SECURING UNTRUSTED AND MALICIOUS SOFTWARE

INTRODUCTION

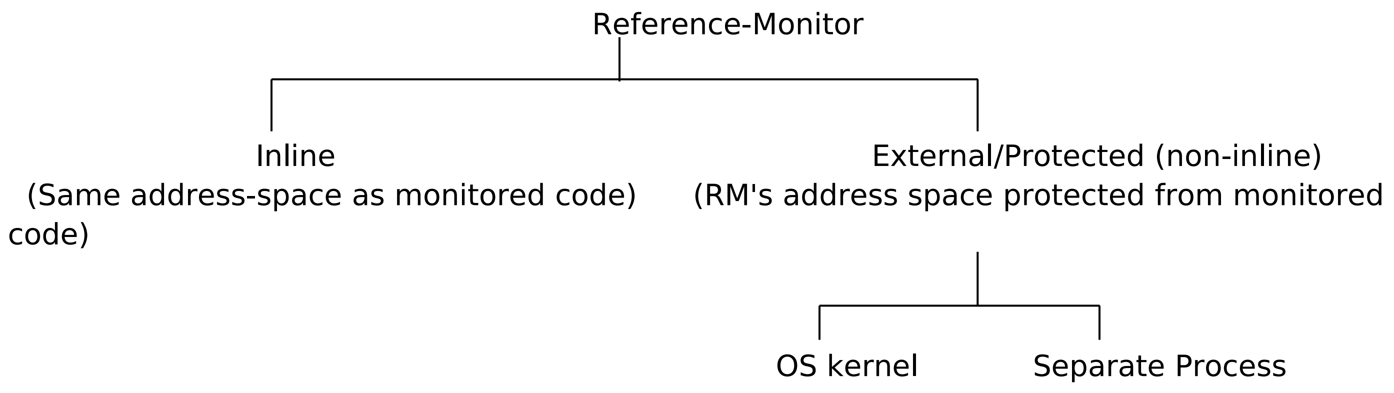
The first question that arises is “What do we mean by "security" and by "securing" software.” One way of defining the phrase "securing software" is enforcing a desired policy. This definition would then entail the capability of specifying the policy, which defines what this software can and cannot do.

Note that at the conceptual level, there is no difference between a policy for securing untrusted software from the ones studied earlier, say, MAC, DAC, Role-Based policies etc. However, when it comes to securing the untrusted software, there exists a need for finer granularity and deeper level of control than the standard operating system primitives.

Security policies can be enforced using the concept of reference monitor. The reference monitor should be protected from being compromised by untrusted code.

One of the properties that’s required of the reference monitor is that of "complete mediation". The requirement basically says that the reference monitor should be able to intercept/examine every operation that is being made by the code under inspection. The idea is that if certain operations are missed out, then it would possibly result in a failure to enforce the security policy.

There are two basic types of reference monitors (RM)


Reference-Monitor
Inline
(Same address-space as monitored code)
External/Protected (non-inline)
(RM's address space protected from monitored code)
OS kernel
Separate Process

*	Inline Reference Monitor (IRM) - IRM runs in the same address-space as the code on which the policy is to be enforced. The key issue here is that a mechanism is needed to protect the data and control flow used by the monitor from being compromised by monitored code.

*	External/Protected Reference Monitor – In non-inline reference monitors, the reference monitor is in a protected address-space. That can be either within the OS Kernel or a separate address-space.

The main distinction is that in an external/Protected RM, the monitored code cannot affect the data and/or the control flow of the monitor. While in the IRM, the monitor and the monitored code are in the same address-space and an explicit mechanism is needed for protection of the integrity of RM.

However, do note there are advantages of IRM over non-inline Monitors:

* Performance. In case of non-inline monitors, a context-switch occurs each time a check needs to be performed. This leads to a loss of performance in External monitors

* In non-inline reference monitors, the monitored code needs to be stopped when its data is in consistent state so that the necessary checks can be performed.

  This is not an issue when the checks happen once in a while. However with finer granularity checks, (say) checks after execution of every single instruction of the monitored code, the context-switches can be quite expensive and performance suffers. (On a typical processor, a context-switch involves few-thousand instructions.)

* The main question then arises as to at what granularity should the checks be performed.

