Cut Me Some Security!

Divya Muthukumaran
The Pennsylvania State University
SIIS Laboratory
CSE Department
University Park, PA, USA
muthukum@cse.psu.edu

Sandra Rueda
The Pennsylvania State University
SIIS Laboratory
CSE Department
University Park, PA, USA
ruedarod@cse.psu.edu

Hayawardh Vijayakumar
The Pennsylvania State University
SIIS Laboratory
CSE Department
University Park, PA, USA
huv101@psu.edu

Trent Jaeger
The Pennsylvania State University
SIIS Laboratory
CSE Department
University Park, PA, USA
tjaeger@cse.psu.edu

ABSTRACT

Computer security is currently fraught with fine-grained access control policies, in operating systems, applications and even programming languages. All this policy configuration means that too many decisions are left to administrators, developers and even users to some extent and as a result we do not get any comprehensive security guarantees. In this position paper, we take a stand for the idea that less policy is better and propose that limiting the choices given to parties along the development and deployment process leads to a more secure system. We argue that other systems processes like scheduling and memory management achieve their goals with minimal user input and access control configuration should also follow suit. We then suggest a technique to automate access control configuration using graph-cuts and show that this gets us closer to achieving our goal.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls, Information flow controls*

General Terms

Security

1. INTRODUCTION

Unlike most system mechanisms, computer security requires a tremendous amount of manual specification. From configuration files to access control policies to program input sanitization, the system administrators, OS distributors, application developers, and even users, in some (unfortunate) cases, have to make security decisions. This contrasts markedly with other systems mechanisms, such as scheduling, memory management, and I/O, where the system makes all the necessary decisions for the users with little or no input. We claim that until security becomes (near) "policy-less," security mechanisms will not be effective at thwarting attackers.

Typically, when a new component (system or program) is introduced, it is delivered with an initial security mechanism, often aimed at protection from faults (e.g., discretionary access control for operating systems and same-origin policy for browsers) rather than security against determined attackers. The policy models for these initial mechanisms are often simple to use, but these components are not secure. As a result, attackers and security researchers identify security flaws. Security practitioners then resort to deploying fine-grained enforcement mechanisms with associated fine-grained policies (e.g., SELinux for operating systems [19] and Chrome browser [3]).

The question in this paper is whether we can develop a fundamental approach that can eliminate the need for fine-grained policy specification. That is, can we identify a problem whose solution results in secure functionality with little or no guidance, as is the case in memory management? Or can we solve such a problem from a small number of relatively simple and verifiable decisions from expert parties, as in processor scheduling? With such a problem, we may be able to move away from the vicious cycle of finer-grained policy enforcement into a "policy-less" future.

So how do we enable programs to make correct security decisions without using fine-grained policies? In this position paper, we aim to create a model that parallels CPU scheduling as a guide to understand how to design future security mechanisms and the challenges that we face. CPU scheduling is an operating system mechanism that determines which process to run next, with the aim of achieving fairness. While it is an NP-complete problem in general, it is solved acceptably using a greedy algorithm with default priority inputs by observing process execution. Access control

SafeConfig'10, October 4, 2010, Chicago, Illinois, USA.

policy configuration is a similarly complex problem, with its need to balance function and security requirements, but we have not seen access control studied in the same way as scheduling, i.e., as an algorithm design problem. To study this question, we examine access control configuration as a graph-cut problem: an effective solution mediates all illegal information flows, which is equivalent to finding an appropriate cut in an information flow graph. We find potential in automating access control, but there are also some significant challenges. We hope that this paper motivates people to explore the option of automating access control configuration.

This paper is organized as follows. We examine how we created this policy management problem in Section 2. We then outline the position we take in this paper and sketch out a solution to the problem in Section 3. We propose a graph-cut approach for deploying systems in Section 4, and in Section 5, we assess the challenges in implementing that approach.

2. THE VICIOUS CYCLE OF POLICY

In this section, we describe why adding more policy is not an effective way to manage security.

