Dynamic Monitoring of Malicious Activity in Software Systems *

M. Debbabi†, M. Girard†, L. Poulin†, M. Salois‡, N. Tawbi†

†LSFM Group
Department of Computer Science
Laval University
Québec, Canada
http://www.ulaval.ca/lsfm

‡Defence Research Establishment Valcartier
2459, Pie-XI blvd North
Val-Bélair, Québec, Canada
G3J 1X5
http://www.drev.dnd.ca

October 2000

Abstract

Because of time and budget constraints, organisations are turning more and more to Commercial-Off-The-Shelf (COTS) software rather than developing in-house software. This situation gives rise to great concerns over safety, security, and reliability in critical information systems. This paper presents a research effort to help manage the risk associated with COTS integration through the exploitation of a dynamic monitor. The strategy consists of inserting appropriate drivers to control accesses to critical resources: files, communication ports, the registry, and the creation/destruction of processes and threads. A prototype monitor called DaMon has been designed and developed to run in Windows NT on an Intel box, to demonstrate the feasibility of reactive monitoring to meet a formal security specification.

1 Introduction

A set of tools to circumscribe harmful effects and even to eliminate them would be a significant asset in an environment requiring tight control of security. With the increasingly high use of Commercial-Off-The-Shelf (COTS) software, it is imperative for many organisations to assure themselves that any out-sourced product will meet strict standards of safety, security, and reliability. One has no guarantees that a piece of software does not contain malicious code or that it does only what it is supposed to do. For example, an application could be sending private information without consent. Even if the program specifications do not include sending information externally, proof that it does not do so is hard to come by.

Even in this simple example, it appears obvious that some form of proof of good conduct is required from the vendor. Barring that, other tools are needed to check and control this kind of malicious behaviour in already installed software. At that time, of course, only the executable code and its accompanying compiled libraries are available, which further complicates matters.

To deal with this problem, in 1997 the Defence Research Establishment Valcartier (DREV) contracted a study by the Language, Semantics, and Formal Method (LSFM) group at Laval University. The present MaliCOTS project team consists of about 12 graduate students, four professors, and two DREV scientists. The platform chosen for the project is Windows NT running on an Intel box.

Preliminary studies identified three main areas for investigation:

1. Certifying compilers
2. Static analysis of executables
3. Dynamic analysis of executables

The first area is the most promising in the long term since it works on source code, which provides the greatest flexibility. This approach tries to extract enough information from the source code to allow formal verification, without compromising the intellectual property rights applicable to the code [2].

The second approach is severely limited since one does not usually have access to COTS source code, and disassembling such software poses legal problems. Nonetheless, some investigations are being made by the team to adapt static analysis techniques to assembly-language source code.

The third set of techniques has three possible facets. One is to build a kind of virtual machine that will check every instruction executed by the processor for validity [11]. This approach has the same limitations as static analysis since it works at assembly level. It has another limitation: runtime overhead. A second facet is to do log analysis. This tactic has two major flaws: (1) it is highly limited by what the system can log, although it is possible to extend that capability; (2) it is only reactive, since one cannot log what has not yet happened, thus it is rather limited in its ability to prevent some undesirable action from occurring. The third facet is to build a tool to monitor critical resources for malicious actions. In this case, a security

*This research is funded by a research contract from the Defence Research Establisment Valcartier (DREV), Québec, Canada.

policy can be written to tell the system what is and is not allowed. This is the approach presented in this paper.

This paper is divided as follows. The next section presents the motivations behind the use of such a monitor. Section 3 discusses related work. The fourth section introduces the essential Windows NT information for this work. Section 5 presents the architecture of the monitor. Section 6 explains the security policy language under development. Finally, Section 7 highlights a prototype that implements these ideas.

2 Motivations for a Dynamic Monitor

Static analysis and certifying compilers are powerful techniques to provide some assurance that a program will respect certain properties. The two techniques are quite similar in that they both analyse a program fully before its execution. There is no overhead at runtime and it is much more effective to verify a large set of inputs this way than to run the program for each element of the set.

However, there are cases where these two techniques simply cannot determine if a piece of code respects certain properties. For instance, if a branch in the program depends on user input, there is no way to decide statically if it will be safe for all possible inputs. Another example is the case where a program legitimately copies files received as input. It is difficult, even impossible, to certify statically that such a program will never copy a protected file to an unprotected zone, since all depends on the type of file treated at runtime. There are also some mathematical undecidabilities intrinsic to these techniques. Lastly, there are many types of malicious code that need not be present in the program at the time static analysis and certifying compilers can be applied, including viruses that will attack after the program has been tested and certified.

Dynamic monitoring, on the other hand, can make use of all the information on what is going on in a program at any given time. Although complete monitoring is tedious, error-prone, and introduces often-unacceptable overhead, the limited monitoring presented here is attractive.

The idea is to supervise and control only those specific resources that play a role in the overall security of the operating system. In effect, the operating system is "wrapped." Traditionally, only critical applications have been wrapped this way, leaving the operating system and all other applications defenceless. This makes as much sense as leaving a top-secret document in a safe, but leaving the safe in a public place where anybody has access to it. Sooner or later, someone will crack the safe and get to the document or, more drastically, someone will take the safe and hide it. Either way, access to the document is compromised. In the proposed approach, all applications can be protected equally, because the monitor sees and can stop all access to critical resources. Plus, the level of protection is not compromised if something sinister is added to the program at runtime: a malicious action is blocked no matter what its source.

