Making Trust Explicit in XOS

Mridu Nanda

Abstract

Modern systems place implicit trust in applications, by granting them the maximum possible access to system resources. Existing mechanisms to enforce the principle of least privilege (PoLP) are: (1) user-centric [18], (2) error-prone [6], and (3) incompatible with traditional APIs [7, 19]. As a result, desktop and server systems are plagued with over-privileged applications that are susceptible to confused deputy attack and malware. In this paper, we present XOS, a speculative operating system that enforces the PoLP at the application-level. XOS allows a process to access resources proportional to the trustworthiness of the process. XOS uses inspiration from human interactions to quantify trust: just like a dependable friend is more trustworthy than an erratic friend, XOS trusts a process that behaves in an expected manner more than a process that behaves in an unpredictable manner. By making trust explicit, XOS encourages developers to write security-conscious applications and limits potential damage from insecure applications.

1 Introduction

Applications running on commodity operating systems get access to system resources by virtue of installation. For example, a Microsoft Word application gets de facto access to memory, disk space, display, and CPU time upon installation. Granting applications unvetted access to system resources violates the *principle of least privilege* (PoLP), which asserts that an application should use the minimal set of resources needed to perform its task [17]. Consequently, desktop and server systems are plagued with over-privileged applications that are susceptible to confused deputy attacks and malware.

Discretionary access controls (DACs) limit how applications can use system resources. The DAC model allows a principal, like a user, to specify an access control list (ACL) for an object she created, like a file [18]. When a user installs an application, the application's processes are bound to the user's ID. Therefore, the application's processes can only access system resources with an ACL that mentions the user. For example, if Alice installs a Microsoft Word application, the corresponding processes can only manipulate resources with an ACL that mentions Alice.

Unfortunately, DACs do not support the PoLP:

- *DACs do not set permissions for all system resources.* While DACs traditionally limit file access, DACs do not restrict access to other system resources, like memory or CPU time.
- *DACs are often configured incorrectly.* Configuring permissions is a painful task, and DACs place this burden on the end-user. Users often grant more permissions than strictly necessary, resulting in over-privileged applications [14].
- *DACs assign trust to users, not processes.* However, by trusting users, DACs also implicitly trust the applications that users install [6]. This clearly a problem, since a DAC cannot prevent a user from installing an insecure application. Thus, a malicious process may be granted unfettered access to system resources.

As a result, systems under the DAC model often give applications far more trust than is strictly necessary.

Given the explosion of technical education [10, 13], users are mostly able to avoid installing malicious applications. Unfortunately, it is still common for users to install well-intentioned, insecure applications. Therefore, we require new security primitives that limit the damage an insecure application can inflict upon the operating system. The operating system is the best place to explore such abstractions because it provides fine-grain control over system resources. Furthermore, operating system-based access controls can assign trust to processes directly. Such a system is not susceptible to a lazy sysadmin, who grants users unnecessary access to sensitive resources.

In this paper, we present XOS, a speculative operating system that provides strong access control guarantees. XOS limits the carnage from exploiting an insecure application by granting processes access to system resources on a least privilege basis. In particular, XOS assigns each process a trust value, which determines which system resources the process can access. A process with a high trust value will be allowed to use more system resources than a process with a low trust value. For example, XOS will schedule a process with a high trust value more frequently than a process with a low trust value. By making trust explicit, XOS aims to secure the operating system environment.

XOS uses inspiration from human behavior to quantify trust. While typical systems use binary values of trust, XOS takes a more natural approach and considers a spectrum of trust. Earning trust is difficult. At installation time, an application must present XOS with a policy that declares all the system resources it will access. XOS checks this policy against some known good values, to determine a base trust value for the application's processes. During run-time, XOS's security monitor verifies that a process's actions are consistent with its policy. XOS rewards a process that behaves in an expected manner by incrementing its trust value. Suspicious behavior flagged by the security monitor can cause XOS to decrement a process's trust value. Processes with higher trust values can vouch for processes with lower trust

values; however, if the vouchee violates the system's trust, XOS decrements both processes' trust values.

The rest of this paper is organized as follows. In section 2, we explore previous attempts to enforce PoLP, and how these approaches relate to XOS. Next, we give a high-level overview of XOS's design, which includes a policy engine (Section 3.3), security monitor (Section 3.4), and ML engine (Section 3.5). We conclude with a discussion of XOS's limitations and future work (Section 4).

2 Related Work

In this section we describe previous attempts to provide systems-level support for the PoLP.

