Secure Assertion

(a.k.a. Application Logging, Application-Level Tripwire, Sanity Checks, Custom Intrusion Detection)

Abstract

The *Secure Assertion* pattern sprinkles application-specific sanity checks throughout the system. These take the form of *assertions* – a popular technique for checking programmer assumptions about the environment and proper program behavior. A secure assert maps conventional assertions to a system-wide intrusion detection system (IDS). This allows the IDS to detect and correlate application-level problems that often reveal attempts to misuse the system.

Problem

Any server that is accessible from the Internet will be attacked. No matter how much effort you expend in protecting it, the possibility exists that an attacker will penetrate the system. The most difficult attacks to detect are those that exploit vulnerabilities in the application itself. These attacks are generally not visible to intrusion detection systems because they are unique to the application and not widely known attack on standard COTS component.

Application-level attacks cannot be defended at the system level. The firewall, the operating systems, the intrusion detection system, and even the application server may view all such traffic as legitimate application requests. For example, if a banking application fails to adequately validate electronic funds transfer requests, an attacker might use the system to cause a victim's bank account to become overdrawn. To the system, these requests look to be legitimate. In this case, the application provides the attacker with a user-friendly graphical user interface for remotely causing mischief.

An attacker that discovers an application-level problem may go completely undetected by the intrusion detection system and the system administrators monitoring the event logs. As noted above, all of the system-level protection mechanisms will view the attacker's requests as valid user transactions. Unless the application author has taken care to log application-level events, the administrators will have no visibility into the problems within the application itself.

Solution

The *Secure Assertion* pattern transparently re-links all the application-level sanity checks (assert statements) with a mechanism that integrates into the system-wide intrusion detection or event reporting system. Any failed assertions encountered by the application are automatically reported to the system-wide monitoring console as a possible security-relevant event.

In languages that include exception handling mechanisms, the *Secure Assertion* pattern replaces the Exception base class with a custom alternative that reports any generated exceptions to the system-wide event reporting system.

The *Secure Assertion* pattern also offers developers an interface for reporting detected problems that are discovered and recovered from. For example, a function that scans user input and replaces illegal and dangerous characters should report any such replacements via the provided reporting interface.

The *Secure Assertion* pattern transparently reports events that developers already detect and recover from. It requires no security-specific coding to be useful. However, it provides developers who are security-aware a set of interfaces for communicating the state of the application to systems administrators at run-time. The developer best understands what events are extraordinary. This pattern provides a mechanism for sharing that understanding with the systems administrators at run-time.

Issues

The *Secure Assertion* pattern provides developers with a reporting framework that allows system administrators to be aware of potentially security-relevant events occurring within the application. In order for the pattern to provide value, the developers must use these mechanisms.

Each application will have its own specific appropriate checks. Some useful generic approaches are:

- Whenever client-side form validation is employed, double-check the client form validation on the server. If any mismatches occur, this may be evidence that an attacker is tampering with the forms.

- After any complex calculation that involves client input, perform sanity checks on the result. For example, many Web servers compute which local files to serve based on the client's URL. Before serving the page, it can't hurt to make sure that the file in question isn't in a critical system directory.

- Objects within an application should check that function arguments comply with stated restrictions. For example, before transferring funds, it can't hurt to ensure that the amount to be transferred isn't negative.

These checks are relatively inexpensive to perform, pay for themselves during system debugging, and could alert a system administrator before the system is remotely exploited.

Intrusion Detection Systems

Many sites use standard intrusion detection system (IDS) packages to detect attacks on the system. While the use of an IDS is a valuable security practice, it is important to realize that the value of the IDS is greatly enhanced by site-specific customization. Commercial intrusion detection systems use comparison against a library of known attack signatures to detect attacks on COTS products. They are generally incapable of detecting attacks on the Web application itself. Any attempt to misuse the application itself, such as disabling client-side form validation, password guessing, or hidden fields, or performing a resource consumption attack, will go undetected by even the best IDS.

