Application Security by Design

SECURITY AS A COMPLETE LIFECYCLE ACTIVITY

SecurityInnovation
THE SOFTWARE SECURITY COMPANY
187 Ballardvale Street, Wilmington, MA 01887 +1.978.694.1008 **www.securityinnovation.com**

Table of Contents

Executive Summary

Deployed software is continuously under attack. Hackers have been exposing and exploiting vulnerabilities for decades and seem to be increasing their attacks. This paper describes complete lifecycle activities aimed at producing more secure and robust code that can better withstand attack.

Traditional perimeter defenses are increasingly unable to stop software attacks as more and more hackers focus on the software layer and shy away from attacks against the system and networking layer. Firewalls, intrusion detection and antivirus systems simply cannot solve this problem. Only a concerted effort by the software development community to produce more robust and reliable applications will foil attackers and allow our users and stakeholders to feel confident that they are protected from exploitation.

Secure software is a software development problem. Its solution is the responsibility of every member of the software development team –from managers and support staff to developers, testers and IT staff. Security must be on everyone’s mind throughout every phase of the software lifecycle. A misstep in any phase can have severe consequences.

However, finding a solution is not easy. The problems associated with application security are getting worse with time. Aging legacy software, which was never developed to be secure, is the foundation on which modern, highly connected and business-critical software is operating. The difficulty of patching these older systems and integrating newer applications has served to make the problem worse.

We need to find a better way to think about software development, and develop a new understanding of the engineering processes required to write robust and secure applications, whether they are web-based, server software, or client-side applications.

It is crucial that each phase of the software development process include the appropriate security analysis, defenses and countermeasures that will result in more secure released code. From requirements through design and implementation to testing and deployment, security must be integrated throughout the Software Development Lifecycle (SDLC) in order to provide the user community with the best, most secure software-based solutions.

SOFTWARE DEVELOPMENT & SECURITY

Developing successful software in the 21st century requires a paradigm shift. Rather than adopt the 20th century attitude that software is made to solve problems and people will use it for the greater good, developers need to take the position that their software will be attacked the second it is deployed and they must build applications that can defend themselves in a hostile environment. Developers must consider software security an integral part of the application development process, not an afterthought.

While it is possible to “retrofit” protection into applications post completion, this is the most expensive and least effective approach. Viruses, worms, and other attacks due to security flaws in applications cost businesses billions of dollars in lost productivity, system recovery, and information loss every year. This doesn’t take into account impacts that are more difficult to measure, such as reputation loss or internal morale. The cost of addressing security vulnerability during the development cycle is less than two percent the cost of removing such a defect from a deployed production application[1]. From the standpoint of both cost and effectiveness, considering security as an integral part of the software development lifecycle is the best way to build and maintain robust, reliable, and trustworthy applications.

Software development methodologies, such as iterative, agile and waterfall models, all benefit from a focus on secure practices. In each case, incorporating security-based techniques in each phase of the SDLC will improve quality and resistance to attack in the final product.

The main security activities in each phase of the SDLC include:

- **Requirements analysis** - Close attention to requirements and how systems interact with their environment ensures that a software project starts building on the right foundation. Insecurities introduced in this early phase will only be compounded in later phases. When developers write requirements about what a system must do, they must also consider what a system must not do. When they write use cases, they need to write misuse/abuse cases to describe how a malicious user might interact with the system. When we consider both legitimate and illegitimate users, our guard is raised and the entire downstream process will benefit from this increased diligence.
  Another important aspect of requirements analysis is understanding risk. For security, this means understanding the business risk of a successful exploit against the application, how that exploit may affect users and what business processes would be necessary to manage damage control. The costs of liability, redevelopment and damage to brand image and market share needs to be understood up front.
- **Design** - Although most security defects are born during implementation, the most expensive are those that are introduced in the design phase. A proactive approach of paying close attention to security during the design phase prevents expensive redesign and yields substantial benefits during all later phases of the SDLC. Design considerations include both architectural issues at the system level and at the individual component level. At the system level, we are interested in techniques that help us reduce our software's attack surface, and better understand how potential threats might impact our design choices. At the component level, we are interested in deciding how best to implement each module.
- **Implementation** - Writing code is never a straightforward process even when the requirements and design have been well conceived. Proper error handling, avoiding dangerous code constructs, implementing input validation and encryption, and ensuring secure communications are common implementation tasks that can fail terribly even in the face of a good design. The list of considerations during coding has grown over the years and attention to these details during code reviews and unit testing can be the difference between a smooth deployment and absolute disaster.
- **Testing and deployment** - Security testing is different and few functional testing paradigms address the needs of security testing sufficiently. When security is a concern, testers must pay more attention to the software's operating environment and functional testing of security components must be more intense to compensate for the increased risk associated with those components.
  Secure deployment of software means paying more attention to potential customer environments including operating systems, network connections, configuration, and customized setup. Security may also require that deployment contain logging (for forensics and incident response) and monitoring so that data for any issues that do arise are available to the right people at the right time.

