Malicious Data and Computer Security

W. Olin Sibert
InterTrust Technologies Corporation
460 Oakmead Parkway
Sunnyvale, CA 94086
osibert@intertrust.com

Abstract

Traditionally, computer security has focused on containing the effects of malicious users or malicious programs. However, as programs become more complex, an additional threat arises: malicious data. This threat arises because apparently benign programs can be made malicious, or subverted, by introduction of an attacker's data—data that is interpreted as instructions by the program to perform activities that the computer's operator would find undesirable. A variety of software features, some intentional and some unwitting, combine to create a software environment that is highly vulnerable to malicious data. This paper catalogs those features, discusses their effects, and examines potential countermeasures. In general, the outlook is depressing: as the economic incentives increase, these vulnerabilities are likely to be exploited more frequently; yet effective countermeasures are costly and complex.

1. Introduction

This paper addresses the increasing vulnerability of computer systems, particularly personal computers (PCs), to attacks based on *malicious data*: that is, attacks employing information that appears to represent input for an application program such as a word processor or spreadsheet, but that actually represents instructions that will be carried out by the computer without the knowledge or approval of the computer's operator. This vulnerability comes from two sources: program features that intentionally treat data as instructions and program flaws that allow data to act as instructions despite the program designer's intentions.

A system that has been subverted by such an attack is, in effect, under the control of a malicious program. Protection against such programs has been the focus of traditional computer security measures: file access control, user/supervisor state, etc. Such measures permit a program's activities to be contained to a limited set of computer resources for which the program's operator is authorized. However, as computers (particularly PCs) are used more and more as extensions of their operators (i.e., as agents), the scope of authorization is greatly increased: a malicious program might, for example, cause a financial transaction using electronic commerce software that is *indistinguishable* by any automated means from a transaction the operator would have authorized—except that there was no such authorization. This increasing difficulty of identifying which computer activities are permissible and which are not increases the risk from *all* types of attacks.

The potential scope of malicious program activity in the PC environment is enormous. On one end of the spectrum are traditional "damage" attacks: virus propagation, destruction of data, compromise of other systems on a network. Another familiar attack involves disclosure: of passwords, of personal data, and so forth; but also of non-computer data such as credit card account numbers; see [1] and [2] for a detailed discussion of such a scenario and of how the disclosed data can be returned untraceably to the attacker. On the

other end of the spectrum are "agency" attacks, in which a computer is made to perform actions of which its operator is wholly unaware, such as electronic purchases, transfers of "digital cash," forged E-mail, and so on.

The two types of vulnerability from malicious data—intentional and unwitting—are quite different and require different approaches to remedy. The unwitting flaws can be fixed (although fixing them is rarely simple), but the intentional mechanisms represent a tension between a system designer's desire to provide features and a user's need for safety.

Furthermore, it is fundamentally difficult to distinguish between data and programs. Although many of the vulnerabilities discussed here rely on supplying actual machine instructions to be executed by hardware, others employ instructions that a program intentionally interprets (such as the PostScript language). Drawing a strict line between data and programs is not sufficient.

Section 1 of this paper introduces the concepts and discusses some potential effects. Section 2 catalogs a variety of intentional mechanisms that can be exploited using malicious data; section 3 describes unwitting mechanisms (i.e., flaws) with the same effect. Finally, section 4 discusses some solution approaches, of which disappointingly few seem to be effective.

2. Intentional Vulnerabilities

With the best of intentions, software developers are responsible for blurring the distinctions between programs and data. Most of the mechanisms cataloged in this section share a common characteristic: they provide a useful capability when used in a benign environment, but they were designed with little or no consideration as to how they might be employed by a hostile party (with the notable exception of Sun's Java language; also see section 4.2).

These mechanisms either permit arbitrary files to be modified, or allow arbitrary programs to be executed, or both. The fundamental property they share is an assumption that the operations that are performed should be performed just as if the user had entered them directly at the keyboard: that is, they are executed within a "user environment" that is shared by all other activities the user performs. The difference is that these packages perform the operations without the user's explicit consent, and often without the user's knowledge. Although some of these features are undocumented, documentation is not the issue: it is simply unreasonable to expect a user to scour a 500-page manual looking for potential security risks before using a program.

