An Operating System Analog to the Perl Data Tainting Functionality

Dana Madsen[1]

danammadsen@hotmail.com

Abstract

Several recent Internet security incidents, such as the Melissa and ILOVEYOU virus outbreaks, demonstrate fundamental shortcomings in the security provided by general purpose operating systems deployed in the Internet computing environment. These incidents reveal that the security controls on downloaded software and data are often discretionary, relying on user vigilance and competence for their effectiveness. This does not provide sufficient protection. To address this deficiency, we propose extending operating system functionality to support a *tainting file system* that associates a level of trustworthiness with each software and data file stored on the computer. We also propose developing a set of mandatory trustworthiness policies that govern how trustworthiness is assigned and how files of a given trustworthiness are handled, along with implementing operating system support for such policies. Our scheme will be developed as an overlay for existing Unix environments that support loadable kernel modules. No source code modifications to the operating system will be required. It is anticipated that our approach could be ported to the Windows 95/98/NT environments or directly integrated into a future operating system. The tainting file system is motivated by the data tainting functionality in the Perl programming language and is intended to complement other approaches to computer and network security, including software wrappers, sandboxing, digital signatures, firewalls, and intrusion detection systems. It is envisioned to be one element of a network defense in depth strategy that encompasses all levels of computer and network architecture.

Keywords: Tainting, Integrity, MAC, Mandatory Access Control, Operating System Security, Virus, Mobile Code

1 Introduction

Several recent incidents demonstrate fundamental shortcomings in the security provided by general purpose operating systems deployed in the Internet computing environment. For example, over the past several months, numerous Microsoft Office macro viruses, sent as email attachments, have propagated due to user ignorance or carelessness. In other cases, malicious

[1] The author thanks Dr. Peter Neumann of SRI and the reviewers for their insightful comments on drafts of this paper. All remaining deficiencies are the *sole responsibility of the author*.

software has masqueraded as data files (e.g., the recent ``JPEG" attachment that granted hackers access to ICQ passwords [12]), or apparently benign software has contained Trojan horses (e.g., *picture.exe* [6]). Several vulnerabilities have been attributed to the execution of mobile code on Web browsers[2], and it may not always be apparent to a user that a Web site contains embedded mobile code, much less embedded hostile code.

In each of the above cases, security is discretionary, relying on user vigilance and competence. It is up to the user to scan email attachments for viruses. Users must judge whether downloaded software, such as a ``cool" screen saver, is safe. They must do the same when visiting Web sites. System administrators have limited recourse in these situations.

Relying on discretionary security mechanisms will not maximize security in the Internet computing environment. Users are downloading software and data far more frequently than ever before, from a constantly changing and potentially unbounded set of sources with varying trustworthiness. Hence, there are more opportunities for the introduction of malicious software or data. Even worse, mobile code, and the macro programming functionality found in office productivity suites, blur the distinction between software and data. Executable content embedded in a data file may be invoked without a user's knowledge or consent. *The chances for a security blunder are increasing.*

Unfortunately, existing mandatory security mechanisms in mainstream general purpose operating systems fall short of what is needed. These mechanisms are geared towards protecting users from each other, on defending against unauthorized access by outsiders, and on restricting use of administrative functions by regular users.[3]

General purpose operating systems do not fully protect users from ``foreign" (i.e., downloaded from potentially untrustworthy Internet sources) software and data files. These files are stored under the ownership of the user who downloads them. There is no distinction at the operating system level between those files and files of known trustworthiness. Foreign software could potentially execute with the full permissions and identity of the user who downloaded it. Foreign data could cause applications resident on the user's machine to execute in an undesirable manner.

2 The Tainting Concept

To rectify the above deficiencies, we advocate the development of a *tainting file system* that could be overlayed onto existing mainstream operating systems, without requiring source code

---

[2] In October 1999, Karsten Sohr of the University of Marburg in Denmark detected a flaw in the Microsoft Java Virtual Machine (JVM) implementation that could allow an attacker to illicitly cast objects of one Java type to another Java type [1]. This weakness could be exploited to compromise, modify, or destroy data on a victim host. In April 1999, Sohr uncovered a similar vulnerability in JVM implementations from other vendors for several other operating systems.