[1] John Pescatore, FirstTake FT-23-5794, Gartner Research, July 2004

- **Maintenance** – Secure maintenance practices emphasize the importance of understanding the existing security infrastructure of an application, maintaining documents to ensure that they match the changes being made, and carefully auditing proposed application changes in terms of the risks that they impose on the overall security of the system. These practices become critically important when a security bug is discovered in the field and must be fixed in an already deployed application. While fixing the defect is paramount the use of a well crafted incident response plan will help ensure a smooth process that fixes the problem with minimum risk of introducing additional security defects in the application.

Addressing security in each phase of the SDLC is the most effective way to create highly secure applications. Solid security focused design principles followed by rigorous security focused coding, testing and deployment practices lead to applications that can stand up to attack and will require less maintenance over time. This results in lower ownership costs for both the end user and the application vendor.

This document will discuss best practices for embedding security and security testing into the SDLC.

DIFFERENT DEVELOPMENT PROCESSES

Developing software is a complex process that gets more complicated as the number of players and lines of code increase. To manage the SDLC, many organizations adopt a formal development process – for example the Rational Unified Process (RUP), eXtreme Programming or Microsoft Solutions Framework (MSF). All of these processes can be used successfully, and are better than having no process at all.

The inclusion of security is process agnostic and maps to key phases common to all of the development processes in existence—requirements gathering, design, implementation, testing/deployment and maintenance.

This document does not focus on one particular process, but on the effective use of security best practices during process phases.

The Requirement Phase

Defining requirements is the first step in the software development process. When done properly, it produces applications that meet the needs of a specific audience. Conversely, poorly defined requirements generally lead to applications that do not meet the needs of the target audience, disrupt the development process, and can lead to project failure.

Requirements represent the bedrock of a software project. If requirements gathering and analysis is done correctly, a solid basis for the rest of the project is established. When requirements are poorly understood, a heavy price is paid during each additional phase, putting the project at risk.

In the traditional development lifecycle, requirements come in several flavors including functional requirements (requirements that describe the functionality that needs to be implemented) and physical requirements (requirements that describe time, space and other physical constraints that the software must operate under). Adding security to the mix requires that we think about security requirements as well.

Undertaking an analysis of security requirements means that security is built into the product from the ground up and sets the stage for the development of a secure and reliable product.

BEST PRACTICES FOR SECURITY REQUIREMENTS

The following collection of best practices is used to define security requirements.

Document Key Security Objectives

Simply saying that ‘the product should be secure’ is not helpful to any project stakeholder. A user who sees such a requirement may incorrectly assume that security was part of the design process. It also does not help developers who need to understand specific security objectives in order to derive a design. The best approach is to define specific security objectives and turn these objectives into concrete requirements.

For example, define the statement “the system should disallow unauthorized users access” as an overall security objective. Your objective is then be mapped to a series of requirements. For example:

- A password protection mechanism shall be implemented that enforces strong passwords (at least ten characters in length with a mandatory mix of numeric, upper case, lower case and special characters).
- The password protection mechanism shall time out after three unsuccessful logon attempts from a single user ID.
- The password protection mechanism will make use of randomly generated numbers to foil automated dictionary attacks.

The idea is to give specificity to the security objectives that users will appreciate and that developers can use.

From a security standpoint, it is important to address user tolerance to security requirements that will be implemented in the system. If a specific system requires a lockout of a user account on three unsuccessful logon attempts, how will this inconvenience legitimate users? Will the service calls that such a feature generates overwhelm the help desk and increase maintenance costs? Addressing such issues up front accelerates the development process and leads to increased user tolerance of security controls.

