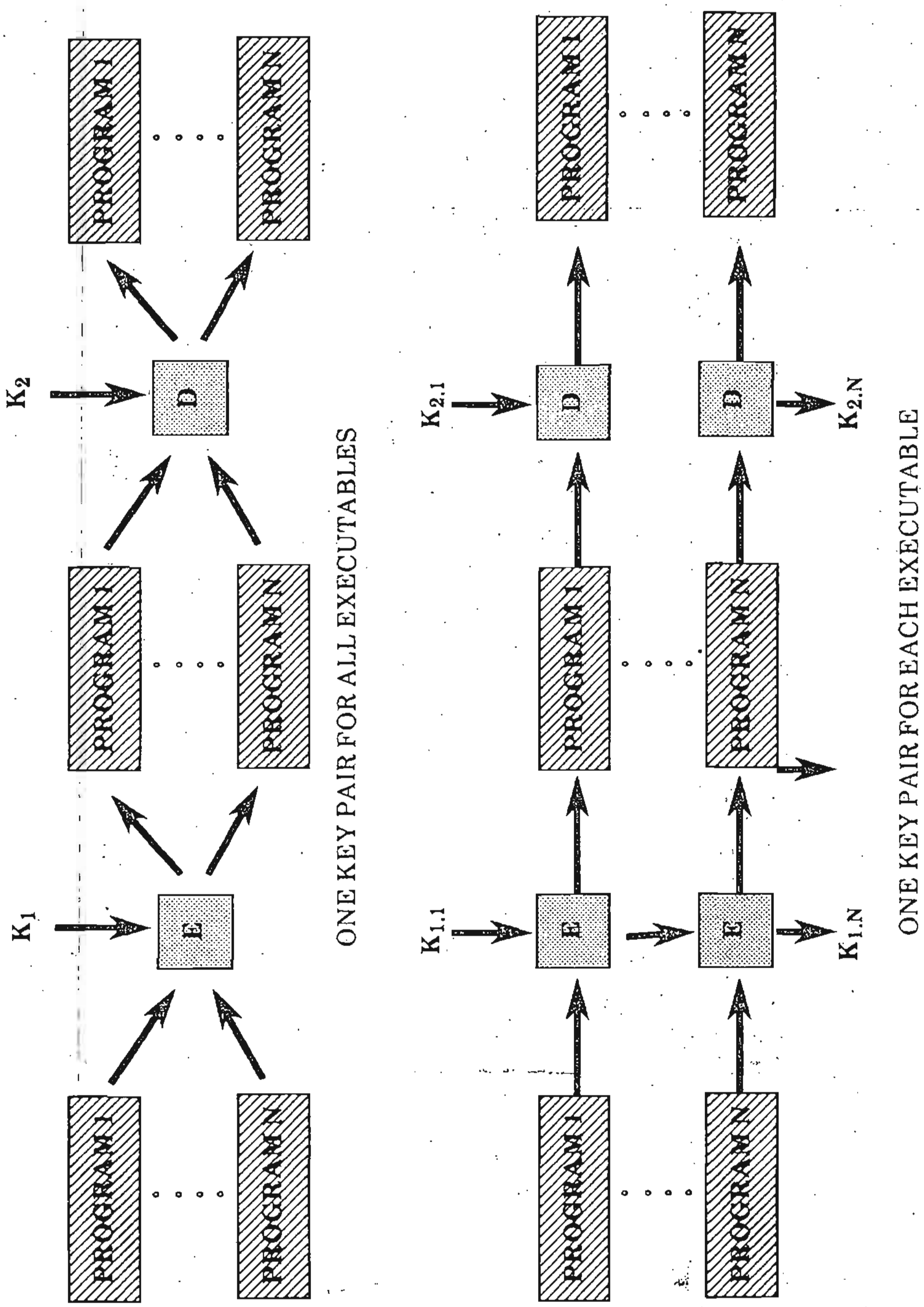


Fig. 7. Key Pairs for Executables

VIRUS DETECTORS

Pre-/Post-fix Testor

The first virus detector relies on the virus always appending itself either to the front or end of a program. A simple virus will likely insert itself at the beginning of a program. This is the simplest action that ensures that the virus will be run before the host program. If the virus appends itself to the end of a program, execution assurance is more difficult. The virus must either depend on control falling through the host to the virus, or code at the top of the host must be changed or added to cause execution of the virus.

The benefit to the SSO is that a simple program can examine a number of files to determine if the first or last **n** bytes are identical. If such a checker determines that several programs have identical pre-fixes, it can assume that a virus has infected them all. The checker must be intelligent enough to discount standard headings or pre-fixes before it starts to examine the code.

