Comprehensive Integrity Protection for Desktop Linux

Wai Kit Sze and R. Sekar
Stony Brook University
Stony Brook, NY, USA

ABSTRACT

Information flow provides principled defenses against malware. It can provide system-wide integrity protection without requiring any program-specific understanding. Information flow policies have been around for 40+ years but they have not been explored in today's context. Specifically, they are not designed for contemporary software and OSes. Applying these policies directly on today's OSes affects usability. In this paper, we focus our attention on an information-flow based integrity protection system that we implemented for Linux, with the goal of minimizing usability impact. We discuss the design decisions made in this system and provide insights on building usable information flow systems.

1. Introduction

Information flow has been proposed 40+ years ago for integrity protection. While these systems [2, 5, 9, 12, 3, 6, 7, 4] can provide principled protections against malware, they are rarely used in practice. This is because these systems often require (1) changing OSes to enforce information flow policies, and/or (2) rewriting applications to handle new security failures. These requirements limit practical adoption of information flow systems. In this paper, we discuss the techniques in making them usable. Our discussion is based on PIP [11], an information flow system focused on usability. Increased usability results from increased application compatibility, combined with the ability to preserve user experience on contemporary OSes that do not support integrity protection. We also propose some demo scenarios to show how our techniques work.

In Section 2, we give a brief overview of PIP. In Section 3, we focus on information flow policies that can improve usability. We describe a technique to distinguish between different types of files, and applying different policies based on these types. We also discuss how to deal with programs that need to handle high and low integrity data simultaneously. We summarize the implementation of PIP in Section 4. Demo scenarios and evaluation are presented in Section 5 and 6 respectively. Related works are presented in Section 7. The paper concludes in Section 8.

The main goal of this demo paper is to illustrate techniques to improve usability of information flow based integrity protection.

SACMAT'14, June 25–27, 2014, London, Ontario, Canada.

ACM 978-1-4503-2939-2/14/06 ...$15.00.
http://dx.doi.org/10.1145/2613087.2613112.

2. Architecture Overview

Basic information flow policy for preserving system integrity is no-write-up and no-read-down. Low integrity processes are not allowed to modify high integrity files while high integrity processes are not allowed to read low integrity files.

PIP is a portable information flow system that has been implemented on Linux and BSD systems. It consists of three main components: existing multi-user support from OSes, a library, and a helper process. We provide an overview on the system architecture here. Details about the system can be found in [11].

PIP relies on existing multi-user support for labeling integrity and enforcing no-write-up on low integrity processes. For simplicity, we consider two integrity levels: high integrity (benign) and low integrity (untrusted). Low integrity subjects are run with a newly created userid called *untrusted*. By default, these untrusted processes have no permission to modify user data. This provides a robust enforcement of no-write-up policy.

No-read-down policy is enforced by a library. Although library enforcement is subject to bypass attacks, note that we are applying it only to benign processes. Such processes run benign code from trustworthy sources, and operate on benign data. It is therefore reasonable to assume that they will not actively seek to subvert policy enforcement, and hence library-based enforcement is reasonable.

To obtain a fully working system, a handful of administrative and utility applications need to be *trusted,* i.e., they should be able to consume untrusted input while continuing to have high integrity. Section 3.5 discusses these applications.

The basic policy outlined so far focuses on preserving integrity, but not usability of programs. For instance, untrusted processes may not be able to read user data as they are treated as a different user on the system. This is where the helper process component comes in: it runs with the privileges of the current user — also called *logged user*, and provides a mechanism for untrusted processes to request controlled access to the user's files. It will allow user files to be opened in read-only mode, and creating files in user directories.

By default, our system only protects integrity, and does not target confidentiality. However, it is possible to enhance the helper process and the rest of the system to enforce stricter policies.

