

Current Approaches to Policy based Security

A. L. Gottlieb

ABSTRACT

The effort to combat the effects of malicious software on unwary users is ongoing. Of the many approaches being studied today, the work being done in application oriented *Security Policies* is gaining wide recognition and is the focus of this report. Specifically, we shall look at two models. First, “Naccio”, conceived by Evans at MIT. Here, we have a system that will change programs to force them into compliance with established security policies. Second, we will examine “*Enforceable Security Policies*”, introduced by Schneider at Cornell University. In this model, applications are literally escorted thru their executions. Along the way, actions to be taken by the application are compared to established policy and if a violation is anticipated, the execution is brought to a halt.

“Naccio”

Naccio detects anomalies in source code then produces a new copy of the source code that is free of any anomalies and presumed safe for execution. Here, an anomaly is detected when the source code is in violation of a *safety policy*. Safety policies are constraints on system resources (e.g. number of open files per process). Upon detection of such an anomaly, Naccio will replace any offending system calls within the source code with calls to a *Policy-Enforcing Library*, I.E. a substitute library of system calls for a given platform where calls are wrapped with code to check and enforce a policy.

Architecturally, Naccio is divided into two main components, the *Policy Generator* and the *Application transformer*. “A policy author runs the policy generator to produce what the application transformer uses to enforce the policy on a particular program” [1].

Policy Generator

The policy generator is a software module run by the policy author. This individual will normally have detailed and timely knowledge of the current levels of system resources. The authors of Naccio are relying on the policy author to predefine a pool of safety policies from which users may select from as needed. This pool reduces the cost and effort of safety policy creation and enforcement. These pooled safety policies may be used repeatedly by many users against many applications. The inputs to the policy generator include: *Resource Descriptions*, *Safety Policies*, and *A Platforms Interface & Platform Library*. The Policy Generator outputs a *Policy Description File* and a *Policy-Enforcing Platform Library*.

Resource Descriptions

The policies generated will be applied to system resources such as files, network connections, threads and displays. Each resource must have a description that identifies the resource and the operations that can manipulate the resource. Resource descriptions are platform independent yet support platform specific resources such as the Windows registry or the */etc/services* file under UNIX.

A Resource Description for the file system:

1. Global resource RFileSystem
2. initialize () /*Called when execution starts.*/
3. terminate () /*Called just before execution ends.*/

4. openRead (file: RFile)
/*called before file is opened for reading*/
5. openCreate (file: RFile)
/*Called before a new file is created for writing*/
6. openWrite (file: RFile)
/*Called before an existing file is opened for writing*/
7. openAppend (file: RFile)
/*Called before existing file is opened for appending*/
8. close (file: RFile)

9. write (file: RFile, n:int)
/*Called before n bytes are written to file*/
10. preRead (file:RFile, n:int)
/*Called before n bytes are read from file*/
11. postRead (file:RFile, n:int)
/*Called after n bytes were read from file*/
12. delete (file:RFile)
/*Called before file is deleted*/

13. observeExists (file:RFile)
/*Called before revealing if file exists*/
14. observeWritable (file,RFile)
/*Called before revealing if file is writeable*/
..// other similar observe<X> operations elided

15. resource RFile
16. RFile (pathname: String)
/*Constructs object corresponding to pathname.*/

Worth noting about the Resource Description above are:

- Line 1: That for all executions, there is only one instance of an RFileSystem.
- Line 15: RFileSystem manipulations such as reading & writing require the RFile parameter, which stores a file's identity.
- Line 10: The preRead & postRead operations are a result of the splitting of the read manipulation. Such splitting is done to facilitate more granular safety policies. Without post facto operations, data gathered pre-operation would be inaccurate.

Resource Descriptions never assume a state of any kind nor are they implemented in any way. They are simple definitions of operations and their responsibility in policy generation is to indicate to a transformed program, the correct resource operation with the correct resource operation argument.

Safety Policies

The LimitWrite Safety Policy:

1. Policy LimitWrite
2. NoOverwrite, LimitBytesWritten (1000000)
3. property NoOverwrite
4. check RFileSystem.openWrite (file: RFile),
5. RFileSystem.openAppend (file: RFile),
6. RFileSystem.delete (file: RFile)
7. violation ("Attempt to overwrite file");
8. property LimitBytesWritten (limit)
9. requires TrackTotalBytesWritten;
10. check RFileSystem.Write (file: RFile, n:int)
11. if (bytes_written > limit) violation ("You cannot do this!!");
12. stateblock TrackTotalBytesWritten
13. addfield RFileSystem.bytes_written : int = 0;
14. precode RFileSystem.write (file: RFile, n:int);
bytes_written += n;

The salient features of the LimitWrite Safety Property include:

- Line 1: The LimitWrite Safety Policy will apply constraints to RFileSystem operations.
- Line 2: The NoOverwrite safety property which prevents the changing of any file.
- Line 8: The LimitBytesWritten safety property which institutes a 1 million byte limit on writing to the file system (I.E. The RFileSystem)
- Line 4: A check clause bound to the NoOverwrite safety property. The clause contains the openWrite, openAppend and delete operations defined in the RFileSystem Resource Description and passes RFile, the parameter defined in RFileSystem that identifies the file.
- Line 7: In the event a check clause operation returning a value indicating a violation of the NoOverwrite property, the violation clause will be executed.
- Line 12: The TrackBytesWritten is called by the requires clause on Line 9; inside the LimitByteWritten property beginning on Line 8. This property is assigned the task of maintaining a current count of the number of bytes written, providing a basis for deciding if the RFileSystem resource can accommodate the next write.
- Line 13: The bytes_written field is added to the RFileSystem Resource Description. This variable will be set to the current number of bytes written when the TrackTotalBytesWritten stateblock is called.
- Line 14: The precode keyword dictates an action to be taken just prior to the write operation. The actions will be the execution of the check clause code which here, will yield the bytes_written variable.

Safety policies are platform independent yet their applicability is limited by the total number of resource operations. “Naccio can detect violations and observe and modify state only at execution points corresponding to resource operations” [1].

Platform Interface & Library

Presently, Naccio implementations are limited to the Win32 and JavaVM platforms. In each case, safety policies are affective when resource operations are consistent with their respective resource descriptions. In this regard, the *Platform Interface* is responsible for providing the linkage between the system call and the resource.

“In the case of JavaVM, the platform interface must describe how native methods in the Java API affect resources” [1]. Native methods installed by applications must be brought under policy control by Naccio. The alternative is to disallow such installations. For non native API methods, constructors and initializes, the choices are to either implement them with a platform interface wrapper; in which case, complete knowledge of what resources will be impacted by the method is critical or let the procedure go unchecked.

Below, is code from the Java API platform interface defining wrappers for the `java.io.FileOutputStream` class:

```
1. wrapper Java.io.FileOutputStream
2. requires RFileMap;
3. state RFile rfile;

4. wrapper FileOutputStream (java.io.File file)
5.     rfile = RFileMap.lookupAdd (file);
6.     if (file.exists())
7.         RFileSystem.openWrite (rfile)
8.     else
9.         RFileSystem.openCreate (rfile)
10.    %%% // marker for original call

11.    ... // Other constructors similar.

12. wrapper void write (byte data[])
13.     if (rfile != null) RFileSystem.write (rfile, data.length);
14.     %%%

15.    ...// Other write methods similar.
```

Line 3: State variable “rfile” will keep track of the Rfile object assigned to the `FileOutputStream` on Lines 7 & 9 in the `FileOutputStream` wrapper.

Line 3: State variable “rfile” will keep track of the Rfile object assigned to the write on Line 13 in the write wrapper. “Wrappers for constructors must set this state to the appropriate value” [1].

Line 5: Retrieves the Rfile object associated with a Java file object.

Line 6: If the resource file exists, it is passed as a parameter to either `RFileSystem.openWrite` or `RFileSystem.openCreate` resource operations. That decision is made by the wrapper’s call to `java.io.file.exists`; the unwrapped version. After the file is either opened or created, execution resumes at Line 10 where the original constructor is called.

Policy-enforcing library

“The policy-enforcing library contains wrapped versions of system calls as directed by the platform interface” [1]. The number and selection of system calls to be wrapped is a function of performance and resource implementation criteria. Individual criterion include calls that do useful work, calls that modify a state that will later become significant and calls that change the programs behavior. Omission of key system calls being wrapped for the sake of performance may leave system resources vulnerable.

The JavaVM policy-enforcing library produced by Naccio:

1. Modified the Java API classes during production of the library.
2. Wrapper implementation was accomplished by renaming the original method. The original method must remain available for calls by both the wrapped method and other methods of the same class. Some programs will call the original (unwrapped).
3. Insertion of the wrapper code into the Java class is accomplished by compiling the code from the platform interface into Java byte codes. The wrapper version now has the name of the original method and will thus, call the original method.
4. Avoidance of duplicate checking on the part of other wrapped API methods by having them call the unwrapped version of a method. Here, wrapped methods can manage their own resource usage.
5. The modified classes are then written to a new directory where the CLASSPATH variable will be set to when a user wishes to run an application passed Naccio.

Policy description file

The Policy description file services the Application transformer by providing:

1. Application transformation rules to be adhered to by the application transformer.
2. The location of the Policy-enforcing library.
3. “Rules directing the application transformer to rename wrapped native methods, and to modify the application to call resource initializers just before execution and to finalizers just before execution terminates” [1].

Application transformer

The application transformer transforms an application in accordance with the directives in the policy description file. As would be expected of any utility operating on object files, the application transformer is very platform dependent. As such, the transformer is responsible for matching the application to the correct policy-enforcing library classes. Applications that execute in their own VM differ from those that run in a VM with other active policies. In the case of the former, simply setting the CLASSPATH environmental variable will bring the modified classes to the forefront instead of the standard Java API. In the case of the latter, the policy-enforcing library containing the renamed classes must be enabled.