As will be shown in the next section, many commercial products are available that use dynamic monitoring. Unfortunately, from experience, none of them uses the technique very effectively. These tools seem to be only ad hoc solutions developed as quick answers to a perceived growing need in the computer user population. They lack a theoretical and coherent background, which the current work tries to provide.

3 Related Work

This section presents related work in dynamic analysis only. For a survey of malicious code detection techniques in general, see [1].

Ironically, there are some commercially available solutions to the problem of COTS security, but most are derived from solutions that address network security. A survey of these along with a classification may be found in [11]. The classification has three categories:

1. Firewall: A tool that controls communication ports according to certain rules. eSafe Protect Desktop uses this technology to block communication ports.

2. Sandbox: Popularised by Java, this technique encloses an application in a virtual environment in which, theoretically, it cannot cause harm. eSafe Protect Desktop also uses this technology to prevent selected programs from accessing specifically enumerated resources.

3. Scanning: Much like virus detection tools, applications are scanned for known sequences of malicious code. Finjan's SurfinShield and Trend Micro's PC-cillin both use this technique.

The first technique is necessary in a secure environment. However, it has major limitations. It cannot protect from anything that passes the barrier. Therefore, it will not protect against malicious applications that are installed from CD-ROMs or other non-communication sources

The second approach is promising, but some say it is too complex to provide good security [5]. The authors disagree with this opinion but agree that a sound and somewhat formal methodology is required to build the sandbox and its rules. The monitor presented in this paper is a derivative of the sandbox technique.

The last approach has proven time and time again to be ineffective, yet it is by far the most common technique used in commercial tools. Known attacks can be scanned-for quickly and easily, so the technique should probably be used in some form, but it cannot protect against unknown attacks, and gathering all known attack signatures in a database is error-prone at best.

This is why researchers have for years been looking for a more proactive approach.

The Naccio project [3] addresses the problem by building tools that take as input an untrusted program and specification file and produce as output a new program that behaves exactly like the original program but with a guarantee that it satisfies the safety policy. This approach has some form of dynamic checking, but it is mostly static. The interest of this approach for the current work is that policies are defined in terms of abstract resource manipulations.

4 Architecture of Windows NT

This section covers the aspects of Windows NT architecture needed to implement a monitor such as the one presented in this paper. The intention is not to explain this architecture exhaustively but rather to situate the present work within it.

Solomon [13] presents Windows NT by explaining its operations, interactions and components. The designers of Windows NT built a layered architecture to allow component modularity, one in which each component has a precise role to play.

One of the significant aspects of the design is the approach to security. Windows NT separates the user-application environment from the kernel environment. User applications have their own reserved environment and it is forbidden for an application to reach the environment of another.

However, this restriction does not apply when code is executed in kernel mode: components of the kernel can reach any environment, reading data from or writing data to any application.

This last characteristic is very significant since it is the cause of many of the vulnerabilities in Windows NT. A malicious program that can succeed in executing kernel code has complete access to the operating system and all applications.

A series of articles on the kernel components of Windows NT by Russinovich [6, 7, 8, 9, 10] explains their operations precisely. He also designed tools to monitor these components. These tools, most of them with source code, are available at `http://www.sysinternals.com`. `Filemon`, `Regmon`, `Pmon`, and `TCPView` are among them. Respectively, they allow the user to monitor accesses to files (hard disk, RAM disk, CD-ROM, floppies, named pipes, and the mail slot), accesses to the registry (the configuration database of Windows NT), the management of processes and threads, and the TCP/IP communication ports. Study of these tools can greatly contribute to one's comprehension of the Windows NT architecture.

Figure 1, reproduced from Microsoft DDK documentation [4], highlights the Input/Output Manager component, central to the monitor's architecture. It is important to note the different drivers needed to process I/O requests, since most of the current work concerns these drivers.

