Protection Wrappers
A Simple and Portable Sandbox for Untrusted Applications

Christian Jensen[*] Daniel Hagimont[†]
INRIA Project SIRAC
655, avenue de l'Europe
38330 Montbonnot – France
Christian.Jensen@imag.fr, hagimont@inrialpes.fr

Abstract

In open and configurable applications, external programs are often used to handle different functions and data formats. This is particularly true for applications that communicate through the Internet, where new protocols and data formats are frequently introduced.

These external programs are often installed quickly and without a full security auditing, even when the sources are available. This makes the users of such applications vulnerable to viruses and Trojan horses introduced by misconfiguration or flaws in the security of these applications.

In this paper we introduce a mechanism called "protection wrappers" that allows an application to run external programs in a restricted environment called a "sandbox". Programs running in a sandbox will execute with the identity of a user with limited privileges. This reduces the potential damage to the system and to the data of the user who originally launched the application.

1 Introduction

The dramatic growth of the Internet and the popularity of the World Wide Web have given birth to a new network community where individual users, academic and industrial institutions, in all countries, are exchanging data and software freely across the network. The Internet was previously used to exchange software and data among a small community of researchers who knew and trusted each other - just like computer hobbyists have exchanged software on diskettes with friends, neighbors, and colleagues - but today people connected to the Internet are receiving data and using software from various unknown sources, e.g. installing and using a new video player found on a Web server.

In principle both programs and data should be carefully verified before being used, the program by the administrator who installs it and the data by the program that manipulates them. However, in many cases software or data are used without prior verification and without authentication of the source.

Internet communication softwares like web browsers or mail readers are increasingly relying on external programs to display images or postscript files, play music or video clips, convert MIME encoded mail, or simply allow users to specify external pagers and editors. These programs are potential Trojan horses for two reasons: first because they may have been written by malicious programmers and secondly because they rarely implement a protection policy that allow them to verify data before operating on them.

Most of these external programs are developed to be used in safe environments where data are generally trusted. Two good examples of this are Ghostscript (`gs(1)`) that allows users to preview their PostScript documents and MS–Word that can be used to prepare reports and write documentation for programs. However, PostScript is a full programming language, that for instance allows programs to access files in the file system, and MS–Word has the ability to create or update macros, based on the definitions found in a document. When these programs are used in the potentially hostile environment of the Internet, where the PostScript document retrieved from a Web–server or the Word document attached to an email may have been carefully prepared by an adversary, these programs can act as Trojan horses that corrupt the users files or helps potential intruders to breach the site security.

It is therefore crucial to provide a protection service that prevents the use of these programs from damaging the machine, and the environment of the user who runs the programs. In this paper, we propose a portable mechanism that isolates programs within a sandbox with restricted privileges. This mechanism works by wrapping the application in a front end program (the wrapper) that implements the need–to–know principle, without modifying the application itself.

A program isolated in a sandbox can initially have few well–defined access rights. Additional rights are then

---

[*] Université Joseph Fourier, Grenoble
[†] INRIA Rhône–Alpes

dynamically granted at run–time only if required. For example, the PostScript previewer can be isolated in a sandbox without access rights to the local file system. When a user invokes the wrapped previewer, a read access right is dynamically added to the sandbox before the previewer is started, thereby allowing the previewer to read the file. When the previewer terminates, this access right is *automatically revoked*.

The rest of this paper is organized as follows: Section 2 contains an analysis of the threat model. In section 3 we present the design of our protection service. Its implementation using protection wrappers is outlined in section 4. Our work is compared to related works in section 6 and we present our conclusions in section 8.

2 Threat Model

The main problem with downloading executable content from the Internet is that it crosses the first and most important line of defense: the system boundary [4], i.e. it allows access to the system as an authorized user.