2.1 Mandatory Access Controls

Mandatory access control (MAC) is a hierarchical access control mechanism that uses labels to enforce security policies [2]. Roughly speaking, every object in the system is assigned a security level (such as `unclassified`, `secret`, `top secret`). To read an object, a subject must possess a security level at least as high as that of the object. For example, a process with `top secret` security clearance can read a `secret` file. However, a subject cannot write to files at lower security levels. From the previous example, the `top secret` process should not read the `secret` file and write its contents to an `unclassified` file, since this will lead to information leakage.

Many operating systems implement MACs. In SELinux, an administrator configures a MAC policy for the machine, thus limiting the attack surface of an exploited system [1]. SELinux is based on Unix, which favors a subtractive permission model (e.g., permissions are removed from already privileged processes) [12]. Consequently, an administrator writing a SELinux policy must define all the permitted behaviors of every application on the system. This process leads to large and unwieldy policies, and places an enormous burden on the administrator.

Asbestos and Eros are two more examples of MAC operating systems. Asbestos supports decentralized MAC by combining aspects of capabilities and information flow policies, thus avoiding the high administrative costs associated with SELinux [7]. Eros is a pure capability-based microkernel [19]. Both of these systems require applications to be ported to fundamentally different application interfaces. In contrast, XOS preserves the POSIX interface.

2.2 Client Side Systems

XOS is inspired by client platform systems that treat applications as distinct untrusted principals [3, 4, 16]. By assigning permissions per-application, client platforms limit each application's access to user-owned resources. For example, mobile devices restrict an application's access to devices such as the camera and the GPS. There are many existing implementations of client-side, per-application, permission granting systems.

Manifests. Android devices require each application to store a manifest file. This file defines the metadata and structure of the application. The file also lists all permissions the application will require to access sensitive user data (such as contacts and SMS) and system features (such as the camera and internet access) [3]. Unfortunately, manifests alone are insufficient for enforcing the PoLP. Studies have shown that application developers often list more permissions than strictly necessary, leading to malware outbreaks [8].

Regardless, we argue that manifests are an important first line of defense against over-privileged applications. XOS adopts Android's permissions manifest by requiring each application to submit a policy that lists all the system resources the application will use. XOS enhances the traditional manifest model by also rating the trustworthiness of the provided policy. By restricting each application's access to the resources listed in its policy, XOS mitigates the effects of an application compromise.

Prompts. By contrast, iOS devices prompt users the first time an application tries to access a sensitive resource [4]. In theory, prompts verify user intent. In practice, prompts are ineffective because users become desensitized to repetitive notifications, granting more permissions than necessary [11, 14]. XOS relieves users from reasoning about application-level permissions.

3 Design

The XOS system assigns an explicit trust value to each process. A process's trust value determines which system resources it can access and which operations it can perform. System resources include memory, disk, file descriptors, ports, and CPU time. Sensitive operations include IPC and certain system calls (such as `mprotect`). The trust abstraction is supported by three components: the *policy engine*, the *security monitor*, and the *ML engine*:

- The *policy engine* accepts a policy, presented by the application at installation time, to determine the base trust value of the application's processes. The policy outlines the system resources and system calls the application will utilize. The application developer can either write the policy herself, or employ a user-level library to automatically generate the policy. The policy engine sends its analysis to the ML engine.
- The *security monitor* intercepts every trap to the operating system, and consults the application's policy before allowing the operation. For example, it will examine the arguments of the `connect` system call, and verify these arguments against the application's policy. The security monitor can generate a trust notification

to the user. A trust notification alerts a user of (1) a process that violates the application's policy (see section 3.4), (2) a process that cannot make forward progress because of a low trust value (see section 3.2), or (3) a process that is infected by "low laying" malware (see section 3.5). It also generates a provenance graph per process, to track the causal relations between system level objects. This provenance graph is stored in the ML engine.

- The *ML engine* takes the application's policy, the analysis generated by the policy engine, and the provenance graphs generated by the security monitor as inputs. It uses these inputs to get a fine-grain picture of a process's activity. The ML engine leverages this information to detect advanced persistent threats (APTs). In the case of an intruder, the ML engine can notify the security monitor, which can raise a trust notification to the application. The ML engine can also alert the policy engine, so that a future process with a similar policy can be assigned a low trust value.

As shown in Figure 1, all three components can update a process's trust value.