Some of the risks posed by these mechanisms can be reduced or eliminated by isolation techniques or by requiring user confirmation. Such solutions, however, reduce the utility of the features and increase complexity for the user. Always requesting a confirmation is little better than never doing so: none but the most paranoid of users will think about it before answering "OK."

2.1. Examples

The following list identifies some of the intentional risks posed by common computer systems and applications:

- **PostScript file I/O primitives.** The PostScript language defines primitives for file I/O. These primitives can be used in a PostScript document to modify arbitrary files when the document is displayed; they extend PostScript's flexibility as a general-purpose programming language at relatively little cost in language complexity, but greatly increase its vulnerability to malicious data. Some PostScript interpreters (e.g., Display PostScript, GhostScript [3]) can disable these primitives (and other non-imaging functions), but doing so is not always simple and can also be seen as inhibiting desired functions.

- **Pathnames in archives.** Arbitrary file pathnames can be stored in common archive formats (i.e., using a maliciously modified `tar` or `PKZIP` program) so that unpacking the archive potentially, can overwrite arbitrary files.

- **Application startup macros.** Much "desktop productivity" software, such as Lotus 1-2-3 and Microsoft Word, provides the ability to run a macro when a data file or document is first opened.

Often, the macro languages include file I/O primitives or even permit execution of arbitrary commands, thus enabling such a "document" to perform arbitrary actions just because it is viewed. Similar features are also present in older applications (for example, the UNIX `troff` document processor has a request for executing a command line while processing a document).

This problem has been understood theoretically for a long time [4][5], but its first exploitations "in the wild" occurred only recently: the Microsoft Word "Concept" Virus [6]. When opened, any document containing this virus modifies the user's environment so that all future Microsoft Word documents will carry it. It is possible to prevent the automatic execution of start-up macros by pressing the SHIFT key while selecting a document, or by disabling the feature globally. Despite publicity, however, it seems unlikely that most users will know how to perform countermeasures, or that they will always remember to do so.

- **Automatic system actions.** A variation on this theme is the feature in certain operating systems (such as "Autoplay" in Microsoft Windows 95), that automatically invokes a program stored on a CD-ROM or other media when the media is inserted into a computer system. This feature may be convenient for media that one trusts (perhaps with a digital signature to provide proof of origin), but it represents a major risk for arbitrary disks. The "read-only" nature of the media is no protection: with the advent of inexpensive CD-ROM writers, writable CD-ROM has become widely used for data interchange.

- **Executable Mail Attachments.** Many modern mail readers provide the ability to attach arbitrary objects to a message—including executable programs. The obvious thing to do with such an attachment is to select it, as one might select a document attachment in order to view it. Although this act is clearly a discretionary one by the user, it is also a very natural one, and the system gives no hint that it might be more dangerous than, say, viewing an image file.

  An early version of this attack was the "CHRISTMA EXEC" virus that propagated on IBM's internal network in 1987 [5]. The mail system did not facilitate this attack: rather, explicit action was required to write out the file and run it, but even so, users almost invariably followed the instructions and did so without suspicion.

- **Executable Web content.** Some Web browsers (e.g., Netscape Navigator, Sun's HotJava, Microsoft's Internet Explorer) offer the capability of downloading and executing parts of a web page locally. In some cases (e.g., Java programs) the local execution is strongly constrained for security reasons and relatively safe; in others (e.g., Explorer and downloaded OLE controls) there are no restrictions on the code being executed.

3. Unwitting Vulnerabilities

The previous section dealt with purposeful software features that provide an opportunity to introduce malicious programs. On the one hand, it is unfortunate that those features were designed with little attention to risk; on the other hand, it is good that they can be identified, for it is possible to imagine countermeasures that would contain them.

There is another class of attacks that does not have those properties: attacks based on program flaws or inadequate design. Here, the designers did not intentionally create a problem; rather, by failing to provide sufficiently robust software, they unintentionally enabled the problem to occur.

Such unintended risks depend on the same basic properties as the intentional ones: programs run in a user environment that is shared by other programs. To date, the exploitations of these risks have involved primarily multiuser systems, where the environment being attacked is privileged. However, privilege is not necessary for these attacks to be useful; they can introduce malicious software into the environment of an unprivileged user just as effectively.

