Poster: User Request as a means to Automate Authorization Hook Placement

Divya Muthukumaran
Pennsylvania State University
muthukum@cse.psu.edu

Trent Jaeger
Pennsylvania State University
tjaeger@cse.psu.edu

Vinod Ganapathy
Rutgers University
vinodg@cs.rutgers.edu

I. Abstract

We consider the problem of retrofitting legacy software with mechanisms for authorization policy enforcement. This is an important problem for operating systems, middleware and server applications (jointly, *servers*), which manage resources for and provide services to multiple, mutually-distrusting clients. Such servers must ensure that when a subject requests to perform a security-sensitive operation on an object, the operation is properly authorized. This goal is typically achieved by placing calls (termed *authorization hooks*) to a reference monitor [1] at suitable locations in the code of the server. At runtime, the invocation of a hook results in an authorization query that specifies the subject, object, and operation. The placement of authorization hooks must provide *complete mediation* of security-sensitive operations performed by the server. If this property is violated, subjects may be able to access objects even if they are not authorized to do so.

In the past decade, several efforts have attempted to place authorization hooks in a variety of servers. For example, discretionary access control mechanisms deployed in the Linux kernel were found to be insufficient to protect the security of hosts in a networked world. The Linux Security Modules (LSM) framework remedies this shortcoming by placing authorization hooks to enforce more powerful security policies. Even user-space servers can benefit from similar protection. For example, the X server manages windows and other objects for multiple clients. Accesses to such objects must be mediated, void of which several attacks are possible. The X server has also therefore been retrofitted with LSM-style authorization hooks [4]. Similar efforts now abound for other server applications (e.g., Apache, Postgres, Dbus, Gconf) [3], [5], [6], [2], operating systems, and virtual machine monitors.

Unfortunately, these efforts have been beset with problems. This is because the identification of security-sensitive operations and the placement of hooks is a manual procedure, largely driven by informal discussions on mailing lists in the developer community. There is no consensus on a formal definition of what constitutes a security-sensitive operation, and no tool support to identify their occurence in large code-bases. Not surprisingly, this *ad hoc* process has resulted in security holes, in some cases many years after hooks were deployed [8]. The discussion of which hooks to deploy can often last years, e.g., the original hook placement for the X server was proposed in 2003 [4], deployed in 2007 [9], and subsequent revisions have added additional hooks. What we therefore need is a principled way to identify security-sensitive operations and their occurrence in code, so that legacy servers can be automatically retrofitted with authorization hooks.

Prior work to address this problem has focused both on verification of authorization hook placement to ensure complete mediation and on mining security-sensitive operations in legacy code. The work on mining security-sensitive operations is the most closely related to work, and uses static and dynamic program analysis to identify possible hook placement locations. However, these mining techniques rely on domain-specific knowledge (e.g., a specification of the data types that denote security-sensitive objects), providing which still requires significant manual effort and a detailed understanding of the server's code base.

The main contribution of this work is a novel automated method for placing authorization hooks in server code that significantly reduces the burden on developers. The technique can identify optimal hook placements, in a manner that both minimizes the number of authorization hooks placed, as well as the number of authorization queries generated at runtime, while providing complete mediation. To develop the technique, we rely on a key observation that we gleaned by studying server code. In a server, clients make requests, which identifies the objects manipulated and the security-sensitive operations performed on them. *We observe that when a client makes a* ***deliberate choice*** *of an object from a collection of objects managed by the server, that automatically signals the need for authorization.* What security-sensitive operations are performed on the retrieved object(s) is determined by the code path that the server takes, which is also an upshot of the user's choice.

Based upon this observation, we design a static program analysis that tracks user choice to identify both security-sensitive objects and the operations that the server performs on them. Our analysis only requires a

specification of the statements from which client input may be obtained (e.g., socket reads), and a language-specific definition of object containers (e.g., arrays, lists), to generate a complete authorization hook placement. It uses context-sensitive, flow-insensitive data flow analysis to track how client input influences the selection of objects from containers: these are marked security-sensitive objects. The analysis also tracks how control flow decisions in code influence how the objects are manipulated: these manipulations are security-sensitive operations. The output of this analysis is a set of program locations where mediation is necessary. However, placing hooks at all these locations may be sub-optimal, both in terms of the number of hooks placed (e.g., a large number of hooks complicates code maintenance and understanding) and the number of authorization queries generated at runtime. We therefore use the control structure of the program to further optimize the placement of authorization hooks.

We have implemented a prototype tool that applies this method to C programs using analyses built on the CIL tool chain [7]. We have evaluated the tool on programs that have manual mediation for comparison, such as the X server and *postgresql*. We also evaluated the tool in terms of the minimization of programmer effort it entails and the accuracy of identification of objects and security sensitive operations in programs.

To summarize, our main contributions are:

- An approach to identifying security-sensitive objects and operations by leveraging a novel observation — that a deliberate choice by the client of an object from a collection managed by the server signals the need for mediation.
- The design and implementation of a static analysis tool that leverages the above observation to automate authorization hook placement in legacy server applications. This tool also identifies optimization opportunities, i.e., cases where hooks can be hoisted, thereby reducing the number of hooks in the source code, and by eliding redundant hook placements that would otherwise result in extra authorization queries at runtime.
- Evaluation with four significant server applications, namely, the X server, postgresql, PennMush, and memcached, demonstrating that our approach can significantly reduce the manual burden on developers in placing authorization hooks. In case of two of these servers, the X server and postgresql, there have been efforts spanning multiple years to place authorization hooks. We show that for these servers, our approach can automatically infer hook placements that are comparable to those placed manually with few false positives and negatives.

REFERENCES

[1] ANDERSON, J. P. Computer security technology planning study, volume II. Tech. Rep. ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, October 1972.

[2] CARTER, J. Using GConf as an Example of How to Create an Userspace Object Manager. *2007 SELinux Symposium* (2007).

[3] D.WALSH. Selinux/apache. http://fedoraproject.org/wiki/SELinux/apache.

[4] KILPATRICK, D., SALAMON, W., AND VANCE, C. Securing the X Window system with SELinux. Tech. Rep. 03-006, NAI Labs, March 2003.

[5] KOHEI, K. Security enhanced postgresql. SEPostgreSQLIntroduction.

[6] LOVE, R. Get on the D-BUS. http://www.linuxjournal.com/article/7744, Jan. 2005.

[7] NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction, 11th International Conference, CC 2002* (2002), Springer, pp. 213–228.

[8] TAN, L., ZHANG, X., MA, X., XIONG, W., AND ZHOU, Y. Autoises: automatically inferring security specifications and detecting violations. In *Proceedings of the 17th conference on Security symposium* (Berkeley, CA, USA, 2008), USENIX Association, pp. 379–394.

[9] WALSH, E. Integrating x.org with security-enhanced linux. In *Proceedings of the 2007 Security-Enhanced Linux Workshop* (Mar. 2007).