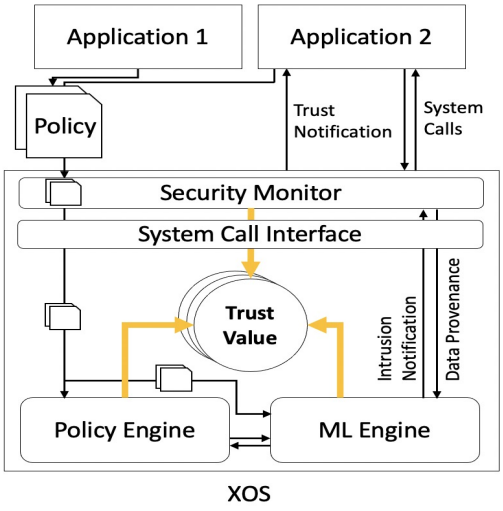

Figure 1. XOS's architecture. The security monitor, policy engine, and ML engine receive copies of each application's policy. The security monitor intercepts every system call and checks if the application is in accordance with its policy. The security monitor can send trust notifications in case of a policy violation or trust deadlock. The security monitor also generates data provenance. The ML engine accepts data provenance and the policy engine's analysis to help detect low-laying malware. The ML engine can send notifications to the security monitor and can alert the policy engine to suspicious policies. The policy engine, security monitor, and ML engine can update every process's trust value.

3.1 Threat model

The primary principals in our threat model are XOS, the applications, and the users. If XOS is installed on a server, then client-side code and data center operators are also principals. XOS enforces the PoLP by requiring applications to earn the operating system's trust.

We consider users, who have accounts on the machine running XOS, to be semi-trusted. The machine's administrator is responsible for assigning permissions to users through DACs. DACs are orthogonal to the access controls enforced by XOS; while DACs assign trust to users, XOS assigns trust to processes. We assume that users will be savvy enough to avoid installing malicious applications. We also assume users are not actively attempting to subvert XOS; this precludes a user from exploiting a side channel to extract information from XOS, beyond what should be allowed by the process's trust value. However, XOS does not rely on users to enforce the PoLP; regardless of Alice's permissions, XOS will ensure that any application Alice installs can only access system resources proportional to the application's trust level.

At installation time, XOS assumes that applications are untrusted. By analyzing an application's policy at installation time, the policy engine can starve an overtly insecure application by assigning it a low base trust value. For example, the policy engine will set a low base trust value to an application that lists `evil.com` in its policy. XOS's security monitor will raise a trust notification if an application uses resources not declared in its policy. Therefore, the security monitor defends against a careless application that presents an incorrect policy to the policy engine. The security monitor also defends against an insecure application infected by malware; the malware might use resources that the application did not declare in its policy.

A key challenge in XOS is assigning trust to insecure applications susceptible to "low laying" malware. Consider an insecure application that presents a reasonable policy to the policy engine and gains a baseline trust value. Suppose malware attacks the application; however, this malware avoids detection from the security monitor by using resources already declared in the policy. Instances of such malware have grown increasingly common. For example, in an advanced persistent threat (APT), an attacker tries to gain control of a system while remaining undetected for a long time [5]. XOS's ML engine makes use of provenance-based anomaly detection to defend against this style of attack [9].

In the case of a distributed system, we assume that client-side code will forward the user's request to the server running XOS. A data center operator has physical access to servers, which enables direct manipulation of server RAM. Therefore, our current design assumes that data center operators are trusted.

3.2 Trust Value

A trustworthy application will:

1. Do the right thing
2. Do what it says it will do

Trust values assigned by the policy engine and ML engine approximate the first metric, while the trust values assigned by the security monitor approximate the second metric.

An application's policy sets an upper bound on per-process system resource consumption. However, a new process will not be able to access all of the listed resources immediately. Instead, each process will only be able to access resources proportional to its trust value. XOS stores an internal mapping between resources and trust values, called a *trust reference*. XOS has a default global trust reference; however, the administrator can override this trust reference with a customized global trust reference, or customized per-application trust references.

Consider a text editor application that provides the policy in Figure 2. Suppose the policy engine deems the policy to be untrustworthy, and assigns a low base trust value to the application's processes (see Section 3.3). In this case, a process will be disallowed from using port 80 immediately after installation: even though the process did not violate the system's trust (e.g., port 80 is in the application's policy), the process did not earn enough trust to access ports, according to the default trust reference.

A process can experience *trust deadlock* if it has a low trust value and cannot make forward progress due to its trust level. Suppose the text editor's process cannot make forward progress without using port 80. In this case, the process will neither have enough trust to use port 80, nor be able to earn more trust due to its dependence on port 80, resulting in a deadlock. The security monitor sends a trust notification to alert the user of a trust deadlock.

