Secure Mobile Code Execution Service

Lap-chung Lam, Yang Yu, and Tzi-cker Chiueh – Rether Networks, Inc.

ABSTRACT

Mobile code refers to programs that come into a host computer over the network and start to execute with or without a user's knowledge or consent. Because these programs run in the execution context of the user that downloads them, they can issue any system calls that the user is allowed to make, and thus pose a serious security threat when they are malicious. Although many solutions have been proposed to solve the malicious mobile code problem, none of them are truly effective at striking a good balance between defeating zero-day attacks and minimizing disruption to the execution of legitimate applications.

This paper describes a commercial system called SEES that secures the execution of mobile code that comes into a host computer as an email attachment or as a web document downloaded through an anchor link by running them on a separate guinea pig machine rather than on the user machine. Effectively, it takes an isolation approach to the secure mobile code execution problem. As a result, SEES *guarantees* that no malicious email attachments or web documents that act on behalf of the user that downloads them, can damage the resources of the user machine, or can leak any confidential information. In particular, even zero-day virus cannot cause any harms. We present the design, implementation and evaluation of SEES on the Windows platform, and contrast it with other existing approaches to the same problem.

Introduction

Mobile code refers to programs that come into an end user's computer over the network and start to execute with or without the user's knowledge or consent. Examples of mobile code include a Java script embedded within an HTML page, a Visual-Basic script contained in a WORD document, an HTML Help file, an ActiveX Control, a Java applet, a transparent browser plug-in or DLL, a new document viewer installed on demand, an explicitly downloaded executable binary, etc. Because a piece of mobile code runs in the execution context of the user that downloads it, it can issue any system calls that the user is allowed to make, including deleting files, modifying configurations or registry entries, sending emails, or installing back-door programs in the home directory. The most common type of malicious mobile code is email attachment.

Existing solutions to the malicious mobile code problem fall into two categories: signature-based anti-virus tools and behavior blocking tools [1]. Neither of them can stop malicious mobile code or malware effectively. Because existing antivirus tools are based on signatures, there is always a time gap between when a piece of malware first appears and when the corresponding signature is derived and distributed to user sites. For malware such as the SQL Slammer worm, any time gap that is more than a few hours is unacceptable, because a well-designed worm can take down the Internet within hours.

Behavior blocking technology sandboxes the execution of suspicious applications by monitoring and controlling the system calls the applications make, according to a security policy. If properly configured, behavior blocking can even stop zero-day exploits. However, it is difficult to properly set the sandboxing security policy for each individual application such that all existing applications run smoothly and none of the existing malware can get through. This is especially true when the source code of the applications to be sandboxed is not available. Indeed, almost all existing behavior blocking systems use a single sandboxing policy for all applications running under a user account or on the same system. As a result, these systems tend to trigger many false positives and break perfectly legitimate applications.

This paper describes the design, implementation, and evaluation of a secure mobile code execution system called SEES (Secure Email Execution Service), which takes an *isolation* approach to safeguard an end user machine from two specific types of mobile code, email attachments and documents retrieved via a web browser. SEES identifies incoming email attachments and web documents, and isolates their execution on a physically separate machine (called the guinea pig machine) in such a way that the execution results are displayed on the end user's computer screen with the same look and feel.

Because SEES guarantees that malware embedded in email attachments or web documents never run on an end user machine, it is impossible for them to access, let alone damage, the user's resources. The guinea pig machine itself is carefully configured so that the risks of being permanently compromised and of attacking others when compromised are minimized.

Compared with signature-based anti-virus systems, SEES does not require periodic signature update

and can effectively protect end users from zero-day virus. Even if a piece of malware successfully penetrates and damages the guinea pig machine, the end user's resources still remain intact. Compared with behavior blocking systems, SEES supports fine-grained isolation for specific types of mobile code and uses a physical segregation approach rather than a sandboxing approach. As a result, SEES is less intrusive in that legitimate applications rarely get disrupted because of its security protection.

SEES effectively solves the email virus problem because it addresses the psychological dimension of the problem: People tend to open legitimate-looking but possibly virus-containing email attachments for fear of missing important messages. In addition, more than 90% of virus entering an enterprise is through email. SEES provides an additional level of assurance that even if an email attachment contains virus, it will not be able to inflict any damage upon a user's machine. This ability to tolerate malware allows SEES to reduce the degree of disruption to legitimate application execution to the minimum when compared with other solutions. It also opens up the possibility that a user can experiment with a piece of mobile code, for example, a downloaded executable binary, on the SEES server before she installs it on her own machine.

Related Work

The physical isolation idea of SEES originated from the Spout system [2], which is a distributed execution architecture to secure the execution of Java applets. Spout uses a web proxy to identify Java applets in incoming HTML pages and redirect them to a playground machine. Spout incorporates a Java-based remote display mechanism rather than Windows terminal service for remote execution. A similar approach for secure execution of Java Applets can also be found in [3].

The most widely used tools to protect user machines from malware are antivirus products from Norton [4], McAfee [5], Trend Micro [6]. These antivirus tools can scan files downloaded by web browsers, ftp clients, and email clients in real time. They can effectively remove all known viruses before the viruses can launch the attacks. However, none of them can stop zero-day attacks since they rely on signatures. This deficiency raises a serious security concern because Internet enables malware to spread so fast that even the most prepared antivirus company cannot derive signatures quickly enough to effectively stop them.