3.1. Examples

A few examples of these attacks include:

- **The Morris Worm `fingerd` Attack.** As reported in [7], the "Morris Worm" delivered an executable program over a network connection to

the `fingerd` program and, by overflowing an internal buffer where the request was stored, caused it to be executed.

This attack, part of the incident that brought the Internet to a halt in 1988, relied on the presence of a fixed-size buffer inside the `fingerd` program. The request received from the network was read into the buffer without a check on its length. Because the request was larger than the buffer, it would overwrite other data, including the return address stored in the stack frame. By changing the return address to designate a location within the request string, the attack forced a transfer of control to the attacker's supplied program stored in the request string. When executed, this small program established a run-time environment and carried out the rest of the attack. This attack was very sensitive to initial conditions: it was developed only for one widely-used operating system, and it depended on the stack frame layout in the `fingerd` program, the containing process environment, and so forth. However, given the source code to the `fingerd` program and a laboratory system on which to experiment, it appears that the attack was engineered with only a few days of effort.

This attack on the `fingerd` program was the first widely demonstrated example of forcing an application program with no intentional programmability features to execute machine instructions supplied by an attacker. It required only a modest engineering effort to create, and it was wildly successful. It breached internal security in multiuser computer systems, which is not normally an issue in personal computers, but it pointed the way for similar attacks in different environments.

The Netscape Navigator attack. In late 1995, a flurry of security problems with cryptography and random number generation in Netscape's Navigator program was reported in the mainstream press. Shortly afterward, some members of the "Cypherpunks" group discovered a buffer overflow flaw in the then-current version of Navigator. This attack is notable because it is directed at a personal computer program, where the objective is not to breach multiuser security but to cause a personal computer to act under control of malicious software.

Following techniques similar to those used in attacking the `fingerd` program, an over-length host name in the HTML source of a Web page can be made to overflow an internal buffer and cause an attacker's program to be executed. Although Navigator's parsing of the HTML language itself turned out to be fairly robust, the routine that converted a host name to an Internet address was found to have a fixed-size buffer that could be overwritten by an oversize fabricated host name, and this led to the ability to cause Navigator to branch to an arbitrary execution address.

- **Overflow `syslog` buffer.** Like `fingerd`, the `syslog` program used for system logging in UNIX systems was found in 1995 to be vulnerable to buffer overflow[8]. The attack technique and the objective (run a program in a privileged process—in this case, `sendmail`) are essentially equivalent to the `fingerd` attack. Although much attention has been paid to eliminating such vulnerabilities in the intervening seven years, the continual emergence of examples suggests that it is very difficult to eliminate the problem systematically.

3.2. Scope of Vulnerability

These examples represent the tip of the iceberg. What sort of programs are vulnerable to such attacks? Any program that misbehaves when given bad input data is a potential victim. If it crashes or dumps core when given bad input, it can probably be made to misbehave in a predictable manner, too. If a program's internal data structures can be damaged by invalid input, this often indicates that its control flow can be affected as well—potentially leading to the ability to execute caller-supplied instructions.

Indeed, software developers typically make no claims that any application programs are bulletproof when faced with invalid input data, because such misbehavior is seen only as an inconvenience to users—after all, "garbage in, garbage out." The risk that it would serve as a way to introduce malicious software into the user environment is rarely, if ever, considered.

Examples of such program misbehaviors include:

- The UNIX utility `uncompress` often dumps core when processing invalid compressed input data. The `gunzip` or `PKUNZIP` decompression utilities may have similar problems.

- Programs that process complex data formats, such as MPEG streams, Rich Text Format, or PostScript may produce wildly incorrect output or misbehave when given malformed input data, indicating that their internal data structures have been damaged.
- Application software such as Microsoft Word or Lotus 1-2-3 often fail catastrophically when given a damaged input file, again likely indicating damage to internal data structures.
- File import software (e.g., Microsoft Word reading WordPerfect documents) often is even more fragile than the software processing an application's native format, making it more likely to harbor vulnerabilities. This fragility may occur because these parts of the software are less thoroughly tested, because they are written by third parties, or perhaps because the formats being converted are themselves undocumented.

Although none of these program behaviors is known to the author to have been exploited, the possibility clearly is present, and further investigation is warranted.

