Moderating the Execution of Applications on Win32 Platforms

Matthew Schmid, Frank Hill & A.K. Ghosh
Cigital, Inc.
21351 Ridgetop Circle, Suite 400
Dulles VA 20166
{mschmid, fhill, aghosh}@cigital.com

J.T. Bloch
Widevine Technologies, Inc.
1301 5th Ave. Suite 1300
Seattle WA 98101
jt@widevine.com

Abstract

This paper describes an approach and tool for providing administrative control over the execution of software on a Windows NT/2000 system. The kernel-driver–based approach provides the system administrator with a way of restricting users to running only approved applications. As a result, illegal, pirated, personal, and malicious software executables can be prevented from running on corporate machines. We describe the key issues involved in the development of this tool and the features that make this tool an important part of regaining enterprise-wide control over corporate machines.

1. Introduction

Today, end users have tremendous control over corporate machines, for better and for worse. Capable technical workers require the ability to be able to modify their configurations, create and install new programs. Today’s operating system platforms provide this flexibility. However, in the hands of unscrupulous or non-savvy professionals, this capability can be misused, and even pose a threat to corporate systems. One of the greatest challenges for today’s Chief Information Officer (CIO) and system administrators is maintaining control over the corporate information infrastructure. The CIO is ultimately responsible for the software that runs on his or her company’s machines. Unlike appliances, traditional factory machinery, and even mainframes from yesteryear, today’s desktop computers give every employee the ability to modify his or her own working equipment simply by installing new software on his or her own machine. As a result, the CIO actually has very little control over what software is running on corporate machines. The risks include: pirated software, unlicensed software copies, warez, personal software programs, unauthorized shareware, and malicious software executables. The end user need not be a malicious conspirator, either. An administrative assistant to the Chief Financial Officer (CFO) can be easily duped into running an email attachment that ships financial balance sheets off-site. While today’s corporations often have policies regarding installing and running unauthorized software, they do not have the tools necessary to enforce these policies. In this paper, we describe a tool for this purpose.

In this paper, we describe an approach and tool for providing administrative control over which software executables are allowed to run on Windows NT/2000 machines. The Execution management utility (Emu) is designed to prevent software from executing without the prior approval of system administrative staff. Our approach centers on the notion of execution control lists (ECLs), which are implemented by a loadable Windows NT kernel module that is capable of selectively intercepting process creation requests. The Emu prevents unauthorized programs (i.e., programs that do not appear on the client’s ECL) from executing by intercepting and denying process creation calls from unapproved programs. For additional security, the Emu can be configured to prevent the loading and execution of Dynamic Link Libraries (DLLs). Because our approach intercepts kernel-level requests, the user cannot bypass it without obtaining administrative privileges. Also, while commercial audit programs exist to determine what programs run on a machine, none, to our knowledge, prevent the execution of unauthorized software. It is, of course, possible for a system administrator to configure Windows clients so that the end user does not have local administrative privilege. This configuration option will
stop installation of applications that modify the registry. However, it is easy to circumvent this protection by copying executables directly to the file system and running them without needing to modify the registry. Many instances of malicious warez fall into this category of application. Our approach prevents such workarounds as well as other tricks such as re-naming executable files.

The Emu can assist corporations by enforcing policies on an enterprise-wide level by preventing the execution of unauthorized, unlicensed or pirated software. Games and entertainment software, or even non-standard utilities (for example, the advertising-enhanced browsers offered by AllAdvantage.com), can be banned or carefully controlled. The Emu also addresses the threat of unknown binary executables compiled for the Windows 32-bit (Win32) platform. Malicious and unknown executables downloaded from the Internet, or installed by physical media by insiders will be prevented from running.

The benefits of the Execution management utility are:

- Stop unknown and potentially malicious software from being executed.
- Warn administrators or audit when unknown and possibly malicious code is attempting to execute.
- Allow corporations to be in compliance with their software license agreements.
- Prevent unauthorized software such as personal software, games, and other forms of entertainment, from being run.
- Identify when a permitted software program has been corrupted or infected with unknown viruses.
- Prevent the execution of unauthorized code modules such as DLLs or ActiveX controls.