Pre- and post-fix checkers will evolve into something more sophisticated. Succeeding checkers must have even more intelligence built in. If a virus knew that only the first 20 bytes were compared, it could create its own "unique" header by inserting a jump instruction to the "real" viral code, followed by 15 random bytes. A "smart" checker could detect that the first few bytes were alike in several different programs, and have the ability to compare an arbitrary number of bytes, even shifting sequences between one file and another.

As viruses are designed to be more sophisticated, checkers will have to rely on statistical techniques to detect viruses. A very smart virus programmer might concoct a scheme where the virus apportions itself up into 20 byte chunks, with a 10 byte chunk being random bits (perhaps get clock value and insert). It would know enough to jump around these random pools and to insert "new" values each time it propagates. That way a checker would not find more than 10 bytes alike out of 20. But if the checker has sampled clean executables and knows that 15 percent of a small target subpart (some section of code minus standard headers) is a reasonable amount of identical code to find in different programs, a 50 percent figure may be enough to trigger an alarm. The virus is then forced to have more random bytes, but that takes up more space which further increases the risk of detection; and one or two instructions would start showing up in some unnatural regularity (the jump instruction). This see-saw battle will continue as checkers and viruses become more sophisticated.

An advantage that the simple pre- and post-fix checker shares with the checksum routine is that it can work on object files quite handily. Humans have enough trouble reading high-level source code, let alone machine code. A program that can examine these types of files can be a useful tool. A disadvantage of the pattern matcher (the "smarter" pre-/post-fix checker) is that it can take even more CPU time than the checksum routine.

Run in background mode, the pattern matcher acts to detect viruses. Once a virus is found, it can be used to find other hosts infected by that virus by looking in the same (relative) place for the same bytes in the other hosts.

EXECUTION LIMITATIONS

The next methods discussed will be used primarily to *prevent* viruses from infecting files, as opposed to the above methods which were used to *detect* infections (except for the encryption method).

Access Controls

These routines try to ensure that executables are never written too directly or, if so, then only by a selected group of files. The easiest way to accomplish this, is to set the privileges for executables to execute only. In Unix the privileges would look like --x--x--x, depending on who was given execute rights. Of course, the file may need to be deleted or the program modified and recompiled, all of which potentially allow infection to occur. And if one program normally writes to an executable, this allows any program (with similar privileges) to write to the executable. There are several methods that could obviate the protection of this scheme. A user could delete an executable and then rename one of his own that is infected with the name of the deleted file. If the user knows the path that the system searches to execute the file, he may be able to insert a like-named file into the path structure such that it is found before the OS gets to the "real" program.

If the file to be protected is a user file it may be more appropriate to allow the user to determine who, if anyone, may execute or even write to it. This may be accomplished by the use of User Defined Domains [12] or domain/type enforced systems [4]. The idea behind these two systems is to allow only needed, prespecified access to files.

These systems can constrain unauthorized access but allow those actions that are otherwise required. If the user has a program which writes into an executable, that fact can be encoded within these schemes as permissable, while still denying other files the right to write into that executable. The granularity of access control may be taken all the way down to a program level. That is, one could specify precisely which programs had access to another. Most popular discretionary access schemes allow the granularity to be specified at the user level; one can indicate which users are allowed access but cannot specify which programs of that user are allowed access and which are not.

The use of the domain/type enforcer can further restrict the ability to contaminate executables by restricting those subjects which have the privilege to create executables. SSOs may wish to tightly control this right, granting it only to compilers or other system routines which take some file and transform it to an executable object. Further, they would have to control who could access these transformers.

This defense narrows the vulnerability of the system greatly and allows the SSO to concentrate his attention and efforts. With protected executables, virus originators are forced to examine other levels of the system for their attacks. One way this can be done is to infect at the source code level. Then the originator has only to corrupt the executable (to force recompilation) or wait until some other change is made and the program re-compiled for the viral propagation to be effected.

Flow Models

Flow model protection can be used as a defense against viruses. One way of implementing flow control would be to "tag" information with a number which represents the number of processes which have "touched" it ("flow distance" [5]). Processes have a preset threshold of "shareability." Once information has been touched so many times it will exceed this threshold and be rejected. This policy, at best, only limits the damage that can be done through a virus which sequentially spreads. If program **A** is infected, every other program in the system can be corrupted from it and thus become infected themselves. This policy limits those infections which are spread through long chains of contamination, where program **A** infects program **B** which infects program **C** and so on. A smart virus could void the flow limit (if it were known) by building the same limit minus one into its propagation trigger.

Another way of limiting flow is to tag information with the names of users who have touched it ("flow list" [5]). Then users may indicate who they wish to share with and also condition sharing on the number of names that appear in the list. If one user knows that the person across the hall regularly brings in freeware, he may not accept any information that has been touched by the freeware user.