The security of most machines currently connected to the Internet relies on inviolability of the system boundary through authorization. This is normally achieved through firewalls (e.g. NetWall from Bull [1]), filtering network traffic (e.g. Wietse Venema's TCP Wrappers [2]), and login procedures with good passwords (e.g. Matt Bishop's `passwd+` [3]).

A downloaded program or a program interpreting downloaded data runs under the identity of the user starting the program and has all access rights available to that user. In this case neither firewall, network filtering, nor strong passwords protect the system. Running inside the system boundary, the program can install a Trojan horse (a program that behaves like a known program, but has a security related side effect) or a back door that allows the potential intruder to enter the system at a later time, without passing through the normal authentication procedures.

The main problem is therefore to run programs that handle unverified downloaded content outside of the local hosts perimeter of trust.

In order to better analyze the problems involved, we divide downloaded content into three different categories: passive data, active data and executable programs. This classification is based exclusively on the intended use of the data and as consequence decides what programs are used to handle it. These categories are defined and discussed in the following.

2.1 Passive Data

Data that are intended to be stored on disk, used by programs or displayed on terminals without prior interpretation are considered passive. Examples of passive data are simple mail messages (RFC 822), network news, and images or sound files downloaded with a web browser.

Passive data are generally harmless, i.e., they can cause no harm to well written programs. However, well prepared data may overflow a buffer in buggy programs, which can result in parts of the data being interpreted as code. This was the case with the bug provoked in `fingerd(8)` by the infamous Internet worm [5, 6].

2.2 Active Data

We define active data as data that can introduce modifications to system objects or consume system resources when interpreted by the proper interpreter. Examples of active data are Java applets, PostScript documents, MS–Word documents containing macros, and all sorts of scripts.

Active data pose the same threats as passive data. Furthermore, they may exploit all openings that the interpreter may offer, e.g. the access to the local file system offered by earlier versions of Ghostscript. This problem is especially important for interpreters developed for use in trusted environments, e.g. small networks or single user systems, because they rarely implement a protection policy that allows them to verify data before operating on them.

2.3 Executable Programs

In principle, all executable programs (third party software or binaries downloaded through the Internet) and all source distributions that have not been explicitly audited before compilation should be regarded as untrusted programs, and as such potential Trojan horses. In reality most users and system administrators trust certain third parties (e.g. companies or organization like Netscape or the Free Software Foundation or well known FTP sites like `simtel.net`) to provide safe binaries or program sources.

Untrusted programs run under the identity of an authorized user and can access all resources available to that user.

2.4 Discussion

The only way to protect the system against an attack with passive data is to ensure that all programs that operate on downloaded data are well behaved and verify data before operating on them.

It is generally difficult to verify that all these programs perform well. We therefore propose to isolate untrusted programs in order to reduce the potential damages to the local host issued either by the untrusted software, or by downloaded active or passive data.

It is also generally difficult to determine a priori the resources required by the untrusted program. We propose an implementation of the need–to–know principle, where the access rights of the program are restricted to the minimum and additional access rights are dynamically granted according to the application's requirements.

3 Sandbox Design

We aim at building a mechanism that protects the local system from intrusion or damage caused by the use of downloaded content. Installation of Trojan horses, back doors, and any persistent damage to the system involves modifying data on disk, so we especially focus on limiting access to the local file system.

The access control mechanisms offered by most of the systems connected to the Internet (excluding single user systems without real security, e.g. Windows or MacOS), define access rights in terms of operations (e.g., read, write, and execute) on files, allowed by individual or groups of authorized users of the system. We want to build the sandbox on the protection mechanisms offered by the operating system, so we find it natural to create a sandbox by changing the identity of the user to some unprivileged user before starting the program.

A program isolated in a sandbox executes with the access rights of the unprivileged user. These access rights are restricted to the minimum required for the software to be runnable (mainly the software's configuration files). Additional access rights are dynamically granted to the program at run–time, e.g. when a filename is passed as an argument to the program.