In summary, we are providing a kernel-level protection mechanism to protect corporations and systems against compromise by unauthorized software. Though our approach does address the broad spectrum of malicious executables, it does not address specific threats from script-based attacks, Java applets, or other code technologies that run within interpreters, unless the interpreters themselves are banned from running using the Emu. Furthermore, our approach does not address threats from authorized programs that may include Trojan horses. However, unlike current virus protection technology, our approach is powerful enough to prevent new variations of known malicious software as well as future and unknown malicious executables from executing.

This paper begins by taking a brief look at some related security work with goals similar to our own. With this background in hand, we discuss the tool that we developed for protecting against the execution of unknown and potentially malicious programs. The paper concludes by illustrating some interesting implementation issues that we encountered during the development of this tool.

2. Related Work

There are several application areas where general-purpose machines are used for application-specific functions. Examples are catalog search engines within libraries, Web and email kiosks, and even cash registers in retail applications. While these machines are supposed to run only one or just a few specific programs, they typically run on a general-purpose machine that may have other software applications, or the ability to install new applications. To date, the mechanisms for preventing new software installations have been very weak. Most simply only install the application(s) necessary for the function and count on the good will or lack of knowledge of users to install new software. Some vendors will use enforcement mechanisms that only allow specific applications to run. However, the current state-of-the-art in this area is to allow programs to run only if their file handle or windows name matches the allowed name. The Advanced Security Control (ASC) tool developed by SmartLine [1] works using this approach. This product lets administrators set up rules for the local computer, defining when any program can be executed and by whom. It also has the capability to reject unknown applications. However, its approach renders this product inadequate for our needs. ASC identifies known applications based solely on the application’s executable name. Therefore even a novice user can bypass this security by naming a malicious application after an approved one. We develop a technique in this paper that cannot be defeated by the user, unless the user gains administrative privileges, in which case the user effectively owns the machine. Our technique depends on Windows security from preventing non-privileged users from obtaining administrator privileges.

The closest work of rigor when it comes to preventing unknown or possibly malicious applications from running derives from the computer security domain. The first approach that is often cited when providing protection against malicious code is sandboxing applications. We provide a brief overview of sandboxing techniques in order to distinguish them from our approach. A number of researchers have explored the use of application sandboxes as a technique for restricting the actions of untrusted applications. The basic idea behind a sandbox is specify the resources that an application can access and the type of access that the application is granted. If the application tries to access a restricted resource, the
sandbox interferes and rejects the access attempt. Researchers at the University of California at Berkeley developed the Janus prototype that sandboxes applications running on a Solaris operating system [2]. The sandbox operates by intercepting system calls, and deciding to allow or deny them based on the parameters to the call. The decision-making logic is contained in a configuration file that is specified for each executable that is sandboxed. Now, a number of commercial sandboxing products exist, including Finjan’s SurfินShield [3], and Aladdin’s eSafe Protector [4].

Another approach to malicious software that has been taken is to periodically check file integrity. The idea is that one can detect intrusions after the fact by frequently checking the integrity of critical system files or key target programs.

One file system integrity-checking tool is Tripwire [5], which is currently marketed as a commercial product. This product maintains cryptographic hashes of critical files and periodically verifies these hashes, notifying an administrator if any changes have been made to the files. This technology is used primarily for detecting program corruptions or changes due to intrusions. Unfortunately, this product does not combine any kind of execution control with its integrity checking. Therefore, an executable that has been compromised by a virus (and has therefore had its signature altered) will be allowed to execute until the Tripwire system has detected this change, and an administrator has removed the compromised executable. A significant amount of damage could be done before the problem is detected. Furthermore, new executables loaded on the system (such as malicious software) can run without detection.