[3] For example, authentication mechanisms are intended to prevent logins from unauthorized users. Memory is protected, so that one process cannot affect memory belonging to another process. A user may restrict access by other users to his files.

modification to those operating systems. Specifically, we propose that standard file attributes, such as access permissions[4], be augmented with one more attribute that reflects the level of trustworthiness associated with a file. Foreign software and data could then be marked as being potentially tainted with malicious content.

It is further proposed that a mandatory trustworthiness policy be supported at the operating system level. The support mechanisms should be capable of being overlayed onto existing mainstream operating systems, without requiring source code modifications to those operating systems. The mandatory trustworthiness policy will specify the level of trustworthiness to be assigned to each file, based for example on the conditions under which the file was created. In some cases, the file may have been created from data directly downloaded from an untrustworthy site on the Internet. Alternatively, it may have been created by an application program based on another, untrustworthy file used by that application. In both cases, the newly-created files may not be fully trustworthy. The trustworthiness policy will restrict the handling of files based on their level of trustworthiness. These restrictions will apply to the execution of downloaded (or otherwise untrustworthy) binary files and to the use of downloaded (or otherwise untrustworthy) data by active processes. It is also desirable to have some means for changing the trustworthiness attribute associated with a file based on conditions outlined in the trustworthiness policy. Support for the mandatory trustworthiness policy should be flexible, allowing it to be tailored by system administrators to specific situations.

The tainting file system concept proposed in this paper is intended to protect against the inattentive or incompetent user who, without malicious intent, introduces malicious content into a file system. Although our approach provides some protection against deliberately malicious users or intruders, this is a positive side effect and is not our present focus.

We have attempted to minimize the impact of tainting on pre-existing (non-tainting) operating system trustworthiness by avoiding source code modifications to the underlying operating system. Nonetheless, we have introduced two new, situation-dependent issues with potential relevance to system security.

First, existing applications, migrated to a tainting environment, may not react well to constraints imposed based on their trustworthiness. This is particularly true for non-robust software that doesn't always check for errors returned by system calls.

Second, we have introduced new, trust-based ``channels of influence", which might conceivably be exploited in specific and limited cases to undermine system security. For example, a process is affected by the trustworthiness of the files it reads. Hence, that process can be maliciously constrained by illicitly reducing file trustworthiness. Opportunities for trust-based attacks are reduced by applying the principle of least privilege when specifying user permissions.

Note that the tainting mechanism itself could become a target for attack. How it is protected depends on the particular operating system into which it is incorporated. In section 4, we describe issues related to the protection of our tainting mechanism under Linux.

---

[4] read, write, execute, etc.

At first glance, the functionality provided by a tainting operating system may seem redundant to existing application-level protection schemes. For example, the Java virtual machine allows restrictions to be placed on the execution of Java applications. However, a tainting mechanism at the operating system level provides a unified approach to trustworthiness that covers *all* application software and that can be tailored to a specific context. It also provides defense in depth for the cases where application-level protections fail.

Realistically speaking, the tainting mechanism *should* be developed in such a way that it can be added to existing mainstream operating systems, such as Unix and Windows NT, without requiring source code modifications to those operating systems. This may not be as secure as a new operating system designed from the ground up with such features. However, it should still be possible to ``raise the bar'' for the sophistication required in malicious code to damage a system. Additionally, an overlay on existing operating systems can provide an excellent prototype environment for future operating system designs.

3 Related Work

Several technologies that partially address some of the technical threats associated with a highly-connected, Internet computing environment are available either as research prototypes or commercial products. However, most of these technologies function at the application level, and all could be strengthened by an operating system that included tainting functionality. Below we look at the relationship between tainting and the Perl programming language, software wrappers, sandboxing, digital signatures, firewalls, and intrusion detection systems. We also consider how tainting relates to other trusted operating system design concepts, most of which are not found in mainstream operating systems.

The tainting file system extends to the operating system level the data tainting mechanism found in the Perl programming language [14]. The underlying principle in Perl is that data derived from outside a program may not be used to affect something else outside the program.