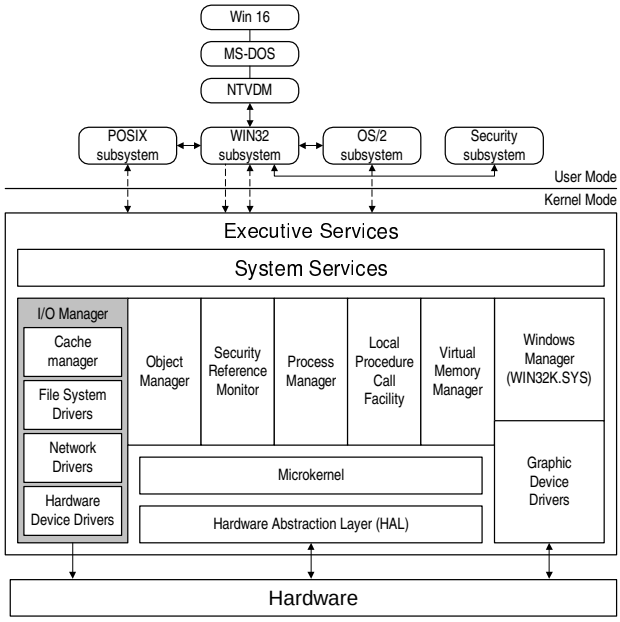

Figure 1: Windows NT Architecture

4.1 Types of Drivers

Conceptually, there are two levels of drivers in Windows NT. Low-level drivers are attached directly to a device such as a hard disk or a sound card and are responsible for the management of the hardware. High-level drivers are those with more general functions, such as the File System Driver (FSD). This driver receives requests to access files and distributes relevant tasks according to the file system in place, the partitions and the equipment. In all cases, it is possible to design a driver to replace another driver. The addition of a driver under Windows NT makes it possible to intervene at various levels. In practice, this introduces a third type of drivers called intermediate drivers.

Intermediate drivers are targeted in the work described because they can filter access requests. The design of this type of driver is different from the other two because they are not concerned with certain functions such as the initialisation of hardware. It is enough to define the various functions and to attach the driver to the driver directly below. In this way, the added driver can filter access requests before they are transmitted to a lower driver.

There are three types of intermediate driver. The generic driver uses the standard calls of the I/O Manager to send requests to other drivers. The second, the filter, is transparent and intercepts requests made to other drivers, also using standard I/O Manager calls. The third type is the joined driver, which forms a pair with another driver and interacts closely with it across a private interface.

Choosing the right type of driver is significant because the design of the monitor depends on the type or types used.

A driver supervising accesses to files, for instance, is regarded as a filter because its role is to intercept requests addressed directly to the driver below. Figure 2 shows an

example of layered drivers. This example refers to the file system filter driver and is explained further in Subsection 5.2.

5 Architecture of the Monitor

To develop an effective monitor, it is necessary to correctly identify the critical resources to be monitored, to make sure that a resource guard is well-positioned to receive all significant events, to create an expressive and complete language to state security policies, and to ensure that the implementation is safe and does exactly what it is supposed to do, and no more.

5.1 Identification of Critical Resources

Critical resources are usually recognisable because they contain significant or strategic information. A file containing all passwords can be regarded as *significant information*, whereas access to the creation of processes could be regarded as *strategic information*.

Although the inherent criticality of a particular resource is usually easily appreciated, this judgment must often be adjusted to account for the threats to which it is likely to be exposed.

On the Microsoft security web site (`http://www.microsoft.com/security`), users are warned about the discovery of new types of attack made against the firm's products and the solutions to these vulnerabilities. The U.S. Computer Emergency Response Team (CERT, `http://www.cert.com`) and its Canadian equivalent CanCERT (`http://www.cancert.ca`) also disseminate information about network security incidents. Another resource to learn about new attacks is the Common Vulnerabilities and Exposures (CVE) list, hosted by the Mitre Corporation (`http://cve.mitre.org`).

If different types of attack target the same resource, one must suspect that the resource is significant or strategic. Starting from information of this nature, it is possible to identify the sensitive resources that are potential targets for malicious attack.

The critical resources that have been identified for Windows NT are:

1. Files: Accesses to files are critical. System-critical, classified and personal information is stored in files.

2. Communication ports: It is essential to monitor what goes in and out of the box.

3. Registry: Windows NT configuration and much user information is stored in the registry. It is possible, for example, to modify the registry so that an unwanted application is executed at boot time without the user's knowledge.

4. Creation/destruction of processes and threads: It is important to know who creates what and when. Denial-of-service attacks often use an endless loop in which they create processes and/or threads to consume all resources.

Table 1: Typical actions carried out on critical resources

Resources	Actions
Files	Executed on hard disk, RAM disk, CD-ROM, floppies, named pipes, and mail slot - Create (Open)* - Read - Write - Rename - Copy - Execute - Transfer
Communication ports	Executed on ports and commonly used protocols - Create (Open)* - Send - Receive
Registry	Executed on keys, sub-keys, and values - Create (Open)* - Read - Write
Processes and threads	Executed on processes, threads and memory - Create - Stop - Kill - Modify a thread's state - Allocate memory

* The same call is used to create or open a resource, only parameters change.

At this time, it is believed that complete management of access to memory is not necessary: coherent management of these four resources will suffice. Table 1 shows these resources along with some of their corresponding actions.

5.2 Positioning the Filters

One way to collect information for a resource guard is to intercept relevant calls to the operating system libraries (`ntdll.dll`, for example). The major problem with that approach is that these libraries run in user mode. Such a "guard" would thus not be able to see everything executed in kernel mode. Since it is also possible to bypass user-mode libraries, this is definitely not a comprehensive solution.

To avoid possible bypassing, the guard needs to be low enough in the calling structure that it sees all calls made to a resource, but high enough that it does not need to manage lower devices. Intermediate drivers in Windows NT allow this positioning.