Many desktop machines suffer from malware intrusion. Kathleen [7] did a survey on 17 network-based intrusion detection systems, and only SessionWall from Computer Associates contains a scanner engine to detect viruses embedded in network traffic. However, this scanner still relies on signatures, and therefore cannot handle zero-day exploits. Tripwire [8] is an IDS running on UNIX-like systems that can detect zero-day viruses. Tripwire computes a hash value for each important system file or binary, and uses it to detect changes to system files. However, it cannot detect viruses that do not modify any system files.

WindowBox [9] from Microsoft is the closest system to SEES. WindowBox implements the sandbox mechanism using a desktop object. WindowBox modifies the Windows 2000 kernel to restrict the access of suspicious applications only to objects created in the same desktop. A user may choose to create many desktops such as work desktop, game desktop and personal desktop, and decide what applications can run on each desktop. If a virus is run on one desktop, it cannot access the network or data created on other desktops.

There are, however, two problems with WindowBox. First, applications running on different desktops still share application configuration files and registries, which a virus may corrupt to cause damage. Second, WindowBox requires users to explicitly decide what applications to run on which desktop and therefore may pose usability problems in practice. In contrast, SEES has neither of these problems because it uses physical isolation and automatic redirection.

Many existing academic sandboxing research systems such as Janus [10], Consh [11], Tron [12], and MAPbox [13] are implemented on top of UNIX-like system. Janus allows users to set permissions on path, environment variables, network access and display. When a process runs under Janus, Janus uses a debugging mechanism to monitor each system call made by the process, and checks the system calls against the user defined security policy. Consh extends Janus by adding a virtual file system and a virtual network system. Besides setting up a security policy, Consh also needs to setup the virtual file system correctly for running applications safely. Instead of setting security policy for a user or a program, Tron allows users to specify different security policies for different instances of the same program. MAPbox groups applications into behavior classes such as editor class, compiler class, mailer class, and browser class, and a special sandbox is built for each behavior class.

Another sandbox example is the secure web browser [14], which is built on top of an OS called SubOS [15], which offers process-specific protection mechanisms. An object downloaded by the web browser is assigned with a sub-userid, and the object is opened or executed on the context of the chosen sub-userid. All of these sandboxing systems require users to set up security policy and choose what and when to sandbox. Such a sandboxing model does not work on Windows environments since many Windows users are used to point and click and they do not have enough computer knowledge to decide what and when to sandbox and setup the sandbox environment correctly. In contrast, the SEES system does not change

the way users use the Windows system. It automatically selects and sandboxes email attachments and web documents.

Secure Mobile Code Execution

A secure mobile code execution service needs to address two fundamental issues: identifying a piece of mobile code when it comes in, and insulating its execution from the host computer that downloads it.

The majority of malware comes into a user's machine because the user clicks on something. Email attachment is the most common channel. Other possible channels include web browsers, ftp programs, peer-to-peer file sharing applications, and messaging applications such as IM and IRC. Another common way through which malware penetrates into a user's computer is by exploiting the automatic download capability of Microsoft's Internet Explorer (IE). Many web pages contain mobile code such as Java scripts, VB scripts, and ActiveX control. If the security level of IE is set to low, IE automatically executes the embedded mobile code when a user visits those pages.

IE also contains vulnerabilities that enable a web page to automatically install a piece of malware on a system even when the security level is set to the highest level. Similar vulnerabilities existed in Microsoft Outlook and Outlook Express. For example, Outlook could execute mobile code contained in an email attachment without a user clicking on it. Finally, by hijacking the control of a server program through such vulnerability as buffer overflow, malware can take over the machine on which the server program runs and potentially spreads to other machines. One such example is the SQL Slammer worm, which exploits a vulnerability in the Resolution Service of Microsoft SQL Server and Microsoft Desktop Engine (MSDE).

The ideal solution to the mobile code identification problem is for each network application to inform the operating system when it downloads and executes a piece of mobile code. Unfortunately neither existing applications nor existing operating systems provide such support. One possible approach to approximate this ideal is to apply binary or source-level program transformation techniques to automatically embed such notification mechanisms into existing network applications without any programming efforts. Because there are a large number of entry points malware can use to infiltrate a Windows PC, SEES chooses to focus on the two most common types of exploit points: email attachments and web objects downloads through IE.

SEES employs API interception techniques to monitor Win32 API calls made by email clients and web browsers, and take proper actions when these programs open or save files. The mobile code identification mechanism used in SEES is independent of email client programs and web browsers.

Once a piece of mobile code is identified, the issue is how to sandbox its execution in such a way that malware cannot cause damage and legitimate applications can run without any glitches. The key problem here is how to set up the sandboxing policy accurately so as to eliminate both false positives and negatives, and automatically so that the security administration overhead is reduced to the minimum.

Commercial behavior blocking products tend to err on the false positive side in that they tend to apply the same sandboxing policy to all applications executed by a user or on a given machine. An ideal solution to this problem is to apply program analysis techniques to extract application-specific sandboxing policy automatically from arbitrary application programs, for example, the PAID system [16]. However, this approach is not always feasible because the source code of network applications is not always available and introducing such a sophisticated sandboxing mechanism may be impossible for certain user sites.

Traditionally, the scope of sandboxing is a machine or a user account. In the Windows environment, all known behavior blocking systems apply their sandboxing mechanism to all processes running under a user account by limiting their system call privilege. This approach invariably breaks certain benign applications because the sandboxing policy cannot possibly cover the needs of all current and future legitimate applications. For example, the Word program needs read/write access to its application-specific directory and the document directory under the user's home directory. Prohibiting Word from accessing those directories may break the functionality of Word and inconvenience the user.