The proposed tainting file system is also an extension of (and is intended to complement) the Generic Software Wrapper concept developed by Fraser et al [5] at TIS Labs at Network Associates, Inc. Generic Software Wrappers can provide protected, non-bypassable, kernel-resident software extensions for augmenting security without requiring modifications to operating system or application source code. A key advantage of their approach is that distinct types of security enhancements, tailored to *specific* system resources and *specific* users, can be readily integrated into a unified framework. Augmenting their work with a tainting file system would provide security tailored to a specific combination of system resources, user trustworthiness, *and* data trustworthiness.

Sandboxing, which provides a constrained execution environment for untrustworthy software, is closely related to the tainting file system concept. It is found in several commercial products, including the Java virtual machine, SurfinShield Corporate 4.5 Desktop Security Software [9], and eSafe Protect Desktop 2.1 from Aladdin Knowledge Systems [13]. However, these products provide application-level sandboxing functionality. As Loscocco et al [8] point out, they cannot protect *themselves* against tampering by other applications and instead rely on the local file system for this protection. In contrast, the tainting file system resides at the operating system level.

Note that tainting and application-level sandboxing are not mutually exclusive. For example, tainting could provide another layer of protection for cases where application-layer mechanisms fail (such as the recent JVM security vulnerabilities uncovered by Sohr [1]). Ideally, tainting and application-level sandboxing would both be elements of an overall defense in depth strategy that addresses security at all architectural levels of a computing environment.

A complementary technique to sandboxing is the use of digital signatures to establish the trustworthiness of downloaded mobile code. Digital signatures could be integrated into the tainting file system, perhaps as one criterion for establishing levels of file trustworthiness.

Some might argue that firewalls can (or should) protect against untrustworthy executable files. However, firewalls are not fully effective in filtering out malicious mobile code at this time. The tainting file system can contain the effects of malicious code passing through a firewall. By hardening the operating systems of hosts behind a firewall against attack, tainting contributes to an overall network defense in depth strategy.

In any case, Linger et al [7] point out that in the long term, firewalls may become an increasingly inadequate protection mechanism due to a long term shift in computing paradigms. They suggest that in the future, most computing resources will be resident on unbounded network infrastructures, controlled by a multitude of service providers, and combined according to dynamic architectures. It will therefore be more difficult to isolate resources and establish security perimeters for deployment of firewalls. A tainting file system is well-suited to this environment, since it is in part motivated by the trend described by Linger et al.

Tainting functionality could supplement role-based access control (RBAC) schemes. In RBAC implementations, access decisions are based on the roles that individual users have in an organization. Associated with each role is a set of allowable operations. The definition of an operation can capture complex, security-relevant details or constraints that cannot be realized by a simple mode of access [11].

A principle RBAC objective is protecting the integrity of information by constraining who can perform which functions on which information, under which conditions [3]. Consequently, roles in an RBAC environment may be trusted to varying degrees. It might be worthwhile to ensure that only trustworthy data are accessed under some trusted roles. It could also be useful to consider less trustworthy the data created or modified under less trusted roles. In some cases, the level of trust accorded to the *binding* of a role to an individual might be dynamically varied. For example, it could change based on the trustworthiness of data or remote hosts accessed by the individual while in that role. All the preceding mechanisms require some means of tracking the trustworthiness of data, which our tainting concept provides.

Functionality developed to implement a tainting file system could also be adapted to support a military security policy as formalized in the Bell-La Padula Confidentiality Model. For example, a classification attribute might be added to files, and mandatory security policies that restrict the handling of files based on classification could be developed. However, tainting and classification are distinct concepts.

The tainting file system is in part intended to preserve the integrity of data on a system from modification caused (directly or indirectly) by untrustworthy software and data. For example, it

could be used to realize integrity properties outlined in the Biba Integrity Model. However, tainting functionality provides security as well as integrity protection. For example, it can be used to prevent an untrustworthy software application from sending data over a network to remote computers.

The LOMAC software package, developed by Fraser [4], provides low water-mark mandatory access control, one of the integrity policies defined by Biba [10]. LOMAC closely resembles our tainting concept. LOMAC assigns integrity levels to objects (files and other entities represented by inodes) and to subjects (processes); two levels are defined in the existing implementation. The primary function of LOMAC is to ensure that (possibly viral) data does not flow from low-integrity objects to high-integrity objects. Secondarily, it ensures that (possibly compromised or malicious) subjects do not interfere with higher-level subjects. LOMAC is implemented for the Linux environment, using loadable kernel modules to preserve compatibility with existing Linux distributions.