3. Improving usability

3.1 Fine-grained policies

Traditional UNIX file permissions does not provide sufficient granularity that enables processes of the *untrusted* user to have read

[†]This work was supported in part by grants from NSF (CNS-0831298, CNS-1319137) and AFOSR (FA9550-09-1-0539).

access to all the files of the logged user. This deficiency can be rectified using Access Control Lists (ACL) on Linux. However, this interferes with normal use of file permissions, e.g., a manual permission change can negate the ability of untrusted code to access user files. Moreover, many applications are unaware of ACLs and may run into compatibility problems, e.g., they may be using `stat` or `eaccess` to check permissions. Finally, ACLs can only allow or deny accesses, but cannot support more complex policies, such as redirecting access to a shadow copy of a high-integrity file. For this reason, PIP uses library and helper process to fine-tune the coarse-grained policy captured by the DAC permission.

3.2 Policy Inference for File Accesses

The basic information flow policy focuses on preserving integrity, but not usability. PIP improves usability by supporting multiple policies. Instead of simply allowing or denying an access, PIP also supports shadowing. Shadowing is crucial for improving usability of untrusted processes: some files need to be read and written whenever a program is executed. Labeling these files as high integrity will prevent untrusted processes from using these files. Labeling them as untrusted will prevent benign processes from using them. Using only allow and deny policies break either benign or untrusted processes. Shadowing maintains two independent copies of the same file and transparently redirects accesses to the file copy based on the integrity of the process. This allows programs to be used as both benign and untrusted.

PIP does not apply shadowing to resolve all file access denials. This is because shadowing of data files causes confusion. Without realizing it, a user may accidentally invoke an untrusted application to edit a high integrity data file. With shadowing, this will work, but subsequently, if the user tries to view the file, a benign viewer will be redirected to the benign, unmodified copy, and the user is left confused as to what happened to the edits he/she made. For this reason, PIP uses the following technique to determine if a file represents a data access or a preference file, and applies different policies.

3.3 Determining file type

PIP relies on file type to determine what policies to apply. Figure 1 shows the file types in PIP. Code and configuration files are easy to identify based on how they are accessed. However, preference files and data files cannot be identified via permission. An important distinction that PIP relies on is how these files are accessed. Data files are specified explicitly by users, while preference files are accessed by programs implicitly.

File type	Permission	Access
code	R–X	-
configuration files	R– –	Implicit
preference files	RW–	Implicit
data	RW–	Explicit

Figure 1: File types in PIP

Implicitly accessed files are those that are not specified explicitly by users. PIP identifies these files by exclusion: the set of files accessed by the application but are not explicitly specified.

Typically, users specify data file accesses in these ways:

- arguments when executing the program
- environment variables
- file names returned by a file selection widget, which captures file names selected by a user from a file dialog box

Identifying explicitly accessed files can be posed as a taint-tracking problem with taint sources listed. Taint propagates in PIP according to the following rule: when a tainted directory is opened, all the file names inside the directory are marked as tainted. Every file access made by the application is then matched against the set of tainted values to identify explicitly accessed files. Implicitly accessed files are those that do not match with the tainted values. PIP relies on Aho-Corasick [1] algorithm to track and identify tainted values efficiently.

With information on whether a file is accessed implicitly by programs or explicitly by users, different policies can be applied to serve users better. While this section focuses on describing how this "implicit-explicit" technique can be used to infer file types and hence be used for shadowing policy, this technique can also be applied to limit the trust on programs that need to handle high and untrusted simultaneously (Section 3.5).

3.4 uudo Inference

By design, PIP does not allow subjects to change their integrity labels. PIP is a Privilege Revision on Invoke (PRI) system and it does not suffer from the problem of self-revocation [4]. Hence PIP avoids causing failures that applications cannot handle gracefully. In PIP, a benign (high integrity) process can read and write into high integrity files and write into untrusted files, but not read from untrusted files. On the other hand, an untrusted process can read and write untrusted files, but it can only read high integrity files. Since it does not allow subject integrity to change, a process has to decide ahead of time what integrity level it wants to be. `uudo` is a program that PIP provides to let users (and benign subjects) execute a program as untrusted.

Instead of requiring users to specify the integrity level for every program executed, PIP relies on `uudo` inference to infer what integrity level a process should be executed with. It involves predicting what files a program will use when executed: if users want to use a program with untrusted files, the process should be executed with userid *untrusted* to avoid violating the security policy. On the other hand, if no untrusted files are involved, the process can be executed with high integrity. By design, an incorrect choice of integrity level only affects usability, but not system integrity.