The goal of our work is quite different from that of the aforementioned projects. We are not concerned with controlling the behavior of programs, but rather, about controlling which programs execute. In the 2000 Virus Bulletin Conference one of the authors suggested that a system like the one that we have developed might go a long way towards preventing viruses and other forms of malicious software [6]. We believe that not only is that statement correct, but that there are a myriad of other benefits as well. Because of the perceived weaknesses in existing software designed to prevent the execution of unknown or undesirable applications, we have developed a prototype tool that provides such control in a secure and easily administered fashion. We believe that this prototype solves many of the problems left by current state-of-the-art software, and provides administrators with a valuable defense against the introduction of unauthorized or hostile applications. The approach is specifically geared towards the enterprise-wide management of Win32 machines, and not towards the individual or home user.

3. Execution Management Utility

The principal idea behind the execution management utility is to restrict a user to executing only an approved and known set of applications. This set typically includes all the operating system software necessary to run application software, operating system services, and a set of desktop applications necessary for the user to perform his or her duties. In addition to the base set of software that many users or groups of users will have in common, individual users may have different software needs. For example, a secretary may need access to a different set of applications than an accountant requires. The Emu provides a client-server architecture that enables an administrator to have complete control over the applications available to any user working on an NT workstation with the Emu client installed.

Why place the execution management utility under the control of system administrators? Bruce Schneier points out that under the best of conditions, users cannot make good security decisions, let alone under the conditions today in which malicious code finds various ways to install and execute on users’ machines [7]. Thus, rather than deploying a solution that counts on the user to make security decisions, the Emu gives that responsibility to an appropriate authority, such as a security officer or system administrator who is responsible for the integrity of the organization’s computing infrastructure.

By giving administrators the ability to restrict the applications that users can access, the Emu can protect an organization against a wide range of threats. Running email attachments or downloading executables from the Internet is a common way for a user to damage a system. The threats posed by such executables range from the intentionally malicious virus or Trojan horse to the well meaning but poorly behaving shareware with which we are all familiar. The damage caused by such executables is made more difficult to identify and repair because the system administrator often does not know what applications an employee may have run.

3.1. The Execution Control List (ECL)

Central to execution management is the Execution Control List or ECL. Each user on the system has his or her own ECL that specifies which programs the user is allowed to execute. The combination of the executable name along with its MD5 hash uniquely identifies the application. Identifying a program through its hash ensures that a user cannot bypass the Emu simply by
changing the name of an unknown executable to the name of an executable that is already part of the ECL. Using hashes for identification has the additional benefit of preventing modified applications from executing. Many viruses propagate by silently attaching themselves to an application and then executing whenever the application is run. If a virus alters a program in any way, then its MD5 hash will be changed. Therefore, the Emu ensures that modified executables will not run, effectively protecting the system from any damage these altered applications can do.

3.2. The Emu client

The Emu client consists of a Windows NT service and device driver that runs on a protected client machine. Fortunately Windows NT provides the necessary security to ensure that the service and device driver will always be executed at boot time, and that only an administrative user can stop them from running. The function of the Emu client is to intercept execution requests and verify whether the requesting user has permission to run the requested application. The device driver is responsible for intercepting the execution request as it is passed to the kernel. Because the device driver is located in kernel space there is no way for a user process to make operating system calls that bypass this component. This is further discussed in Section 4.

After intercepting an execution request, the Emu hashes the requested executable and then looks for this name/hash combination in the user’s ECL. The reason that the name is used in addition to the hash is to help the user to be sure that he or she is actually executing the program he or she intended to execute. For example, the name/hash verification would prevent an attack in which a program that is supposed to encrypt a file is replaced with one that is supposed to delete a file (assuming that both of these are in the user’s ECL). If the executable is found in the ECL, then the Emu client will allow the application to be executed as usual. In this case, the user is completely unaware that the Emu has performed any action at all. If the ECL does not contain an entry for this executable, then the Emu client will prevent the program from executing and pop-up a dialog box to inform the user that the program they tried to run is not permitted. This dialog box is further discussed in section 3.4.

3.3. ECL management and the Administrative Server

Up to this point, we have assumed that the Emu client has direct local access to a user’s ECL. It would be very wasteful, however, to require each Emu client to maintain a collection of ECLs for every user on the system. Furthermore, we want to be able to establish an ECL that follows a user to any Emu controlled machine on a corporate network. This is where the Emu’s Administrative Server (AS) comes into use.