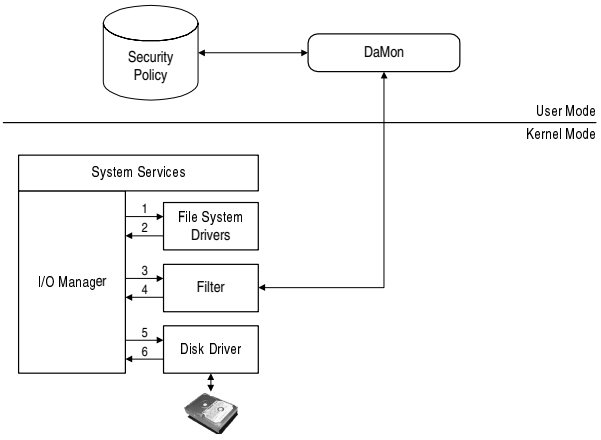

Figure 2: Positioning intermediate driver (filter)

Figure 2 presents the position adopted to manage file requests: a filter introduced between two existing drivers. The filter is inserted between the file system driver (FSD) and the device driver (disk). The file system driver (NTFS in this case) simplifies the filter's task because it already filters out requests based on Windows NT's file-access policy. When access is denied, the chain of calls does not reach lower drivers.

Port guards take a similar approach.

Registry guards and the creation/destruction of processes and threads are a different matter. Since the kernel handles these calls directly, another approach is required to intercept them. The calls to the registry are known and partially documented at `http://www.sysinternals.com`. A solution for hooking the creation and destruction of processes and threads has been implemented but is not yet entirely satisfactory. Information at this level is scarce at best and the team is still looking for better solutions.

6 Security Policies

A significant aspect of dynamic security is being able to react in accord with precise criteria. These criteria express the security policies that must guide the decisions of the monitor. In other words, each guard (filter) having the responsibility to supervise a particular resource must react to a given situation based on the security policies laid down by the user (or by a super user).

The difficulty with the design of a dynamic analysis monitor is that it must react to situations involving more than one resource. For example, if the file guard is in a cocoon and monitors only files, it would be impossible to protect against the sending of a private file to the network without exchanging information with the communication port guards.

The definition and design of rigorous security policies would not allow information leaks of this kind to occur. Schneider [12] proposes an interesting approach: an automaton to manage the collaboration of all components. This concept makes it possible to know the state of any particular sequence of actions. For example, it would be possible to stop the sending of private information in the preceding example if the automaton was in the state "read-in private information" by disallowing the transition to a state "send." This way, the port guard would be informed to deny the send request.

With this concept, the user can now define his security preferences using the language and architecture to be described next.

6.1 Language and Architecture

The Extensible Markup Language (XML) is a flexible way to create common information formats and share both the format and the data. XML is a simpler and easier-to-use subset of the Standard Generalised Markup Language (SGML), which was developed to provide a means to format and maintain legal documents.

The World Wide Web Consortium currently proposes XML as a formal recommendation. It is extensible because one can always define new markup symbols. This makes it a logical choice for the definition of security policies.

There are currently two ways to declare a document's format: Document Type Definition (DTD) and XML Schemas. DTD is the first and best-known document definition standard, but it is far from perfect. Its limitations have warranted the creation of a new approach: XML Schemas.

The purpose of schemas is similar to that of DTDs. Like DTDs, XML Schemas describe elements and their content models so that documents can be validated. But the approach can also process integers, dates, floats and so on directly, rather than treating them as strings, as DTDs do. It adds support for attribute groups and integrates namespaces. It is also an open-ended model, so it is possible to extend its vocabulary and the inheritance relationships among elements.

Schemas will be used in the complete version of this prototype. However, since DTDs are more compact and better known in the community, they will be used in this paper.

A Security Policy Markup Language (SPML) is defined in this section. With this grammar, it is possible to define two types of security policy and use them in a monitoring system. The next subsection presents a few terms needed to define this grammar.

6.1.1 Elements of a Security Policy

A security policy is defined in SPML as one or many rules. A security rule is one particular action while the policy includes all possible actions.

To enforce security, it is necessary to know exactly what is under surveillance and from whom. "Who" identifies the *principal*, the person or application that wants to perform an action. "What" identifies the *resource* that the

```
<!ENTITY Security_Policy_Version "0.5.0.0b">

<!ELEMENT Security_Policy (Type_SP?, Actors+, Request+, Reaction+)>

<!ELEMENT Type_SP ((ID,Author*)|ID_Key)>
<!ATTLIST Type_SP Priority_Level (0 | 1 | 2 | 3 | 4) "4">
<!ATTLIST Type_SP Integrity_Key CDATA #IMPLIED>
<!ATTLIST Type_SP Propriety_Type (Surveillance | Initialisation) "Surveillance">

<!ELEMENT Actor (Principal+, Context?)>
<!ATTLIST Actor Role (Source | Destination | Resource) "Resource">
<!ELEMENT Context ((ID, Description?, ActiveComputer_Context) | ID_Key)>
<!ATTLIST Context Security_Level (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) "9">
<!ELEMENT ActiveComputer_Context (Computer_Name*, OS*, Period_Range*, Disk_Space?,
   Pc_Virtual_Memory_Free?, Pc_Physical_Memory_Free, Pc_CPU_Used, Pc_Write_HD,
   Pc_Read_HD, Pc_Read-Write_HD)>

<!ELEMENT Request ((ID,List_Request+) | ID_Key)>
<!ELEMENT List_Request (((Request_Attribute, Transfer)?, (Request_Attribute, Rename)?,
   (Request_Attribute, Copy)?,(Request_Attribute, Delete)?, (Request_Attribute, Write)?,
   (Request_Attribute, Read)?, (Request_Attribute, Create)?) |
   ((Request_Attribute, Send)?, (Request_Attribute, Listen)?,
   (Request_Attribute, Receive)?))>

<!ELEMENT Reaction ((ID,(Log_Success?, Log_Error?, Log_Read?, Log_Write?),
   (Verify_Status, Pre-Request?, Request_Succeeded?, Request_Failed?)?) | ID_Key)>
```