PIP determines the required integrity level of a process based on a simple technique. If any of the arguments or environment variables corresponds to a low integrity file, PIP would execute the program as untrusted. We found that this simple technique is very helpful because data are typically specified as program's input arguments. For instance, a lot of command line programs are not interactive and input files need to be specified as arguments. Even for GUI programs, they also accept input arguments to specify files to be opened. File explorers (e.g., nautilus) then act as front ends to interact with users: double-clicking on the icons cause them to be executed with file path arguments.

This technique, however, fails if files to be opened depend on the interaction with the programs. Since files to be accessed are not known when deciding process integrity, PIP cannot infer the use of `uudo` if high integrity programs are invoked without arguments.

3.5 Trusted programs

Applications that are designed to handle data from various sources simultaneously require special handling in information flow systems as this violates the basic information flow policy. We provide a discussion on how these applications are handled in PIP, so as to balance integrity protection and usability.

Web browsers.

We designated `Firefox` to protect itself from network inputs and inputs from local files selected using a file dialog by the user. Files selected by user using a file dialog are mainly used for uploading. These files are identified by the "implicit-explicit" mechanism described in Section 3.3, preventing `Firefox` from using untrusted files as non-data inputs. To ensure that downloaded files are associated with the right integrity labels, we developed a `Firefox` addon, which uses a database to map domains to integrity levels.

As a second alternative, we dedicated an instance of the web browser for benign sites. Using policies, the benign instance can be restricted from accessing untrusted sites. In PIP, we manually defined a whitelist of benign sites. A better alternative would use whitelists provided by third parties. Instead of blocking users from visiting untrusted sites, we can invoke the untrusted browser instance to load the pages directly.

Email clients.

Email clients introduce untrusted data into the system through message headers, content, and attachments. Our approach is to trust the email reader to protect itself from untrusted sources. However, attachments are given labels corresponding to the site from which the attachment was received. We developed an addon for `Thunderbird` for this purpose. However, the current email protocol (SMTP) does not protect against spoofing. To provide trustworthy labeling, we could either rely on digital signatures (when present), or on the chain of SMTP servers that handled the email. Such spoof-protection has not yet been implemented.

Software Installation.

Our system relies on correct integrity labeling when new files are introduced into the system. Of particular concern is the software installation phase, especially because this phase often requires administrative privileges. Solutions have previously been developed for securing software installation, such as SSI [8]. We are implementing an approach similar to SSI to protect the software installation phase and to label files introduced during the installation. PIP can then enforce the policies at run time based on the labels.

X-Server and DBus.

Malicious X-clients can abuse X-server APIs to harm other X-clients. One approach we provide is to redirect untrusted X-clients to `Xephyr`, a nested X-server. Another alternative uses X-security-extensions to designate untrusted processes as *untrusted X-client*, to restrict/disable accesses to certain X resources. Since this option trusts the X-server, it is not as secure as the first alternative, but integrates smoothly in terms of user experience.

We are also trusting DBus and some service daemons to handle requests from untrusted processes. For instance, we allow untrusted processes to send desktop notifications and play sounds. Same as X-server, we trust these applications to handle requests from untrusted processes, but not for consuming untrusted files.

File utilities.

Files belonging to different integrity levels co-exist. Utilities such as `mv`, `cp`, `tar`, `find`, `grep`, and `rm` may need to handle files of high integrity and untrusted *at the same time*. We designated these file utilities as able to protect themselves when dealing with untrusted data such that their functionalities can be preserved.

Instead of trusting these utilities to consume any untrusted data, PIP can further reduce the set of files by relying on the "implicit-explicit" technique described in Section 3.3. When users invoke a command, data files are specified as input arguments[1].

[1]When globbing is used in shell command, the shell process will expand it to the set of file names matching the pattern.

