PreEmptive

Enterprise Application Hardening

Abstract

This white paper enumerates the essential characteristics of an enterprise obfuscation solution and assesses its suitability and value within a layered security program and as a component of a risk-based IT control framework.

Specific topics include:

- The value in protecting your applications before you release them
- A summary of application protection technical capabilities and considerations
- Application lifecycle workflow and process requirements
- IT Controls and Governance, Risk and Compliance
- Development, quality assurance and support best practice
- Guidance as to when enterprise app hardening can be an effective IT control
- Enterprise app hardening evaluation criteria
- Guidance for incorporating hardening into a DevSecOps process

Table of Contents

Three Reasons to Harden Enterprise Applications

Intellectual Property Protection

Whether you're building applications for sale, as a key part of a larger financial or manufacturing business, or as part of a line of business apps for internal use, there is likely to be IP (trade secrets) within your software. And possession of functioning source code provides transparent access to any IP that is coded within the application. So, a hacker that works for a competitor might be able to steal your technological advances by reviewing your source code.

From a legal perspective, there are three common ways to protect the IP embedded in your code:

- Patents
- Copyright Protection
- Trade Secrets

Although patents offer the strongest protection, patenting software requires a massive certification process that is slow, expensive, and difficult. For the vast majority of software builders, a patent just isn't a workable IP protection solution. In contrast, copyright protection is automatic. You don't need to mark up your code and copyright law is the basis of most software licenses. However, it comes with its own big issues; it's limited to copying and distributing content. You can't copyright algorithms, innovations, or inventions, so if someone else's code looks nothing like yours or that organization can demonstrate they developed their code in isolation from yours, they're in the clear. And with managed code, where someone else can generate the same algorithms as yours in multiple languages, copyrighting an application offers little real protection. That leaves trade secrets, which have a lot going for them. There's no certification, they last forever, and they include concepts, innovations, etc. that give your business financial and competitive advantage. That's why trade secret protection under the law is increasingly the regulatory strategy of choice for many development organizations.

However, trade secret protection has two significant limitations. First, some major jurisdictions – like India, for example – simply don't recognize the legal concept of a trade secret. The other limitation of trade secret protection is even more fundamental: unlike copyrights and patents, it only covers things that are actually secret. Once something becomes public, it can no longer be protected under trade secret law. More specifically, the definition of trade secret theft requires that possession of a trade secret be achieved through improper means, such as bribery, blackmail, or espionage. Recently enacted trade secret laws, both in the United States (the DTSA) and the European Union, specifically permit reverse-engineering of any legally acquired product. If reverse-engineering your application yields your source code, and therefore, your algorithms, such algorithms may no longer be secrets and no longer covered by trade secret protection.

Application Integrity Protection

Whether a hacker is trying to pirate your app, steal your data, or alter the behavior of a critical piece of infrastructure software as part of a larger crime – inspecting and/or modifying an application can play an essential role. As part of a layered protection strategy, companies should have mechanisms in place that add anti-debug and anti-tamper functionality directly into an application to protect, detect, and respond to attacks on the application's integrity.

Consider how the following exploits that stem directly from debugger hacks cross data, operational, and IP risk boundaries:

Debugger Hack	Resulting Vulnerability and Risk
Bypassing encryption and other techniques used during data transmission and/or storage exposes otherwise secured data.	Unauthorized data access leads to data loss, loss of revenue, privacy breaches, regulatory non-compliance, and trade secret theft.
Insert and modify data within your application	Interrupt application flow circumventing controls and governance and voiding authorization and access controls.
Trace logic and the flow of your application	Expose intellectual property for reuse and exploitation
View encryption functions, the values of dynamic keys and when and how sensitive information is saved to your file systems and databases	Security and operational breach exposes data and systems beyond any one application.

Application Data Protection

Exploiting application vulnerabilities to gain unauthorized control over private data is a widely recognized, common attack technique.

In an ideal world, development would release vulnerability-free applications that were also immune to native and managed debugger hacks, profilers and reverse-engineering tools. We do not live in an ideal world.