One way to ensure your security objectives are well focused is to look at each from the perspective of risk. Risk can be thought about from two different angles:

- **Risk to the customer** needs to be carefully considered. How will a breach of the system affect a customer's ability to run his or her own business? Will a breach put a customer's data at risk? Will it put them in violation of any corporate standards or privacy laws? Thinking proactively about your customer's security concerns will help you mitigate, or at least prepare contingencies, for such a worst-case scenario.
- **Risk to market share and brand reputation** is an important consideration for any independent software vendor. How will a breach of your system affect your installed base? What opportunities would a breach represent for your competitors? What type of damage control would help alleviate these risks?

Separate Security Requirements from Functional Requirements

Security requirements should be enumerated separately from functional requirements to facilitate explicit review and testing. It is often the case that later stages of the SDLC employ specialists for reviewing and testing security functionality. Mixing security requirements with functional requirements will make this process harder and more time consuming.

For Every Use Case, Write a Misuse Case

Use cases are a best practice for analysis of functional requirements and understanding how they impact the system. Use cases describe legitimate users and how they interact with the application to get real work done. Misuse cases (or abuse cases) describe attackers (malicious users) and how they intentionally misuse the system.

For example, a use case for an online ecommerce application might spell out a session in which a user performs a search for a product, finds the product, and places an order. Misuse cases would spell out attempts by an attacker to jump past the credit card entry page directly to the purchase confirmed page. These misuse cases are the basis for threat modeling which will add more detail to these attack scenarios in the design phase.

Denial of Service Scenarios

Include expected performance capabilities and requirements and methods for mitigating heavy loads. Many performance requirements are focused on quality of service (QoS) and the user experience around system responsiveness. In the context of security, performance requirements speak to a system's ability to stand up to a denial of service (DoS) attack.

For example, consider the timing out of user passwords after three unsuccessful logon attempts as discussed above. This time-out feature is intended to protect against password guessing attacks, but could lead to denial of service if an attacker manages to lock out every single user.

Write Requirements for Industry Standards and Regulatory Rules

Laws like Sarbanes-Oxley (http://www.sarbanes-oxley.com/) and HIPAA (http://www.hipaa.org/) put constraints on data collection and may affect how you design an overall security strategy for your product line. Be sure that everyone on a project understands the standards and regulatory requirements that apply to the product being developed.

The Design Phase

Designing secure applications is more straightforward when requirements are spelled out as suggested above. However, the process is still far from over. Even with the best requirements, software design is a challenging activity and must be performed with great care and clear goals. The overall goals for the security design process are:

- Spell out the threats to a system in details sufficient for software developers to understand and code against.
- Provide a system architecture that mitigates as many threats as possible.
- Employ design techniques that force developers to consider security with every line of code they write.

BEST PRACTICES FOR SECURE DESIGN

Use the following collection of best practices to guide software design:

Perform Threat Analysis

Threat modeling is an engineering technique used to describe threats to the software application. Threat models are methodically developed in order to understand the risk to a system from malicious users or applications. Threat modeling allows development teams to anticipate attacks by understanding how an adversary chooses targets (assets), locates entry points and conducts an attack.

A well-implemented threat model will profile the assets that need to be protected, how the threats to these assets, the attacks that could be used to realize a threat, and the conditions under which the attack may be successful. A threat model shows how adversaries view the system, its components, and where they are likely to attempt its exploitation. Not only is this important for identifying potential threats, but also in understanding what application defenses must be defeated in order for a threat or series of threats to be realized.

For example, an ecommerce system may have the following threat, “A malicious user is able to gain an undeserved credit on their account.” A threat model would specify this as:

- **Threat:** A malicious user is able to gain an undeserved credit on their account
- **Attack:** User sets item_price field to a negative value
- **Conditions:** Server-side validation doesn’t exist on item_price field or server-side validation is non-existent

A threat model will consist of multiple threats each of which can have multiple attacks. Each attack can then have multiple conditions. The conditions indicate how to defend against the threat in the design, and can be used during testing to generate security test cases.