2.1 Deploying Reference Monitors

Ten years ago, conventional operating systems distributors were finally convinced that discretionary access control was insufficient to protect a system from determined attackers. The operating systems security community had long known about the virtues of the reference monitor concept [5], an unbypassable reference validation mechanism that enforces a mandatory access control (MAC) policy. As a result, a reference validation mechanisms were added to Linux, FreeBSD, and ported to Mac OS X.

It was soon determined that security mechanisms in the OS alone were insufficient. Important access control decisions were also being made in user-space programs, such as the X window server [22], so access control mechanisms were added to these programs. Also, ad hoc access control mechanisms in browsers were being replaced with comprehensive reference monitors [3]. Concurrently, a number of independent movements had also been ongoing that added comprehensive access control to programming language runtimes, such as Java [12] and security-typed languages [17].

2.2 Making Access Control Decisions

Access control mechanisms enforce access control policies, and the major problem is now the need to express and manage all these policies. While UNIX discretionary access control model was never sufficient to express policies that would thwart determined attackers, the model is one that users and system administrators seem to understand. Information flow security models, such as Bell-LaPadula [7] and Biba integrity [8], are also logical for users, but while these models do provide strong guarantees against attackers, they fail to express the functional requirements of conventional systems. As a result, flexible access control models were adopted in many cases, such as Type Enforcement [9] by SELinux [19] and DTE [21]. We saw a similar evolution that led to fine-grained access control models for Java [12], Windows [20], and applications (browsers [3]), even where strong security guarantees are not enforced.

From our experience and that of others [1, 2, 4], we have learned that people find it hard to use these fine-grained policy models effectively to deploy secure systems. While SELinux can express a comprehensive, mandatory policy, people cannot use it effectively. SELinux policies released with Linux distributions enforce a *targeted* policy, where only network-facing daemons are confined (i.e., those that do not need to be fully trusted [10]), but other local processes are unconfined. This approach was derived from the AppArmor model [18], which focuses on a usable approach to prevent network attacks. The result is that the deployment of these approaches does not achieve strong security guarantees. This is also a similar case for Windows Mandatory Integrity Control model, which enforces Biba integrity, but only for writes, resulting in an incomplete model.

Our take away is that if people are given the flexibility to choose among function, usability, and security, they will choose the former two every time. Wurster and Van Oorschot discuss this problem in the context of application developers [23], proposing that developers only be given development choices that lead to a secure program. That is, developers, OS distributors, and system administrators should be given a small, coherent set of choices that enable them to provide their function in a secure manner.

But, it is not clear that we are converging on this goal. A recent proposal is an even more powerful model, Decentralized Information Flow Control [15, 25] (DIFC). This model is an information flow model based on the Decentralized Label Model [16, 11], which enables strong security guarantees (modulo management of delegated rights) while permitting function not authorized in classical models (e.g., controlled declassification of secrets and endorsement of low-integrity data). To achieve this, the DIFC model is even more expressive than previous models, representing each security requirement in the form of a tag and constructing labels from sets of tags. Thus, while we may be converging on what we need to express, we are not converging on how this can possibly be expressed and maintained. A completely new approach to managing security is necessary.

3. HOW CAN WE SIMPLIFY MANUAL EFFORT?

We feel that we need to take a cue from existing operating systems mechanisms, in particular CPU scheduling, to solve the access control configuration problem. The CPU scheduling mechanism has the following stages:

- First, it *models* the scheduling state of all processes, such as a set of queues in a multi-queue scheduler.
- The CPU scheduler then *inputs* high-level configuration parameters, such as priorities, and runtime process behavior, such as time slices, to construct an instance of the model for that system.
- The scheduler *solves* the scheduling problem by choosing the next process to schedule and by updating the model instance.
- Finally, the CPU scheduler *enacts* the scheduling decision by scheduling the selected process and updating queues.

We envision that a system mechanism to configure access control would have to implement each of these steps.