The protection wrapper (a front end program) implementing this mechanism must parse the arguments of the program to determine if any additional access rights should be added to the sandbox before the program is started. The addition of access rights is controlled by the security policy compiled into the protection wrapper. The wrapper then assumes the identity of the unprivileged user and runs the program. All additional access rights must be revoked, when the wrapped program terminates.

Different policies can be defined for the same program by constructing different protection wrappers. The protection wrapper can restrict the environment of the untrusted program further, by controlling the environment variables available to the program and/or the working directory of the program.

4 Implementation

We have written a number of protection wrappers by hand, in order to verify the feasibility of our proposition. Our experiences with these wrappers are described in the following.

4.1 Protection Wrappers

Protection wrappers rely on the protection mechanisms of the underlying operating system, in our case primarily file access control implemented by Access Control Lists (ACLs). ACLs allow fine grained access control for files, i.e., individual users can be granted or denied specific access rights to a file.

This mechanism is used by the protection wrapper to grant the unprivileged user access to the files specified on the command line, before taking on the identity of that user. Furthermore, it is used to revoke access rights after the wrapped program has terminated.

The overall structure of a protection wrapper is described in pseudo–C in figure 1.

```
main(int argc, char** argv)
{
  parse_arguments();
  grant_rights();
  setup_environment(&environment);

  if (!fork()){
    /* in child */
    setuid(unprivileged_uid);
    setgid(unprivileged_gid);
    execve(PROGRAM, argv, environment);
  } else
    wait();

  revoke_rights();
  return();
}
```

Figure 1: Protection wrapper structure

In practice the functionality of `parse_arguments()` and `grant_rights()` are performed by the same procedure. When the wrapper recognizes a file name parameter (in `argv`), it adds the `unprivileged_uid` to the ACL for that file, with the access rights compiled into the protection wrapper. It also adds the file name and the access right to the list of rights to be revoked when the wrapped program terminates.

4.2 Using Protection Wrappers

The protection wrapper has two main functions: changing the identity of the caller to an unprivileged user and dynamically granting access rights for that user in the ACLs of files needed to run the program (i.e., configuration files and files passed as arguments to the program).

In order to change the identity of the caller, the wrapper itself must execute as a privileged user (e.g. `root` on a Unix system), so ordinary users cannot be allowed to create their own wrappers. An ordinary user who wishes to run a downloaded program must therefore ask his system administrator – or use a dedicated tool – to install a wrapper for the downloaded program before he can run it securely. Once a protection wrapper is installed, the untrusted program can be replaced by updated versions without changing the wrapper, as long as the new versions respect the original interface. However, if an ordinary user is allowed to update a wrapped program, he is effectively allowed to run almost any program anonymously (i.e., as the unprivileged user). In general, this does not augment his privileges, but can be used to mask his real identity when performing questionable actions. The right to update a wrapped program should therefore be restricted to a few trusted users.

Different normal users can use the same protection wrapper to execute the same program as the same unprivileged user, or different wrappers can be created for each of the normal users.

Using a single wrapper has the advantage of saving disk–space and being simple and easy for the system administrator to manage (he only has a single wrapper to do and it can be installed transparently to the ordinary users of the system), but all instances of the wrapped program will run as the same user and with the union of all access rights dynamically granted to that unprivileged user. In this case, an exploitable program can be used by a local user (anyone with legitimate access to the system) to obtain temporary access rights for those files other users have passed as argument to a wrapped program. Furthermore, a single unprivileged user effectively hides the identity of the real user, when several instances of the program are running simultaneously.

Another possibility is to have different protection wrappers associated with different unprivileged users. Each real user will run the wrapped program as a different unprivileged user, which prevents interference from other users and restores accountability. However, this method requires one or more "pseudo–users" defined for each real user of the system, which is both resource consuming and makes it difficult to detect security vulnerabilities (the potentially large number of pseudo–users makes it difficult to audit the system).