The Administrative Server maintains a centralized repository of ECLs for all users on a system. The server fields requests for user ECLs and distributes them to Emu clients via the local area network. Each ECL that the AS maintains contains a cryptographic signature of the ECL’s contents. When an Emu client receives a user’s ECL from the AS, it immediately tries to verify the signature on the ECL. If the signature is valid (indicating that the ECL originated from the AS and has not been modified) then the ECL is accepted for use by the client. An invalid signature results in the rejection of the ECL.

The Emu client may request an ECL from the Administrative Server under several different circumstances. The first occasion is when the Emu client does not have a local copy of an ECL for the requesting user. For efficiency, the Emu client does not have to contact the AS each time a user attempts to run an application. The Emu client keeps a time-stamped local copy of any local user’s ECL, and the Emu client uses this copy of the ECL. It is up to the administrator to determine how often this local ECL copy should be refreshed from the AS, which is the second situation in which the Emu client would request an ECL from the server. The third reason that the Emu client would request a user’s ECL from the Administrative Server is when the server has signaled the client to refresh its ECL. This might occur, for instance, after the administrator has finished making changes to an ECL and wants the client to use the newly updated ECL.

Maintaining a copy of the local user’s ECLs enables the Emu client to continue functioning when the Administrative Server is not available. This may occur when the client machine is temporarily removed from the network, the AS’s machine is malfunctioning, or the system is experiencing network problems.

A default Execution Control List, locally cached by all Emu clients, provides a very restrictive ECL used only when the Emu client does not have an ECL for the requesting user and cannot obtain an ECL for that user from the Administrative Server. Providing a default ECL ensures that any user who can log onto an Emu-protected workstation will be provided with a basic ECL that has been pre-defined by the administrator.
3.4. Role of the Administrative Server

To make itself useful on an enterprise level, the Emu provides a means for centralized administration. An administrator can manage policies and control Emu functionality from a single machine running the Emu’s Administrative Server. As previously discussed, the AS maintains a central repository of ECLs and is responsible for distributing digitally signed copies of these policies to any Emu client that requests them. Another role of the AS is to handle users’ requests to add applications to their ECLs. A user can generate such a request when the Emu client rejects the execution of an application that is not on the user’s ECL. When the Emu client prevents an application from executing, it generates a dialog box indicating that the requested executable is not on the current ECL, and it asks the user if she would like to request it to be added to their ECL. If the user opts to submit a request, then an ECL add request is forwarded to the AS. If the AS is unavailable, then the execution request is simply rejected.

The add request consists of the user name, program name, and program hash. The AS maintains a queue of add requests awaiting approval or rejection by the administrator. To make the administrator’s job easier, the AS maintains a database of known application hashes. If an add request is for a known application, then the AS informs the administrator of this positive identification. If the program’s hash is unknown, then the administrator may want to investigate the application further before allowing a user to add it to their ECL. If the administrator chooses to accept an add request, the user’s ECL is updated and the workstation that requested the change is signaled to update its policy.

3.5. ECL creation

Once the Emu has been installed on a workstation, only the programs listed in the ECL can execute on that workstation. In practice, we need a facility to legitimately add programs to this list. The Emu provides three methods for adding programs to this list: the learning mode, the ECL editor, and the user request. We have already discussed how the user request functions.

The most straightforward technique for developing an ECL is by using the ECL editor. This simple utility allows an administrator to select executable files that should be added to a user’s ECL. The administrator may also use the ECL editor to modify existing policies. Additionally, the editor can make changes to groups of policies all at once. This functionality is useful when, for example, the administrator decides that all users in a group should be able to execute certain applications. However, developing an entire policy by using the ECL editor would be extremely tedious, so the Emu provides other methods for policy development.

The Emu’s learning mode provides a means for developing an ECL policy that is customized to individual users. An administrator uses the Administrative Server to put a user into learning mode or take her out of learning mode. Once in learning mode, an ECL is generated by observing the application usage patterns of a user. During learning mode, the Emu passively observes program execution and does not attempt to enforce any ECL restrictions. Each application that the user executes is added to his or her current ECL. The administrator is the only person with access to learning mode.