There are two ways to resolve trust deadlock. One option involves the application to revising its policy to earn a higher base trust level. This approach encourages developers to write security-oriented applications. Unfortunately, this approach also requires user intervention. In particular, a user must uninstall, download, and re-install the application so that the new policy can be processed by the policy engine. Another way to resolve deadlock involves a trusted application vouching (e.g., transferring some of its trust) to the deadlocked application (see section 3.6).

3.3 Policy Engine

During installation, an application must present a policy to XOS. The policy engine examines the policy to assign a base trust value to the application. An example policy for a text editor application might look like Figure 2.

The policy engine uses three metrics to rank the trustworthiness of a policy.

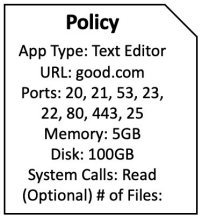

Figure 2. An example policy for a text editor application

- **Whitelists/Blacklists**: The policy engine checks some values against known good values and known bad values, which are stored in whitelists and blacklists, respectively. The machine administrator is the only user that can modify these lists. When examining Figure 2, the policy engine might find `good.com` on a whitelist, increasing the policy's trustworthiness.
- **The thorough developer**: An application developer can configure as many, or as few, of the optional policy fields as she likes. For example, the policy in figure 2 does not place an upper bound on the number of files the application will access. However, the policy engine rewards a thorough developer by assigning her application a high trust value. Detailed policies help the security monitor flag anomalous behavior caused by malware. This metric also encourages good program hygiene by forcing the developer to reason about resource consumption.
- **Similar Applications**: The policy engine uses the `App Type` field to compare the policy to an existing application's policy. For example, the policy engine would compare Figure 2 to a preexisting text editor's policy. The engine uses the older application's policy, base trust level, and current trust level to infer the new application's base trust level. For example, the policy engine might find that the pre-installed text editor application currently has a high trust value. However, since the pre-installed application did not require any ports, the policy engine might assign the new application a lower base trust value than the pre-installed application's base trust value.

The policy engine sums the result of all three metrics to come up with a base trust value for the application's processes. The policy engine also sends its analysis to the ML engine, to aid with intrusion detection.

3.4 Security Monitor

The security monitor intercepts every context switch into the operating system, and checks if the requested operation is compliant with the application's policy. For example, a process in the text editor application might call `fork`; however, the security monitor will disallow this system call since `fork` is not listed in the text editor's policy (Figure 2). In a naive

implementation, the security monitor would add overhead to every system call. Our system uses the insight that a highly trusted process will require less monitoring. Therefore, for a given process, the overhead imposed by the security monitor is inversely proportional to the trust value of that process.

Unfortunately, low-laying malware, such as APTs, may abuse the high trust level of a process to avoid the security monitor. To monitor such cases, the security monitor collects provenance at every system call. Provenance captures casual relationships between systems-level objects. The security monitor sends this data to the ML engine for further processing (see Section 3.5).

The security monitor can generate a trust notification to the user. A trust notification alerts a user of (1) a process that violates the application's policy, (2) a process that is trust-deadlocked, or (3) a process that is infected by "low laying" malware (see section 3.5).

3.5 ML Engine

The ML engine's primary goal is to detect low-laying malware. We assume that the ML engine is trained with "good system behavior", which can be captured during a modeling period before the machine is in use. During runtime, the ML engine ingests provenance collected by the security monitor. Data provenance represents system execution as a directed acyclic graph (DAG) that describes information flow between system subjects (e.g., processes) and objects (e.g., files and sockets) [15]. The ML engine adopts UNICORN, a provenance-based anomaly detector, to identify low-laying malware [9]. The ML engine can flag suspicious behavior and send an intrusion detection notification to the security monitor, which can forward the message as a trust notification to the user.

The ML engine also receives every application's policy and the accompanying analysis produced by the policy engine. The ML engine can help assess the quality of a policy since it receives fine-grained information about each process. For example, the ML engine can alert the policy engine of a policy that is susceptible to low-laying malware.

3.6 Transferring Trust

The `fork` and `exec` system calls can effect a process's trust value. In particular, a `fork`-ed process will inherit the parent process's trust level, and an `exec`-ed process will inherit the application's base trust level.