4.3 Protection Wrapper Generation

Writing protection wrappers for all untrusted applications is a tedious and error prone task. Furthermore, all protection wrappers have the same structure, so we are therefore working on a tool that generates source files for a protection wrapper, based on the protection definitions described in a configuration file.

The syntax of the protection wrapper configuration file will be similar to the one illustrated in figure 2.

The IDENTIFICATION division identifies the wrapped program with a full path name and the identity (both UID and GID) of the unprivileged user.

```
IDENTIFICATION
  PROGRAM = <full path name>
  UID = <uid>
  GID = <gid>
ENVIRONMENT
  DIRECTORY = <full path name>
  LIMIT <resource> = <limit>
  {ENV <variable name> = <value>}
PARAMETERS
  {<param> = [access right] <type>}
```

Figure 2: Configuration file syntax

The ENVIRONMENT division allows customization of the runtime environment of the wrapped program. The wrapper can change directory (maybe even introduce a `chroot` jail[1]), set limits on resource usage with `setrlimit(2)`, or specify values for environment variables, e.g. `$path`.

The PARAMETERS division specifies all the parameters accepted by the program and the access rights that should be granted to the sandbox. It is also possible to impose restrictions on some of parameters values, if the wrapped program has well known weaknesses that cannot be repaired easily.

5 Evaluation

In the following we present our experiences with programming protection wrappers. The evaluation covers both qualitative aspects (do protection wrappers achieve their goals?) and quantitative aspects (what is the performance overhead of wrapping an application?).

5.1 Qualitative Evaluation

Protection wrappers allow users to run untrusted programs under the identity of an unprivileged user, thereby reducing the risk of compromising themselves or the system on which they run. Moreover, the dynamic transfer of access rights only allows the wrapped program to access those files it actually needs; the access rights are automatically revoked when the program terminates.

The main problem with protection wrappers is that they can only wrap entities that are visible from the outside of the program, i.e., well–known configuration files or files passed as arguments on the command–line. Filenames supplied interactively to the program cannot have the access rights added automatically by the protection wrapper.

This problem is best illustrated by an example. In the PostScript previewer `ghostview(1)`, PostScript files can either be specified as arguments on the command line

[1] `chroot` Jails are described in 6.1

or interactively through a file dialog box. It is possible for the protection wrapper to grant temporary access rights to the files passed as arguments, but not for those entered interactively through the file dialog. The file dialog is normally implemented as one or more functions (a widget or a library), that is linked with the program, and will be loaded with the application at runtime. This library becomes an integral part of the wrapped program and will run with the access rights of the unprivileged user.

However, by specifying the environment of the wrapped program, we can control the path used by the system to load shared libraries. It is therefore possible to use an approach similar to protection wrappers for the shared libraries.

Interactive programs pose two problems for such a wrapper mechanism: first of all, each program may have its own input mechanism (so it is difficult to know what libraries and functions to wrap) and secondly, once in the wrapped program it is impossible to get the real user's access rights back (so we need something similar to the setuid mechanism for the shared libraries).

One way of solving the problem of different input methods is to require all programs to use a single standard interface, e.g. the Motif file selection dialog box [7]. A library that wraps the standard input method and changes the ACLs for the selected files, can then be substituted for the normal library by setting the library path of the wrapped program. The original protection wrapper and the library wrapper described here have opposite purposes: the protection wrapper is used to reduce the access rights of the wrapped program and the library wrapper is used to reinstate the access rights of the real user to a single library within the wrapped program.

The library wrapper must execute with special privileges in order to reinstate the access rights of the real user and manipulate the ACLs of the files entered through the file dialog. Furthermore, it must be able to identify the real user of the program and match that identity with the user who enters the filename. We are currently investigating these problems.