Enforceable Security Policies

To date, operating system based security policies have been applied to issues concerning:

1. Availability: Denying a process the ability to deny a resource to other processes.
 2. Access Control: Prohibiting operations processes may commit against objects.
 3. Bounded Availability: Programs must release acquired resources after a fixed period.
 4. Information flow: Restricting inferences about objects based on system behavior.
- More recently, application dependent security policies that act upon an applications state Provide a finer granularity in policy accuracy than the operating system bases policies described above which act upon approximations.

Targets & Valid Enforcement Mechanisms

Within the context of an executing application (*target*), Schneider offers a class of enforcement mechanisms that monitor the steps taken by such an execution. If a violation of a security policy is foreseen during the Execution Monitoring (*EM*) then the target is halted. Objects, modules, processes, subsystems and entire systems all are viable targets for EM and reference monitors, firewalls and virtual memory are all potential mechanisms. “The execution steps being monitored may range from fine-grained actions (such as memory accesses) to higher level operations (such as method calls) to operations that change the security-configuration and thus restrict subsequent execution”. [2]

Mechanisms that are excluded from Execution Monitoring

Execution monitoring mechanisms may not include:

1. Mechanisms using additional information about an execution than what can be gathered by simply observing its progress. Compilers and theorem-provers are precluded from EM mechanism status because they report on a non-running executable and their output include the following:
 - A. Predicts future execution steps expected of an executable
 - B. Predict alternative execution paths.
 - C. Predict the set of all possible executions.
2. Mechanisms which modify or alter a target before execution.

Security policies

“We define a security policy to be a predicate on sets of executions”. [3] Here, an execution is a sequence of instructions or states. A security policy applies to an illegal instruction or state in a given execution.

What is a Property?

A property is a policy that makes decisions based on the observation of a single programs step by step executions. A property may not speculate on possible executions of the program; only what is observed. A property may not relate two different executions. Suppose a program is run with an input, *input_1* and produces an output, *output_1*. The next run of the program should yield a different *output_2* from a different *input_2*. If this is not true then this is cannot be a property.

What is a Safety Property?

A *safety* property states “nothing bad may happen”, and any continuation of the execution after such an event will not correct the violation. Such is the case with access-control where once a policy forbidden resource has been accessed; the policy violation cannot thereafter be un-done during the same execution. [3] When a policy violation occurs, execution must be terminated. The violation must be observed in a finite period of time.

What is a Liveness Property?

A *liveness* property states that during the steps of an execution, additional steps may be added upon acquisition of a resource. But the steps here are finite so during this finite period of time, “nothing bad can happen”. So, if rejection of an execution occurs, it must happen after some finite period of time and EM cannot wait for the execution to complete before deciding to reject.

NOTE: A security policy, to be enforceable, must satisfy all the above conditions.

Security Automata

“All safety properties can be characterized by automata”. [4] Similar in form to the non-deterministic finite-state automata and belonging to the Buchi class [Eilenberg 1974], are what we will refer to as *security automata*; introduced by Alpern and Schneider [1987]. These automatons are fully accepting of safety properties.

Operationally, when a target is executed, a simulation of the security automaton that serves an enforcement mechanism in EM is executed in tandem with the target. For every step during the execution of the target that is about to happen, an input symbol

representing that step is created then sent to the simulated automata. Within the automaton, if the transition function permits a successful transition to a new state based upon the input symbol, execution of the target will continue. However, if a transition based on that input symbol is unsuccessful, then a policy violation has occurred and execution of the target will be halted.

Considerations concerning the use of Security Automata

1. Some safety properties demand targets be permitted to execute until completion before a termination decision can be made. This fact results in the cumulative growth of the security automaton and the proportional reduction of memory.
2. If a policy violation is detected by the security automaton, can the enforcement mechanism actually halt the execution of the offending target?
3. If the target either disrupts or avoids the security automaton, the enforcement mechanism will not be informed of a policy violation and the target will continue.

References

- [1] David Evans, Andrew Twyman, “Flexible Policy-Directed Code Safety”, *IEEE Security and Privacy*, [1999, May 9], <http://naccio.cs.virginia.edu/sp99.pdf>, 1999
- [2] Fred B. Schneider, “Enforceable Security Policies”, *ACM Transactions on Information and System Security*, [2000, Feb], <http://naccio.cs.virginia.edu/sp99.pdf>, 1999
- [3] Lujo Bauer, Jarred Ligatti, David Walker, “More Enforceable Security Policies, *Foundations of Computer Security Workshop*, Copenhagen, Denmark, [2002, July], <http://www.cs.cornell.edu/html/CS513-sp99/NL18.html>
- [4] Lynette Millet, Borislav Deianov, “Enforceable Security Policies, *CS 513 System Security—Lecture Notes*, [1998, April 6], <http://www.cs.cornell.edu/html/CS513-sp99/NL18.html>
- [5] Alpern, B, Schneider, F. B. 1987. Recognizing safety and liveness. *Distributed Computing*. 2, 117-126. <http://www.cornell.edu/fbs/fullist.htm>
- [6] Eilenberg, S. 1974. *Automata, Languages, and Machines, Volume A*. Academic Press, Inc., New York, NY.