Flow model protection is just a way of limiting or conditioning the accesses allowed to executables. Systems that allow users to set the privileges to their executables provide mechanisms for limiting viruses (as noted above), since viruses can only exploit the privileges that they naturally obtain (excepting any security flows that can be actively exploited). If the virus is allowed to change accesses while still under program control, this will not affect them very much. If the OS requires a trusted path[4] connection to change privileges, the system is more secure. Regrettably, flow model protection is a prime example of a security/functionality trade off. The more secure the system in terms of this model, the less sharing (functionality) is possible. Conversely, the more sharing allowed, the less security is added by flow controls.

Labeling

Labeling certain executables at the lowest level –1 [1] on a system which has mandatory security will also prevent those executables from being infected from viruses at higher levels. This works because mandatory security prevents any subject from writing to an object which has a lower classification level than the subject. Thus if the executable has a level which is less than everyone else's, nobody can write to it. But this method requires that each executable be downgraded to be protected, as well as requiring the data that these executables use be at the same lowest –1 level. This is a counter-intuitive method of using levels to protect information. Also, all of the executables downgraded to that level must be virus-free, as they could potentially write to each other.

ROMs

Installation on a read-only device will allow SSOs to use the physical qualities of the medium to prevent writing to executables. Of course, this method incurs problems if the executables must be modified. It must then be possible to write another executable which will be executed instead of the old version. But if this is possible, then it may also be possible for someone to create a contaminated version of the executable and write it out to be executed instead of the "correct" version. The goal of keeping development systems separate from operational ones is much the same here; ROMs are generally "safer." Naturally, the original source code and the compiler must be protected from contamination as well as the transition to executable code and the underlying microcode.

FUTURE DEFENSES

Future defenses (also called second generation defenses) are those which, in general, utilize "artificial intelligence" programming techniques. The methods discussed include

4. "A mechanism by which a person at a terminal can communicate directly with the Trusted Computing Base. This mechanism can only be activated by the person or the Trusted Computing Base and cannot be imitated by untrusted software." DoD Trusted Computer System Evaluation Criteria DoD 5200.28-STD, December 1985.

a program which examines other programs and determines whether malicious software is imbedded within it, a very smart audit program which looks for viral activity in the system activity, and a program which examines other programs and looks for changes in the coding style which would indicate a change of authorship.

Virus Filter

We believe that the task of writing a computer program which would examine other programs and determine whether or not the examinee is infected with a virus is impossible. However, it is possible to detect certain types of viruses in certain environments as well as locate sections of code which look "suspicious." The program that does this must know a lot about what viruses look like and also be cognizant of the system environment within which it is operating.

To write this program, instructions and usage must be researched. Viruses have certain properties which many (if not most) other programs do not. For example, many viruses will need to call system routines to find the names of executable files to infect, whereas many user programs already know which programs that they will access. The implementation of these properties should show up in the instructions of the virus. Moreover, the *clustering* (appearance in close proximity) of these instructions, as in a virus which appends or inserts itself in toto, would be a significant fingerprint. A normal user program may have many of the same instructions that a virus does but is more likely to have them spread throughout.

This program would attempt to locate viral-like code, assign some value as to the perceived likelihood of it being a virus, and then pass that information (and the section[s] of suspicious code) back to a user for any final decision or action.

Auditor

The audit routine would determine when a program(s) was suspicious by examining their behavior. It may sample the global system state to establish whether viruses had infected programs. Certain viruses may be easily detected through their behavior from a single program, where the effects of others may not be seen except through an aggregation. The auditor would also be comparing and analyzing behavior through time, since viruses may construct their triggers to mask their propagation properties. [2]

An auditor which uses templates of user activity and then compares current actions against this template has already been proposed. [6, 8] An authorized user may spend most of his time doing "real" work or computing, where a masquerader may spend an inordinate amount of time browsing through directories or checking statuses. An implementation of this type of auditor could simply sound an alert when the compared difference was great enough, or it could provide more information to the SSO to more closely predict exactly what type of attack is (or has) occurred.

Author Checker

The last second generation viral defense is a program which examines code and then tries to answer questions such as "how many authors does this program have?" and "where does one author's code end and another's begin?" Certain techniques exist to answer these questions for noncomputer-like documents. Such techniques would look at such items as the length of sentences or paragraphs, the tense and inflection, the use and type of certain grammatical characteristics or ploys, not to mention simple handwriting analysis. A program could be constructed to examine source code with similar intentions.

Perhaps it would examine indentation, the use of comments, loop construction, or even characteristics of variable names.