5.2 Quantitative Evaluation

The performance of the protection wrapper mechanism has not been our primary concern. However, it is unlikely that people will use the protection wrappers if they impose a large overhead on the wrapped programs. We have measured the overhead for a simple program that takes two filenames as arguments and copies the content of one file into the other, i.e., it requires the additional access rights for two files while the program is running. The measurements were performed on a BULL Estrella workstation (PowerPC 604, 100MHz) running AIX–4.1.4. The cost of the mechanism is shown in Table 1.

Action	Measured Time
ACL manipulation	36 ms
fork + exec	6 ms
total overhead	42 ms

Table 1: Overhead of the protection wrapper mechanism

Our measurements show two things: the overhead of the mechanism is small compared to the runtime of many programs and secondly that most of the overhead is due to the manipulation of ACLs.

The total overhead can be broken into the following categories: ACL manipulation, spawning the wrapped program (`fork(2)` and `exec(2)`) and parsing the parameters in the wrapper in order to know which access rights to transfer [2].

The test program was very simple so the time required for parsing the parameters a second time was negligible (around 0.2 ms). However, we do not expect this overhead to become dramatically higher in more complex programs.

We expect the time required to spawn the wrapped program to be constant, although it may increase slightly if a large environment is passed along to the wrapped program.

The main cost of our protection wrapper mechanism is the manipulation of ACLs. This cost will increase with the number of ACL manipulations, i.e., configuration files and filenames passed as arguments on the command line.

6 Related Work

The basic idea for the protection wrappers (i.e., to confine programs to a limited environment (a sandbox) at run–time) is inspired by the tradition of using `chroot(1)` to restrict the environment of programs in Unix and by the Janus system developed at Berkeley. It uses the same sandboxing mechanism as the suEXEC mechanism in the Apache web–server. We call this mechanism a "`setuid` jail" in analogy with the `chroot` jail. We examine these sandboxing solutions in the following.

6.1 `chroot` Jails

The `chroot` jail is probably one of the oldest mechanisms to control the environment of a program that handles untrusted content. It relies on the `chroot` program or library function, that changes the perceived root directory of an application. This means that programs can be confined to a subtree of the file system hierarchy. Any file

[2]The shown wrapper does not control the environment of the wrapped program, so the overhead will be slightly higher if this functionality is build into the wrapper.

name used by a chroot'ed program will be interpreted relative to the perceived root directory. This is often used to prevent public services like anonymous ftp or Web–servers from accessing sensitive information, e.g. system configuration files.

The advantage of this model is that the operating system guarantees that only resources available in the sandbox can be corrupted. The disadvantage is that everything the program needs to run (configuration files, dynamic libraries, ...) has to be copied into the sandbox before the program is run, and any results have to be copied back out from the sandbox (making sure that no valid data is overwritten) when the program terminates.

6.2 Janus

Janus [8] uses the operating systems process tracing capability to monitor all system calls from processes running in a sandbox. System calls that are potentially dangerous, e.g. `open(2)`, is only allowed to proceed if it is explicitly allowed by a security policy and not disallowed by any other policy.

These policies cover different areas of system security, e.g. access to the file system or network connections. Security policies are statically specified in configuration files that are read when the system is initialized. This means that Janus has the same disadvantage as the `chroot` jail mentioned above, i.e., access rights to local resources cannot be granted dynamically according to the invocation parameters. Moreover, Janus relies on the ability to filter all system calls by a monitoring process, either through the `ptrace(2)` system call or `/proc` file system. These features are widely available on Unix systems, but porting this mechanism to other systems seems difficult.

6.3 suEXEC

The Apache http–server provides the ability to run Common Gateway Interface (CGI) programs as a different user from the one running the web–server. This allows web–servers, that host many different users, to launch programs started from a web–page with a user id different from its own (e.g. the identity of the user owning the web–page). Programs started by the web–server would otherwise have access to all resources managed by the web–server.

The suEXEC mechanism implements half of the functionality of a protection wrapper: it allows a user to run a program under a different user id[3], but there is no mechanism to dynamically grant additional access rights to the wrapped program at runtime.