  For example, In StackGuard, every function call and return is monitored. Note that every function has only few tens to hundreds of instructions. Hence if we have context-switches at similar granularity, that is for function call and return, then performance would not be good (again because context-switches are far more expensive)

<u>**GRANULARITY W.R.T. EXTERNAL MONITORS:**</u>

MONITORING SYSTEM-CALL:

* Another feasible granularity level can be enforcing policy at system call level.

  * Monitoring at system-call is a "natural" way of enforcing policy using a non-inline monitor. Following are some of the reasons:

1. RIGHT GRANULARITY

   * When a system call is performed, there is already a context-switch in progress. The control is being transferred to the kernel. Checks at this junction would thus prevent additional overheads.

2. System calls are designed in a way that allows policies to be expressed easily.

   (a) In a well designed OS, the number of system-calls is relatively less, measured in few hundreds.

   For example: Linux has 319 system calls, FreeBSD has 330. Thus the number of operations that need to be monitored is reasonably small.

   (b) System calls are designed to be orthogonal. There is just one system call for a specific purpose. Suppose, to the contrary, there are 10 different ways of doing the same thing. In such a case, if that thing is to be prevented, then the policy would have to examine all the 10 different methods

   (c) Since the system-calls incorporate the concept of user-space and kernel-address-space, the issue of what data is relevant and what needs to be monitored is reasonably well-defined.

   For instance, if a file is to be opened, the relevant data are the parameters to the system call. Hence there is no need to examine any other data in the monitored process's address-space. However, if process's data is monitored independently, we could have a TOCTTOU (Time of Check to Time of Use) issues. Especially in multi-threaded processes, where the check could ensure the correctness of data at a specific time, only to have the data changed by another thread by the time of use of that data.

(d) System calls are understood by a larger set of people than those that understand kernel internals. This means that more people are likely to be able to write system call policies, as opposed to policies based on deeper events within the OS kernel. For instance, the Linux Security Module (LSM) on Linux defines several security related events within the OS. This need for expertise makes it more difficult to write policies. (It should be noted, though, that if one has the expertise, then LSM can avoid several pitfalls associated with system call policies.)

MORE ABOUT THE RIGHT ABSTRACTIONS (Side discussion)

To clarify this point, consider the following example:

Example:

In SELinux, these monitoring operations have been moved deeper inside the operating system.

The basic goal of SELinux is the same. That is, enforcement of the security policy on user-level processes. In a way, this is a better approach, because it’s closer to the resources.

Moreover, on the system-call level, when a policy needs to be specified, there arise certain issues:

For example: name to object translation in the operating system. We might want to specify that a certain file should not be modifiable. At times we need to protect the name of the object, at other times we need to protect the object itself without any regard for the name.

This distinction is hard to express at the system-call level. Certain system-calls use file-name while others use file-descriptors. However when within the kernel, we can be more specific about the operation intended.

However the key point is that there are also disadvantages to such an approach.

DISADVANTAGES OF SELINUX APPROACH:

* There are a lot more operations inside the kernel and thus specifying a policy becomes more complicated.
* The policy (specified using SELinux) is very closely coupled to the operating system.

  SELinux uses a Linux Security Module which identifies the places within the kernel where the security checks should be performed and provides a framework/programming-environment to develop a reference monitor. Thus each time a security sensitive operation is to be performed; the reference monitor is called with all the relevant parameters.

Thus when writing the reference monitor, we would need to know the points in the kernel where the security policy has to be enforced and understand what each interception point means.

As compared to this, then system call level granularity has the following advantages:

ADVANTAGE OF SYS-CALL GRANULARITY:

In comparison to the above, while writing policies at the system call layer, although some level of system knowledge is required (for example, file-manipulation in windows is different from unix), the system call interface generally tends to be the same across Unix-flavors of operating-systems.

Thus sys-call based policies are easier to write as compared utilizing hooks with operating system to enforce policy.

3. CAN BE DONE WITH EXISTING OS'ES

Another important point is that this technique can be implemented on existing operating systems.