The basic problem is that increasingly complex and ill-defined data semantics are difficult to process, so it is no surprise that application software fails when presented with bogus input data. Software that responds correctly to all incorrect input is far harder to create than software that simply responds correctly to correct input.

Application software development contrasts with the design philosophy of network protocols, where a basic assumption is that all possible bit sequences will be encountered, so all must be handled reasonably. It is partly for this reason that implementations of complex network protocols often have a long development period before they are truly robust.

3.3. Exploitation Techniques

The known exploitations for invalid input data are known primarily because they were used to breach *system* integrity in multiuser systems. These attacks are more difficult to construct than those that exploit known software features. They require constructing executable programs "by hand," tailored to run in a largely unknown environment. Although doing so is awkward, it is by no means beyond the abilities of a moderately sophisticated attacker.

The most fruitful exploitation technique seems to be buffer overflow: provide more data than a program expects, so that it will overwrite internal storage for program variables or addresses, and the program will misbehave in a deterministic—and possibly controllable—manner. Another technique involves providing data with out-of-range values. Such inputs can cause calculated branches to go to unintended destinations, or can cause values to be stored outside of array bounds. All these vulnerabilities offer the potential to cause a transfer to the attacker's supplied executable code, from which point the attacker can do anything that the attacked program can do.

It is important to note that malicious data representing machine instructions does not require arbitrary binary values. For example, the Intel 80x86 opcode set and the MS-DOS executable file format permit a valid executable program to be constructed entirely from printable ASCII characters. Such a program can perform arbitrary actions when executed, yet it requires no special transfer mechanism—it can be delivered as ordinary unformatted E-mail. The first known example of such a program[1][9] contains a small executable header that decodes the rest of the program text—transferred in UNIX `uuencode` format—into a memory buffer, then transfers to it.

Of course, a successful exploitation is quite difficult. It is necessary first to understand how the program is misbehaving, then to determine what input data will create predictable misbehavior, then to craft input data that contains an appropriate attacking program. The analysis stages require an understanding of the software that comes most readily from source code, but as most personal computer applications are not distributed in source code form, techniques such as disassembly and emulation are required. Experimentation plays a critical role, also.

---

1. For example, when saved as a text file, the following five lines of text (Copyright © 1994 by A. Padgett Peterson) form an executable MS-DOS program that prints a short text message:

```
XP[@PPD]5‘P(f#(f((f?5!QP^P_u!2$=po}l=!!rZF*$*$ =0%GF%!!%PP$P$Ps-
%gmZ$rl6lW$rm6mWlVl6m=ldmAlv%fmvmB$Vm6lW$Vm6mWl6m6m=ld%ylVmqlJmq
lRmqlNmqlBlWl6m6l/m’l/m3mWl7m7mrm4mql:lXl7m7mAl2lYl1m6lZl6m2mPm&
mPl&%o%[$‘$U%^$\$\%bl&%Y$X%[%\$Z%Yl&$q%b%\$a%^l&%W$^%^$\%‘l&$p%‘
$a$a%^%‘l&$‘%[%‘%Yl&$p%b$\$\%b%Y$|l&$[%‘%W%Yl&%b$Xl&%z%z${l$pp
```

4. Solutions

Solving the problems posed by unsafe or malicious data requires fundamentally different techniques from traditional computer security approaches, because the objective is different. Traditional approaches focus on isolation and protection of resources: that is, on preventing activity whose nature is known in advance. Protection from malicious data, on the other hand, requires distinguishing among program activities that are in accord with the operator's intent and those that the operator would not want to occur. This problem—of divining the operator's intent—seems unlikely to be solved.

Addressing the malicious data problem seems instead to require a return to fundamentals:

- Avoid building unsafe features into computer programs. This would reduce the incidence of "intentional" problems.
- Use programming techniques and languages that encourage construction of robust programs. This would reduce the frequency and severity of "unwitting" vulnerabilities.

Aside from these techniques—which would represent a fundamental change in commercial software development—there are relatively few external, system-level techniques that offer much hope for improvement. The problem of safe execution of mutually suspicious programs remains a difficult problem in computer system design [10]. Even if such solutions were readily available, it is unclear whether users could be expected to exercise the necessary discipline to protect themselves. After all, it is not unreasonable to expect that computer systems, like other complex appliances, should be safe to use without detailed understanding of their internal operations.