On the other hand, it is extremely difficult if not impossible to devise a sandboxing policy that can accurately capture and anticipate the requirements of all non-malicious applications such as Word that are going to run on a machine or under a user account. In addition, sometimes even a single process may need to be sandboxed differently at different times. For example, many Windows applications, such as Word, open multiple documents in the same process to reduce resource consumption. This means that when a user opens a local Word document and an email attachment that contains a Word document, she needs to choose between sandboxing both documents and sandboxing neither document.

Instead of sandboxing, SEES chooses to execute mobile code in a different execution environment than the one that downloads it. Specifically, mobile code runs on a physically separate machine called the guinea pig machine under a low-privilege user account, and the result of execution is sent back to the user machine through a remote display mechanism. This execution architecture provides the same look and feel for benign programs but provides physical isolation for potentially malicious programs.

This approach has several advantages. First, it allows centralized management and enforcement of

security policies, and thus reduces administration workload. Specifically, properly configuring the guinea pig machine is all is needed to defend an enterprise against malicious code embedded within email attachments or web documents. Second, the security policies on the guinea pig machine can be loosened to avoid unnecessary disruption to legitimate applications without compromising security. This additional latitude results from the fact that the guinea pig machine is potentially dispensable and is logically separate from the rest of the intranet.

SEES Implementation

SEES System Architecture

Figure 1 illustrates the system architecture of SEES, which consists of a SEES server and a SEES client. The SEES server runs on a stand-alone machine and provides the isolated execution environment for mobile code. The SEES client takes control when a user opens an email attachment or a web document. Whenever a SEES client needs to open a file that potentially contains mobile code, it sends the file to the SEES server, which opens the file and displays the results on the SEES client's screen. To the user, the look and feel is the same as if the file is opened locally.

The SEES server consists of three components as shown in Figure 2, Execution Manager, Security Control Manager, and System Call Monitor. The Execution Manager allows a SEES client to run a piece of mobile code on the SEES server, and provides the same look and feel as if it is executed locally. The Security Control Manager provides an isolated execution environment so that the side effects of mobile code are completely segregated from the rest of the SEES server. The System Call Monitor is a traditional sandboxing mechanism that protects the SEES server itself from malicious code by monitoring and controlling system call invocations according to a predefined security policy.

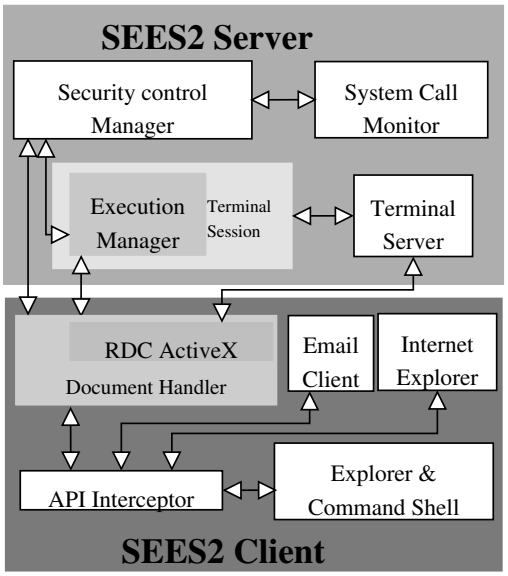

Figure 2: The detailed software architecture of SEES. The Execution Manager runs a piece of mobile code on the SEES server on behalf of a SEES client. The Security Control Manager provides an isolated execution environment to segregate the side effects of mobile code from the rest of the SEES server. The System Call Monitor protects the SEES server from malicious system call invocations according to a predefined security policy. The main component of SEES client is the API interceptor, which intercepts the save and open operations of application programs and redirects mobile code to the SEES server for execution.

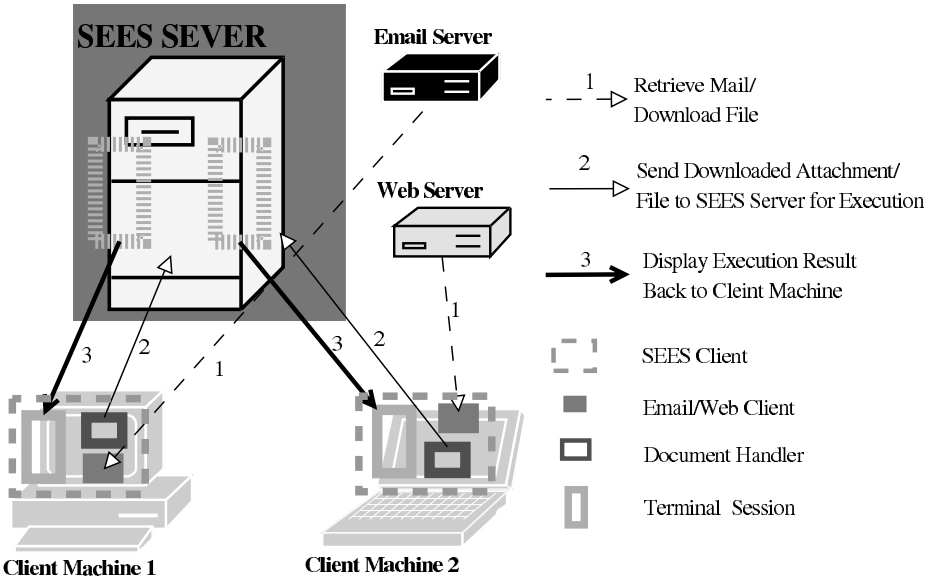

Figure 1: In the SEES architecture, mobile code downloaded from an email client or Internet Explorer runs on a separate guinea pig machine called the SEES server, but the results of mobile code execution are sent back to the end user machine through a remote display mechanism such as Windows terminal service.