Figure 3: Excerpt of the security policy grammar

principal wants to use. A *request* is made by a principal to use a resource. A security rule defines *reactions* to be performed when the access is granted or when it is denied.

A principal may be an application, if this application uses a resource. An application can also itself be a resource. To clarify this situation, the term *actor* is used to identify a principal who has a role in the security policy. A *role* may be a principal, a destination, or a resource role, depending on the context of the request.

6.2 Definition of SPML

Figure 3 shows an excerpt of the security policy grammar.

The `Security_Policy_Version` entity identifies the SPML version. It is useful to keep track of versions to know if a tool can interpret a particular version.

The `Security_Policy` element is the root of a security policy. It identifies the type of the policy, the actors within it, the possible requests to verify, and the reactions to take on authorised and unauthorised requests.

6.3 Resolution Algorithm

The definition of the SPML is used in two ways, depending on the `Type_SP` attribute: to initialise the database and for the surveillance phase. The initialisation phase is, of course, required for all databases and will not be described further.

The algorithm for the surveillance phase of the monitor is shown in Figure 4. When a guard receives a request, it notifies the monitor, who identifies the players, checks if an automaton exists for this context and creates one if not. Then it verifies if the automaton is already in an "unauthorised" state — this can happen if multiple invalid requests are made, for example — and if it is, it cancels the request and executes the appropriate reaction. If the automaton is in a correct state it gets the corresponding security rules (SR) from the database, resolves the user rights according to heritage and priority rules, executes the appropriate reaction(s) and changes the state of the automaton. Then it waits for the next request.

In the actual implementation, of course, some caching and other optimisation actions are performed.

6.4 Policy Examples

Going back to the example, there are many ways to express the rule "once critical information is read, it cannot be sent to the network." This subsection proposes three.

For the sake of brevity and clarity, a simplified notation is used rather than actual SPML. Evidently, a real-life policy would have to take more complex cases into account.

The first thing to do is to initialise the policy database. Each principal has a name, a security level (0–public, 1–secret, 2–top secret) and a status. The status can be "active", which means that the principal is alive in the policy, or "under surveillance," which means that it is under close watch. One principal is defined as a group containing all principals of level secret or higher. Another principal is a document named `Secret.doc`, which requires a secu-

```
Initialisation phase
    Principal = GroupSecret, Contains(Security Level >= 2), Status = Active
    Principal = Secret.doc, Security Level = 2, Status = Active

Solution 1
    SR1:    Actor Source = GroupSecret;
            Actor Resource = Secret.doc;
            Authorised Request = Open, Read, Write, Rename
                Authorised Reaction = {};
            Unauthorised Request = Send
                Unauthorised Reaction = "Send email with context to security
                                         administrator".

Solution 2
    SR1:    Actor Source = Security Level;
            Actor Destination = (Security Level lower than Source Security Level);
            Authorised Request = {}
                Authorised Reaction = {};
            Unauthorised Request = Send
                Unauthorised Reaction = "Send email with context to security
                                         administrator".

Solution 3
    SR1:    Actor Source = GroupSecret;
            Actor Resource = Secret.doc;
            Authorised Request = Open, Read, Write, Rename
                Authorised Reaction = (Actor Source Status = "under surveillance");
            Unauthorised Request = {};
                Unauthorised Reaction = {}.
    SR2:    Actor Source = (status = "under surveillance");
            Actor Resource = Secret.doc;
            Authorised Request = Close
                Authorised Reaction = (Actor Source Status = "Active");
            Unauthorised Request = Send
                Unauthorised Reaction = "Send email with context to security
                                         administrator".
```

Figure 5: Examples of Security Policies

```
Receive an Actor's Request from a guard
Identify Actor and Request
IF automaton for context does not exist
    Create automaton state for context
IF automaton state == "unauthorised"
    Cancel Request
    Execute unauthorised reaction
ELSE
    Use context to get SR from database
    Resolve user rights
    IF Request satisfies SR
        Let request through
        Execute authorised reaction
    ELSE
        Cancel Request
        Execute unauthorised reaction
    Take automaton transition
Wait for next request
```

Figure 4: Surveillance algorithm

rity level of secret to be accessed. This is the initialisation phase shown in Figure 5.

Figure 5 also shows the three different solutions, which are explained below.

The monitor automatically blocks an unauthorised request and notifies the user, so the reaction of blocking the request is not mentioned in the policy.

