

Replace this file with `prentcsmacro.sty` for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the [ENTCS Macro Home Page](#).

The S3MS.NET Run Time Monitor

Lieven Desmet¹ Wouter Joosen¹ Fabio Massacci²
Katsiaryna Naliuka² Pieter Philippaerts^{1,3} Frank Piessens¹
Dries Vanoverberghe¹

Abstract

This paper describes the S3MS.NET run time monitor, a tool that can enforce security policies expressed in a variety of policy languages for .NET desktop or mobile applications. The tool consists of two major parts: a bytecode inliner that rewrites .NET assemblies to insert calls to a policy decision point, and a policy compiler that compiles source policies to executable policy decision points. The tool supports both singlethreaded and multithreaded applications, and is sufficiently mature to be used on real-world applications.

This paper describes the overall functionality and architecture of the tool, discusses its strengths and weaknesses, and reports on our experience with using the tool on case studies as well as in teaching.

Keywords: security, bytecode rewriting, .NET, MSIL

1 Introduction

In today's networked world, code mobility is ubiquitous. Even mobile phones and Personal Digital Assistants increasingly support the installation of third party applications from a variety of sources. This support for applications from potentially untrustworthy sources comes with a serious risk: malicious or buggy applications can lead to denial of service, financial damage, leaking of confidential information and so forth. The research community has developed a variety of countermeasures for addressing the threat of untrusted mobile code. One important class of countermeasures addresses this risk by monitoring the application at run time, and aborting it if it violates a predefined security policy.

This paper reports on a tool developed within the *Security of Software and Services for Mobile Systems (S3MS)* project. It implements a monitor for the .NET platform through bytecode inlining. Several of the key algorithms implemented in the tool have been proven formally correct, and the implementation is sufficiently

¹ DistriNet Research Group, Department of Computer Science Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium

² Department of Information and Communication Technology, Universit di Trento, Via Sommarive 14, I-38050 Povo (Trento), Italy

³ Email: Pieter.Philippaerts@cs.kuleuven.be

mature to handle real-world applications for both the .NET Compact Framework (for mobile devices) and the full .NET Framework (for desktops and servers).

2 Tool Architecture

2.1 Overview

As case studies [10] show, the major security concerns of users about third-party applications are invocations of functionality that incurs a monetary cost, and treatment of sensitive data. Access to this functionality as well as to the sensitive information is provided by calling system API methods. A simple way to prevent the application from causing harm is to suppress the calls to the potentially dangerous methods, effectively *sandboxing* the application. However, in this way the useful functionality that the application can provide is also hampered. To allow this functionality to the application, without compromising security, the access to the sensitive system calls (later called *security-relevant* methods) should be regulated by the *policy*, which grants access to security-relevant methods according to specified rules. These rules can include conditions on the environment (e.g. time) or on the previous access requests of the application (as in history-based access control [4]).

To define what functionality is considered security-relevant, we rely on the policy. For example, if the policy prohibits network accesses after sensitive information was accessed by the application, then the security-relevant API calls are “starting a connection” and “accessing sensitive information”. All other operations, such as creating files, are irrelevant to this policy and need not to be monitored. Note, that some operations are more likely to be listed as security-relevant than others. For instance, GUI operations are unlikely to be listed as security-relevant by any realistic policy, and therefore will be executed without any monitoring overhead.

The S3MS.NET run time monitor consists of two key components (Figure 1). The *inliner* rewrites potentially dangerous applications. It scans the bytecode to find security-relevant API calls, and wraps additional code around such calls. This additional code checks whether the application is allowed to perform this call. If so, the wrapper code will silently allow the application to continue. If not, the application will be interrupted.

The second component, called *the policy compiler*, generates an executable version of the policy that the user has created. The dotted line in Figure 1 signifies that the wrapper code inserted by the inliner will call functions in the executable policy, generated by the policy compiler.

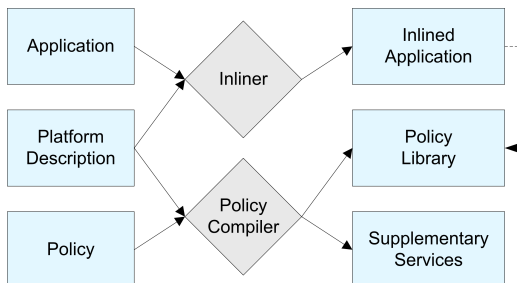


Fig. 1. The architecture of the tool

The tool supports multiple platforms – most notably the .NET Compact Framework and the .NET Full Framework – and hence both tool components additionally take a platform description as input.

We discuss each of the two components in some more detail.