The distinction between the tainting file system and LOMAC is one of emphasis and implementation. The tainting file system emphasizes the vulnerability of the inattentive or incompetent user in the highly-connected and unbounded Internet computing environment. LOMAC emphasizes the threat of malicious users, compromised root daemons, and viruses. The tainting file system is first concerned with restricting actions taken using the identity and permissions of a user that are based on untrustworthy data. LOMAC is first concerned with strict-sense integrity, preventing the flow of data from low-integrity objects to high-integrity objects. The tainting file system enforces a low water-mark policy for subjects *and* objects. As an object is manipulated by various subjects, its trustworthiness can fall, and subjects reading from that object encounter increasing restrictions on their activities. Whether and how these restrictions are lifted is situation and policy dependent. However, as explained in section 4, a ``sticky bit'' associated with each object can be used to selectively enforce a low water-mark policy for subjects only. LOMAC enforces a low water-mark policy for subjects only, meaning that object integrity never decreases. Finally, we have devoted significant attention to defining the role of a tainting file system in a network defense in depth strategy and in defining trustworthiness levels and mandatory security policies which are well suited to the unbounded computing environment described by Linger et al [7].

Application-level isolation, as discussed by Fayad et al [2], bears superficial resemblance to our operating system level tainting file system. In both cases, data are categorized (according to integrity in Fayad et al; according to trustworthiness in the tainting file system). Additionally, entities accessing data (users in Fayad et al; processes in the tainting file system) are classified according to trustworthiness.[5]

However, Fayad et al focus on *isolating* untrustworthiness, while we allow more dynamic *interaction* between varying levels of trustworthiness. Fundamental to the approach of Fayad et al is a requirement that high integrity be maintained for certain data. Hence, such data are copied when accessed by a suspicious user, and those copies are discarded if that user is later shown to

[5] Note that in the tainting file system, trustworthiness is primarily associated with files and processes, rather than users.

be untrustworthy. In the tainting file system, there is no hard requirement for maintaining a high level of file trustworthiness. Rather, the emphasis is on tracking how file trustworthiness changes. Hence, there is no need to copy files when they are accessed by an untrustworthy process.[6]

In general, an analog to the tainting file system could be implemented above the operating system level, either in a high level language (as in Perl) or an application-specific context (e.g., a database). However, the integrity of the tainting mechanism would then be reliant on an untrustworthy underlying file system. This is the same weakness noted above and by Loscocco et al [8] in connection with sandboxing. Alternatively, our tainting concept might be augmented with some of the ideas proposed by Fayad et al.

Finally, note that the tainting functionality is a form of dynamic labeling, where the trustworthiness label of a file changes over time based on conditions surrounding file creation or modification. One of the earliest dynamic labeling schemes was found in the ADEPT-50 processor developed by Weissman [15], though ADEPT-50 labels reflected object classification.

4 A Baseline Tainting File System Design

We present an initial design for adding a tainting file system to Linux, based on loadable kernel modules. We also outline an example tainting-based security policy oriented towards the Internet computing environment. In general, a system administrators should be able to tailor the policies they use to their specific situations.

Files are assigned a level of trustworthiness based on the conditions under which they are created and/or modified. We succinctly characterize these conditions by assigning a level of trustworthiness to each executing process. The trustworthiness of a process is used to set or update the trustworthiness of all files created or modified by that process. The trustworthiness of a process is initially established based on the trustworthiness of the binary files from which that process executes. It is dynamically affected by the data sources from which the process reads, including files, network connections, and interprocess communication (IPC) mechanisms. Figure 1 illustrates our approach.

The execution of active processes is constrained based on their trustworthiness. Which actions and system resources are denied depends on the particular trustworthiness policy in use.