Developing custom policies through the use of the learning mode enables an administrator to limit a user to those programs that are necessary and sufficient to perform his or her job. These functions include operating system utilities and services that start up on machine boot. After allowing a user to work under learning mode for a given period of time, the administrator may then review the ECL that has been created and make any desired changes. Defining an appropriate length of time for a user to remain in learning mode will depend on circumstances specific to each work environment.

A final method of developing an ECL for a user is by white-listing all executables that exist on a user’s machine. After installing all of the software that a user needs, an administrator can use the Emu to add all of the existing executables to a user’s ECL. This essentially creates a trusted computing base that consists of well-known software that an administrator has personally installed. The Emu will prevent the user from running any software that was not part of this original installation. The administrator can further fine-tune a user’s ECL to remove any applications that the user should not be running (for example registry editing tools).

4. Developing the Execution Management Utility

This section describes our kernel-level wrapping approach to execution control. We discuss the rationale behind our approach and briefly examine other wrapping techniques. We also highlight some of the interesting implementation issues that we encountered and present our solutions to these problems.

To achieve our goal of moderating execution requests we require a means of intercepting those requests and filtering them according to the rules set up by the EMU. The technique that we will use is known as function
wrapping. A wrapper can be loosely defined as code that is invoked in place of the intended function, allowing for additional processing to take place both before and after the targeted function call.

There are several levels at which a function wrapping system can be implemented. Wrapping can be performed in user-space by intercepting function calls between Windows applications and the Win32 subsystem. In [8] the authors present a user-level wrapping approach for Windows operating systems. They demonstrate that their approach can be applied to issues involving security, data integrity, and application integration. Microsoft Research has also made available a generic tool for performing library call wrapping at the user level [9]. Similarly, in [10] the authors describe a user-level wrapping system for the Solaris and FreeBSD operating systems. Their goal was to use software wrappers to harden COTS software for use in mission critical systems. Neither of these tools is ideal for our purposes because we are interested in wrapping Windows NT kernel-level system calls.

The benefit of working with user-level wrapping systems is that they allow for easy development and do not require any modifications to the operating system. Additionally the level of information that is available as function parameters is often more useful than that found within the kernel. The drawback of user-level wrappers is that they are very difficult to implement securely and are often trivial to circumvent. One reason for this is that while most well-behaved Windows applications make system calls through the Win32 API, it is also possible to bypass this interface by making calls directly to the kernel services. Nothing in the Windows NT kernel prevents making direct calls to executive objects. A malicious program cognizant of a user-level wrapping system may attempt to bypass the system by making direct calls to operating system objects, rather than going through the standard Win32 API.

Implementing our wrapping system within the Windows NT kernel will provide us with the security and non-bypassability needed for an effective implementation of the Emu. In [11] the authors implemented a kernel level wrapping system for the Linux operating system. They have suggested that the non-bypassable nature of their approach is ideal for security applications such as restricting a process’s resources. On the Windows NT operating system the only method of extending the OS with user-written code is to develop a device driver. Writing a device driver provides access to internal operating system functions and data structures not accessible from user mode. Device drivers are installed via the I/O subsystem.

4.1. Implementing the kernel driver

System services are operations performed by the operating system kernel on behalf of applications or other kernel components. These operations are implemented as system services because they may affect processes other than the caller. For instance, manipulating CPU hardware, starting and stopping processes, and manipulating files are sensitive operations generally implemented in the kernel.

In Windows NT, user applications invoke system services by executing an interrupt instruction. Code in the kernel takes control of the machine in response to the interrupt and performs some useful activity for the calling process before relinquishing control. A kernel entity known as the dispatcher initially responds to the interrupt instruction, determines the nature of the interrupt, and calls a function to handle the request. Two tables in kernel memory describe the locations and parameter requirements of all functions available to the dispatcher. One table specifies handlers for user requests; the other specifies handlers for requests originating within the kernel. The calling process places information about the requested system service on the stack along with any parameters required for completing the operation [12].