The SEES client is implemented on top of Remote Desktop ActiveX Control. When a user invokes a piece of mobile code, the SEES client first consults with the Security Control Manager to obtain a low-privilege account, and executes the mobile code under this account on the SEES server. There are several implementation alternatives to supporting the remote execution mechanism, including Windows terminal server, Linux server running Wine or Crossover Office and VNC, and Windows server running multi-user VNC or X Windows.

The Windows terminal server is the best choice in term of performance overhead and usability, but it requires expensive licensing charge. A less expensive way is to use a Linux server running Wine and VNC (LWV), which is slower and requires much more memory. Currently Wine can successfully run many Windows applications such as Microsoft Office. Still, there are many other Windows applications that cannot run under Wine. Finally, we have developed an experimental version of multi-user VNC for the Windows platform, but its performance for interactive applications is still inferior to Windows terminal server.

Mobile Code Identification

The main task of the SEES client is to identify potentially dangerous contents downloaded from the network and send the contents to the SEES sever when users invoke them. Since mobile code can have many different forms, and they can come into a computer from many channels, there is no universal mechanism that can identify mobile code accurately. Our current approach is to treat the files downloaded by Internet Explorer or an email client and with dangerous MIME type such as .exe and .doc as dangerous contents.

The easiest way to identify downloaded contents is to use a proxy server to monitor and parse the incoming contents, and mark the contents as dangerous if the contents have certain MIME type. The first version of SEES used this approach. More concretely, a POP3 proxy server is used to intercept all incoming emails and rename an email attachment if it contains dangerous MIME type such as .exe and .doc. The POP3 proxy server adds a .sees extension to each dangerous attachment so that when a user clicks on a renamed attachment, the SEES client is invoked instead of the corresponding application.

However, this proxy server approach has two major drawbacks. First, the .sees extension tends to create confusion because it is visible to the end users. Second, many email servers, such as Microsoft Exchange Server and IBM Lotus Domino, use different protocols between themselves and email clients. Worse yet, emails could be encrypted or email servers could require a secure connection. As a result, the proxy server approach inherently entails significant implementation complexity that cannot be easily removed.

To avoid these problems, SEES employs a client-side Win32 API interception mechanism to identify email attachments and downloaded web documents. The software architecture of the SEES client is shown in Figure 2. Specifically, the SEES client intercepts file open and file save operations of email clients and IE. When the user double-clicks on an email attachment or a link on an HTML page, the email client or IE calls the ShellExecute family of API to open the attachment/file. The API interceptor intercepts the ShellExecute family of API calls and re-directs the attachment/file to the SEES server. If the user attempts to save an attachment/file to a target file, the API interceptor intercepts the GetSaveFileName family of API calls to append an .sees extension to the target file if it is in a FAT file system, or flags a ‘‘dangerous’’ flag of the target file if it is in an NTFS file system.

The .sees extension and the unused flag are meant to indicate to the system that the file is potentially dangerous. Later on, when a user opens this file through the Windows Explorer or command shell, the IShellExecuteHook component intercepts the ShellExecute family of API calls, examines whether the file is dangerous, and re-directs it to the SEES server if it is dangerous. The advantage of using API interception to identify downloaded files is that a single mechanism can work with many different applications as long as they use the ShellExecute family of API. Currently, we are extending this interception mechanism to the peer-peer applications and instant messenger applications.

API Interception

When an email client or IE opens a file, the Win32 API interceptor intercepts the file open call and sends a request to the SEES server to open it. If an email client or IE needs to save a file, the interceptor marks the file as dangerous. When a user opens a dangerous file, the interceptor also sends the file to the SEES server. The ShellExecute family of APIs in shell32.dll of the Windows platform are the most commonly used APIs to perform file operations on a file. Although they are not the only APIs that can be used to open a file or to execute a file, they are indeed used by all email clients we have tested, as well as IE, Explorer, and the DOS shell.

Intercepting Win32 API calls means taking over the program control when these APIs are called without modifying the monitored applications. The most commonly used interception mechanisms are *Proxy DLL*, *EAT Patching*, *IAT Patching*, and *Shell Extension*. Proxy DLL replaces an original DLL with a proxy DLL that contains a call stub for each exported function in the original DLL. The replacing proxy DLL assumes the name of the original DLL, while the original DLL is renamed.

When an application uses the original DLL’s name to load a DLL, it is the proxy DLL that gets loaded instead. All the calls made to the functions in the original DLL are routed to the exported stubs in the proxy DLL. The proxy DLL can simply forward the calls to the original DLL, perform some operations

before forwarding, forward the calls to someone else, or reject the calls. Proxy DLL is the simplest way to intercept Win32 APIs. However, this technique requires that the function prototypes of all the exported functions in a DLL be available. SEES does not use this approach because some function prototypes of the DLLs we wan to intercept are not available.

The Portable Executable (PE) format [17] is the binary format used by both executable files and DLLs on the Windows platform. Each DLL PE file contains a table called *Export Address Table* (EAT) that stores the entry point of each exported functions. The addresses stored in a DLL's EAT are used for an application to call the functions the DLL exports. To intercept a function exported by a DLL, one can add to the DLL file a new section to store the intercepting function's code, and modify the EAT entry of the intercepted function to point to the intercepting function.