6.4.1 Solution 1 - Check the Actor Name

This is the simplest solution. Upon request, the `Actor Source` attribute is instantiated with a member of the `GroupSecret` group. All members of the "secret" group can open, read, write, and rename the file `Secret.doc`, but cannot send it. If they try, the request is blocked and a message is sent to the administrator.

6.4.2 Solution 2 - Check the Security Level

This policy protects against the sending of protected information to a destination having a lower security clearance.

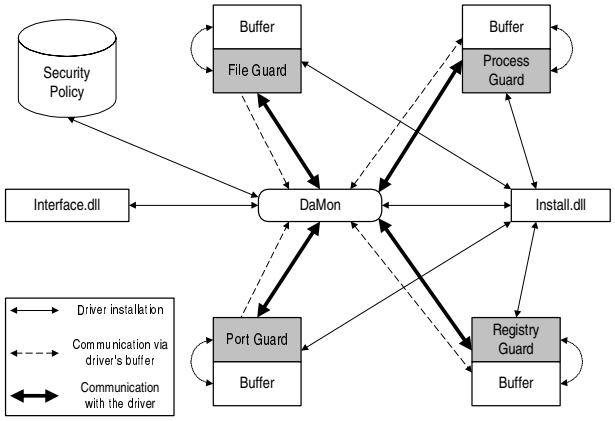

Figure 6: DaMon architecture

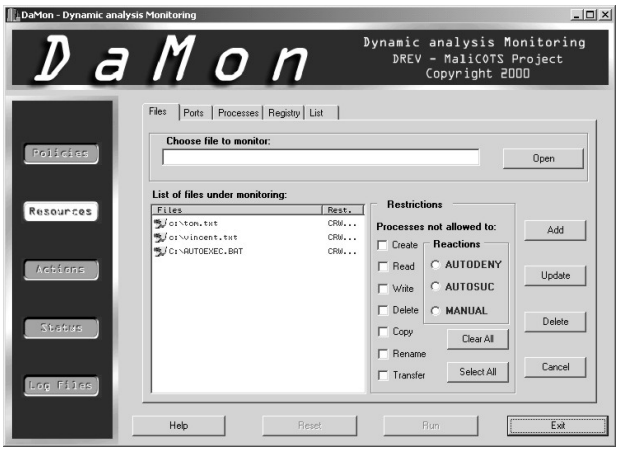

Figure 7: The file-monitoring interface of DaMon

Any such attempts are blocked and a message is sent to the administrator. This policy is general and can be applied whether the destination is known or not. Of course, all unknown destinations would be assigned a security level of zero.

6.4.3 Solution 3 - Check the State of the System

This is the most interesting solution. It uses the state of a system to protect it. If a principal reads or opens `Secret.doc`, then its status is set to "under surveillance." While under surveillance, the second security rule specifies that a principal cannot send anything. When the file is closed, sending privileges are re-established by changing the status back to `Active`.

7 DaMon: A Monitoring Prototype

The DaMon[1] prototype implements several of the ideas expressed in this paper, including a rudimentary implementation of security policy and the provision of means by which a driver can prevent certain activities.

7.1 Architecture of DaMon

The architecture presented in Figure 6 is based on the characteristics previously mentioned.

The core of the application (`DaMon.exe`) communicates with all other components. Certain modules are implemented in the form of dynamically linked libraries (DLLs) that are loaded on the core's request.

`Instdrv.dll` has the role of loading and initiating communications between the core and the drivers. This is done only once when the application starts.

`Interface.dll` regroups all the information concerning the display of information.

Since the interfaces for all resources are very similar, only the aspect of file monitoring is presented here. Other screen captures of the prototype are shown in Annex A.

[1] Dynamic Analysis Monitoring

7.2 Monitoring Files

Figure 7 shows the file-monitoring interface of DaMon. This interface indicates the files that are currently being monitored, along with their associated permissions. In the case shown, for example, the file `c:\autoexec.bat` is under surveillance. Note that the full path is required because files in different locations can have the same name — `readme.txt` or `setup.exe`, for example. The files to be monitored are specified in this interface.

7.2.1 Starting File Monitoring

Clicking on the `List` tab opens the file-monitoring interface itself (Figure 8) and clicking the start button begins surveillance.

`autoexec.bat` has been set up as a file under surveillance, and Figure 8 shows the results of reading it — by executing the command `sysedit.exe`, for example. The `Actions` column lists the basic operations included in I/O calls needed to simply read a file. A lot of information is included, allowing a very good level of granularity. With these basic operations, it is possible to intercept a request, check to see if it is allowed under the security policy and react accordingly.

8 Conclusion

Use of COTS software is increasing, largely because of time and budget constraints. This situation poses great concerns about safety, security, and reliability in critical information systems.

Some research has been done to address this problem, most of it involving static techniques that attempt to provide a proof of some sort that an application will behave correctly for all possible inputs. A number of powerful solutions have emerged from this research, including certifying compilers.