Our approach to tracking trustworthiness and constraining active processes is based on intercepting certain system calls. The set of intercepted system calls can be grouped into three classes, based on whether the system calls affect file or process trustworthiness or constrain an active process. Taken together, the set of intercepted system calls in each class effectively wrap each active process in the system, as illustrated in figure 2.

---

[6] However, as explained in section, our ``sticky bit' provides optional support for maintaining a specific degree of file trustworthiness.

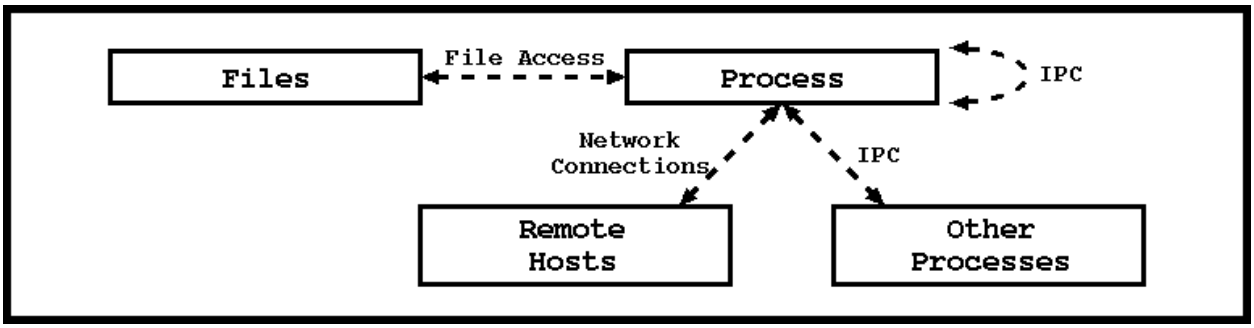

Figure 1: Conditions affecting file trustworthiness.

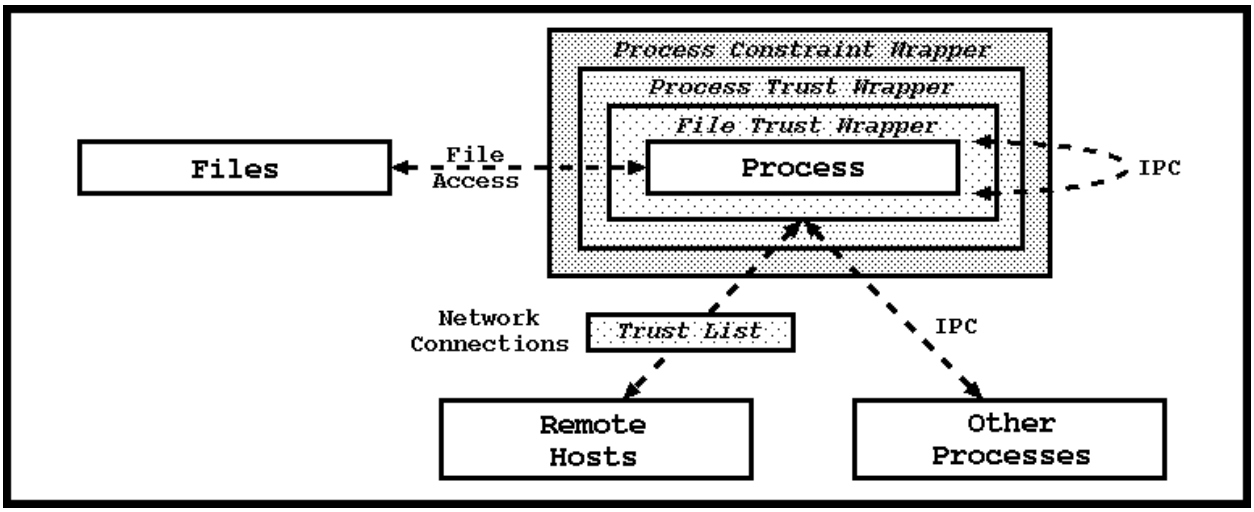

Figure 2: Tainting is implemented by effectively wrapping all active processes.