To help prevent trust deadlock, a high trusted application can transfer trust to a low trusted application. Each application's policy can include a list of vouchees. For example, Google Chrome may vouch for Adobe Flash Player. During a trust deadlock, Adobe Flash Player may request some trust from Google Chrome. The voucher is penalized if the vouchee breaks the system's trust. For example, if Adobe Flash Player violates its policy, then the system will decrement both Adobe Flash Player and Google Chrome's trust values. Of course, a key challenge will be to analyze Chrome's incentives for vouching for Adobe Flash Player.

4 Discussion

The current iteration of XOS assumes that users will avoid installing malicious applications. A stronger threat model would allow a user to install any type of application. In this case, XOS will have to prevent malicious applications from earning the system's trust. Preventing malicious applications from earning trust will be difficult. For example, a malicious application can easily earn a high base trust value by presenting a reasonable, yet over-inclusive policy. A malicious application could also steadily accumulate the system's trust, as long as it does not violate its policy. Neither the policy engine nor the security monitor can provide enough detailed insight to determine the intent of an application. Therefore, we envision the ML engine as the primary defense against malicious applications.

XOS will have to be heavily optimized to be deployed on commodity desktop and server systems. We proposed a heuristic to reduce the overhead of the security monitor. However, the ML engine may also impose significant overhead on the overall system. We look to previous work in system-level data provenance analysis to make this feature efficient.

References

[1] [n.d.]. *What is SELinux?* Retrieved March 1, 2020 from https://www.redhat.com/en/topics/linux/what-is-selinux

[2] 2010. *Mandatory Access Control.* Retrieved March 1, 2020 from https://docs.oracle.com/cd/E19109-01/tsolaris8/816-1041/uguide1-32763/index.html

[3] 2021. *App Manifest Overview.* Retrieved March 1, 2020 from https://developer.android.com/guide/topics/manifest/manifest-intro#perms

[4] 2021. *Requesting Permission.* Retrieved March 1, 2020 from https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/requesting-permission/

[5] A. Alshamrani, S. Myneni, A. Chowdhary, and D. Huang. 2019. A Survey on Advanced Persistent Threats: Techniques, Solutions, Challenges, and Research Opportunities. *IEEE Communications Surveys Tutorials* 21, 2 (2019), 1851–1877. https://doi.org/10.1109/COMST.2019.2891891

[6] Ryan Ausanka-Crues. [n.d.]. Methods for access control: advances and limitations. ([n. d.]).

[7] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. 2005. Labels and Event Processes in the Asbestos Operating System. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom) *(SOSP '05)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/1095810.1095813

[8] Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. The Effectiveness of Application Permissions. In *Proceedings of the 2nd USENIX Conference on Web Application Development* (Portland, OR) *(WebApps'11)*. USENIX Association, USA, 7.

[9] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. 2020. Unicorn: Runtime provenance-based detector for advanced persistent threats. *arXiv preprint arXiv:2001.01525* (2020).

[10] David Harley and Andrew Lee. 2007. Phish Phodder: Is user education helping or hindering?. In *Virus Bulletin Conference Proceedings*. 1–7.

[11] Ka-Ping Yee. 2004. Aligning security and usability. *IEEE Security Privacy* 2, 5 (2004), 48–55. https://doi.org/10.1109/MSP.2004.64

[12] Maxwell N Krohn, Cliff Frey, M Frans Kaashoek, Robert Tappan Morris, and David Ziegler. [n.d.]. Make Least Privilege a Right (Not a Privilege).

[13] Xin Luo and Qinyu Liao. 2007. Awareness education as the key to ransomware prevention. *Information Systems Security* 16, 4 (2007), 195–202.

[14] Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. 2010. Do Windows Users Follow the Principle of Least Privilege? Investigating User Account Control Practices. In *Proceedings of the Sixth Symposium on Usable Privacy and Security* (Redmond, Washington, USA) *(SOUPS '10)*. Association for Computing Machinery, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/1837110.1837112

[15] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. 2018. Runtime Analysis of Whole-System Provenance. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1601–1616. https://doi.org/10.1145/3243734.3243776

[16] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. 2012. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *2012 IEEE Symposium on Security and Privacy*. 224–238. https://doi.org/10.1109/SP.2012.24

[17] Jerome H Saltzer and Michael D Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[18] Ravi S Sandhu and Pierangela Samarati. 1994. Access control: principle and practice. *IEEE communications magazine* 32, 9 (1994), 40–48.

[19] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A Fast Capability System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (Charleston, South Carolina, USA) *(SOSP '99)*. Association for Computing Machinery, New York, NY, USA, 170–185. https://doi.org/10.1145/319151.319163