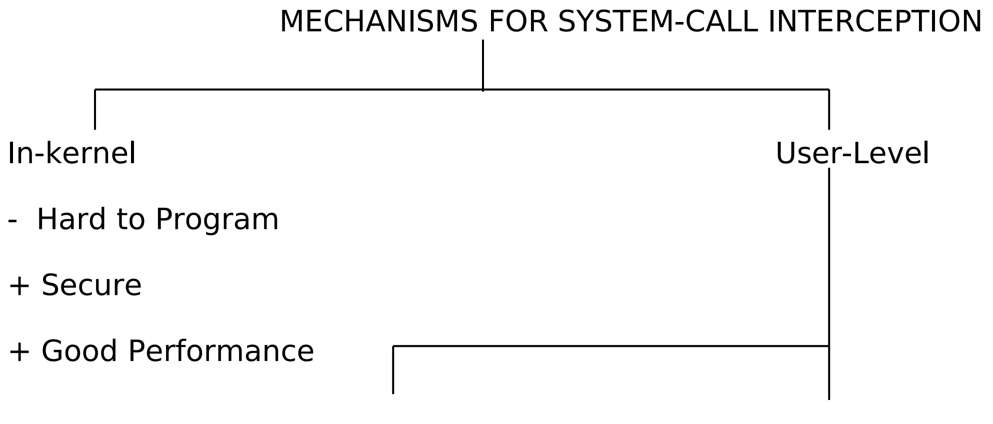
System calls are basically a software interrupts that get dispatched to appropriate kernel routines. Thus, in principle, there exists a dispatch table to guide these system-calls. By replacing this table with another table of choice, we could effectively intercept all systems calls.

Thus there exists a backward compatibility in implemented a system-call level reference monitor

* For all these reasons, system-call interception, that is having system-call level granularity, is quite popular and a lot of research has focused on the use of this technique. On Linux, LSM is very popular as well; but on Windows, security tools rely much more commonly on system call interception.

MECHANISMS FOR SYSTEM CALL INTERCEPTION:

* The following is a classification for mechanisms for implementing system-call interception.



Library-Interception	Kernel Supported Interception
+ Easy to Program	+ Easy to program
- Not Secure	+ Secure
+ Good Performance	- Bad performance

I KERNEL-LEVEL SYSTEM-CALL INTERCEPTION:

* This method involved doing call interception in the kernel, say, by replacing the system-call dispatch table with one that routes the system call via the monitoring code for every system call. In fact, a number of security tools including anti-virus and application firewall products operate by doing system call interception at some level.

ADVANTAGE:

* Good performance. Since there is no context switch and just the data is being routed through another method in the kernel.

DISADVANTAGES:

* Kernel coding is much more difficult as opposed to user-level code.

* Operating Systems may not support replacing system call tables because this capability can be exploited by malicious code.

For example: Rootkits. The objective of a rootkit is to hide its presence from typical system monitoring tools, and to carry out their malicious activities while remaining hidden. Rootkits may be used to hide the presence of spyware or bot software on a system.

A popular way of implementing rootkits is to intercept system calls made by any process. So for example if any directory is being listed, the rootkit would intercept the data returned by the system call (used for directory listing) and remove any root-kit related files from the result. A similar strategy can be used to hide the presence of rootkit processes from process monitoring utilities.

Thus on Linux, the symbol that denotes this system-call table is not visible. Thus within the kernel, we cannot refer to this table by name. (But we can search for the table based on some kind of pattern matching on its contents.) On Vista too, the dispatch table was supposed to be protected, but this was relaxed later on since AV vendors complained.

II USER-LEVEL SYSTEM CALL INTERCEPTION:

1. Library-Interception:

* This method implies intercepting system calls by intercepting calls to the user-level library that makes the system call. This is because in a typical operating system, programs do not directly make system calls. Making a system call usually involves setting up the appropriate registers and executing the software interrupt

instruction. Hence it is easier to place this code in a library and for the user programs to call these library functions.

In Linux, this library is "libc". In Windows, similar role is played by "ntdll.dll".

* The basic idea is that while searching libraries, the OS searches for a specified set of directories. In Linux, the environment variable "LD_LIBRARY_PATH" stores the path of the directories to be searched for libraries. Thus by placing the instrumented library (which will intercept system calls and invoke the reference monitor on each call) earlier in the path, calls to libc can be intercepted.

  NOTE:

  Just as an aside, on Windows, the system call interface is not public and it not documented by Microsoft. Parts of the interface have been reverse-engineered and documented by third parties, but not the complete interface. Even for those operations that are documented, not all parameters are documented. The documented library is the Win32 library (kernel32.dll). As a result, there are many tools on Windows that rely on library interception of calls to this API.

ADVANTAGES/DISADVANTAGES of library interception:

* Performance. Since there is no context-switch involved, performance is adequate.

* Since this is at user-space, coding is relatively easier as compared to Kernel-level coding.