[3] The suEXEC mechanism can be used to change the identity to any user (except root) when running any program, so it naturally comes with a lot of restrictions on who can use it and how it can be used.

7 Future Work

The implementation of protection wrappers has raised a number of interesting problems that we are currently investigating or planning to investigate in the near future. We are especially focusing on three problems: protection wrapper generation, wrapper environment control and access rights for interactively specified parameters.

We are currently working on a generator for protection wrappers. This generator will allow us to automatically generate the sources for a protection wrapper from a specification similar to the one illustrated in Figure 2. Based on our experiences with this generator, we plan to create a tool that will allow ordinary users to specify and install their own protection wrappers.

The protection wrapper allows us to control the environment of the wrapped program, by setting or changing the values of environment variables. We are currently investigating how much of the real user's environment should be inherited by the wrapped program and how this control can be used to improve the security of the wrapped application.

The problem of interactively specified parameters has to be solved in order to make the protection wrapper mechanism generally applicable. We plan to limit our support to programs that use a standard input method or where the input functions in the source code are easily identifiable and located in a way that makes it easy to split them out into a separate shared library. We need to find an efficient way of executing this shared library with the access rights of the real user, and associate this library with the real users input device.

8 Conclusions

In this paper we have presented some of the security problems of composing applications from external programs. The main problem is that these programs are potential Trojan horses because they rarely implement a protection policy that allow them to verify data before operating on them.

We have proposed a mechanism for isolating untrusted components (programs) in sandboxes, where they can do little harm to the system or the users running the applications. Several mechanisms have been proposed to isolate untrusted code, but none of these provide dynamic granting of access right.

We have outlined the implementation of this sandboxing scheme on an Unix style system, but most of the system mechanisms we used are general purpose operating systems mechanisms. We plan to implement it on other systems, in particular Windows NT.

Our experience with protection wrappers show that they incur a relatively small overhead on the overall runtime of an application.

However, confining all programs to separate sandboxes would probably cripple the system in a way that would make it unacceptable for most civilian uses (and the military will generally have the resources to audit all useful programs). It is therefore often a question of trust and the individual security policy defined for the system, that decides which programs to isolate in sandboxes.

Acknowledgments

This work was partially supported by CNET (France Télécom). We would also like to thank the anonymous reviewers for their helpful suggestions and valuable comments on ways to improve this paper.

References

[1] F. Soinne: "Netwall Version 3.2". *Bull Netwall Whitepaper available from http://www-frec.bull.com/ospbuhp.htm.*

[2] W. Venema: "TCP Wrappers". *The TCP Wrapper distribution is available from ftp://ftp.cert.org/pub/tools/tcp_wrappers.*

[3] M. Bishop: "Anatomy of a Proactive Password Changer". *Proceedings of the Third UNIX Security Symposium, pp. 171–184, August 1992.*

[4] M. Gasser: "Building a Secure Computer System". *Van Nostrand Reinhold, 1. edition pp. 19–21, New York, 1988.*

[5] M. W. Eichin and J. A. Rochlis: "With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988". *Presented at the 1989 IEEE Symposium on Research in Security and Privacy, 1989.*

[6] E. H. Spafford: "The Internet Worm Program: An Analysis". *Purdue Technical Report CSD-TR-823, December 1988.*

[7] P. M. Ferguson and D. Brennan: "Motif Reference Manual (The Definitive Guides to the X Window System, Vol 6B". *O'Reilly & Associates, December 1993.*

[8] I. Goldberg and D. Wagner and R. Thomas and E. A. Brewer: "A Secure Environment for Untrusted Helper Applications". *Proceedings of the 6th USENIX Security Symposium, pp. 1–13, 1996.*

[9] Apache Documentation: "Apace suEXEC Support". *Apache HTTP Server Version 1.3, available from "http://www.apache.org/docs/suexec.html".*