Title

Has Access Control Become the Weak Link?

Permalink
https://escholarship.org/uc/item/1bw6970f

Journal
IEEE Security & Privacy, 22(5)

ISSN
1540-7993

Author
Jaeger, Trent

Publication Date
2024

DOI
10.1109/msec.2024.3427588

Trent Jaeger
The University of California, Riverside

Has Access Control Become the Weak Link?

In the early 2000s, computer systems were under threat from a variety of Internet worms. This malware attacked network-facing programs by exploiting their memory errors, hijacking their execution to perform malicious operations and propagate the malware to other systems. One key change that commercial systems adopted to prevent such attacks was in access control enforcement. With the enhanced access control (and other defenses), defenders were able to prevent Internet worm attacks, but a variety of other significant and catastrophic attacks (e.g., ransomware) have emerged since that time. But our access control infrastructure is essentially the same as that used to combat Internet worms. In this column, I want to look more closely at the current state of access control enforcement and where we might go from here.

A little review of the situation in the early 2000s. At this time, commercial systems (e.g., UNIX-based systems and Windows) did not yet take security threats seriously. Commercial systems controlled access to their resources (e.g., files) using what is known as *discretionary access control* (DAC). DAC enforcement has two key limitations that allowed Internet worm attacks.

First, DAC authorizes each process's request to access a system resource based on a user identity associated with the process (i.e., user ID). The idea is that users run processes and should get access to their own files. However, system processes were not associated with a specific user and can act on behalf of any user. As a result, DAC systems traditionally assigned an all-powerful user (e.g., root or administrator) as the user identity for system processes. These all-powerful system processes are authorized to perform any operation.

Second, DAC also allows a process to administer some the DAC permissions. Typically, a process can modify the DAC permissions of resources it is said to *own*, which are often resources created by a process with that user identity (i.e., could be another process with that user identity). However, researchers found that by allowing processes to manage permissions, it is impractical to determine whether a particular security policy will ensure that a security requirement (e.g., preventing a process from obtaining a particular permission) in typical DAC administration (i.e., the undecideability result for the *safety problem*[4]) is enforced. Methods to ensure access control changes comply with security properties, such as the take-grant model[7] and access control constraints,[12] were not adopted.

The combination of these two limitations of DAC enforcement were key factors in permitting Internet worm attacks to succeed. On the one hand, systems of the time had many network-facing system processes, which all ran under an all-powerful user identity. Thus, if any one of these programs had a vulnerability, the associated host was effectively compromised. Such vulnerabilities were plentiful then and remain an issue today, so something had to be done reduce the permissions assigned to network-facing system processes. However, even after reducing the permissions associated with a process, DAC administration makes it impractical to determine whether we can confine a hijacked process to prevent host compromise in general. An adversary could direct a network-facing process they have hijacked to change the permissions to resources that it owns and create new resources to plant attacks (more about these attacks next) to attack other system processes. One must use DAC very carefully to

Digital Object Identifier 10.1109/MSEC.2024.3427588
Date of current version: 13 September 2024

avoid such attacks, since programs are typically not hardened for such threats comprehensively.

Commercial systems could not prevent Internet worms reliably with their current DAC enforcement.

Using Access Control to Prevent Internet Worm Attacks

Fortunately, there had been several key findings in access control research that could be drawn upon to prevent Internet worm attacks in commercial systems by this time. First, researchers advocated restricting processes to just the permissions they need to perform their functionality, often called *least privilege.*[11] By applying least privilege, a compromised system process may lack permissions needed to compromise the host (e.g., install a rootkit). However, enforcing least privilege alone cannot ensure that the host is protected from a compromised process, as the permissions needed for functionality may still be sufficient to compromise the host, directly or indirectly.

Alternatively, researchers proposed an access control approach that ensures the enforcement of security properties (e.g., secrecy or integrity), where processes are restricted only to operations that comply with those security properties. This approach, which is commonly called *multilevel security*[1] although it is more generally expressed as lattice-based access control,[3] limits processes to operations that only use data that it is authorized to see (i.e., at its "level" or lower) and that cannot leak any data to less privileged processes (i.e., at a lower level). Comparing the two approaches, least privilege aims to restrict processes to only necessary permissions, hopefully blocking operations that may violate security properties, while multilevel security aims to restrict functionality within operations that must satisfy security properties.