When an application calls an intercepted function, the intercepting function is activated, and the intercepting function can choose to abort the function call, forward the function call to the original function, or perform some other operations. However this EAT patching technique cannot be easily applied to system DLLs since the Windows File Protection (WFP) mechanism discourages such DLL modifications by nullifying the effects of these modifications. Even if all backup versions of a system DLL are replaced, Windows OS can still restore the DLL to the unmodified version through the Windows Updates mechanism. Another problem of this approach is that modification to a DLL has to be compatible with future versions of the DLL.

Each executable PE file includes an *Import Address Table* (IAT), which has an entry for each imported function (a function exported by a DLL). After an executable file is loaded into memory, this table is filled with the addresses of the imported functions. When an application makes a call to an imported function, it first looks up the corresponding IAT entry, and then uses the address contained within to jump to the target function. IAT Patching modifies the IAT entry of an intercepted function to point to the new intercepting function. All intercepting functions are implemented in a DLL, and the intercepting functions are loaded into memory by a method known as *DLL injection*.

There are three ways to inject a DLL to the address space of a running process: 1) using the Win32 API SetWindowsHookEx, 2) using the Win32 API CreateRemoteThread, and 3) using the AppInit_DLLs registry. All these three mechanisms force the Windows OS to load a specified DLL automatically. Each Windows DLL has a function called DllMain, which is called automatically by the Windows OS after a DLL is loaded. Therefore, the code for patching the IAT table can be implemented in DllMain. However, this IAT patching technique only works with statically loaded DLLs. A DLL can be loaded dynamically by using LoadLibrary and the entry point of a function exported by a DLL can be obtained by GetProcAddress. To support dynamically loaded DLLs, both LoadLibrary and GetProcAddress must also be intercepted via IAT patching.

The IShellExecuteHook interface is a shell extension that can intercept any calls made to ShellExecute (EX). This is the documented approach to extend the behavior of the ShellExecute(EX) API with low overhead, but it cannot intercept other Win32 API calls. SEES uses both IAT patching and the shell extension method to intercept the ShellExecute family of APIs and other APIs. The ShellExecute family of APIs sometimes are used to perform some other operations beside opening an attachment. The SEES client analyzes the arguments used in these API calls to filter out unwanted cases.

Saving Files to Local Disk

Even though mobile code runs on the SEES server, it is essential that the user feels that it is executed locally. Towards that end, when a user attempts to save a file from an application running on the SEES server, the file save interface should show the file system on the user's machine rather than that on the SEES server. To implement local save for a remote execution mechanism such as Windows Terminal Server, one needs to re-direct the file save operation from the SEES server to the requesting SEES client. More concretely, when the Execution Manager intercepts a save operation from applications running on the SEES server, it requests the SEES client to launch a save as dialog on the SEES client machine, thus providing the illusion that the user is saving the attachment/file on the local disk.

After the user picks a local file name for the save operation, the Execution Manager first stores the file on the server's disk, and then transfers it to the SEES client, which then saves the copy to the file location the user specifies. However, since the SEES server cannot always detect if an opened attachment is a virus or not, after the user saves the attachment to the local machine, it is also marked as dangerous so that it will be sent to the SEES server next time when the user clicks on it again.

Isolation of Mobile Code Execution

Because the SEES server is responsible for executing mobile code on behalf of all SEES clients within an organization, it is essential to protect it from malicious mobile code. That is, it should not be possible for any mobile code to bring down the SEES server and deny the mobile code execution service to other hosts. To provide such protection, SEES adds the following checks for every mobile code execution request from the SEES clients:

- Only certain IP addresses are authorized to be a SEES client.
- Each SEES client can only make a finite number of mobile code execution requests.

- The total amount of memory and disk usage by a SEES client is limited.

The SEES server executes each piece of mobile code on a low-privilege account, which allows its processes to read/write its home directory and to have read access to certain system applications and files. As a result, no mobile code can steal information from the SEES server or corrupt the system data structures such as registries, DLLs, and applications. To prevent mobile code from leaving any permanent effects on the SEES server, all the modifications to the registries and file system made by a piece of mobile code are erased after the execution is done.

More specifically, after installation, the SEES configuration tool copies the registry files NTUSER.DAT and UsrClass.dat of each account into a safe place. After execution of each piece of mobile code, these two registry files are restored automatically. As a result, when a piece of mobile code starts, it always start with a ''clean'' execution context, in terms of registry values and home directory contents, and will never get ''infected'' by other malicious mobile code.

A key advantage of the SEES architecture is that the SEES server can use a more lenient security policy when executing mobile code, as long as such policies never bring down the server. That is, the SEES server only needs to protect itself from denial-of-service (DOS) attacks, and can afford to err for other types of attacks, since the server itself is not supposed to contain any valuable information. Because of this additional latitude, SEES is much better than existing behavior blocking systems because it can minimize disruption to legitimate applications.

To take this idea to the extreme, we are currently exploring a *namespace virtualization* mechanism that allows each piece of mobile code to modify whatever files and registries it wants, and yet these modifications are never visible to the SEES server or other pieces of mobile code. This mechanism ensures no legitimate mobile code will be disrupted while protecting the SEES server from malicious mobile code.

System Call Monitoring

As an additional layer of defense, the SEES server also includes a system call monitor that checks all the system calls made during the mobile code execution against a pre-defined sandboxing policy. Any system calls that violate the sandboxing policy are denied, and an alert message is sent to the user.

