Notes on application-orientated access control

Adrian Spalka and Hanno Langweg
Department of Computer Science III, University of Bonn, Germany
adrian@cs.uni-bonn.de

1. Abstract

The protection qualities of discretionary access control systems realised by today's prevalent operating systems are based on an assessment of the trustworthiness of users. By starting a program a user transfers his trustworthiness to it, ie, there is the tacit assumption that the program's trustworthiness at least matches that of the user. However, malicious programs are a growing source of threat. They perform operations without the user's consent and often in contravention of his interests. To eliminate this danger we examine program-orientated protection strategies. We then present, firstly, a small enhancement to the operating system and, secondly, an addition to the operating system, which support both a user and an application with high security demands in the enforcement of authenticity and integrity even in the presence of malicious programs.

2. Introduction

The security of computer systems depends on three factors: availability, confidentiality and integrity. Moreover, owing to its importance in a growing number of today's applications, authenticity is added as the fourth factor.

This work addresses problems posed to security by malicious programs, eg Trojan horse programs, viruses and worms, which run on a user's computer or in his account without his consent and, most often also, without his knowledge. There are many ways a malicious program can exploit to enter a user's computer. Today's most popular operating systems, eg Windows 2000 or Linux, have an access control system which grants rights to a program based on the identity of the user who started it; these operating systems do not support program-based access rights control. Thus, once in place, a malicious program is endowed during its execution with the user's rights and privileges and can perform a whole range of malicious operations unintended by the user, eg delete files, modify programs or interfere in the communication of programs.

Current protection mechanisms, such as firewalls, virus scanners and email filters, aim only at preventing a malicious program from entering the computer. Yet, the practice demonstrates that this protection is often insufficient.

Trojan horse programs do not usually exploit implementation vulnerabilities, such as buffer overflows. They use the application programming interface of the operating system to pursue their malicious goals. If a Trojan horse cannot utilise an interface owing to access restrictions, it must bypass the control mechanisms. One way to do it is to have a sufficiently privileged application access an interface or make the user authorise a transaction. If this application does not prevent its being controlled by a Trojan horse, then the Trojan horse has successfully elevated its privileges.

There exist generic approaches to strengthen an access control system's protection of confidentiality, which considerably reduce the potential damage inflicted by a malicious program. However, today's operating systems are far from being ready to implement them. Though equally important, the protection of authenticity and integrity received far less attention. Thus, if present at all, access control systems in their present shape place no specific restrictions on the actions of a malicious program.

Moreover, generic protection mechanism are likely to be too stringent since they cannot take an application's communication semantics into account. We therefore believe that an application should also have the ability to take care of its own protection demands.

In a first step we suggest to give an application more control over communication with other components to limit interference of malicious programs. To give an example we propose its realisation in the Microsoft Windows messaging model.

3. Rationale for application-orientated access control

3.1. Previous and related works

Protection mechanisms so far concentrate on separating applications from each other (Balfanz 2001, Pfitzmann 2001) or forcing a user to assign trust to applications and to operate them by help of different accounts. Often big frameworks are proposed that would require significant modifications of existing operating systems. This is an obstacle as regards bringing more security to the users' computers.

Approaches more promising in this respect focus on slight modifications of existing operating systems and on adding functions on top of an established operating system basis. Wrappers for existing programs are an example for this (Fraser 1999, Dean 2002):

However, current approaches usually limit applications in their collaboration with other programs. Thereby opportunities are reduced that could be exploited positively when using context-aware controls directed to the applications.

3.2. A shift of paradigm

Today's discretionary access control systems (DAC) record rights to protection units with respect to users only, but in fact they control access to these units with respect to programs. For this to work out properly a DAC system must assume that each access to a protection unit by a program is assuredly performed on the behalf of the user who started it. We must stress that this is a tacit assumption. For the administration of his protection units a DAC system offers to a user only the possibility of granting and revoking rights to other users, and not to programs. Consequently, the DAC system leaves to the user the problem of establishing that a program performs only the type of access intended by him – the protection offered by a DAC system relies on this shift of responsibility.

In theory, the type of access performed by a program can be checked by examining its source code, and eventually that of dependent libraries. However, in practice there are many compelling reasons which render such a check impossible. Since a user cannot establish a program's type of access beforehand, ie check if it is malicious, he must decide on another way to deal with this threat at runtime. We can think of employing technical and social measures in a security system. A technical security system based on control and monitoring can, to give an example, in a first step determine operations that lead to a violation of confidentiality and then, on these grounds: set up general constraints on these operations; control these operations at runtime and, possibly with the user's intervention, grant or deny it; and log these operations and let the user inspect the log later

As a technically supported social measure the program, for example, can carry information on its author, eg a digital signature, which the user can verify. The user's decision to run or not to run it can then be based on his assessment of the author's credibility.