Attack scenarios that describe what should be guarded against include:

- User saves Web page locally
- User identifies item price on page and modifies it
- User resets all relative page links to absolute links
- User reloads page and purchases item

This simple threat now has far more depth and detail, allowing developers to think through possible defensive scenarios to foil the attacker at every single opportunity, and allowing testers to develop test cases that can be used to prove the existence of the threat in the application.

Identify Design Techniques that Mitigate Risks

All security risks are mitigated in either the development phase, (see the next section), or in the design phase. As a general rule, the earlier a risk is mitigated the better. This means that design is the first place to initiate defensive thinking.

For example, the risk represented by the threat described above can be mitigated by the design principle of validating all user input on the server. When properly implemented, the entire threat and all of its associated attacks points are rendered moot because, no matter what an attacker changes on the client machine, the negative price value will be validated on the server and the attack will be thwarted.

There are many design principles, such as using server-side validation, that developers can use to mitigate large classes of common security problems – for example, the principle of least privilege (all code should run with the lowest permission levels it needs to perform its assigned task); and defense in depth (place multiple barriers in the path of a determined attacker).

It is a good practice to use design techniques recommended by the development platform vendor and over time build a set of proven design techniques for the applications you are developing.

Finally, consider the role of the programming language you choose in mitigating risks. Languages that target a virtual machine such as Java and C#, reduce the risk of buffer overflow vulnerabilities, while strongly typed languages reduce the risk of integer overflow and many data-oriented vulnerabilities without requiring the implementation of additional protective measures.

For more information on design techniques for security see the Microsoft patterns & practices Security Design Guidelines index - http://msdn.microsoft.com/library/en-us/dnpag2/html/SecurityGuidelinesIndex.asp.

Identify Components Essential to Security

Designing the overall component structure of a system is a key activity in the design phase. Each component identified in the design must be carefully analyzed for its security needs. Some components will require protection. Other components will provide protection in the form of input validation, authentication, or cryptographic routines. These latter components must be the focus of extra attention from security engineers.

For every component that implements a security function, engineers must carefully consider how that function is going to be implemented, reviewed and tested. Test planning for security components is a crucial task that needs to be given attention as early as possible in the SDLC. Planning tests only after a component is implemented is too late and adds additional risk to the process.

Build a Security Test Plan

Security test plans should be separate from functional, performance, load and other testing concerns. Security testing often requires specialized personnel and tools that are beyond the normal function of a QA team. Furthermore, security is important enough that it cannot and should not be buried in a test plan containing other types of tests. Security tests need to be highly visible and verifiable by external auditors who are likely (even preferably) not concerned with other aspects of quality.

Security test plans differ from functional test plans in the following sense. Where functional test plans specify what *should* occur when a user applies a series of inputs, security test plans specify what *should not* occur. This fundamental difference in scope and function makes the two types of test plans incompatible. Separate them to call attention to the importance of security testing and the fundamental differences in how such tests are applied.

A security test plan can be derived directly from the results of threat modeling. The threat model will describe the assets to be protected, the attacks that may be used to gain access to or damage assets, and the conditions under which the attacks are likely to be successful. A security test plan must first be concerned with testing these already identified conditions and attacks. If the conditions hold true and an attack succeeds then you have found a valid threat that should be addressed.

Once testing begins, the test plan should be expanded to cover additional attacks and conditions that were not discovered during the threat modeling process. The act of security testing almost always uncovers information about the application that was not discovered in the threat model. The threat model is a good starting place; do not allow it to limit your testing efforts.

Plan for Incident Response

Expectations of perfect security are unrealistic. All software is a target for exploitation and all software, given enough effort, can eventually be exploited. Ensure you have a plan for dealing with security issues when they occur.

Incident response procedures need to be documented and understood. During a security event, organization and preparedness can be the difference between further harm to your user base and a quick, decisive reaction that reestablishes the trust relationship with users by allowing everyone to recover from the attack as quickly as possible.

Failure to plan for future security incidents is a common mistake made by development organizations. Without a plan in place your team may be forced to add unexpected delays to current development tasks in order to respond to a critical customer security issue. A security incident response plan not only helps set reasonable expectations on future development, it also ensures that the security defect is fixed more accurately and deployed more quickly. Setting expectations, and roles, before the incident occurs will help your team focus on the problem at hand rather than force them to develop process and procedures for security response in parallel with fixing the security defect.