4.1. External Solutions

This section briefly discusses some of the solution techniques that can be applied externally to contain or reduce the effects of malicious data:

- **System isolation.** A computer system that is not connected to a network and used for only one purpose is unlikely to be vulnerable to malicious data, and even if attacked, would not be able to do much damage. This approach is, by default, what has protected most personal computers—but increasingly these computers are networked and used for many activities.
- **Virtual machine environments.** Suspect or untested software can be run under control of a virtual machine monitor; this approach in effect is the same as running many isolated systems. As long as the virtual machines remain isolated, this technique contains the problem effectively, but as soon as data is transferred among them, they become vulnerable. Maintaining the necessary isolation requires a generally infeasible degree of discipline on part of the operator. It is not reasonable to expect personal computer operators to maintain a constant state of suspicion.
- **Automated filters.** Known examples of malicious data can be detected and filtered out. For example, Secure Computing Corporation's Sidewinder product can analyze all traffic coming across a network firewall and reject patterns that it recognizes as malicious (such as virus-infected executables or malformed HTML documents). Similarly, some virus detection products are now capable of detecting the known examples of the Microsoft Word virus described in section 2.1.
- **Capability-based operating systems.** Capability systems were a major focus of operating system research in the 1970s [11]. In principle, such systems can safely contain the effects of malicious data more effectively than virtual machine monitors because they exercise control over resources at a finer grain. However, capability systems have the same drawback of requiring considerable discipline to use effectively and also require special hardware and/or programming techniques to use effectively. Although a few capability-based systems were introduced in the 1980s (from companies such as BiiN, Intel [12], and Key Logic [13][14]), these were not commercially successful, and they are no longer actively marketed.
- **Dynamic monitoring.** The virus protection field deals with some of the problems that can be caused by malicious data. One of the techniques developed for virus protection is dynamic monitoring of program activity: pattern matching of program opera-

tions against acceptable types of operations [15] (e.g., files to which a program is expected to write, as opposed to those to which it should not). A user can be presented with the opportunity to permit or deny such actions.

- **Digitally signed executables.** Public-key cryptography can be used to sign application software and certify is as "safe" as judged by some certifier—where one of the "safety" properties would be that the application cannot be corrupted by malicious data. This technique has been proposed as a way of marking executable Web content as safe to use. Unfortunately, it simply moves the burden of assurance to a certifier without making the analysis any more tractable; it also places an unreasonable burden on users, who must decide which certifiers are trustworthy. Because even major mass market application software appears susceptible to malicious data attacks, it is not clear what value this type of certification technique could add.

4.2. Internal Solutions

In the long term, internal solutions seem to offer more hope for addressing these problems:

- **Safe application design.** Defense against intentional mechanisms that permit malicious data to be introduced requires that application designers pay more attention to system safety. That is, they should avoid features that introduce unconstrained programmability into an application.

- **Safer languages.** The most important defense against malicious data is programs that are more resistant to it. An important part of this resistance involves use of languages and environments that are themselves robust, with bounds checking, pointer validation, memory management, and so forth. The Java language [16] is one such; others (e.g., Ada and Python [17]) also have extensive robustness features.

  The Java language is particularly interesting because of its program validation mechanism and its utility for enforcing type safety rules to contain features that could introduce intentional vulnerabilities. Unfortunately, current versions of Java do not live up to the promise of safe execution. Although some of the problems reported in [18] and detailed in [19] result from simple implementation problems related to specific execution environments, two design flaws have been reported that breach the type safety of the language itself. The lack of a formal basis for Java's claimed type safety and security properties is troubling.

- **Non-von Neumann computer architectures.** The principal mechanism for unintentional malicious data flaws is the ability to execute data: an attacker supplies malicious instructions as data and causes a branch to them. If instructions are clearly distinguished from data, the attack is much harder. Unfortunately, the prevalent use of interpreters, sometimes with multiple levels of interpretation, makes this approach unworkable on a hardware level.