Lastly, the user can resort to purely social measures. If the program is used for a long time by him or other users and no improper behaviour was observed (and reported), the user can extrapolate from this fact the assumption that the program will continue to work properly. If the program is new or used only a little, the user can still do nothing, close his eyes and hope that it will not cause any intentional damage.

When it comes to program-orientated protection the DAC system implements only the last, purely social strategy – though the word implements seems misplaced for this strategy comprises no technical measures and, therefore, requires no effort on the side of the DAC system.

The absence of technical protection measures against programs at runtime and the user's inability to establish a program's proper access behaviour beforehand is regarded in DAC systems as a 'protection quality', the name of which is trust. To say it bluntly, the meaning of trust-based protection here is that the user is helpless and the DAC system does not care.

The protection system we have in mind controls two sources of danger: malicious users who start programs with the intent of violating security and malicious programs started by the victim user ignorant of the programs' hidden malicious intentions. The first one can be averted by today's DAC systems. To eliminate the second danger we propose to develop a program-orientated protection strategy, which forms a central part in a new control system with advanced security properties.

Two practical methods of supporting an application's ability of taking care of its own protection against malicious programs are presented in the subsequent sections. The first one is a fairly basic enhancement to operating systems, which, however, has far reaching implications on an application's power to detect a malicious interference in a communication with other components. The second one is an addition to operating systems, which prevent malicious programs from compromising the integrity of critical data. Both methods are a part of our general aim of enabling security aware applications to protect its execution even in the presence of malicious programs on standard operating systems.

4. Operating system enhancements

4.1. Individual vs central protection

Applications can employ two kinds of protection against being controlled by untrustworthy programs. Central protection refers to mechanisms of the operating system imposing access restrictions on collaboration. Individual protection comprises methods of the application itself to determine whether or not an interaction is acceptable.

While central protection, eg, by way of 'wrappers' for COTS software, is applicable to all applications and frees the application developer from thinking about implementing security mechanisms, there are drawbacks. As regards filtering the communication between programs it is difficult to determine which communication is 'good' according to a security policy. This involves knowledge about the communicating programs. As with firewalls, the filtering is on a coarse basis and neglects the semantics of the communication. While one message may be security-relevant in a small number of cases, it may serve a useful and harmless purpose during the rest of the time. For instance, querying the contents of a window has to be blocked only when there is confidential content.

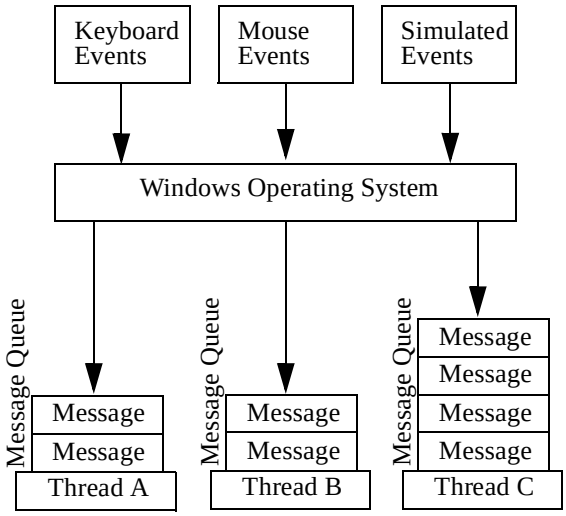
Central and individual protection can be combined. Central protection can serve as a core enhanced by fine-grained individual protection at the application level.

4.2. Windows messaging model

It has been shown that applications running on the Microsoft Windows or X-Windows platform can be vulnerable to attacks by Trojan horse programs. A popular attack is to control applications remotely by simulating user actions.[1] This is done by placing messages in an application's message queue that are not created by the operating system in response to user actions like typing on the keyboard. Thereby confidential information can be conveyed to rogue applications or transactions can be placed with e.g. a home-banking or electronic signature application.[2]

Messaging between applications is a potential vulnerability. As a counter measure, messaging can be restricted at the platform or application level. The X-Windows system, for instance, allows to disable conveyance of messages placed by the *SendEvents* function.[3] In Microsoft Windows, certain types of messages can be restricted for a desktop. However, blocking messages completely is a very strict method. There may be occasions, like computer-based training, in which remote control of another application or parts of it is desired.