The Implementation Phase

The focus on security in the requirements and design phase sets the stage for writing secure code. Getting the requirements and design phases right is the most important way to ensure that this happens.

However, mistakes can still occur – code is developed by humans and humans are imperfect. In addition, developers often have preconceived notions about how to write code that do not mesh with security goals. As developers, we tend to do things the way we did them last time. This complacency can often lead to unintended side effects, like choosing the wrong string manipulation APIs just because it was what was used for the last job.

Following secure implementation best practices and ensuring that those best practices are up to date will help ensure that the hard work put into requirements analysis and secure design is not squandered during development.

BEST PRACTICES FOR SECURE IMPLEMENTATION

Use the following collection of best practices to guide software implementation.

Coding Standards

All commercial development, from giant server applications written in C to Java applets and HTML, should adhere to rigid coding standards. If code runs or plays a part in gathering information for a mission-critical application, it is too important to leave to chance and should be controlled by coding standards that are constantly kept up-to-date.

Obtaining the right standards and keeping them current with the latest best practices and security wisdom should be a top priority for software development organizations. Because these standards are the clearinghouse for security best practices, their maintenance and dissemination must take a high priority. Every developer should commit as much of the standard to memory as possible and have a copy readily available for reference whenever there is question about a specific practice. Such diligence assures that as new developers arrive into a group and experienced developers depart a group, the collective knowledge of best practices will not be lost.

Coding standards for security should contain safe handling of strings and integer results, methods of input validation, handling of temporary files, authentication of code libraries, proper error handling, etc. Constructing such a standard is an excellent educational practice for developers. Enforcing these standards as an everyday practice for all software developers within an organization will ensure that a large number of security bugs are avoided as the code is written.

For more information on secure coding standards for .NET code, see the Microsoft patterns & practices Security Question List for managed code - http://msdn.microsoft.com/library/en-us/dnpag2/html/PAGQuestionList0001.asp

For more information on secure coding standards for C and C++, see *Writing Secure Code* - http://www.amazon.com/gp/product/0735617228/sr=1-1/qid=1137101922/ref=pd_bbs_1/104-3196078-6421533?%5Fencoding=UTF8

Code Reviews

Code reviews are always a best practice for software development, but the normal code review performed for functional verification is different than a code review performed for security purposes. In fact, each type of review is so important that they should not be combined. A functional review should look at functional issues, and a separate security code review should look only for security issues.

All code developed by the team should be security reviewed regularly both individually by the development manager and as a team. The key objectives of the code review are:

- The design goals are being met
- The security objectives are being met
- The implementation is robust

Several techniques are used as part of the code review and include both automated and manual processes. The automated steps include code scanning to locate use of non-constrained methods, unchecked return values, methods without exception handling and other key patterns. Careful attention should be paid to the following "hot" areas:

- **Hard coded secrets** - Scrutinize code for embedded text "secrets" associated with variable names such as `password` or `creditcard`.
- **SQL Injection** - Ensure that any input used by an embedded SQL query is validated and that the SQL query is parameterized.
- **Information Disclosure** - Look for potential exposure of undesired information through error dialogs or logfiles. Failure to clear secrets from memory, transmitting clear text over the network, storing clear text on disk are all sources of information disclosure.
- **Cross-Site Scripting** - Look for this web vulnerability that is caused when an attacker inserts script into the application, which is then executed in the application's security context, allowing the attacker to collect user information.
- **Input/Data Validation** - Look for input validation that is incomplete or that could be tampered with by an attacker (such as client-side validation in a Web application). Also, watch for authentication based only on file names, IP address or other insecure validation mechanisms.

There are many other things to watch for – this is not an easy task and the above list is far from complete. The use of a checklist, a set of questions to ask while reviewing the code, and a good tool that allows you to scan the code for common problems will contribute immensely toward a successful code review.

Automated Static Analysis

Think of static analysis as an automated code review where a tool processes the source or binary and lists potential problems that a human developer then manually investigates.

Static analysis for security purposes is crucial because history has shown that it detects many problems that would be difficult to find in any other way. For one thing, security bugs are smaller in scope than functional bugs, meaning that the static analysis procedure has to look for fewer potential problems, thus reducing the number of false positives. Security bugs also tend to be more costly than functional bugs (or at least they are more visible) and thus an investment in preventing them is more easily justified.