2.2 The Inliner

The inliner loops over the bytecode of an untrusted application looking for calls to security-relevant methods (SRM). Identifying such calls statically in the presence of dynamic binding and delegates (a form of type safe function pointers supported by the .NET virtual machine) is non-trivial. The tool implements the algorithm by Vanoverberghe and Piessens [8].

Before and after each call to a SRM, a call to the executable version of the policy, called the *policy decision point (PDP)*, is injected. A PDP is a Dynamic Link Library that manages the security state associated with the application. It can be thought of as an implementation of a security automaton [6] that reacts to the start and return (both normal and exceptional) of SRMs.

Listings 1 and 2 show the effect of inlining on a simple program that sends an SMS. If the method to send SMS's is considered security relevant, the inliner will transform it as shown. Note that the tool operates on the level of bytecode, not on source level, but we show the results as they would look at source level to make the transformation easier to understand.

```
SmsMessage message = ...
message.SendSMS();
```

Listing 1. Example code that sends an SMS message on a mobile phone.

```
SmsMessage message = ...
PDP.BeforeSendSMS(message);
try {
    message.SendSMS();
    PDP.AfterSendSMS(message);
} catch (SecurityException se) {
    throw se;
} catch (Exception e) {
    PDP.ExceptionSendSMS(message, e);
    throw;
}
```

Listing 2. The SMS example code, after inlining.

Before each SRM call, a *'before handler'* is added, which checks whether the application is allowed to call that method. If not, an exception is thrown. This exception will prevent the application from calling the method, since the program will jump over the SRM to the first suitable exception handler it finds.

Likewise, after the SRM call, an *'after handler'* is added. This handler typically only updates the internal state of the PDP. If an exception occurs during the execution of the SRM, the *'exceptional handler'* will be called instead of the *'after handler'*. In summary, the different handler methods implement the reaction of the security automaton to the three types of events: calls, normal returns and exceptional returns of security-relevant methods.

2.2.1 *Inheritance and Polymorphism*

The simplified code shown above does not deal with inheritance and dynamic binding. Support for this was implemented by extending the logic in the PDP to consider the type of the object at runtime, instead of only looking at the static type that is available during the inlining process. When a security-relevant virtual method is called, calls are inlined to so-called *dynamic dispatcher methods* that inspect the runtime type of the object and forward to the correct handler. The details, and a formal proof of correctness of this inlining algorithm is presented in [8].

2.2.2 *Multithreading and Synchronization*

Inlining in a multithreaded program requires synchronization. Two synchronization strategies are possible: strong synchronization, where the security state is locked for the entire duration of a SRM call, or weak synchronization where the security state is locked only during execution of the handler methods.

Our tool implements strong synchronization, which might be problematic when SRMs take a long time to execute, or are blocking (e.g. a method that waits for an incoming network connection). To alleviate this problem, the tool partitions the handler methods according to which security state variables they access. Two partitions that access a distinct set of state variables can be locked independently from each other.

2.3 *The Policy Compiler*

The policy compiler is the component that translates source policies, written by the user or the system administrator, into an executable policy decision point.

The tool supports two different policy languages, one that represents security automata by means of an explicit declaration of the security state, and guarded commands that operate on this state, and another one that is a variant of a temporal logic. Both languages extend history-based access control by introducing the notion of *scopes*. A scope specifies whether the policy applies to (1) a single run of each application, (2) saves information between multiple runs of the same application or (3) gathers events from the entire system.

2.3.1 *ConSpec*

ConSpec ([1]) is directly based on the notion of security automata, and is similar to Erlingsson’s PSLang[7] policy language. Like PSLang, a ConSpec specification includes the definition of state variables and the definition of what state transitions are caused by each of the security relevant events. An SRM is executed if the state allows it, and the state is updated accordingly before or after the execution of the SRM. ConSpec extends PSLang with support for multiple scopes.

2.3.2 *2D-LTL*

An alternative to ConSpec is the 2D-LTL policy language [5], a temporal logic language based upon a bi-dimensional model of execution. One dimension is a sequence of states of execution inside each run (session) of the application, and another one is formed by the global sequence of sessions themselves ordered by

their start time. To reason about this bi-dimensional model, two types of temporal operators are applied: local and global ones. Local operators apply to the sequence of states inside the session, for instance, the “previously local” operator (Y_L) refers to the previous state in the same session, while “previously global” (Y_G) points to the final state of the previous session.

3 Experience and Discussion

The S3MS.NET run time monitor was developed in the European FP6 project, Security of Software and Services for Mobile Systems (S3MS). The tool is a component of a comprehensive security architecture for mobile devices [2] that supports a novel paradigm for developing trustworthy applications, the security-by-contract paradigm [3].

The implementation of the tool, as well as supporting documentation and examples can be found at <http://www.cs.kuleuven.be/~pieter/inliner/>.