We need to model the function of processes, gather system-specific inputs regarding access, measure the system as it runs, solve the problem of determining what accesses should be allowed, and enact those access decisions. To study options for solving the access control configuration problem, we examine it as a graph-cut problem. We recently modeled the problem of placing mediation statements (e.g., declassifiers and runtime checks) in legacy code [14] as a graph-cut problem. We found that configuring access control in general is the problem of finding the "right" cut in the graph to block attackers while at the same time permitting necessary function. Our study shows that there are both benefits and challenges in approaching access control configuration as a graph cut problem.

In this discussion, we envision a graph-cut configuration mechanism could work as follows. First, default function of the system (in terms of interaction between processes) would be shown as an information flow graph. As a community, we have a pretty good idea how processes interact, so we believe that such a graph could be collected from runtime experience. It need not be a perfect representation of system function, as a mechanism must be robust to changes. Second, the security requirements of the data added to the system would be specified as *imports* and *exports*. Third, we compute a graph-cut that mediates all illegal accesses while preserving necessary function. Mediation may occur through a variety of means, including the blocking of flows, runtime mediation of flows (e.g., sanitization), etc. Thus, the choice of how to enact the cut is left open. We envision that a set of possible options will be collected over time to enable function while protecting the system from security risks, which the cut aims to mitigate.

4. WHY GRAPH-CUTS?

In graph theory, given a graph *G=(V,E)*, a *vertex cut* of this graph with respect to a source and a sink is a set of vertices whose removal will divide the graph into two parts, one containing the source and the other containing the sink, such that the sink can no longer be reached from the source. In *mediation placement* our goal is to find points in the program where mediators such as declassifiers and endorsers can be placed to prevent illegal information flows. We showed in previous work [14] that this was tantamount to obtaining a vertex cut of the information flow graph of the program with respect to the offending sources and sinks of information. We developed a tool to convert source code of programs into information flow graphs, generate a vertex cut of that graph and return the corresponding expressions as complete and minimal positions for placing mediation statements such that all illegal information flows in the system with eliminated. Our tool currently works on both C and Java programs and cuts can be generated for 20K SLOC programs in less than 90 seconds, and selected placements correspond to manual placements 80% of the time. The graph-cut tool can be applied to any problem that can be represented as an information flow graph.

We identified four basic steps to creating and solving a graph-cut problem.

1. *Building a model:* We need to build an information flow graph of the entire system showing the interaction between different processes. For individual programs, we can build the information flow graph from a static analysis of the source code. For system wide information flows we need to monitor the system usage to gather flows and any auditing utility can help with this. For example, we can use a utility like SELinux *audit2allow* which reads logs to identify accesses that were denied by the policy in order to build a policy that allows necessary functionality.

2. *Gathering inputs: Imports* and *Exports* are used to mark security sensitive data that enter and leave the system respectively and represent the sources and sinks in the graph-cut problem. Users need to specify what the security requirements of *imports* are with respect to *exports*, for example, *imports* labeled *password* should not reach exports labeled *public* without declassification. Initially, *imports* and *exports* need to be specified just like scheduling priorities, but over time they may become well-known. For example, it is well known that web-servers receive untrusted input.

3. *Solving the problem:* We compute a vertex-cut of the information flow graph with the *imports* as sources and *exports* as sinks. This solution provides completeness by construction (i.e., the cuts suggested cover all illegal flows in the program). Generating a min-cut will also guarantee a minimal solution.

4. *Enacting the cut:* Once we have the cut, we need to build a security solution that resolves the security error as per the security requirement. Typically, we would need to make programs declassifiers or endorsers by changing the source code. For this, we envision having a template of filters to choose from. For example, for *imports* labeled *password* that need to be kept secret, the tool can automatically pick an encryption function from the template to insert at the cut location. Whenever new security requirements are added, corresponding filters are added to the template library.

5. CUT-PROBLEM CHALLENGES

The following challenges must be addressed to generate access enforcement from graph-cuts.