Unit Testing for Security

Unit testing is rarely a formal process in development organizations. Unlike more formal system tests that are conducted by a separate test organization, unit tests are carried out by the development team – often the very same developer that wrote the code in the first place.

For security, many top software vendors have found that this is not sufficient for modules that pertain to security and that significant benefit is realized by formalizing the unit testing process. Practices like developers crosschecking each other's code by providing unit tests to other developers, and maintaining a unit test library, are considered best practices.

At a minimum, all components that enforce security (processing untrusted inputs, parsing file formats, encrypting network communications, authenticating and authorizing users) should be diligently unit tested for robustness, reliability and security. Other places to insist on a unit testing process are hard to reach error code paths, code that processes sensitive information, and code that is network-enabled.

For more information on unit testing and the Nunit testing framework see the Nunit home page - http://www.nunit.org/

Defect Management

Defect management is a critical process that enables consistent communication across development teams. The key security goal of defect management is to make sure that all identified security defects are prioritized, sized and assigned to someone to be repaired within a specified time frame. Security defects should be retested both from a regression point of view and with new test cases that ensure that fixes were properly made and did not break any existing functionality.

For security vulnerabilities, the following should be performed:

- Ensure that every bug is fixed in its entirety. Security bugs tend to get an immediate "knee-jerk" reaction from testers who scramble to write a bug report as soon as the defect is found. More care is required. Often, rushed bug reports contain only the tip of the iceberg when it comes to what insecure behaviors might occur. Developers should be careful to reproduce the insecure behavior and investigate all such behaviors to ensure that a reliable fix is made that mitigates all aspects of the defect.
- Investigate whether the defect could exist in similar functions elsewhere in the product. Look for the same bug in similar functionality throughout your application. Bugs tend to travel in groups; make sure you have eradicated them all.
- Retest all security fixes by re-reviewing the code, re-running scanning tools, reapplying unit and system tests, and then re-testing with new test cases to ensure that the fix is reliable and no new security bugs were introduced in the process.

Effective defect management ensures that defects have owners and that everyone who needs to be in the loop is involved and working towards a common endpoint.

Testing and Deployment Phase

Even applications that have been built with a process that includes security in every phase need significant testing. Unit testing will help uncover many common vulnerabilities, but security focused testing, called “penetration testing” needs to take place during the integration and system testing phases. Penetration testing provides feedback on the types of issues that slip through the cracks, and finds problems before the code is shipped to the field.

BEST PRACTICES FOR SECURITY TESTING

Use the following collection of best practices to guide software testing.

Look for What is Not There

Specification-based testing is the polar opposite of penetration testing. In the former, testers are guided by a test plan based on a written specification. In other words, testers look for functionality that is supposed to be present. For example, functional testers apply input A and watch for output B because that is the specified behavior.

However, for security, this is not enough. The fact that the software correctly produced output B might actually mask a security defect because it gives us the false sense of security that the software is correct.

For security testing, we are more concerned with functionality that is *not* supposed to be there such as unintentional side effects and behaviors that are not specified in the test plan or design. This is critically different than testing an application for failure, a common practice in function testing. While some security bugs do manifest themselves as system failures (most notably the buffer overrun) many are much more subtle. An effective security tester will always test with an attack in mind, attempting to get at an asset of known value, and will look for every clue that may allow the attack to succeed.

For example, if we are testing a music player, we do not care whether the music plays well – that is the job of functional testing. Our goal is to ensure that no music artifacts are stored in memory or on the disk where they might be stolen by a determined music pirate. The primary goal in security testing is to think like an attacker and look for chinks in the application’s defenses.

Test Outside the GUI and Public Interfaces

Testing is all too often a matter of forcing as many inputs through an application’s GUI or APIs as possible. However, for security, this simply is not enough. In reality, the inputs that arrive to an application through its public interfaces are far outnumbered by inputs that arrive from the network and from the file system. In addition, for security, these “invisible” inputs are often far more important since that is the first place attackers will look in order to gain an advantage. Tools like file corruptors and network fuzzers are very important in this space as they allow these other, less visible, interfaces to be more thoroughly tested.