Another key innovation used in preventing Internet worm attacks is the development of system-administered access control policies, called *mandatory access control* (MAC). MAC polices cannot be modified by any of the user processes or system processes, excepting those expressly permitted to administer the system's security properties. Multilevel security policies are mandatory by design as the system defines the security properties that must be followed, so they cannot be changed by normal processing. On the other hand, since least privilege aims to allow all functionality necessary, least privilege policies may be either discretionary or mandatory. Researchers devised methods for MAC enforcement of least privilege, such as *type enforcement.*[2]

After the emergence of Internet worm attacks, researchers proposed that improvements in access control could prevent such attacks.[10] While multilevel security offers the potential to prevent attacks entirely, this approach often prevents operations necessary for implementing common functionality, such as bidirectional communication between processes at different levels (i.e., with different security properties). As a result, commercial systems adopted the least-privilege approach, and Linux adopted MAC enforcement of least-privilege policies, with key examples being SELinux and AppArmor. Access control policies were designed to confine system processes, especially network-facing processes, to the permissions deemed necessary for them to function only (e.g., found by dynamic analysis). While early policies often led to unexpected denials (e.g., preventing system boot in some configurations), eventually system distributors expanded policies sufficiently to make them practical.

Access Control Policy Analysis

A challenge with applying least privilege access control is that we

Digital Object Identifier 10.1109/MSEC.2024.3441458

do not know which authorized operations may permit attacks, such as attacks on secrecy and integrity prevented by multilevel security. To answer the question of whether least privilege access control still permits attacks, access control policy analysis was proposed. I want to briefly explain what access control policy analysis is and what it is for before we examine where we are now.

In general, access control policy analysis[6] is analogous to program analysis, where we take the artifact (i.e., the access control policy) and compute a data-flow graph $G = (V, E)$ implied by the authorized operations. Since MAC policies are immutable, this data-flow graph represents all the possible authorized flows. In this case, the data-flow graph describes the flows among subjects S and objects O identified by the policy, where $V = S \cup O$. For access control analysis, a data flow (i.e., an edge $(u, v) \in E$, where $u, v \in V$) is created in two ways: 1) when the access control policy authorizes a read operation on object o by subject s an edge (o, s) (i.e., a flow from the object to the subject) is added to the data flow graph and 2) when the access control policy authorizes a write operation on object o by subject s an edge (s, o) (i.e., a flow from the subject to the object) is added to the data flow graph. All system calls can be classified based on whether their effects cause reads, writes, or both.

Access control policy analysis compares the data flows resulting from an access control policy to a security policy representing the operations considered secure. The idea is to identify any authorized operation that may enable an adversary to attack a program, e.g., causing a data leak (violate secrecy) or illicitly modifying critical program data (violate integrity). This is exactly the set of operations that are deemed insecure in a multilevel security policy, so access control policy analysis often uses a multilevel policy to find the unsafe operations authorized by a least privilege policy. While commercial systems do not use multilevel policies, they sometimes assign security levels to processes to limit permissions, such as Android's privilege levels and Windows' Mandatory Integrity Control.

Where Are We Now?

Twenty-plus years after the introduction of least privilege MAC enforcement in commercial systems, we continue to apply that approach. However, adversaries have become proficient in employing attacks that exploit weaknesses in least-privilege enforcement. Most of these attack types have been known for a long time (i.e., prior to Internet worms), but have become a common part of the adversary's arsenal. The prevalence and effectiveness of such attacks means that additional defenses are required.

Consider mobile systems, such as Android, which allow the use of third-party applications. As adversaries may entice mobile users into running their malicious apps, a challenge for mobile systems is to prevent attacks even when adversary-controlled code may run on the system. Mobile systems change the threat model from trying to confine network-facing processes to also confining third-party apps. This problem is complicated by the prevalence of Android original equipment manufacturers (OEMs) who extend their versions of Android systems in a variety of ways to provide value-added functionality for their customers, including functionality for use by third-party apps.

There are several threat vectors that must be considered to protect Android (and OEM) programs from exploitation, but let's examine one. Third-party apps often share the filesystem with Android programs, which may put the latter at risk should they use a file or symbolic link created by a malicious third-party app. For example, malicious apps can plant files to control Android programs (e.g., library code, updates, and configurations) or plant symbolic links to redirect Android programs to operate on adversary-chosen resources. Whenever Android programs share resources that may be modified by third-party apps, this creates threats to Android security.

By using access control policy analysis, one can identify such threats from the combination of Android access control policies.[8,9] While the presence of a threatening data flow (e.g., from a third-party app to an Android program) does not directly mean that a program has a vulnerability, many vulnerabilities to these kinds of threats have been detected recently. This comports with my personal experience that programs often fail to address such threats, as they underestimate the amount of authorized sharing resulting from least-privilege MAC enforcement. Note that Android systems have extensive, fine-grained access control, so these threats could be even more numerous on other systems.

