Enforcing IRM Security Policies: Two Case Studies*

Micah Jones
University of Texas at Dallas
Email: micah.jones1@student.utdallas.edu

Kevin W. Hamlen
University of Texas at Dallas
Email: hamlen@utdallas.edu

***Abstract*—SPoX (Security Policy XML) is a declarative language for specifying application security policies for enforcement by In-lined Reference Monitors. Two case studies are presented that demonstrate how this language can be used to effectively enforce application-specific security policies for untrusted Java applications in the absence of source code.**

I. Introduction

In-lined Reference Monitors (IRM's) [15] are an emerging paradigm for enforcing a powerful and versatile class of software security policies in the absence of source code. In an IRM framework, a *rewriter* automatically transforms untrusted applications (e.g., Java bytecode binaries) in accordance with a client-specified security policy. The rewriting process involves inserting dynamic security guards around potentially security-relevant operations in the untrusted code. The inserted guards preserve policy-adherent behavior but detect and prevent policy violations at runtime; impending violations trigger a remedial action (e.g., premature termination).

IRM's are useful for enforcing *safety policies* [9], [13]. Informally, a safety policy is any policy that prohibits untrusted applications from exhibiting "bad" events. This includes all access control policies (where "bad" events are unauthorized accesses of security-relevant resources), as well as time-bounded availability policies (where "bad" events are a failure to make progress after a certain number of seconds or cycles).[1]

Traditionally, safety policies are enforced at the operating system, virtual machine, or hardware levels; however, this approach can be impractical when policies are application-specific or when they refer to fine-grained events that are not efficiently observable from outside the untrusted process. For example, an access control policy that constrains system calls performed by a web browser plug-in is not easily enforceable by the operating system when both the plug-in and the browser run in the same address space. Furthermore, enforcing such a policy traditionally requires modifying the operating system or virtual machine to support it, which is time-consuming, error-prone, and can introduce compatibility problems. In contrast, IRM systems yield rewritten, *self-monitoring* applications without modifying the operating system, system libraries, or virtual machine. This makes them highly flexible and well-suited to enforcing such policies with minimal overhead.

*This research was supported by AFOSR YIP award number FA9550-08-1-0044 and a grant from Texas Enterprise Funds.

[1]Safety policies are sometimes contrasted with *liveness policies*, which instead say that some "good" event must eventually happen. Non-time-bounded availability policies are examples of liveness policies.

Much of the past work on IRM's has relied upon specification languages in which policies are expressed partly imperatively, such as those that include code fragments for the rewriter to insert into the untrusted code (e.g., [3], [4], [6]). Such policies are extremely difficult to analyze and reason about because they specify code-transformations instead of code properties. Other work has expressed policies purely declaratively, but at the cost of significantly reduced power (e.g., by limiting the language of security-relevant events to method calls [1], [7], [10]). Specifying and enforcing more powerful classes of declaratively specified security policies using IRM's has remained an open problem in the field.

To address this deficiency, we have been developing Security Policy XML (SPoX)—a purely declarative yet powerful IRM policy specification language that supports the following:

- **An Aspect-oriented event language:** Security-relevant *events* are specified in SPoX using *pointcuts* from Aspect-Oriented Programming (AOP) [11]. A *pointcut* defines a set of machine instructions. Our Java implementation of SPoX uses a pointcut language based on AspectJ [12] to specify security-relevant operations. This allows policy-writers to enforce policies that regard dynamic method calls and their arguments, memory usage limits, numerical computations, and other important security properties observable at the bytecode instruction level.
- **History-based policies:** Many security policies constrain event histories rather than individual events. For example, to enforce data confidentiality a policy might constrain only those network send operations that occur after reading a confidential file. SPoX expresses history-based policies with *security automata* [15]—finite state automata that accept permissible event sequences. Since the state space of real security policies is often large, SPoX uses a linear constraint system to efficiently represent resource counters and other common policy primitives.
- **Separation of concerns:** In order to keep the language purely declarative, SPoX specifications encode only the policy to be enforced and not a particular enforcement strategy. This makes it easier both to write the policy and to derive meaningful guarantees for the enforcement system. In past work [8] this has allowed us to derive a formal denotational semantics for SPoX that could be used to formally verify rewriting algorithms and the code they produce. Such verification is important for applying IRM certification systems [2], [10].

In this article we relate some preliminary experiences using SPoX to enforce security policies for real-world Java bytecode applications. While our present implementation is limited to Java, we believe that the language design can be easily adapted to many other languages and platforms. Our Java-based rewriter parses SPoX policies and applies them to arbitrary Java binaries. In section II, we discuss two interesting case studies. Section III concludes by summarizing what we have learned and suggesting future work.