The software architecture of SEES's system call monitor is shown in Figure 3. The current SEES prototype monitors only two system calls, NtOpenFile and NtCreateFile, which are used for both files and network connection operations. To intercept system calls on a Windows NT-like environment, we modify KeServiceDescriptorTable [18], which is the system call dispatch table data structure in the kernel. The System Call Interception module first changes the table entries corresponding to NtOpenFile and NtCreateFile to point to two SEES hooking functions respectively and saves the original function pointers. Consequently, all calls to NtOpenFile and NtCreateFile go through SEES's hooking functions.

When the hooking functions intercept a system call, the Call Source Identification module needs to identify the process that makes the system call, since only processes that execute mobile code and the child processes they create need to be sandboxed. This

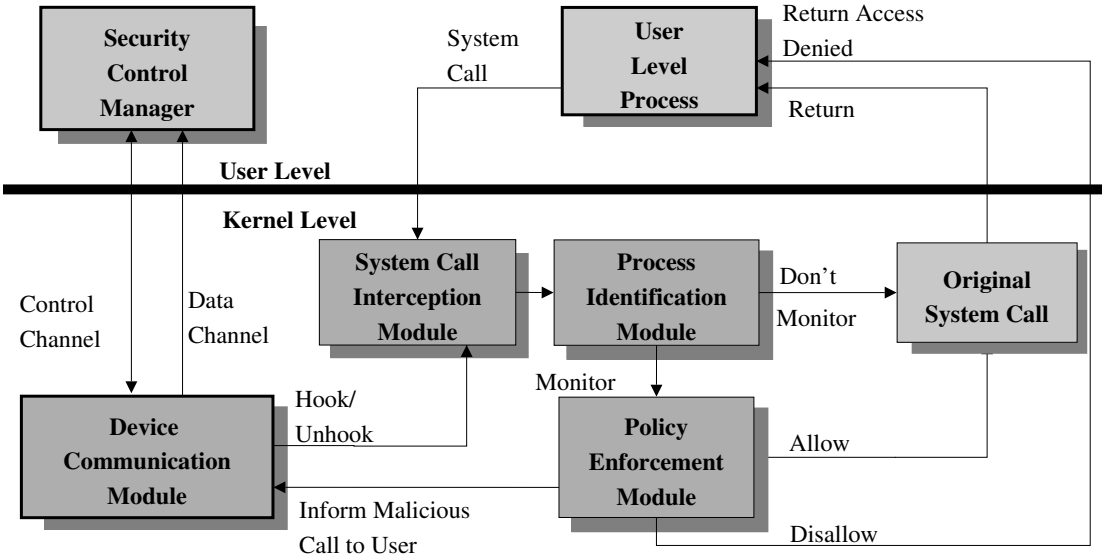

Figure 3: The detailed software architecture and control flow of the System Call Monitor driver, which is embedded into the kernel. The System Call Monitor only sandboxes the applications that run under a low privilege SEES user account according to the security policy set by a system administrator. It allows the system applications and the applications that are not executed under a SEES user account to run normally.

module obtains all the relevant information, such as process ID, terminal session ID, full path of the binary image of the current process, security identifier (SID) of the current user, filename or object name (network object) to be opened, and desired access mode, from the process structure of the current process making the system call. The Policy Enforcement module then uses the collected information to check for access violation. These policy rules are set up at the configuration time and loaded into the system call monitoring driver during initialization. The Device Communication module is responsible for informing the user when a sandboxed process violates the security policy.

Evaluation

Scalability

The main scalability concern about SEES architecture is the fact that all mobile code's execution is concentrated on a single server using Windows Terminal Services (WTS). In this subsection, we will examine the start-up latency of individual applications and CPU/memory consumption on the SEES server. To be sure, WTS actually supports server clustering to improve overall throughput and fault tolerance. We use a DELL desktop with Intel Pentium 4 2.4 GHz CPU, 768 MB PC2700 DDR memory as the SEES server and an Acer desktop machine with Intel Pentium 3 650 MHz CPU and 384 MB PC133 SDRAM memory as a SEES client. The operating systems used on both machines are MS Windows 2000.

We tested different types of documents using applications in Microsoft Office Suite and the result shows that when the available physical memory on the SEES server is more than 20 MB, the start-up latency between local execution and SEES-model execution is only about one second on average. When the number of active terminal sessions increases, e.g., increasing to 30 or 40 sessions, there is still no noticeable latency difference between the two cases. When the available physical memory is below 20 MB, the application startup latency increases significantly, i.e., 5 seconds or longer, because of extensive swapping. This means the architecture based on WTS does not add to additional usability problem in terms of latency as long as the SEES server is installed with enough memory.

The SEES server's CPU usage is also related to its memory consumption. When the SEES server's available physical memory is more than 20 MB, its CPU usage usually reaches a peak value between 50% and 90% when a terminal session starts or terminates, but quickly decreases to a lower average value. However, after the available physical memory becomes smaller than 20 MB, the CPU usage remains at a high peak value and that is when the start-up latency starts to deteriorate.