Tainting additionally requires several kernel memory resident data structures. A *File State Table* tracks the trustworthiness of each open file and which processes are reading or writing which files. A *Process State Table* tracks the trustworthiness of active processes and which files are being accessed by each process. Entries in the file and process state tables have a ``sticky bit'' field. When the sticky bit is set, any action that would lower the trustworthiness of the corresponding file or process is denied. Finally, the *Process Trust Profile* specifies how execution of processes at each level of trustworthiness is constrained.

File trustworthiness is a persistent file quality, in the same manner as the more conventional read, write, and execute permission attributes. Somehow the trustworthiness attribute must be overlayed onto the existing file system. This might be done by adding a fixed-size header field to each file or by storing the attributes in a single, well-guarded data file. The best option has not yet been determined.

Our example trustworthiness policy defines four levels of trustworthiness that are applied to both files and executing processes, as illustrated in figure 3. When a process executing at trustworthiness level *x* creates or modifies a file, then the maximum trustworthiness of that file is *x*. When a binary file of trustworthiness level *y* is executed, the corresponding process is initially placed at trustworthiness level *y*. Both file and process trustworthiness may change over time

based on system conditions, according to a low water-mark policy for subjects and objects [10]. We also associate trustworthiness with connections to remote hosts.

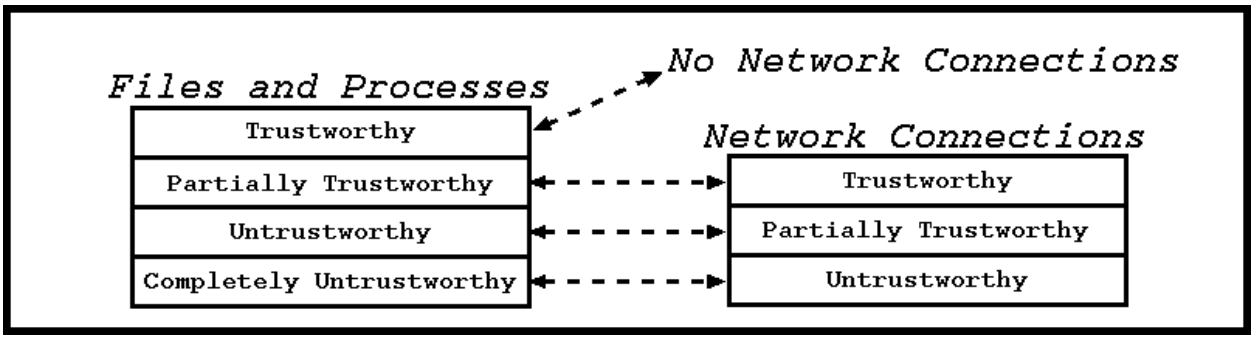

Figure 3: Trustworthiness levels for files, processes, and network connections.

Certain security restrictions are enforced based on file and process trustworthiness. A completely untrustworthy file should never by read by any executing process, nor should it be executed. Untrustworthy or partially trustworthy files do not execute with, or cannot be used to cause effects with, the full permissions and identity of any user. Trustworthy files are created by user-owned processes that do not read from network connections. Such files can act or cause actions with the full permissions and identity of the user to whom they belong. Process restrictions are analogous, except that completely untrustworthy processes are halted immediately, and trustworthy processes do not read data from any type of network connection.

We do not fully trust processes that read from network connections, nor do we fully trust data derived from network connections. Network connections (either TCP or UDP) are assigned a level of trustworthiness identical to trustworthiness of the remote host. Untrustworthy network connections should not be allowed. If a remote host becomes untrustworthy, then all existing network connections to that host are immediately considered untrustworthy. The set of untrustworthy and trustworthy remote hosts is explicitly specified by the system administrator. By default, hosts not on those lists are considered partially trustworthy.

Our trustworthiness policy enforces several additional, specific constraints on active processes, based on the trustworthiness of those processes, as described below:

Trustworthy Process: Any file created by a trustworthy process will, by default, have its world permissions cleared, though this can be explicitly overridden.