Nevertheless, there will always be situations in which static analysis techniques are powerless to prevent mali-

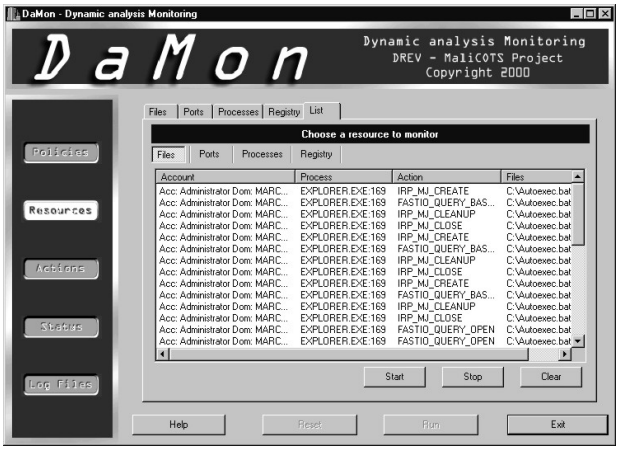

Figure 8: Monitoring `autoexec.bat`

cious action. The most convincing examples are viruses. No matter how good a static analysis is, it will never be able to detect malicious code that is inserted after the analysis has been completed.

This is why a good watchdog is required that will enforce a security policy dynamically. It is the authors' belief that the monitor presented in this paper fits this description.

The prototype, called DaMon, is already capable of stopping certain malicious actions based on combined accesses to critical resources (files, communication ports, registry, processes and threads) according to rudimentary security specifications. A more complete and expressive security specification language using XML is under development and will be fully integrated with the prototype in the coming year.

References

[1] J. Bergeron, M. Debbabi, J. Desharnais, B. Ktari, M. Salois, N. Tawbi, R. Charpentier, and M. Patry. Detection of Malicious Code in COTS Software : a Short Survey. In *First International Software Assurance Certification Conference (ISACC'99)*, Washington D.C., Mar. 1999. Section C1.

[2] R. Charpentier and M. Salois. Detection of Malicious Code in COTS Software via Certifying Compilers. In *Commercial Off-The-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS"*, Neuilly-sur-Seine Cedex, France, Apr. 2000. NATO, RTO.

[3] D. Evans and A. Twyman. Flexible Policy-Directed Code Safety. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 1999. IEEE. `http://naccio.cs.virginia.edu`.

[4] Microsoft Corporation. *Microsoft Windows NT 4.0 Driver Development Kit Documentation*, 2000. `http://www.microsoft.com/ddk/`.

[5] D. Neeley. How to Keep Out Bad Characters. Security Management Online, 1998. `http://www.securitymanagement.com/library/000599.html`.

[6] M. Russinovich. Inside NTFS: NT's Native File System — Past, Present, and Future. *Windows NT Magazine*, 1(27), Jan. 1998. `http://www.winntmag.com/Articles/Index.cfm?ArticleID=3455`.

[7] M. Russinovich. Windows NT Architecture, Part 1. *Windows NT Magazine*, 1(29), Mar. 1998. `http://www.winntmag.com/Articles/Index.cfm?ArticleID=2984`.

[8] M. Russinovich. Windows NT Architecture, Part 2. *Windows NT Magazine*, 1(30), Apr. 1998. `http://www.winntmag.com/Articles/Index.cfm?ArticleID=3025`.

[9] M. Russinovich. Windows NT Security, Part 1. *Windows NT Magazine*, 1(31), May 1998. `http://www.winntmag.com/Articles/Index.cfm?ArticleID=3143`.

[10] M. Russinovich. Windows NT Security, Part 2. *Windows NT Magazine*, 1(32), June 1998. `http://www.winntmag.com/Articles/Index.cfm?ArticleID=3492`.

[11] M. Salois and R. Charpentier. Dynamic Detection of Malicious Code in COTS Sofware. In *Commercial Off-The-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS"*, Neuilly-sur-Seine Cedex, France, Apr. 2000. NATO, RTO.

[12] F. B. Schneider. Enforceable Security Policies. Department of Computer Science, Cornell University, New York, `http://www.cs.purdue.edu/homes/jv/smc/bib/content.html`, Jan. 1998.

[13] D. A. Solomon. *Inside Windows NT*. Microsoft Press, 2^{nd} edition, 1997.