When the number of active terminal sessions increases, the memory usage on the SEES server increases linearly, as shown in Figure 4. In this test, we created a series of new terminal sessions, each running a WORD instance that opens a 865KB document,

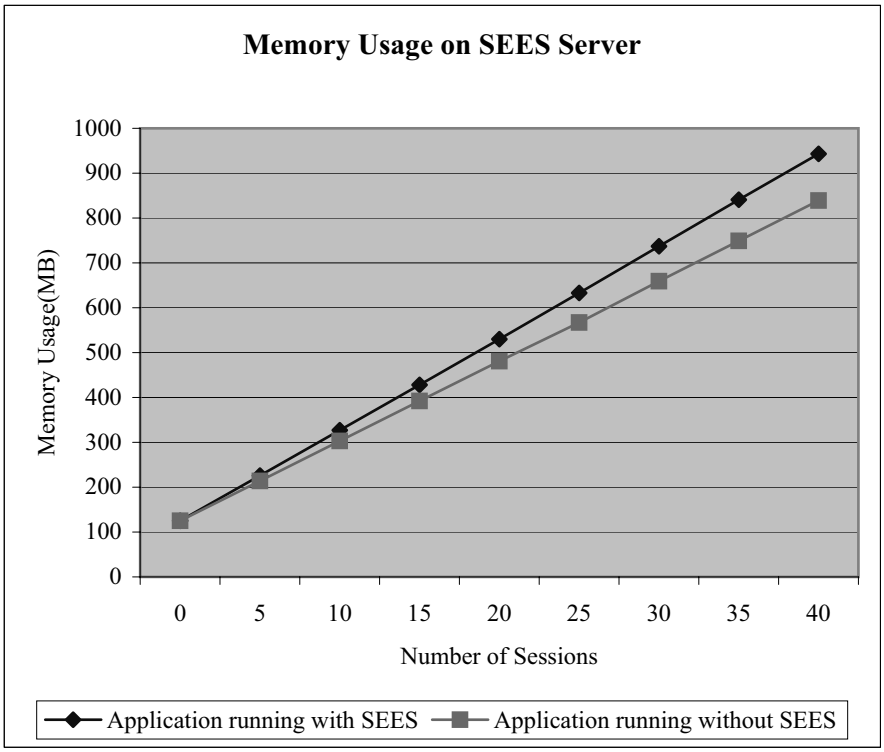

Figure 4: The physical memory consumption of the SEES server increases linearly at a rate of 20 MB per new terminal session. In this case, each terminal session runs a WORD instance that opens a 865KB document.

and the rate of increase in the SEES server's memory consumption is about 20 MB per new session. To isolate the contribution of the SEES server, we removed the SEES server control and opened the same WORD document in new terminal sessions, and the increment in physical memory usage is about 18 MB per session. This shows that the SEES server itself consumes only about 2 MB. The WORD application in this case consumes 8 MB because it opens a document containing many images. This means that there is a fixed overhead of about 10 MB associated with each new terminal session even when it is not running any application, and a remote-display scheme better than WTS can be exploited to reduce the memory overhead.

Attack Analysis

Saving Attachment Directly From Email Client

Instead of opening an email attachment, users can also save an email attachment directly from an email client application. This operation is different from the ''Local Save'' functionality of SEES and must be identified properly. One solution is to intercept this operation of email client applications and mark the file to be saved.

Time-bomb Malicious Code

A time-bomb malicious code will not be triggered until some later time or when a specific user operation takes place. As a result, the system call monitor on the SEES server may not immediately detect a time bomb's malicious behavior. One way to resolve this problem is to mark the file saved through ''Local Save'' so that all subsequent invocations of these files will still be executed on the SEES server.

Benign Installer and Malicious Executable

It is difficult to distinguish a benign application installer from a malicious executable, because both can modify system directories and registries. As a result, the SEES server may falsely mistake a legitimate application installer for malicious mobile code. We are working on a namespace virtualization technique that provides a process its own virtual execution environment by logging all its updates to system resources, and committing these updates only when it is sure of the process's legitimacy.

User Context Contamination

Because a malicious email attachment can always update the private registry entries or home directory of the user account under which it runs, these updates can potentially infect future email attachments that execute under the same user account. To address this problem, the SEES server provides an account refreshing mechanism, which cleans up the home directory and refreshes the private registry entries as soon as an existing session terminates.

Attacks Against the SEES Server

The home directory of a SEES user account on the SEES server is configured to be writable for processes under that user account. If a malicious email attachment keeps creating files, it may consume excessive disk space on the SEES Server. To prevent this attack, SEES sets a disk quota limit for each account using NTFS's quota management. If an attacker somehow gets hold of the user ID and password of a user account on the SEES server, she can bombard the SEES server with many terminal sessions. The SEES server solves this problem by limiting the total number of terminal sessions per host. In addition, the attacker can log into the terminal server to eavesdrop the current applications running under the same user account or to browse related network shares. To stop such attacks, the SEES server ensures that terminal sessions under the same user account always come from the same host, and the network share component is disabled.

Conclusion and Future Work

SEES stands out among both research and commercial solutions to the secure mobile code execution problem because of its unique capability of both stopping zero-day virus and minimizing disruption to execution of legitimate applications. It achieves this through accurate identification of specific types of mobile code and physical isolation of the execution of these mobile code. The end result is that SEES can guarantee that no email attachments and web documents can can act on behalf of the user that downloads them, can damage the resources of the user machine, or can leak any confidential information.

As we discussed in the section on the fundamental issues of secure mobile code execution, there are many other mobile code entry points that the current SEES implementations do not capture and therefore cannot isolate. For example, it is difficult to identify and sandbox mobile code embedded in an HTML page or an email body, especially when the HTML page or email is encrypted.

The main problem is that this type of mobile code runs in the same address space as the downloading application, in this case Internet Explorer or Outlook, and requires the sandboxing mechanism to use different sandboxing policies at different times for the same application. As another example, mobile code embedded within objects being exchanged through FTP applications, peer-to-peer file sharing applications, IRC and Instant Messaging applications becomes increasingly prevalent, and thus needs to be identified and sandboxed properly.