Microsoft Windows uses an internal messaging model to control Windows applications. Messages are generated whenever an event occurs. For example, when a user presses a key on the keyboard and releases it or moves the mouse, a message is generated by the operating system. The message is then placed in the message queue for the appropriate thread. An application checks its message queue to retrieve messages.[4]



The system passes all input for an application to the various windows in the application. Each window has a function, called a window procedure, that the system calls whenever it has input for the window. The window procedure processes the input and returns control to the system. All aspects of a window's appearance and behaviour depend on the window procedure's response to these messages.

All messages can be evaluated by the application before they are processed. If an application detects that a certain message may violate its security policy (e.g. reading confidential information), it can modify its response to that message. While the standard behaviour might be to report information stored in window elements, the window procedure might send an empty buffer instead.

In the model, it is not possible to distinguish between

1. Cult of the Dead Cow (2001). *Back Orifice 2000.* http://bo2k.sourceforge.net
2. Spalka, A., A. B. Cremers and H. Langweg (2001). 'The Fairy Tale of »What You See Is What You Sign«. Trojan Horse Attacks on Software for Digital Signatures'. *Proceedings of IFIP Working Conference on Security and Control of IT in Society-II:75-86*.
3. Bråthen, R. (1998). 'Crash Course in X Windows Security'. *GridLock* 1(1998):1. http://www.hackphreak.org/gridlock/issues/issue.1/xwin.html
4. Microsoft (1998). *Microsoft Windows Architecture for Developers Training Kit*.

messages placed in the queue by the operating system and messages placed by another application.

4.3. Enhancements to the Windows messaging model

We propose to enhance the Windows messaging model to assist applications in the decision on how to react to messages sent to them. The solution is backward-compatible in that it allows old applications to execute without modifications.

Each message is assigned its origin as an additional parameter. Hence, it can be determined by the receiving application whether the message originated from the operating system or from another application. This involves all messages placed in the message queue by way of the *SendMessage*, *SendMessageCallback*, *SendMessageTimeout*, *SendNotifyMessage*, *PostMessage*, *PostThreadMessage*, *SendDlgItemMessage* functions.

```
MSG_ORIGIN_WINDOWS                  = 0x01
                    // message originates from the OS
MSG_ORIGIN_APPLICATION              = 0x02
                  // message is placed by an application
MSG_ORIGIN_JOURNALPLAYBACK          = 0x10
                  // played by a journal playback hook
MSG_ORIGIN_CBT                      = 0x20
                        // issued by a CBT application
```

Messages that are said to originate from the operating system include those resulting from the user's use of input devices. An application can register as a computer-based training (CBT) application. It hence indicates that messages placed by it in the message queue do not reflect the user's intention.

```
BOOL SetCBTMessageStatus(BOOL bIsCBT);
                        // if successful returns TRUE
```

The functions to retrieve messages from the application's message queue – *GetMessage* and *PeekMessage* – are augmented by *GetMessageEx* and *PeekMessageEx*.

```
typedef struct tagMSGORIGIN {
        DWORD           dwOriginClass
                        // origin of message
        DWORD           dwThreadID
                // id of thread that placed the message
} MSGORIGIN, *PMSGORIGIN, *LPMSGORIGIN;

BOOL GetMessageEx(
        LPMSG           lpMsg,
                        // message information
        HWND            hWnd,
                        // handle to window
        UINT            wMsgFilterMin,
                        // first message
        UINT            wMsgFilterMax,
                        // last message
        LPMSGORIGIN     lpOrigin
                        // origin of message
);

BOOL PeekMessageEx(
        LPMSG           lpMsg,
                        // message information
        HWND            hWnd,
                        // handle to window
        UINT            wMsgFilterMin,
                        // first message
        UINT            wMsgFilterMax,
                        // last message
        UINT            wRemoveMsg
                        // removal options
        LPMSGORIGIN     lpOrigin
                        // origin of message
);
```

With this additional information a program can block communication with untrustworthy programs. We give an example how this works with a home banking application in which transactions are placed and authorised by the user. Depending on the type and origin of a message the home banking application acts differently.

Assume that a transaction is prepared to be sent to the bank. The user maybe wants to use a helper application to fill in some details. The details of the transaction shall not be available to other programs. Hence, the home banking application processes all messages sent by the operating system, but poses restrictions on messages of different origin. While messages that modify the contents of the window may be allowed, querying of information is ignored and submitting the transaction for further processing cannot be forced by other programs. If messages are marked as computer-based training (CBT) no restrictions are applied. However, the transaction is then marked as being placed by a CBT application and will consequently not be transferred to the bank later.

5. SWORM – an operating system add-on

In contrast to security features that are provided directly by the operating system these can be facilitated by an application itself.