If a commercial IDS is not employed, deliver these assertions via the system's error logging facilities (syslog or NT event logs). These logs should be reviewed routinely. In addition, other security-relevant events, such as failed logins, should be reported via the same mechanism.

The knowledge of the application should be used in tuning any commercial intrusion detection system. For example, many standard intrusion detection systems are capable of alerting on specific URL patterns. If the application does not use cgi-bin scripts or .asp files, it is a good idea to tune the IDS to alert on any requests for these types of resources, since they are obviously not legitimate client requests.

The efficacy of an IDS depends on the fact that it is unknown to the attacker. Don't let the attacker know, through messages or any other means, about the details of the IDS system. If any warning messages are communicated to the attacker, they should be generic in nature.

Integrity Checking

Integrity checks alert the system administrator to the fact that the application has been tampered with. Depending on site policy, they may prevent the Web server from starting up if it appears that the site may have been penetrated.

As with Intrusion Detection Systems, there are commercial integrity checking packages, most notably Tripwire. However, as with IDS, custom integrity checks can greatly increase the efficiency of a commercial package.

For example, an application can be designed to maintain a tripwire-like checksums file for the key application components. Those checksums are stored in some obscure, application-unique location. Whenever the application starts, it checks that the files in question have not been tampered with. When a legitimate system upgrade it installed, the administrator will know to update the checksum file. But any attempts by an attacker to alter the system will result in the administrator being notified.

Monitoring Concerns

The best logging system in the world will serve no useful purpose if it is not actively monitored. (See the *Log for Audit* pattern for advice on auditing issues that will affect the type and amount of log data being collected.)

A little customization can go a long way. Often just a few application-aware checks are necessary. Don't overwhelm the system administrators responsible for monitoring the logs. Overly sensitive sensors that constantly fire have a tendency to inure the reader and could cause legitimate attacks to go unnoticed.

The *Account Lockout* and *Network Address Blacklist* patterns should be used by the system administrator monitoring these events. If the application is clearly under distress, an IP blocking rule should be inserted until the problems can be further investigated.

This pattern does not encompass any form of automated response. It is possible to add such a response, although developing the intelligence to respond appropriately to arbitrary text messages is extremely challenging.

Examples

A number of application servers (including BEA WebLogic, Netscape Application Server, and Apache Tomcat) provide developers with a logging function that will append alerts to a system log. The Extended Log Format defines specific classes of log events, including security events. These logging mechanisms can be integrated directly into Intrusion Detection Systems such as ISS RealSecure. These systems allow logged events to be securely forwarded to a remote monitoring console.

Trade-Offs

Accountability	This pattern might improve accountability indirectly by preventing exploits that would circumvent authentication.
Availability	This pattern could impact availability adversely if local policy dictates a fail-secure approach, in which case detected violations will cause the system to become unavailable.
Confidentiality	See Accountability.
Integrity	This pattern improves integrity by removing opportunities for compromise of the application and helps ensure that exploitation of remaining weaknesses will not go undetected.
Manageability	This pattern will increase the management overhead of the application because it will require that log auditing be performed along with other security tasks. If the various sensors are too finely tuned, a significant administration burden will result.
Usability	No effect.
Performance	Properly implemented secure assertions should not significantly impact performance.
Cost	Assertions require that developers spend additional time and effort inserting checks. However, they usually pay for themselves in reduced debugging effort.

Related Patterns

- *Choose the Right Stuff* – a related pattern that provides more details about the trade-offs between developing custom components and using standard parts.

- *Client Input Filters* – a related pattern that describes filters to detect or fix problems with client input; these filters are an important class of application Tripwire and must report those events.

- *Network Address Blacklist* – a related pattern that discusses a mechanism for blocking network access to remote systems that have triggered numerous alerts.

References

[1] Cohen, F. and Associates. "The Deception Toolkit Home Page and Mailing List". http://all.net/dtk/dtk.html, 1998.

[2] Ranum, M. "Intrusion Detection and Network Forensics". USENIX '99, Monterey, CA, June 1999.