Annex A: Screen Captures of the Prototype

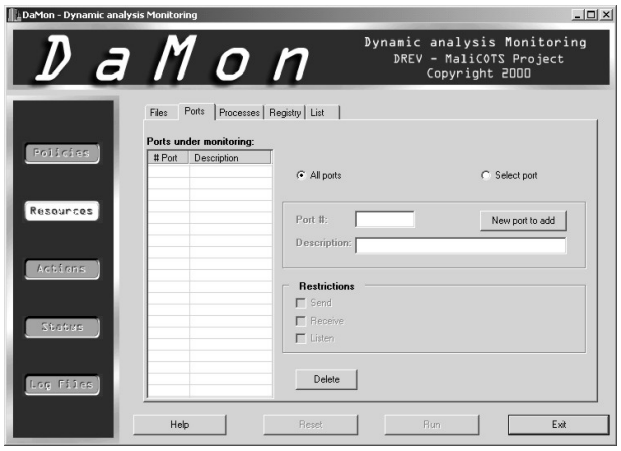

Figure 9: Configuration of port monitoring

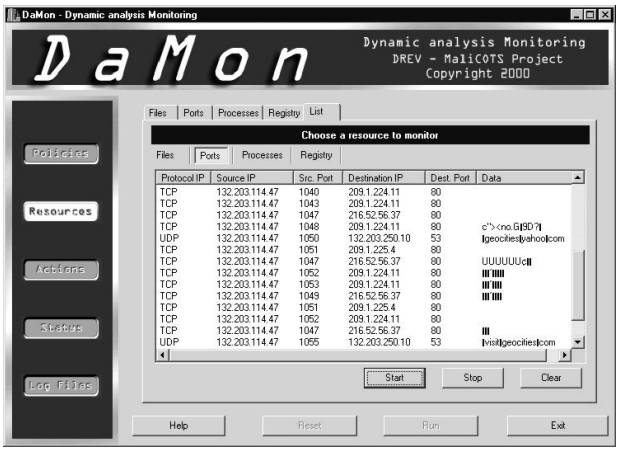

Figure 12: A view of captured TCP and UDP activities

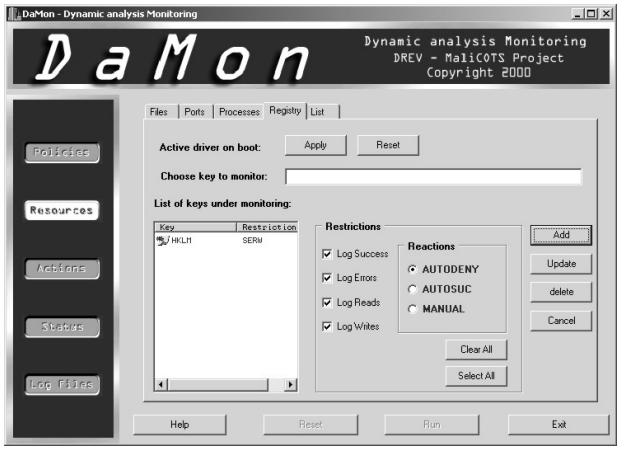

Figure 10: Configuration of registry monitoring

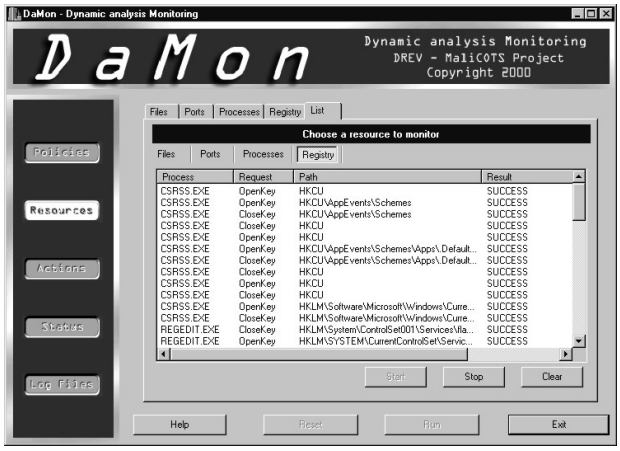

Figure 13: A view of accesses to registry keys

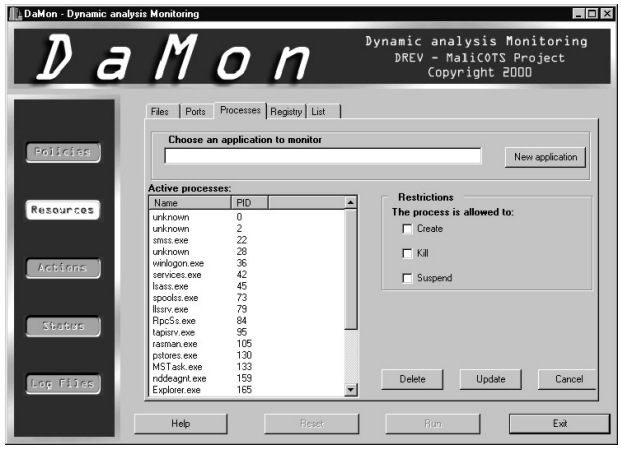

Figure 11: Configuration of process monitoring, with a snapshot of the processes currently running

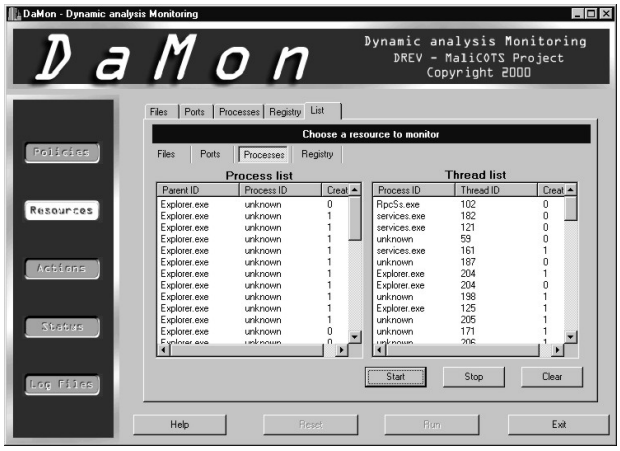

Figure 14: A view of captured creation/destruction of processes and threads