We give an example of such an add-on. In case an application wants to protect the integrity of data against other applications that are executed on behalf of a user, it is not advisable to use access rights depending on a user account. Here we introduce a SWORM storage medium. SWORM stands for 'software-based write once read multiple'. After a file has been stored on the medium it cannot be altered or deleted and thus its integrity is preserved against malicious

programs. This is especially useful when electronic signatures for legal transactions are considered.

WORM media are available as hardware devices, eg, a CD-R drive. However, if data is to be fixed for a comparably short period, ie, as temporary storage, the handling of disks and the associated costs are not acceptable. WORM functionality can be provided in software, too, reducing costs and increasing ease-of-use.

A cheap implementation of the SWORM is a device driver that provides a virtual WORM medium on top of the Windows NT file system. One folder is protected by access rights that allow the system account to freely access the folder but denies access for everyone else. In case the access rights are changed this is detected, the driver stops working and issues a warning to the user and the administrator.

The device driver allows the creation of files and read access to them but it does not allow to rename or modify files after they are closed. After a specified period of time the files are deleted.

If access rights are not available (eg, when using Windows 95/98) we store the SWORM files in memory protected by the operating system. Hence, the contents of the SWORM are available only until the machine is shut down. As long as the operating system is running we achieve a SWORM security level comparable to the Windows NT implementation of the SWORM.

The SWORM driver discussed above could be provided by the operating system. However, there is no need to wait for the OS vendor to include the driver; an application that needs data integrity against malicious programs could add this driver during installation.

6. Conclusions

Malicious programs are a more and more popular form of attacks. Security aware applications that execute in today's operating systems must therefore have more means to design and enforce it own protection.

We have shown that small enhancements to the Microsoft Windows messaging model can avert common attacks by malicious programs aiming at taking control over security-relevant applications. Unlike central and inflexible protection mechanisms, our solution gives information and responsibility to the application and the application developer. Our approach is also compatible with computer-based training which requires controlling other applications.

We have also shown that protection of integrity of data, which plays an increasingly important role in e-services, can be effectively supported with a storage that prevents modification of created data.

In conclusion, protection methods which try to prevent a malicious program from entering a computer, such as fire-walls, are, of course, necessary yet not sufficient. We are convinced that an adequate level of protection can only be attained if new security measures are designed – and implemented – that enable security aware applications to perform their task reliably even in the presence of malicious programs.

7. References

Balfanz, D. (2001). *Access Control for Ad-hoc Collaboration*. PhD thesis, Princeton University.

Bråthen, R. (1998). 'Crash Course in X Windows Security'. *GridLock* 1(1998):1. http://www.hackphreak.org/gridlock/issues/issue.1/xwin.html

CERT Coordination Center (1999). *CERT Advisory CA-99-02-Trojan-Horses*. http://www.cert.org/advisories/CA-99-02-Trojan-Horses.html

Cult of the Dead Cow (2001). *Back Orifice 2000*. http://bo2k.sourceforge.net

Dean, J.C., and Li, L. (2002). 'Issues in Developing Security Wrapper Technology for COTS Software Products'. *Proceedings of International Conference on COTS-based Software Systems 2002.* LNCS 2255.

Fisher, J. (1995). *Securing X Windows*. UCRL-MA-121788. CIAC-2316 R.0. http://ciac.llnl.gov/ciac/documents/ciac2316.html

Fraser, T., Badger, L., and Feldman, M. (1999). 'Hardening COTS Software with Generic Software Wrappers'. *1999 IEEE Symposium on Security and Privacy.*

Gligor, V.D. (1999). '20 Years of Operating Systems Security'. *1999 IEEE Symposium on Security and Privacy.*

Lacoste, G. , Pfitzmann, B., Steiner, M., and Waidner, M., ed. (2000) *SEMPER – Secure Electronic Marketplace for Europe*. Berlin et al: Springer-Verlag.

Microsoft (1998). *Microsoft Windows Architecture for Developers Training Kit*.

Microsoft (2001). *Microsoft Developer Network Library.*

Pfitzmann, B., Riordan, J., Stüble, C., Waidner, M., and Weber, A. (2001). *The PERSEUS System Architecture*. IBM Research Report RZ 3335 (#93381) 04/09/01, IBM Research Division, Zurich. http://www-krypt.cs.uni-sb.de/~perseus/

Spalka, A., Cremers, A.B., and Langweg, H. (2001). 'The Fairy Tale of "What You See Is What You Sign". Trojan Horse Attacks on Software for Digital Signatures'. *2001 IFIP Working Conference on Security and Control of IT in Society-II.* 75-86.