	LOC				
	C		header		Other
	Ubuntu	+PCBSD	Ubuntu	+PCBSD	Both
Shared	2208	130	737	27	39
helper	703	16	106		
uudo	68	52			
library	2206	163	492	30	74
Total	5185	361	1335	57	113

Figure 2: Code complexity on Ubuntu and PCBSD

A side effect of making these utilities as trusted is that their outputs have high integrity labels. This is not desirable for applications like `cp` and `tar` as integrity labels on original files are lost. We solved this problem by setting appropriate flags to preserve the integrity information. This is relatively easy as the integrity information is encoded as group ownership in PIP.

4. Implementation

We implemented the system on Ubuntu 10.04. A prototype is also developed for PCBSD 8.2. Figure 2 summarizes the implementation complexity. +PCBSD corresponds to the additional number of lines of code required in order to support PCBSD. Shared corresponds to code shared across multiple components.

5. Usage/Demo Scenario

Here are some scenarios to illustrate the usability of PIP.

Watching a movie.

We opened a movie torrent from an untrusted website. `Firefox` downloaded the file to the temporary directory and labeled it as untrusted. The default BitTorrent client, `Transmission`, was invoked as untrusted to start downloading the movie into the Download directory. Once the download completed, we double-clicked the movie to view it. `vlc` was started as untrusted to play the movie. Realizing that the movie had no subtitles, we located `subdownloader` for downloading subtitles. Since our installer considers Ubuntu's universe repository as untrusted, the application was installed as untrusted, and hence operated only in untrusted mode. We searched and found a match. Clicking on the match resulted in launching an untrusted `Firefox` instance. We went back to `subdownloader` to download the subtitle, and then loaded this file into `vlc` to continue watching the movie.

Compiling programs from students.

Some students submit their programming assignments. Teaching assistants for the course need to download their projects, extract them, compile them and execute the binaries in order to grade the assignments. In this experiment, we considered an attack that creates a backdoor by appending ssh key to `authorized_keys` so that a malicious student can break into TA's machine later.

With protection from PIP, when the TA received the submission as an attachment, it was marked untrusted. As the code was unpacked, compiled and run, this "untrusted" label stayed with it. So, when the code tried to append a public key, it was stopped.

Resume template.

We downloaded a compressed resume template from the Internet. When we double clicked on the tgz file, `FileRoller`, the default archive manager started automatically as untrusted because the file was labeled as untrusted by `Firefox`. We extracted the files to Documents directory. We then opened the file with `texmaker` by selecting "Open With", since `texmaker` was not the default handler for tex file. `texmaker` was started as untrusted and we started editing the file. We then compiled the latex file and viewed the dvi

Document Readers	Adobe Reader, dhelp, dissy, dwdiff, evince, F-spot, FoxitReader, Geegle-gps, jparse, naturaldocs, nfoview, pdf2ps, webmagick
Editor/ Office/ Document Processor	Audacity, Abiword, cdcover, eclipse, ewipe, gambas2, gedit, GIMP, Gnumeric, gwyddion, Inkscape, labplot, lyx, OpenOffice, Pitivi, pyroom, R Studio, scidavis, Scite, texmaker, tkgate, wxmaxima
Games	asc, gbrainy, Kiki-the-nano-bot, luola, OpenTTD, SimuTrans, SuperTux, supertuxkart, Tumiki-fighters, wesnoth, xdemineur, xtux
Internet	cbm, evolution, dailystrips, Firefox, flickcurl, gnome-rdp, httrack, jdresolve, kadu, lynx, Opera, rdiff, scp, SeaMonkey, subdownloader, Thunderbird, Transmission, wbox, xchat
Media	aqualung, banshee, mplayer, rhythmbox, totem, vlc
Shell-like	bochs, csh, gnu-smalltalk, regina, swipl
Other	apoo, arbtt, cassbeam, clustalx, dvdrip, expect, gdpc, glaurung, googleearth, gpscorrelate-gui, grass, gscan2pdf, jpilot, kiki, otp, qmtest, symlinks, tar, tkdesk, treil, VisualBoyAdvance, w2do, wmmon, xeji, xtrkcad, z88

Figure 3: Software tested

document with `evince` by clicking on the "View DVI" button in `texmaker`. We then viewed pdf and `AdobeReader` was automatically invoked as untrusted. The document was rendered properly.

Stock charting and analysis.