For more information on testing outside of the normal interfaces see *19 Attacks to Break Software Security* - http://web.securityinnovation.com/19-attacks-for-breaking-applications/.

Also see *How to Break Software Security* - http://www.amazon.com/exec/obidos/ASIN/0321194330/qid=1092252380/sr=ka-1/ref=pd_ka_1/002-2780814-6580820.

Dynamic Analysis/Fault Injection

Static analysis analyzes code at rest and necessarily misses cases where the code only fails in its operational environment. Therefore, dynamic analysis is the next step – analysis automation is attached to a running binary to identify problems that would be difficult for unaided, manual testing to detect. Dynamic analysis is to testing what static analysis is to a code review.

Dynamic analysis requires tools that can identify and instrument the running executable and include tools such as Web proxy tools, API call interception, debuggers, etc. They uncover many of the invisible behaviors like memory manipulation, file access, library loads, etc., that do not appear in plain view on an application’s GUI or API.

A key aspect of dynamic analysis for security is the concept of fault injection where faulty inputs like return calls from the operating systems and third party components can be modified. Fault injection’s main purpose is to drive the application to execute its error code paths and other typically under-exercised code. Attackers often target such under-exercised code because they know that little developer or test time was spent looking at it making it a prime location for security problems.

Test the Deployment Environment

An otherwise secure application can be left exposed by a single misconfiguration or mistake in the setup process. If you are deploying an application to a server (such as a web application, or client/server app) check the server and all of the services you will be dependent upon for security. Use deployment checklists provided by server software vendors and build your own checklist of common problems in the application you are deploying. For instance, you will want to scan the server for open ports, review configuration files for a web server, and ensure attackers cannot gain access to sensitive files or directories.

For more information on security deployment review see Microsoft patterns & practices Security Deployment Review Index - http://msdn.microsoft.com/library/en-us/dnpag2/html/SecurityDeploymentReviewIndex.asp

Test Incident Response Procedures

Do not wait until post deployment to find out if your incident response mechanisms are up to the task. Run breach simulation drills with any priority one security bugs found during system test. Pretend the bug you just found was the cause of a major breach and test your organization’s reaction to fixing the bug, testing the fix, and understanding what problems you might encounter in pushing the patch to the field. Prerelease is definitely the time to fix any missteps made during these simulations.

Conclusion

Secure software does not happen by accident. It happens when every developer, tester and manager on a project takes security seriously during every phase of the software development lifecycle. Security is not something that is addressed at the end of a product cycle, nor is it a specific milestone that occurs during project execution.

Security must be everywhere. It should begin at project inception and be on the mind of every engineer during requirements analysis, design, coding, testing and deployment. This is the only way that security can be reliably improved in every product you build.

Security cannot be separated from the software development lifecycle. Companies that realize this and obtain the knowledge and tools required to act on this realization will gain an advantage over their competitors and protect their customers from the dark side of the hacking community.

References

BEST PRACTICES

Microsoft patterns & practices Security Design Guidelines index - http://msdn.microsoft.com/library/en-us/dnpag2/html/SecurityGuidelinesIndex.asp

Microsoft patterns & practices Security Question List for managed code - http://msdn.microsoft.com/library/en-us/dnpag2/html/PAGQuestionList0001.asp

Writing Secure Code - http://www.amazon.com/gp/product/0735617228/sr=1-1/qid=1137101922/ref=pd_bbs_1/104-3196078-6421533?%5Fencoding=UTF8

Nunit home page - http://www.nunit.org/

19 Attacks to Break Software Security - http://www.sisecure.com/pdf/19-attacks.pdf

How to Break Software Security - http://www.amazon.com/exec/obidos/ASIN/0321194330/qid=1092252380/sr=ka-1/ref=pd_ka_1/002-2780814-6580820

Microsoft patterns & practices Security Deployment Review Index - http://msdn.microsoft.com/library/en-us/dnpag2/html/SecurityDeploymentReviewIndex.asp

REGULATORY STANDARDS

Sarbanes-Oxley Financial and Accounting Information Disclosure Act (http://www.sarbanes-oxley.com/)

Health Insurance Portability and Accountability Act (HIPAA) – (http://www.hipaa.org/)