Space limitations make it impossible to discuss related research in this paper. We refer to the public S3MS deliverables at <http://www.s3ms.org> for a detailed overview of related work. Here, we limit ourselves to a brief summary of our experiences with the tool.

In the context of the S3MS project, we gained experience with the tool described in this paper on two case studies:

- a “Chess-by-SMS” application, where two players can play a game of chess on their mobile phones over SMS.
- a multiplayer online role-playing game where many players can interact in a virtual world through their mobile phones. The client for this application is a graphical interface to this virtual world, and the server implements the virtual world, and synchronizes the different players.

In addition, the tool was used to support a project assignment for a course on secure software development at the K.U.Leuven. In this assignment, students were asked to enforce various policies on a .NET e-mail client.

Based on these experiences, we summarize the major advantages and limitations of the tool.

A major advantage of the tool, compared to state-of-the-art code access security systems based on sandboxing (such as .NET CAS and the Java Security Architecture) is its improved expressiveness. The main difference between CAS and the approach outlined here, is that CAS is stateless. This means that in a CAS policy, a method call is either allowed for an application or disallowed. With the S3MS approach, a more dynamic policy can be written, where a method can for instance be invoked only a particular number of times. This is essential for enforcing policies that specify quota on resource accesses.

A second important strength of the tool is its performance. A key difference between CAS and our approach, is that CAS performs a stack walk whenever it tries to determine whether the application may invoke a specific sensitive function or not. Because stack walks are slow, this may be an issue on mobile devices (CAS is not yet implemented on the .NET Compact Framework). The speed of the S3MS

approach mainly depends on the speed of the *before* and *after handlers*. These can be made arbitrarily complex, but are usually only a few simple calculations. This results in a small performance overhead. Microbenchmarks [9] show that the performance impact of the inlining itself is negligible, and for the policies and case studies done in S3MS, there was no noticeable impact on performance.

Finally, the support for multiple policy languages and multiple platforms makes the tool a very versatile security enforcement tool.

A limitation is that we do not support applications that use reflection. Using the reflection API, functions can be called dynamically at runtime. Hence, for security reasons, access to the reflection API should be forbidden, or the entire system becomes vulnerable. We do not see this as a major disadvantage, however, because our approach is aimed at mobile devices, and the reflection API is not implemented on the .NET Compact Framework. Also, by providing suitable policies for invoking the Reflection API, limited support for reflection could be provided.

A second limitation of the approach implemented in the tool is that it is hard and sometimes even impossible to express certain useful policies as security automata over API method calls. For instance, a policy that limits the number of bytes transmitted over a network needs to monitor all API method calls that could lead to network traffic, and should be able to predict how much bytes of traffic the method will consume. In the presence of DNS lookups, redirects and so forth, this can be very hard.

A final limitation is that the policy languages supported by the tool are targeted to “expert” users. Writing a correct policy is much like a programming task. However, more user-friendly (e.g. graphical) policy languages could be compiled to Conspec or 2D-LTL.

References

- [1] Aktug, I. and K. Naliuka, *Conspec - a formal language for policy specification*, Electr. Notes Theor. Comput. Sci. **197** (2008), pp. 45–58.
- [2] Desmet, L., W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens and D. Vanoverberghe, *A flexible security architecture to support third-party applications on mobile devices*, in: *CSAW*, 2007, pp. 19–28.
- [3] Desmet, L., W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan and D. Vanoverberghe, *Security-by-contract on the .NET platform*, Inf. Secur. Tech. Rep. **13** (2008), pp. 25–32.
- [4] Edjlali, G., A. Acharya and V. Chaudhary, *History-based access control for mobile code*, in: *Proceedings of the 5th ACM conference on Computer and communications security*, 1998, pp. 38–40.
- [5] Massacci, F. and K. Naliuka, *Multi-session security monitoring for mobile code*, Technical Report DIT-06-067, UNITN (2006).
- [6] Schneider, F. B., *Enforceable security policies*, ACM Trans. Inf. Syst. Secur. **3** (2000), pp. 30–50.
- [7] Úlfar Erlingsson, “The inlined reference monitor approach to security policy enforcement,” Ph.D. thesis, Dep. of Computer Science, Cornell University (2004).
- [8] Vanoverberghe, D. and F. Piessens, *A caller-side inline reference monitor for an object-oriented intermediate language*, in: *FMOODS*, 2008, pp. 240–258.
- [9] Vanoverberghe, D. and F. Piessens, *Security enforcement aware software development*, Information and Software Technology (2009).
- [10] Zobel, A., C. Simoni, D. Piazza, X. Nuez and D. Rodriguez, *Business case and security requirements*, Public Deliverable of EU Research Project D5.1.1, S3MS- Security of Software and Services for Mobile Systems, Report available at www.s3ms.org (2006).