Why do these types of vulnerabilities continue to emerge? One issue is that least-privilege access control is hard to get perfect. If a program is denied any permission that it really needs, then the program will fail to operate correctly, leading to functionality problems. As a result, least-privilege access control must overapproximate the permissions needed by each program, resulting in a significant number of threats (e.g., over 1,000 for Android OEMs[9]) Another issue is that configuring least-privilege access control is hard. For example, we observe that many Android OEMs utilize the Linux DAC to try to prevent sharing rather than the MAC enforcement, reintroducing its risks. As a result, the responsibility for preventing threats from turning into vulnerabilities falls upon programmers. Although Linux application programming interfaces have been extended to prevent some vulnerabilities (e.g., `openat`), the complexity and variety of threats that programmers face makes it difficult to address them all.

Where Do We Go From Here?

Given the current situation, the question is what actions should be taken to reduce number of vulnerabilities resulting from least-privilege MAC enforcement. My hope in developing access control analysis was that operating systems distributors would use such methods to detect threats and make changes to reduce the number of threats, e.g., deploy program configurations that create fewer threats. However, access control analysis identifies threats rather than vulnerabilities, so these threats are not always removed. As a result, it has largely fallen upon programmers to defend against such threats, but many of these threats are not addressed sufficiently.

It appears unlikely that improvements in access control alone will be sufficient to prevent these exploits. As we saw for Android, refining least-privilege MAC enforcement is already too complex for many OEMs. In addition, programs may really use many of these permissions that are under threat, so adopting a more restrictive policy or a policy that prevents all security violations (e.g., multilevel security), would prevent many operations deemed necessary in commercial programs. In practical terms, we are probably committed to some form of least-privilege access control, but how do we make it more effective?

While access control policy analysis alone does not identify vulnerabilities directly, we envision that some combination of program analysis and access control policy analysis should be considered. Joint program and access control analysis has been proposed in the past to check whether the information flows of the program were compliant with information flows authorized by SELinux access control.[5] To make access control enforcement more robust, we could envision automating analysis of whether the threats found by access control analysis could actually cause program vulnerabilities. Challenges remain to determine how to the two types of analyses could be combined effectively. In addition, to make impact on the security of systems, we need to improve the ease of responding to problems when they are identified.

The bottom line is that the least-privilege MAC approach to access control has limitations that need to be addressed to improve our security posture beyond the current state in tangible ways. It appears that efforts from multiple communities (e.g., program analysis, access control, filesystems, and so on) will need to be brought together to prevent such attacks. ■

References

1. D. E. Bell and L. J. LaPadula, "Secure computer system: Unified exposition and Multics interpretation," MITRE Corp., Bedford, MA, USA, Tech. Rep. MTR-2997, Mar. 1976.
2. W. E. Boebert and R. Y. Kain, "A practical alternative to hierarchical integrity policies," in *Proc. 8th Nat. Comput. Secur. Conf.*, 1985, pp. 18–27.
3. D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976, doi: 10.1145/360051.360056.
4. M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Commun. ACM*, vol. 19, no. 8, pp. 461–471, 1976, doi: 10.1145/360303.360333.
5. B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel, "From Trusted to Secure: Building and executing applications that enforce systems security," in *Proc. USENIX Annu. Tech. Conf.*, May 2007, pp. 205–218.
6. T. Jaeger, R. Sailer, and X. Zhang, "Analyzing integrity protection in the SELinux example policy," in *Proc. 11th USENIX Secur. Symp.*, Aug. 2003, pp. 59–74.
7. A. K. Jones, R. J. Lipton, and L. Snyder, "A linear time algorithm for deciding security," in *Proc. 17th Annu. Symp. Found. Comput. Sci.*, Houston, TX, USA, 1976, pp. 33–41, doi: 10.1109/SFCS.1976.1.
8. Y.-T. Lee et al., "PolyScope: Multi-policy access control analysis to triage Android Scoped Storage," *IEEE Trans. Dependable Secure Comput.*, early access, Sep. 2023, doi: 10.1109/TDSC.2023.3310402.
9. Y.-T. Lee et al., "PolyScope: Multi-policy access control analysis to compute authorized attack operations in Android systems," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 2579–2596.
10. P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J.Turner, and J. F. Farrell, "The inevitability of failure: The flawed assumption of security in modern computing environments," in *Proc. 21st Nat. Inf. Syst. Secur. Conf.*, Oct. 1998, pp. 303–314.
11. J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proc. IEEE*, vol. 63, no. 9, pp. 1278–1308, Sep. 1975, doi: 10.1109/PROC.1975.9939.
12. J. Tidswell and T. Jaeger, "An access control model for simplifying constraint expression," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2000, pp. 154–163, doi: 10.1145/352600.352622.