Although the physical isolation approach in the current SEES implementations provides strong protection, it has two disadvantages. First, it requires an expensive infrastructure, namely the Windows Terminal Server. Second, it cannot be easily generalized to a mobile computing environment, because the SEES server infrastructure may not always be available. To address these problems, we are currently developing a logical isolation approach that relies on system call

monitoring and virtualization techniques, and thus does not require a separate guinea pig machine. We expect this approach to be more scalable and portable while providing the same degree of protection as the physical isolation approach.

Author Biographies

Dr. Lap Chung Lam is the Chief Engineer of Rether Networks Inc. He received his B.A. in CS and mathematics from SUNY at New Paltz, and Ph.D. in CS from Stony Brook University in 1997 and 2005 respectively. He received a best paper award from the Program Analysis for Security and Safety Workshop (PASSWORD) co-located with ECOOP 2006. Dr. Lam's current research interest focuses on computer security, software protection, and program analysis. He can be reached electronically at lclam@rether.com .

Yang Yu is a Ph.D. candidate in the Computer Science Department of Stony Brook University. He received his B.S. and M.S. in computer science from Tsinghua University, Beijing, China in 1999 and 2002 respectively, and M.S. in computer science from Stony Brook University in 2005. He has received a Best Paper Award from 2005 Annual Computer Security Applications Conference (ACSAC). His current research interest lies in operating system and system security. He may be reached at yyu@cs.sunysb.edu .

Dr. Tzi-cker Chiueh is a Professor in the Computer Science Department of Stony Brook University, and the Chief Scientist of Rether Networks Inc. He received his B.S. in EE from National Taiwan University, M.S. in CS from Stanford University, and Ph.D. in CS from University of California at Berkeley in 1984, 1988, and 1992, respectively. He received an NSF CAREER award in 1995, an IEEE Hot Interconnect Best Paper award in 1999, a Long Island Software Award in 1997 and 2004, and a Best Paper Award from 2005 Annual Computer Security Applications Conference (ACSAC). Dr. Chiueh has published over 140 technical papers in refereed conferences and journals. His current research interest lies in wireless networking, computer security, and storage systems.

Bibliography

[1] Conry-Murray, Andrew, *Product focus: Behavior-blocking stops unknown malicious code*, 2002, http://www.networkmagazine.com/shared/article/showArticle.jhtml?articleId=8703363&classroom= .

[2] Chiueh, Tzi-cker, Harish Sankaran and Anindya Neogi, ''Spout: A transparent distributed execution engine for java applets,'' *IEEE Journal of Selected Areas in Communications*, Vol. 20, 2002.

[3] Malkhi, D. and M. K. Reiter, ''Secure execution of java applets using a remote playground,'' *IEEE Transactions on Software Engineering*, Vol. 26, 2000.

[4] *Symantec: Norton antivirus 2004 professional*, 2004, http://www.symantec.com/nav/nav_pro/features.html .

[5] *McAfee: Mcafee virusscan*, 2004, http://us.mcafee.com/root/package.asp?pkgid=100 .

[6] *Trend Micro: Officescan*, 2004, http://www.trendmicro.com/en/products/desktop/osce/evaluate/features.htm .

[7] Jackson, Kathleen A., ''Intrusion detection system (ids) product survey,'' *Los Alamos National Laboratory report LA-UR-99-3883*, 1999.

[8] Kim, G. H. and E. H. Spafford, ''The design and implementation of tripwire: A file system integrity checker,'' *ACM Conference on Computer and Communications Security* pp. 18-29, 1994.

[9] Balfanz, Dirk and Danie R. Simon: ''Windowbox: a simple security model for the connected desktop,'' *Proceedings of the 4th USENIX Windows Systems Symposium*, pp. 37-48, 2000.

[10] Goldberg, Ian, David Wagner, Randi Thomas, and Eric A. Brewer, ''A secure environment for untrusted helper applications,'' *Proceedings of the 6th USENIX Security Symposium,* San Jose, CA, 1996.

[11] Alexandrov, Albert, Paul Kmiec, and Klaus Schauser, ''Consh: A confined execution environment for internet computations,'' *USENIX Annual Technical Conference*, 1999.

[12] Berman, Andrew, Virgil Bourassa, and Erik Selberg: ''Tron: Process-specific file protection for the UNIX operating system,'' *Proceedings of the 1995 USENIX Technical Conference*, pp. 165-175, 1995.

[13] Acharya, Anurag and Raje Mandar, ''Mapbox: Using parameterized behavior classes to confine untrusted applications,'' *Proceedings of the Tenth USENIX Security Symposium*, 2000.

[14] Ioannidis, Sotiris and Steven M. Bellovin, ''Building a secure web browser,'' USENIX Annual Technical Conference, *FREENIX Track*, pp. 127-134, 2001.

[15] Ioannidis, Sotiris and Steven M. Bellovin, ''Subo operating systems: A new approach to application security,'' *Technical Report MS-CIS-01-06*, University of Pennsylvania, 2000.

[16] Chiueh, Tzi-cker, *Paid: Program-semantics aware intrusion detection*, 2003, http://www.ecsl.cs.sunysb.edu/paid/index.html .

[17] Microsoft Coporation, *Microsoft portable executable and common object file format specification*, 1999, http://www.microsoft.com/whdc/hwdev/hardware/PECOFF.mspx .

[18] Schreiber, Sven B., *Undocumented Windows 2000 Secrets A Programmer's Cookbook*, Addison-Wesley, pp. 266-268, 2001.