We wanted to study trend of a stock and we searched the Internet about how to analyze. We came across a tutorial on an unknown website with a `R` script. We installed `R` and downloaded the script. When we started `R`, we found that it is a command line environment and is not so user-friendly for beginners. We then installed `RStudio`, a front-end for `R`, from a deb file we found on another unknown website. Our installer installed `RStudio` as untrusted because `Firefox` labeled the deb file as untrusted. After we started `RStudio`, we loaded the script and realized that it required several `R` libraries. We installed the missing `R` libraries. These libraries were installed in a shadow directory since `R` implicitly accessed the library directory. After installing the libraries, we generated a graph. We saved the graph in the Pictures directory, and edited the graph with `GIMP`.

6. Evaluation

We tested about 100 software packages spanning multiple categories listed in Figure 3. All of these programs can run as benign, as well as untrusted. They all worked without any problems or perceptible differences. Usability of these programs depends on the type of programs. We focus our discussion on usability for the first two categories. More detailed discussions can be found in [11].

Benign document readers can only open high integrity files. Untrusted readers have no restriction in opening. We believe this does not affect usability because these document readers are usually invoked via file explorers (e.g., double-clicking an icon in nautilus). Our `uudo` inference technique (Section 3.4) can infer the required integrity level. A difference between benign and untrusted document readers is when performing a "SaveAs": Benign readers can create high integrity copies while untrusted readers can only create low integrity copies.

When invoking editors via file explorers, usability is preserved because PIP can infer reliably the files to be edited. However, editors can violate information flow policies when they are used to edit both high and low integrity files simultaneously. Usability depends on what integrity the editors are in. Benign editors cannot open low integrity files. On the other hand, untrusted editors tend to open high integrity files in read-only mode automatically when they cannot open them in writable-mode.

7. Related Work

Since the Biba [2] integrity model proposed, researchers have been working to improve usability of information flow systems. Low-water-mark is an extension to the Biba model that allows entities to downgrade from higher integrity to lower, such that more usage scenarios can be supported. LOMAC [4] improves on the low-water-mark model to address self-revocation problems.

Instead of focusing on usability, Decentralized Information Flow Control (DIFC) systems (HiStar [12], Flume [5], and Asbestos [3]) extend functionalities by allowing applications to create their own labels. This model, however, requires applications (or even the OS) to be rewritten in order to take advantage of the system.

UMIP [6], PPI [9] and IFEDAC [7] are more recent systems developed for Linux OS and compatible with existing applications. However, they do not address the usability issues discussed here.

8. Conclusion

PIP system being demonstrated provides systematic integrity protection for Linux. This paper presented the PIP system architecture, and describes in some depth the challenges posed by traditional information flow techniques, and the techniques we developed in PIP to address them. We also proposed some demo scenarios to illustrate how the techniques we introduced are useful in improving usability. PIP has been tested with hundreds of software packages. It is an open-source project, with the source-code as well as a virtual machine image being downloadable from our website [10].

9. References

[1] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. In *Communications of the ACM 18(6)*, 1975.

[2] K. J. Biba. Integrity Considerations for Secure Computer Systems. In *Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts*, 1977.

[3] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. In *SOSP*, 2005.

[4] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *S&P*, 2000.

[5] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *SOSP*, 2007.

[6] N. Li, Z. Mao, and H. Chen. Usable Mandatory Integrity Protection for Operating Systems . In *S&P*, 2007.

[7] Z. Mao, N. Li, H. Chen, and X. Jiang. Combining Discretionary Policy with Mandatory Information Flow in Operating Systems. In *TISSEC 14(3)*, 2011.

[8] W. Sun, R. Sekar, Z. Liang, and V. N. Venkatakrishnan. Expanding Malware Defense by Securing Software Installations. In *DIMVA*, 2008.

[9] W. Sun, R. Sekar, G. Poothia, and T. Karandikar. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *S&P*, 2008.

[10] W. K. Sze. Portable Integrity Protection System (PIP). `http://www.seclab.cs.sunysb.edu/seclab/pip`.

[11] W. K. Sze and R. Sekar. A Portable User-Level Approach for System-wide Integrity Protection. In *ACSAC*, 2013.

[12] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *OSDI*, 2006.