- **Sheer complexity of applications.** One reason that these attacks have not been more widely perpetrated is that they are *difficult*, because much application software is not available in source code form and is extremely complex. An attacker must understand a great deal about a program's internal operation to be able to fabricate malicious data that will cause predictable types of misbehavior. Although not a defense one would like to rely on, it has been reasonably effective.

5. Conclusions

The general outlook for malicious data as a computer security problem is unclear. The potential vulnerabilities are legion, but exploitation poses great practical difficulties. Unfortunately, defense also poses great difficulties, and as the economic incentive for creating malicious software increases, it seems likely that attackers will attempt to exploit these vulnerabilities.

The most effective technical solutions appear to require pervasive change in the way that computer software is built. The near-term alternatives all involve giving up many of the "general-purpose tool" properties that make personal computers so effective in the first place.

6. References

[1] Garfinkel, Simson, "Program shows ease of stealing credit card information," San Jose Mercury News, 29 January 1996

[2] Sibert, Olin, "Risks (and lack thereof) of typing credit card numbers" Risks-Forum Digest, volume 17, issue 69, 7 February 1996, available by anonymous FTP from `ftp.sri.com` in `/risks/17/risks-17.69`

[3] Computer Emergency Response Team, *CERT Advisory CA-95:10*, 31 August 1995, available by anonymous FTP from `info.cert.org` in `/pub/cert_advisories/CA-95:10.ghostscript`

[4] Hoffmann, Lance J., *Rogue Programs: Viruses, Worms, and Trojan Horses*, Van Nostrand Reinhold, 1990

[5] Ferbrache, David, *A Pathology of Computer Viruses*, Springer Verlag, 1992

[6] Computer Incident Advisory Capability United States (Department of Energy), *CIAC Alert G-10a: Winword Macro Viruses*, available from `http://ciac.llnl.gov/ciac/bulletins/g-10a.shtml`

[7] Eichin, Mark W., and Rochlis, Jon A., "With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988," in *Proceedings, 1989 IEEE Computer Society Symposium on Security and Privacy* page 326–343, 1–3 May 1989, Oakland, California

[8] Computer Emergency Response Team, *CERT Advisory CA-95:10*, 19 October 1995, available by anonymous FTP from `info.cert.org` in `/pub/cert_advisories/CA-95:13.syslog.vul`

[9] Peterson, A. Padgett, personal communication, 15 February 1996. Mr. Peterson reports, "I described it in an internal Martin Marietta memo on security threats presented in 1988 as 'theoretically possible' but did not construct a working prototype [until 1994]."

[10] Rotenberg, Leo J., *Making Computers Keep Secrets*, Ph.D. Thesis, Massachusetts Institute of Technology, 1973, published as MIT Project MAC Technical Report TR-115, February 1974

[11] Levy, H. M., *Capability-based Computer Systems*, Digital Press, Maynard, Massachusetts, 1984

[12] Organick, Eliott I., *A Programmer's View of the Intel 432 System*, McGraw-Hill, New York, 1985

[13] Hardy, Norman, "KeyKOS Architecture," *ACM Operating Systems Review*, Volume 19, Number 4, October 1985

[14] Rajunas, Susan, et al., "Security in KeyKOS," in *Proceedings, 1986 IEEE Computer Society Symposium on Security and Privacy*, 7–9 April 1986, Oakland, California

[15] Pozzo, Maria, and Gray, Terrence, "Managing Exposure to Potentially Malicious Programs," in *Proceedings of 1986 National Computer Security Conference*, 15–18 September 1986, National Bureau of Standards, Gaithersburg, Maryland pages 75–80

[16] Sun Microsystems, Inc., *Java Language Specification*, available as `http://www.javasoft.com/JDK-beta-2/psfiles/javaspec.ps`

[17] van Rossum, Guido, *Python Reference Manual*, Dept. AA, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, available as `http://www.python.org/doc/ref/ref.html`

[18] Computer Emergency Response Team, *CERT Advisory CA-96:07*, 29 March 1996, available by anonymous FTP from `info.cert.org` in `pub/cert_advisories/CA-96.07.java_bytecode_verifier`

[19] Dean, Drew; Felten, Edward; and Wallach, Dan, "Java Security: From HotJava to Netscape and Beyond," in *Proceedings, 1996 IEEE Computer Society Symposium on Security and Privacy*, 6–8 May 1996, Oakland, California