Our method of controlling new process creation relies on our ability to instruct the dispatcher to call a specific function when a user process invokes certain system services. This approach requires constructing a device driver that is loaded into the kernel, dynamically, or as part of the boot sequence. When our driver is loaded, it modifies an entry in the table consulted by the dispatcher to handle an interrupt instruction. In our case, we are interested in intercepting calls to the ZwCreateProcess function. This function is responsible for starting all user- and kernel-mode processes. The modified entry causes the dispatcher to call our function instead of ZwCreateProcess [13]. Our function will be called whenever a user-mode application is executed. The signature of our function is identical to that of ZwCreateProcess, so the kernel interface exported to applications is not altered [14].

Once the dispatcher calls our function it determines whether to allow or block the process creation. If our wrapping function wishes to allow the creation of a new process, it invokes ZwCreateProcess with the user’s parameters and passes the results to the caller. Otherwise, it returns a value indicating failure. The wrapping function may use any method it desires to evaluate the creation request.
In our current system architecture, the wrapping function contacts a user-mode application, obtaining a ruling on the disposition of the process creation request. The wrapping function provides information about the requesting and requested processes (the current process and the process it wishes to create). The application engages its own logic and informs the wrapping function of its decision. This logic may include consulting a list of condoned executables, prompting the user or some other authority for permission to proceed, or requiring proof of authorization (i.e., a password) from the user or from some other source.

4.2. Safely managing kernel wrappers

The simplest method of wrapping involves making a local copy of the relevant system service entry and placing your wrapper function in its place. This method also works well for one or many people installing wrappers. In effect, this method creates a chain of functions, each of which thinks that it is calling the real System Service. Unfortunately, this chain can easily become disrupted if device drivers begin unwrapping that system service. For an example of this problem see Figure 1.

Furthermore, it was useful during the development of the Emu to load and unload our device driver on the fly. Unfortunately, unloading a device driver after wrapping a system service has its difficulties. Specifically, because a user mode program may choose to call a system service at any time, it is very difficult to tell whether your wrapper is in use at a particular point in time. Unless a device driver can be absolutely certain that a wrapper is not currently in use, that driver cannot be safely unloaded.

In order to perform service wrapping properly, we would like to develop a wrapping package that has two important qualities. First, it should allow multiple device drivers to concurrently wrap the same system services. These multiple wrappers can be thought of as layers over top of the original system service. Control must descend through these layers when a call is made to a system service. Second, the wrapping package should provide a framework under which drivers can safely wrap and unwrap the system services.

The first requirement, that of allowing multiple concurrent device drivers, has several possible solutions. We decided to create a device driver that exports functionality to all other parts of the kernel. It allows other device drivers to call exported functions that add to, remove from, and descend through layers of functions on top of the existing system services. This hides the implementation details of actually tinkering with the system service table and allows our wrapping package to carefully synchronize access to the layers surrounding the system service table.

Because these functions are synchronized, they help to fulfill the second requirement of providing for safe wrapping and unwrapping of system services. Specifically, actions that add or remove wrapper layers are considered to be writers, while actual calls to system services that descend through the layers are considered to be readers. Consistent with the traditional synchronization problem, multiple readers are allowed to descend through the wrapper layers simultaneously, but the actions of the writers, which modify the wrapper layers themselves, must be serialized. By tightly

![Figure 1. This pseudo-code demonstrates a flaw with the simple wrapping approach.](image-url)
controlling the synchronization of readers and writers, we can allow a device driver to safely unload after removing all of its layers of the system service table.

5. Discussion and conclusions

The approach and tool described here provide an enterprise-wide management facility for controlling the software that is allowed to run on users' machines. It returns control over information assets back to the CIO and system administrative staff and enables enforcement of corporate policies with regard to software. Bear in mind that because the control mechanism is based on actually running the software, it does not prevent unauthorized software installations — it merely prevents their execution.

In addition to preventing unauthorized software from running, we believe it is effective in stopping new and unknown malicious software executables from causing damage. One of the biggest problems in the anti-virus industry today is dealing with new variants of known malicious executables and novel malicious software. Currently, the only available approach is reactive. Only once a malicious executable is released in the wild and causes enough damage to be noticed can an anti-virus vendor release a signature to detect its presence on corporate machines. By this point, the damage may have already been done — particularly with a fast-spreading worm. With an Emu-like utility, new and unknown executables such as some classes of viruses and worms can be stopped from running in the first place.