DISADVANTAGES:

* The main disadvantage is that this is not a secure mechanism. That is an untrusted code cannot be relied on to use this library. It could directly make system calls on its own. (On UNIX, such direct system calls are the most common means used by exploit code. On Windows, the use of Windows API is more common.)

* Even if the instrumented library is used, the checker is still running in the untrusted code's address-space and thus the code could again easily compromise the checks.

* As a result, this mechanism is inappropriate for use with untrusted code, or trusted code that contains an arbitrary code injection vulnerability. But it can be used on benign applications (which are trusted not to be malicious) that do not suffer from code injection vulnerabilities, or have been protected against injected code attacks.

  Consider a program written in Java or PHP, there do not exist binary code-injection vulnerabilities. However these might have injection of script-code. These are fine because launching of such scripts still requires benign code to make a system call or a function call. These operations can still be monitored --- since the benign code has not been compromised in any way before this call, it is reasonable to assume that it will not attempt to defeat library interception mechanisms.

  As a concrete example, consider an information flow policy that dictates that untrusted data should not be read or executed by a benign program. This means that there is no mechanism to exploit any vulnerabilities in this program to circumvent a reference monitor that enforces this policy using library interception.

EXAMPLE (OF LIBRARY-INTERCEPTION TECHNIQUE):

* In Windows, at times security tools including anti-virus tools use library interception method. Since the system-call interface is undocumented it becomes difficult to decide what should be allowed and what should not be allowed. Hence there is fair amount of usage of library-call interception method as a protection even against malicious/untrusted code, simply because having some protection is better than having no protection at all.

Of course, the flip-side is that having a false sense of security brought about by such a protection mechanism may cause more harm than good.

2. Kernel supported Interception user-level:

* Many UNIX-based operating systems provide kernel support for performing system call interception at the user-level.

* For example: In Solaris, there is a “proc” mechanism, which helps with system call interception. In Linux, we have the "ptrace" mechnism.

Ptrace is available in all Unix-based operating systems and had been originally provided for the purposes of debugging. This facility in Linux has been expanded to provide a way to intercept system calls as well.

* The basic idea is to have a "debugger" process examine the memory/state of the "debugged" process. Typical operations are "peek" (examining memory contents) and "poke" (modifying the state of the debugged process). Examining the registers of the process and single-stepping are other capabilities.

- The debugger can set "break-point" at desired points in the process under observation. This breakpoint can be set by writing (say) an invalid instruction at the desired breakpoint location. When the debugged process runs, when the breakpoint instruction is executed, a UNIX signal will be generated. Ptrace allows the debugger process to intercept this signal. At this point, it can examine memory interactively as desired by the user.

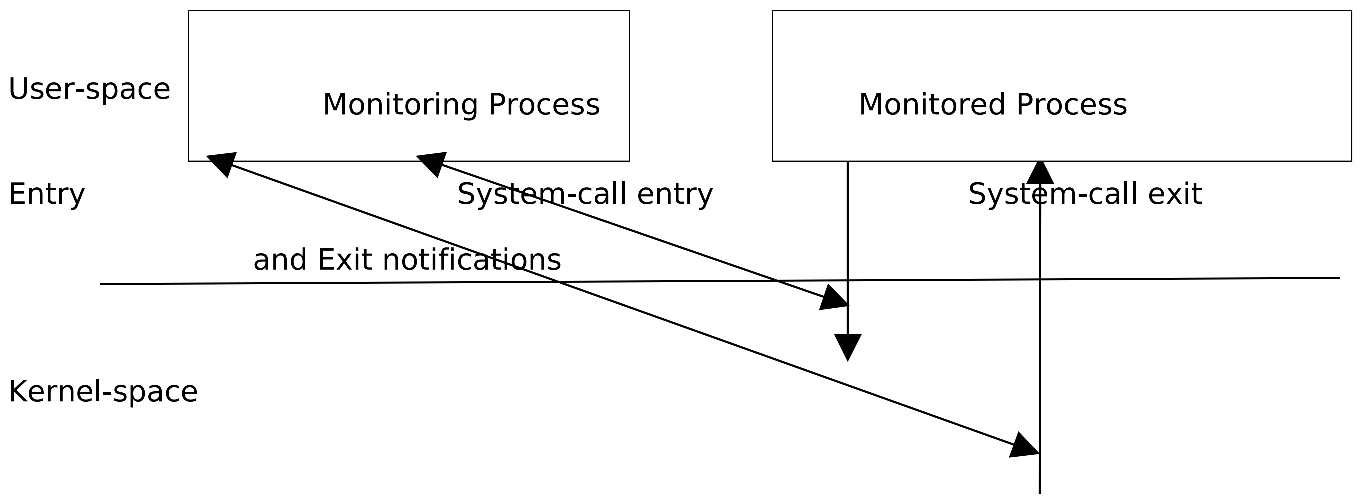


DIAGRAM FOR THE PTRACE OPERATIONS

* The ptrace interface has a couple of instructions which instruct/allow the monitored process to run until it makes a system call. When the process makes a system call, the control is handed over to the debugger/monitoring process.