Partially Trustworthy Process: A partially trustworthy process has never interacted with a partially trustworthy or untrustworthy remote host, even indirectly via IPC or reading from a file created by virtue of a past interaction. Therefore, the chances that this process will cause unauthorized or unacceptable damage to data on the local system are small. On this basis we define the following three constraints*: (i)* We allow the process to have the same access to any file or directory stored on a user account that ``world'' would have. By default, any file or directory created by a partially trustworthy process will have a mode of 0777, though a partially trustworthy process could certainly specify more restrictive permissions. However, a partially trustworthy process cannot undo any stricter permissions that it might set. *(ii)* The chances that the process will maliciously exfiltrate sensitive data to a remote host are also small. We do not allow data to be written by a partially trustworthy process to an

untrustworthy remote host. We allow unconstrained writing of data to partially trustworthy and trustworthy remote hosts. *(iii)* A partially trustworthy process should not have *suid* capability.

Untrustworthy Process: There is some potential that an untrustworthy process will cause unauthorized damage to the local system or exfiltrate data from the local system to a remote host. However, there is still some value in having it execute in a ``read-only" mode. Therefore, an untrustworthy process should not be able to create, delete, write, or change the permissions of any file whatsoever. An untrustworthy process should not be able to write data to an open network connection. Additionally, it should not be able to use *suid* functionality.

Completely Untrustworthy Process: No constraints are applied because the process is automatically and immediately halted when it becomes completely untrustworthy.

Figure 4 shows an example scenario for the tainting design and the trustworthiness policy defined in this section. The left side of the figure is a list of actions taken by two processes *P1* and *P2* that affect a file *F*. The right side of the figure shows the resulting trustworthiness of those entities, with lighter shades representing lower levels.

In general, system administrators should be able to specify the trustworthiness policies used. For instance, the example policy described in this section may be too stringent for some environments. Certain public information accessed over the network may actually be trustworthy. There might be a need to access files imported from legacy systems or stored on remotely-mounted file systems that do not support tainting.

Recall that the tainting mechanism itself may be targeted by an adversary. In our prototype Linux implementation, there are two minimal requirements for protecting the tainting mechanism: secure the loadable kernel modules used to implement tainting and defend the file trustworthiness attributes. The security of both depends on the security of access to root-level privileges. In the first case, anyone who gains root access can disable tainting functionality or replace it with his own Trojanized modules. In the second case, the attributes will be stored in the file system, since they are persistent, and by definition root has access to the entire file system. Use of smart cards might alleviate both situations by better differentiating between authorized and unauthorized root users. For the present, we duck this issue by noting that tainting provides additional protection for an inattentive or incompetent user and is not specifically intended to defend against malicious users.