Although executables make up one major category of malicious programs, macro viruses and other interpreted forms of mobile code such as scripts also pose great dangers to corporate systems. Because interpreted languages execute through another application (the interpreter), the Emu currently cannot discriminate among various interpreted scripts. Thus, we do not assert protection against malicious scripts. However, the Emu can prevent malicious scripts from running by preventing its interpreter from running. The Emu enables an administrator to decide which interpreters a user should be able to run. The Emu can prevent the interpreter itself from running, thereby preventing all scripts depending on that interpreter from executing. For instance, the administrator can easily prevent the Windows Scripting Host from running on client machines. This would prevent most scripting-based viruses from running. However, it would also prevent useful scripts from running, too. So the solution is not ideal, though it may be practical for a large number of users. Emu does not provide protection against macro viruses because it cannot allow a user to run Microsoft Word and prevent Word from executing potentially dangerous macro code. However, many interpreters are beginning to address these security concerns themselves, for example by giving the user the ability to disable macros. Most web browsers allow users to turn off the execution of mobile code, and some third party products put this control in the hands of the administrator, not the desktop user. While we do not address the mobile code security problem, our approach will work hand-in-hand with other third party mobile code protection technologies.

In addition to monitoring the launch of standalone executables, the Emu is also capable of monitoring the loading and execution of Dynamic Link Libraries (DLLs). An application is often made up of a base executable that loads DLLs that contain additional functionality. These DLLs may be well-known system libraries, or they may be proprietary third party software that is distributed as part of the application. Furthermore, application extensions (commonly known as plug-ins) may arrive as DLLs, not as separate executables. Thus a well-protected system must restrict the DLLs that can be loaded in addition to the applications that can be launched.

The current system of DLL enforcement restricts a user from loading any DLLs that were not on the system at the time that the Emu was set up. On a Windows system, it is significantly harder for an administrator to keep track of all of the DLLs that a user should be able to execute. It is not always clear which DLLs an application may be dependent on, and it is almost impossible for an administrator to make an informed security decision about whether or not a particular DLL should be run. The approach that the Emu takes is to restrict the introduction of new DLLs. If using a new piece of software requires the introduction of some new DLLs, then the administrator can add these DLLs to the Emu’s DLL ECL at the time of installation.

The execution management utility provides security-conscious administrators with a valuable defense against the introduction of unauthorized programs. It ensures that no user can execute a program (knowingly or unknowingly) without the explicit consent of the administrator. This greatly reduces the threat posed by unwitting employees, malicious insiders, and even malicious software. In addition to security considerations, the Emu also provides control over the distribution of illegally obtained applications and the use of entertainment programs on corporate resources.

There are numerous Emu features that make it manageable at the enterprise level, including centrally managed execution control lists and client-server communication. The kernel based wrapping approach
ensures the non-bypassability of the Emu and results in negligible performance overhead. Security features incorporated in the Emu ensure that even a malicious adversary cannot circumvent the execution management utility’s execution control lists.

Finally, it is important to note that our approach is not a substitute for security protection technologies such as malicious code scanners and sandboxing techniques. Rather, our approach can work together with other techniques. For instance, if a user receives an unknown executable it will be denied execution by the Emu client. The user may then request an administrator permission to run the unknown executable. A system administrator can scan the code against a virus scanner or malicious code classifier. Furthermore, the administrator can choose to run the unknown executable within a sandboxed environment in order to determine if it is trustworthy. If the administrator decides the application can be trusted, then she may subsequently add the executable to the user’s ECL to allow future execution. The important protection our approach provides is a first line of defense against users running unknown and possibly malicious executables.

In today’s Internet-enabled work environment, it is essential that administrators have full control over the programs that are executed by their user community. The Emu’s ability to control which programs are allowed to execute is a significant step toward regaining control over the enterprise information infrastructure.