As system routines transform the source code in preparation for machine execution, such analysis would become more difficult, although not impossible. Once the source code is verified to be uninfected, the object code (source code run through the compiler) needs to be tested. Here, certain of the above characteristics cannot be used. The compiler would strip out comments, for instance, but the basic structure of the program would remain. If a program is optimized, that would increase the amount of personal characteristics filtered out (or masked), decreasing the confidence level of finding and identifying differences.

CONCLUSIONS

Certain measures may be undertaken to provide SSOs with some assurance that programs or executables cannot undetectably be infected with computer viruses. These measures rely upon the changes that must occur for infection to take place. Once the protection of the routines and the data that they require (the list of checksums or the list of public keys) is assured, these routines provide a high degree of assurance that viral activity will be prevented or detected. Other, more sophisticated mechanisms are possible but require further research before implementation.

P.L. 86-36

[redacted]

DOCID: 3896732

REFERENCES

[1] [redacted] "The Susceptibility of Multics to Viral Attacks," *Cryptologic Quarterly*, Fall 1985.

[2] [redacted] "Computer Virus Organization: A Definitive Taxonomy and Anatomy of Computer Viruses," *Cryptologic Quarterly*, Fall 1986.

[3] [redacted] "Why Use a Virus Instead of a Trojan Horse?" Informal Technical Note, 1987.

[4] Boebert, Earl and Richard Kain. "A Practical Alternative to Hierarchical Integrity Policies," *Proceedings of the 8th National NBS/NCSC Computer Security Conference*, 1985.

[5] Cohen, Fred. "Computer Viruses," *Proceedings of the 7th National NBS/NCSC Computer Security Conference*, 1984. P.L. 86-36

[6] Denning, Dorothy. "An Intrusion-Detection Model," *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, 7–9 April 1986.

[7] Diffie, Whitfield and Martin E. Hellman. "New Directions in Cryptology," *IEEE Transactions on Information Theory*, Vol. IT-22, No. 6, November 1976.

[8] Halme, Lawrence R., and John Van Horne. "Automated Analysis of Computer System Audit Trails for Security Purposes," *Proceedings of the 9th National NBS/NCSC Computer Security Conference*, 1986.

[9] Karger, Paul and Roger Schell. "Multics Security Evaluation: Vulnerability Analysis," ESD-TR-74-193 Vol. II, June 1974.

[10] Rivest, R.L., A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," *CACM*, Vol. 21, No. 2, February 1978.

[11] [redacted] "Computer Virus Infections," *Cryptologic Quarterly*, Fall 1985.

[12] Smith, Terry A. "User Definable Domains as a Mechanism for Implementing the Least Privilege Principle," *Proceedings of the 9th National NBS/NCSC Computer Security Conference*, 1986.

[13] Thompson, Ken. "Reflections on Trusting Trust," *CACM*, Vol. 27, No. 8, August 1984.

Appendix

Viral Defenses and System Security (U)

The effectiveness of these defenses will vary depending on the security of the system they inhabit. An **A1**[1] system should be able to adequately protect a list of keys, for instance, where a **D**[2] system may not. There are two questions to answer when examining viral defenses and system security: one, is a specific viral defense necessary in an **A1** (or above some level) system? and two, would a defense do any good in a **D** (or below some level) system?

The answer to both is yes. There is already [1] a paper which details a vulnerability in a **B2** level system. It is obvious that without specific mechanisms which can be used to defeat viruses, a system built to an A1 level of security is still vulnerable to viruses. This vulnerability is probably not the risk of disclosure but that of integrity or denial of service. That is, a system built to **A1** with no additional security functionality is susceptible to certain classes of computer viruses. However, it is true that an **A1** system provides the assurance that when viral defenses are added they are much less likely to be subverted.

A **D** level system may still benefit from the addition of viral defenses. There are three ways that defenses may be used. First, it may be announced that they are being installed. Although this would allow a cognizant viral designer to create "defense-resistant" viruses, any *imported* viruses stand a good chance of being caught. Second, defenses may be added surreptitiously. Whereas this incurs the limitations of depending on secrecy instead of strength for security, it is arguably better than announcing its emplacement. The third method requires the SSO to logout all users from the system, perform a shutdown, boot the OS from a physically protected medium, and then perform the check for viruses. Of course, this last is only applicable to those defenses which attempt to ferret out viruses from the appearance of the host program and could not work for those which rely on the programs behavior to detect viruses.

An SSO may find either of the first two methods adequate in a benign environment, but must implement the last if warranted. It may also be reasonable to use method one or two during the month but at the end of the month effect the more secure sweep.

1. See DoD TCSEC DoD 5200.28-STD.
2. Ibid.