  The monitor can then examine the system call being invoked, the parameters being passed to the system call and can even modify the register values and thus the parameters if it so wishes.

* Once the system-call is executed, the monitor is again notified of the result. Thus the monitor can again examine the results and also modify them if need be.

DISADVANTAGES of kernel-assisted user-level interception:

* The disadvantage of this method is the extra context-switches. For every system call there are two additional context switches (as compared to a purely in-kernel approach). Thus for processes that make a lot of system calls, the slowdown factor can be significant, sometimes more than 100%.

ADVANTAGES:

* The monitoring code is running at user-level and thus is relatively easy to implement.
* The mechanism is secure, and can be used with untrusted code.

III HYBRID TECHNIQUES:

* The basic idea of hybrid techniques is to improve performance by cutting down on the number of context-switches. Thus if the number of context-switches (in the ptrace mechanism) could be reduced by (say) a factor of 10, then performance would not be an issue.

* Thus if we can ensure that the most frequent operations are checked in the kernel and the rest of the operations are monitored at the user-level, we can achieve best of both worlds.

* Thus for example, we could have a policy that dictates that reads and writes are not security sensitive, and that only file open operations need to be checked. Thus reads-writes need not be sent to the user-level at all. On the other hand file-open operations can be sent to a user-level monitor.

  If reads-writes comprise 70-80% of the system-calls and only 20-30 % operations are file opens, then the improvement in performance is considerable.

INLINE REFERENCE MONITORS

ADVANTAGES

* Finer granularity
* Good Performance
* Much Better visibility:

The basic concept is that if we are running in the same address space then its much easier to understand and interpret memory contents, as opposed to running in a different address space.

DISADVANTAGES:

* Separate mechanism to protect reference monitor's data/logic. Note that the traditional solution to this problem, protecting data and code of process, called for each process to have an address-space of its own.

* Many years ago a technique called SOFTWARE-BASED FAULT-ISOLATION was developed.

  In operating systems, in addition to security, there is another reason for having separate address-spaces. And that is fault-isolation. For example if one process corrupts its memory state, with separate address-spaces for kernel and other processes, those remain uncorrupted.

  For example: Windows-95 did not have separate address spaces for kernel and processes. This contributed to system instability.

  Hence the term "fault-isolation" is used synonymously with memory-protection

* The basic idea of this technique is that software-based techniques are used to enforce memory protection.

* The technique involved two components as follows:

SOFTWARE-BASED FAULT-ISOLATION

Compiler	Verifier
* Generate code that checks the target of each memory reference	Needed because the code is compiled at a different site; the site running the code may not trust the compiler.

The compiler would generate code such that before the monitored process performs any memory access, (say) a write, checks would be performed to ensure that the location of the write was permissible, (say) was not within a certain range (that is used to store IRM's data and/or code).

Another important point to note is that code could still evade the all the memory checks (say) by jumping over the memory checks and thus skipping the checks. Hence another important point is for the verifier to ensure that the code should not evade the checks by jumping over them.

The above requirement is a pretty difficult one. The software-based fault isolation technique was originally developed for RISC instruction sets where problems are generally simpler including the disassembly of binary code. The way the above technique was implemented in the original work was that a register was dedicated for all memory accesses. This register was always guaranteed to contain a valid address before any control-flow transfer instruction. Hence, even if jumps attempted to jump to a location after the check, it is OK since the register contents are guaranteed at the point of jump, i.e., it points to memory allowed to be accessed by the code.

An alternative approach is to statically analyze all control flow transfers and ensure that there are no jumps that bypass the memory checks. But this is hard to do --- there are indirect jumps, calls and returns, and their targets cannot be statically checked easily. (In the next lecture, we will talk about a technique called “control-flow integrity” that attempts to do this.)