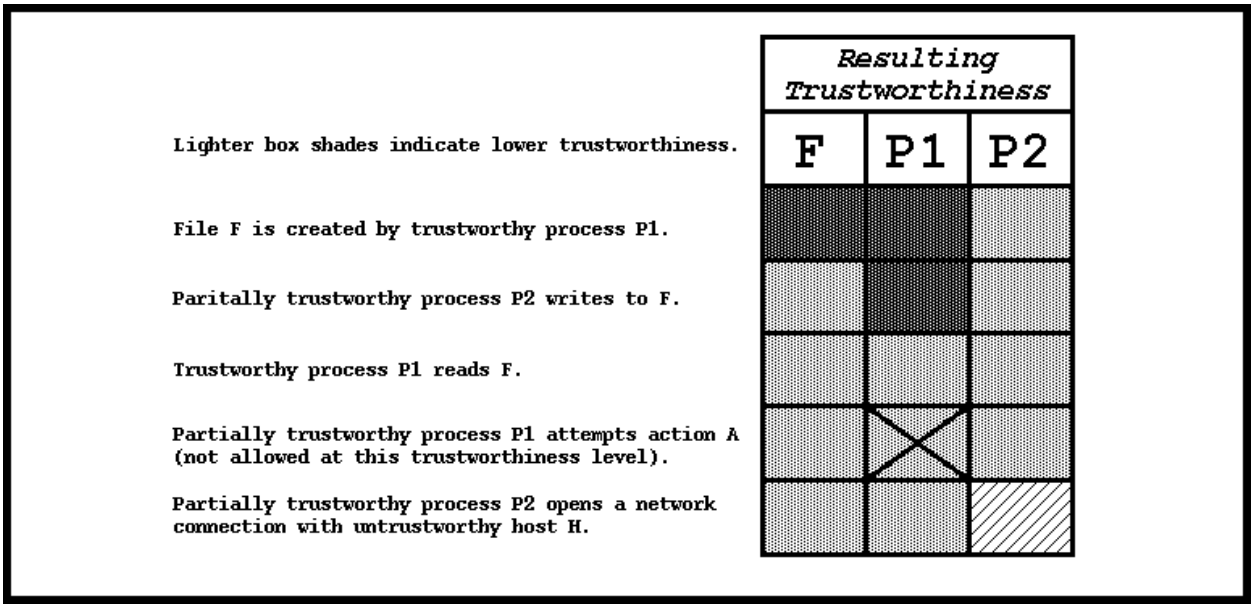

Figure 4: Example scenario for the trustworthiness policy defined in this section.

5 Conclusion

We have proposed the idea of a tainting file system to address deficiencies in general purpose operating systems deployed in the highly-connected, Internet computing environment. This scheme adds a new file attribute that denotes the level of trustworthiness of a file. It can be applied to compiled or interpreted software files or to data files. The scheme also includes a mandatory trustworthiness policy. This policy associates a level of trustworthiness with all files based on the conditions under which they were created or modified. It enforces limitations based on the assigned level of trustworthiness, including limitations on executable files and limitations on programs accessing data files of a given trustworthiness. Finally, it controls changing the level of trustworthiness associated with a given file.

The tainting file system concept complements other approaches to computer and network security, including software wrappers, sandboxing, digital signatures, firewalls, and intrusion detection systems. It is envisioned to be one element of a network defense in depth strategy that encompasses all layers of computer and network architecture.

We have expanded the tainting file system concept into a baseline design. Realistically speaking, the baseline design should be implemented in such a way that it can be added to existing mainstream operating systems, such as Unix and Windows NT, without requiring source code modifications to those operating systems. We have suggested an approach for doing so in the Unix environment, though actual implementation is still underway. It should also be possible to develop an implementation for the Windows 95/98/NT environment.

Several design issues remain to be considered. One is compatibility of tainting with existing applications. The basic tainting concept presented here might be generalized in several ways. For example, the number of trustworthiness levels need not be four. File trustworthiness might be established based on other criteria besides the trustworthiness of all processes modifying that

file. Trustworthiness of processes might be set based on a combination of current actions and past history.

References

[1] Dirk Balfanz and Edward Felten. Secure internet programming news. http://www.cs.princeton.edu/sip/history, October 1995.

[2] Amgad Fayad, Sushil Jajodia, and Catherine McCollum. Application-level isolation using data inconsistency detection. In *Proceedings of the 15th Annual Computer Security Applications Conference, 1999.*

[3] David Ferraiolo and Richard Kuhn. Role-based access control. In *Proceedings of 15th National Computer Security Conference, 1992.*

[4] Timothy Fraser. Lomac - low water-mark mandatory access control. Technical Report 0766, NAI Labs, 1999.

[5] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening cots software with generic software wrappers. *IEEE Security and Privacy, 1999.*

[6] Data Fellows Inc. http://www.datafellows.com.

[7] R. C. Linger, N. R. Mead, and H. F. Lipson. Requirements definition for survivable network systems. http://www.cert.org.

[8] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrel. The inevitability of failure: The flawed assumption of security in modern computing envronments. http://www.jya.com (from NISSC98).

[9] Finjan Software Ltd. Surfinshield corporate 4.5 desktop security software: Technical whitepaper. http://www.finjan.com, 1999.

[10] Terry Mayfield, J. Eric Roskos, Stephen R. Welke, and John M. Boone. Integrity in automated information systems. Technical Report 79-91, National Computer Security Center, 1991.

[11] NIST. An introduction to role based access control. NIST CSL Bulletin, *http://hissa.ncsl.nist.gov/rbac*.

[12] SecurityPortal.com. http://www.securityportal.com, September 1999.

[13] Aladdin Knowledge Systems. http://www.esafe.com.

[14] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl.* O'Rielly and Associates, Inc., 2nd edition, 1996.

[15] Clark Weissman. Security controls in the adept-50 time sharing system. In *Proceedings of the 1969 AFIPS Fall Joint Computer Conference, 1969.*