II. Case Studies

In this section we discuss two case studies. For the first, we created a file-oriented access control policy and enforced it on the Columba email client [5]. For the second, we applied an anti-freeriding policy to the XNap p2p file sharing client [14].

A. Columba Email Client

Columba is an open-source Java email application. We enforced a policy that prohibits Columba from creating or accessing files whose names end in `.exe`. Such a policy is useful for inhibiting virus propagation through email attachments.

Effectively enforcing this policy requires constraining calls to Java library methods that access the file system. The number of such methods is surprisingly vast; it includes methods that stream data, those that create and manipulate SQL databases (since those databases reside in files that could have a prohibited name), etc. Listing all such methods would be difficult (and error-prone) for the average administrator, so we developed a pointcut library that identifies these method calls. The following is an excerpt:

```
(pointcut name="fileMethods"
  (or
    (call "java.io.File.new")
    (call "java.io.FileWriter.new")
    (call "java.io.FileReader.new")
    ...
```

SPoX's aspect-oriented design allows such libraries to be maintained modularly by a trusted expert. Policy-writers can then refer to these libraries to compose higher-level, application-specific policies. The disjunctive pointcut above was stored under the name `fileMethods`.

To precisely enforce the policy, the specification must constrain the runtime arguments to these calls. For this we use SPoX's `argval` predicate:

```
(or (and (pointcutid name="fileMethods")
     (or (argval num="1" (streq ".*\.exe.*"))
         (argval num="2" (streq ".*\.exe.*"))))
    (call "java.lang.Runtime.exec"))
```

Our rewriter dynamically decides these predicates by injecting code that calls the `toString()` member of each argument and performs the requested regular expression comparison. Note that this policy also prohibits access to Java's `java.lang.Runtime.exec` method, which could be used by an attacker to execute an arbitrary (untrusted) external application.

Our rewriter required about 28 seconds on an Intel 2.66GHz Core 2 Duo processor and increased the original 2.8MB Columba binary by 3.6%. The runtime performance of the rewritten code could not be measured formally because Columba requires user interaction when executed; however, we did not observe any noticeable performance overhead due to the inserted security checks. Likewise, no behavioral change to the application was observed except in the event of a policy violation—accessing a file with a `.exe` extension forced a premature termination of the application.

Our policy did have the unexpected side-effect of disabling Columba's spell-checker. Upon investigation, we found that this feature used `Runtime.exec` to launch an external spell check application, which was blocked by the policy. Had we wished to allow this program to run anyway, we could have modified the policy to whitelist certain application names as follows:

```
(and (call "java.lang.Runtime.exec" )
     (not (argval num="1" (streq "filename"))))
```

where `filename` is a regular expression denoting names of trusted external applications. For modularity, the list of trusted executables could be maintained as a separate pointcut library.

The policy specification above defends against certain malware propagation attacks, but has several deficiencies that could be remedied by a more sophisticated specification. For example, one could easily extend the regular expressions to prohibit other dangerous file extensions. In addition, our `fileMethods` library was somewhat informally derived. A more rigorous examination of the Java runtime libraries would likely uncover additions to the library that would be necessary to rule out all possible violations.

B. XNap Peer-to-peer File Sharing Client

XNap is an open-source file-sharing client implemented in Java. We enforced an anti-freeriding policy that requires the number of downloads to be at most two larger than the number of uploads during a given session. This is an interesting policy because it is both history-based and application-specific. In addition, the security state space is potentially large—one security state for each possible difference between the number of downloads and uploads.

To enforce the policy, we wrote the following specification:

```
(pointcut name="download"
 (execution "xnap.net.MultiDownload.download"))
(pointcut name="upload"
 (and (execution "xnap.util.UploadQueue.add")
      (argtyp num="1"
              "xnap.net.IUploadContainer")))
(state name="s")
(forall var="i" from=-2147483646 to=2
  (edge name="queue_download"
    (nodes var="s" "i,i+1")
    (pointcutid name="download"))
  (edge name="queue_upload"
    (nodes var="s" "i,i-1")
    (pointcutid name="upload")))
(edge name="too_many_downloads"
  (nodes var="s" "2,#")
  (pointcutid name="download"))
```

The policy above essentially creates a counter where downloads increment the value of state variable `s` and uploads (or, more precisely, additions to the upload queue) decrement it. A user is allowed to download two more files than he has uploaded; additional downloads trigger a policy violation and the program halts. The details of the policy can be understood in terms of the following policy components:

- The `execution` pointcut is like `call`, but matches method entrypoints rather than call sites.
- `argtyp` matches a method's (possibly dynamic) parameter types. This allows the policy to distinguish between identically-named methods that differ only by signature.
- The `state` structure defines a state variable that tracks the current security state. In the above policy, the security state is global, but SPoX also supports per-instance security state for enforcement of per-object security policies.
- Each `edge` declaration defines how the security state changes in response to security-relevant events.
- Within each `edge`, the `nodes` specify the starting and ending security states for the transition. For example, if `s` is in state 2 and an instruction matching the `download` pointcut is about to be executed, the final edge will change `s` to "#", which is a reserved state value that represents a policy violation (i.e., no permitted transition).
- `forall` structures enclose `edges` (and possibly nested `foralls`) to declare a large collection of similar `edges`. This strategy both shortens policy specifications and aids in efficient policy enforcement and policy analysis [8]. Here we declare edges that increment and decrement state `s` as downloads and uploads occur.

Since this is an application-specific policy, its formulation required some knowledge of the internal structure of the application; however, this was easily gleaned without any access to the application source code. We pinpointed the relevant methods for download and upload operations via a cursory examination of the bytecode disassembly.

We're aware of at least two important deficiencies in this policy as we've defined it here. First, since the security state does not persist across application instances, users can exit out of the application and restart it after every two downloads to reset the counter, and thereby increase downloads over uploads over time. Previous work has demonstrated how to implement a persistent security state in an IRM to remedy such vulnerabilities [1]. We intend to explore such an extension to our work in the future. Second, our policy only tallies queued uploads but not completed uploads. A malicious user could freeride by cancelling queued uploads before completion. To close this vulnerability, we could have added extra logic to dictate how download completions and cancellations affect the current security state.

III. Conclusion

We have here presented a powerful yet purely declarative language for encoding security policies for enforcement by IRM's. Some preliminary experiences with our IRM framework were related in the form of two case studies. Each case study enforced a realistic security policy over a pre-existing production-level application. The policies illustrate several of the more interesting components of our policy language, highlighting its versatility.

There are two primary kinds of policies that we have developed during our experiments: those that are general-purpose, usually pinpointing specific Java library method calls (e.g., the Columba policy); and those that are application-specific (e.g., the XNap policy). Formulating each poses unique challenges. General-purpose policies tend to require comprehensive architectural knowledge, such as identifying all system library methods that can access security-relevant resources. This has led us to develop a modular specification language in which common policy components can be separated into external libraries compiled by a system expert. Application-specific policies tend to require at least a superficial understanding of the underlying program structure. This points to a need for good disassembly and visualization tools.

In future work we intend to leverage SPoX's purely declarative formulation to develop tools for analyzing and optimizing security policies. Our past work has indicated that such formal analysis is more tractable than with specification languages that include imperative components [8]. This will lead to robust certification systems [2], [10] for SPoX implementations.

References

[1] I. Aktug and K. Naliuka. ConSpec: A formal language for policy specification. In *Proc. of the 1st Int. Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM'07)*, pages 45–58, 2007.

[2] I. Aktug, M. Dam, and D. Gurov. Provably correct runtime monitoring. In *Proc. of the Int. Sym. on Formal Methods (FM)*, 2008.

[3] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. of the ACM Conf. on Prog. Lang. Design and Implementation (PLDI)*, pages 305–314, 2005.

[4] F. Chen and G. Roşu. Java-MOP: A Monitoring Oriented Programming environment for Java. In *Proc. of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 546–550, 2005.

[5] F. Dietz and T. Stich. Columba (v1.4). www.columbamail.org.

[6] Ú. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proc. of the IEEE Symp. on Security and Privacy*, 2000.

[7] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. of the New Security Paradigms Workshop (NSPW)*, pages 87–95, 1999.

[8] K. W. Hamlen and M. Jones. Aspect-Oriented In-lined Reference Monitors. In *Proc. of the 3rd ACM Workshop on Prog. Lang. and Analysis for Security (PLAS)*, 2008.

[9] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. On Prog. Lang. and Systems (TOPLAS)*, 28(1):175–205, 2006.

[10] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *Proc. of the 1st ACM Workshop on Prog. Lang. and Analysis for Security (PLAS)*, pages 7–15, 2006.

[11] G. Kiczales, J. Lamping, A. Medhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. of the 11th European Conf. on Object-Oriented Prog. (ECOOP)*, volume 1241, pages 220–242, 1997.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of the 15th European Conf. on Object-Oriented Prog. (ECOOP)*, volume 2072, pages 327–355, 2001.

[13] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Trans. on Software Engineering (TSE)*, 3(2):125–143, 1977.

[14] Y. J. Leist and S. Pingel. XNap (v2.5r3). xnap.sourceforge.net.

[15] F. B. Schneider. Enforceable security policies. *ACM Trans. on Info. and System Security (TISSEC)*, 3(1):30–50, 2000.