- **Sources and sinks**: Ensure that all the security-sensitive sources and sinks are identified with little programmer input.
- **Cut-conjunction**: There may be multiple cut problems to solve per component, so we need to merge the solutions.
- **Mediators**: Collect and evaluate the cost of cut options to determine the weights for options.

First, while people can probably identify the key external sources for their programs and systems, we may also need to identify sources and sinks within the program. For example, the Saner project identified that untrusted inputs may need to be sanitized multiple times for different purposes [6], so we need to automate the finding of sinks with different requirements within a program. Second, the graph-cut problem where there are multiple sources and sinks is called the *cut-conjunction problem*, and this problem has unknown computational complexity [13]. We have developed a simple greedy algorithms, but more exploration is

necessary. Third, as described above, a library of mediators will be necessary to automate sanitization, raising two problems: (1) how to apply sanitization methods automatically and (2) how to evaluate the cost of such methods for computing cuts. The Saner system shows one method to model known sanitizers [6], and we envision that costs may be estimated by the complexity of the filtering or declassification rules, analogous to information flow assertions in RESIN [24].

6. CONCLUSION

In this position paper, we argue that people cannot use fine-grained policy models effectively to deploy secure systems. We propose that users should only be provided with a small, coherent set of choices that lead to a secure system. We show how we can take a cue from other system processes such as CPU scheduling to automate the solution to this problem. We model this access control configuration as a graph-cut problem and show that solving the "right" cut problem can take us closer to our goal.

7. REFERENCES

[1] Comments on the Content Security Policy specification. `http://www.mail-archive.com/dev-security@lists.mozilla.org/msg01530.html`.

[2] Do you disable SELinux? `http://stackoverflow.com/questions/97816/do-you-disable-selinux`.

[3] Google Chrome. `http://www.google.com/chrome/intl/en/features.html`.

[4] SELinux may cause mysterious permission problems. `http://drupal.org/node/50280`.

[5] J. P. Anderson. Computer Security Technology Planning Study, Volume II. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), October 1972.

[6] D. Balzarotti *et al.* Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symp. on Security and Privacy*, 2008.

[7] D. E. Bell and L. J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report ESD-TR-75-306, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), March 1976.

[8] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, MITRE, April 1977.

[9] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th NCSC*, 1985.

[10] H. Chen, N. Li, and Z. Mao. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *Proceedings of NDSS '09*, 2009.

[11] D. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5), 1976.

[12] L. Gong *et al. Inside Java 2 platform security architecture, API design, and implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[13] L. Khachiyan *et al.* Enumerating disjunctions and conjunctions of paths and cuts in reliability theory. *Discrete Appl. Math.*, 155(2):137–149, 2007.

[14] D. King *et al.* Automating security mediation placement. In *Proceedings of ESOP '10*, pages 327–344, 2010.

[15] M. N. Krohn *et al.* Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM SOSP*, Oct. 2007.

[16] A. C. Myers and B. Liskov. A decentralized model for information flow control. *ACM Operating Systems Review*, 31(5), Oct. 1997.

[17] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. `http://www.cs.cornell.edu/jif,July2001-2003`.

[18] Novell. AppArmor Linux Application Security. `http://www.novell.com/linux/security/apparmor/`.

[19] Security-Enhanced Linux. `http://www.nsa.gov/selinux`.

[20] M. M. Swift *et al.* Improving the granularity of access control for windows 2000. *ACM Trans. Inf. Syst. Secur.*, 5(4):398–437, 2002.

[21] K. M. Walker *et al.* Confining root programs with domain and type enforcement (DTE). In *Proceedings of the 6th USENIX Security Symp.*, 1996.

[22] E. Walsh. Application of the flask architecture to the x window system server. In *Proceedings of the 2007 SELinux Symposium*, 2007.

[23] G. Wurster and P.C. van Oorschot. The developer is the enemy. In *Proceedings of NSPW '08*, 2008.

[24] A. Yip *et al.* Improving application security with data flow assertions. In *SOSP '09*, pages 291–304, 2009.

[25] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the* 7^{th} *OSDI*, 2006.