Secure coding practices informed by subsequent static analysis and security testing are often effective in striving for this ideal, but even in the best case scenarios, can never guarantee a vulnerability-free application.

Further, secure coding practices do not address risks stemming directly from unauthorized debugging, tampering, or reverse engineering hacks (since these do not rely upon vulnerability exploits for success). Since data in use can't be encrypted, the next best strategy is to restrict unauthorized use of debuggers, rooted mobile devices, emulators and other tools that hackers rely upon to access and modify application-resident data. Preventative, detective, and responsive controls combine to secure your applications and – by extension – the data that flows through them.

If you care about data-at-rest and data-in-motion, then you need to care about data-in-use – it's the same data and it brings with it the same responsibilities, risks and liabilities.

The Three Dimensions of Enterprise Application Hardening

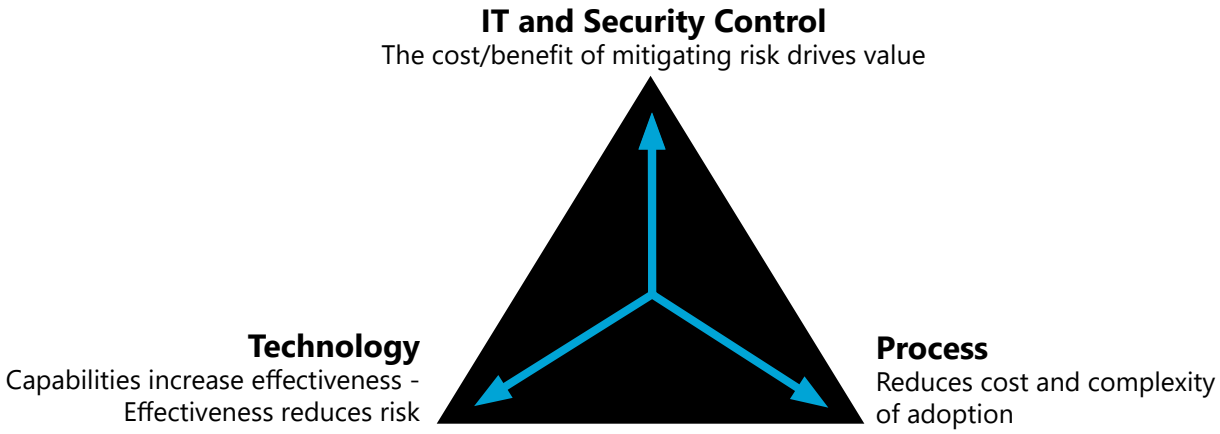


Figure I: The three dimensions of enterprise app hardening

Technology

Application Hardening and Shielding transformations fall into a number of categories including:

- **Renaming**: altering the names of methods, variables, etc. to make source code more difficult to understand. Strong renaming algorithms use overloading to reuse names, forcing every line to be analyzed.
- **Control Flow Obfuscation**: logic and flow are re-expressed making translation into valid C# (or any other language) impossible. Sophisticated approaches provide different levels to strike the right balance between obfuscation and performance.
- **String Encryption**: strings such as login prompts, SQL queries, etc. are encrypted and decryption function calls are injected into the instruction stack before the string is needed.
- **Instruction Pattern Transformation**: converts common instructions created by the compiler to other, less obvious constructs. These are perfectly legal machine language instructions that may not map cleanly to high level languages such as Java or C#. One example is transient variable caching which leverages the stack based nature of the Java and .NET runtimes.
- **Dummy Code Insertion**: inserting code into the executable that does not affect the logic of the program, but breaks decompilers or makes reverse engineered code much more difficult to analyze.
- **Unused Code and Metadata Removal**: removing debug information, non-essential metadata and used code from applications make them smaller and reduce the information available to an attacker. This procedure may slightly improve the runtime performance.
- **Binary Linking/Merging**: this transform combines multiple input executables/libraries into one or more output binaries. Linking can be used to make your application smaller, especially when used with renaming and pruning, It can simplify deployment scenarios and it may reduce information available to hackers.
- **Opaque Predicate Insertion**: obfuscates by adding conditional branches that always evaluate to known results—results that cannot easily be determined via static analysis. This is a way of introducing potentially incorrect code that will never actually be executed, but is confusing to attackers trying to understand decompiled output.
- **Anti-Tamper**: infuses application self-protection checks into your code to verify your application has not been tampered with in any way. If tampering is detected, it can shut down the application, limit the functionality, invoke random crashes (to disguise the reason for the crash), or perform any other custom action. It might also send a message to a service to provide details about the tampering detected.
- **Anti-Debug**: when a hacker is trying to pirate or counterfeit your app, steal your data, or alter the behavior of a critical piece of infrastructure software they will almost certainly begin with reverse engineering and stepping through your application with a debugger. This technique layers in application self-protection by injecting code to detect if your production application is executing within a debugger. If a debugger is used, it can corrupt sensitive data (protecting it from theft), invoke random crashes (to disguise that the crash was the result of a debug check), or perform any other custom action. It might also send a message to a service to provide a warning signal.
- **Reacting to Compromised Environments**: Regulators, standards bodies and IT auditors have become increasingly likely to recommend an absolute prohibition of rooted devices in production environments. This technique injects sophisticated root detection logic as well as the logic your app needs to defend itself against these evolving attacks.
- **Watermarking**: Software watermarking can be used to hide customer identification or copyright information into software applications and can be used to identify owners of the software or track the origin of a pirated copy. Software watermarking is similar to watermarking in other digital such as songs, movies and images. An obfuscator can be a particularly effective tool to insert a watermark because its operation can be, by its very nature, difficult to find or to alter.

Process

The process of application hardening and obfuscation has emerged as the pivotal element in driving the viability and the value of protecting your applications. Without a well-defined and integrated process, the complexities and risks introduced by hardening and obfuscation may ultimately outweigh the perceived benefits that it promises. It can complicate debugging, patch generation and management, distributed development practices and the reuse of libraries, components and web services. Look for vendors that allow you to evaluate and test their application hardening solution with your specific applications. It should integrate into your build process without too much work.

The following components represent the basic building blocks of an enterprise application hardening step that can be inserted within a build process. While other configurations are certainly possible, this approach has proven flexible enough to accommodate multiple libraries and assemblies, patch generation and distributed development workgroups.

- **Input Programs** - Any unprotected .NET, Java, Android or iOS executable or library
- **External Dependencies** - Dependent dynamic link libraries
- **External Configuration** - XML configuration file
- **App Obfuscation, Hardening or Shielding** - Post Compilation Protection System
- **Output Programs** - Protected executables or dynamic link libraries
- **Map File** - XML file containing information on how renaming was done and what attributes were applied.

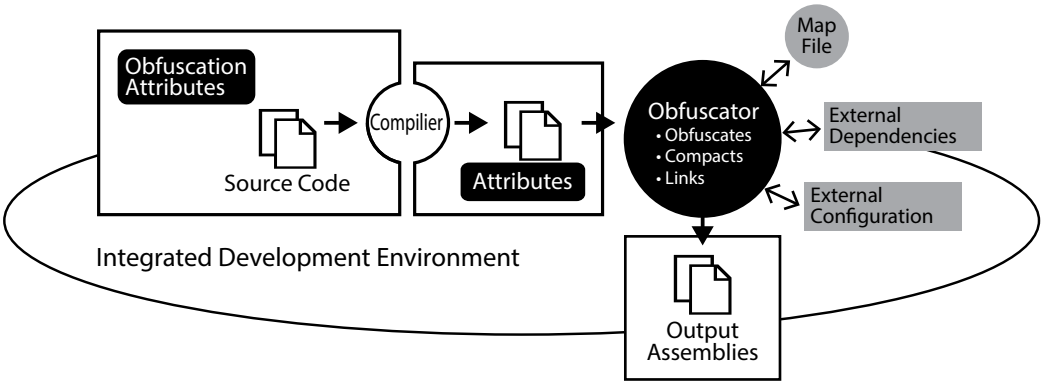


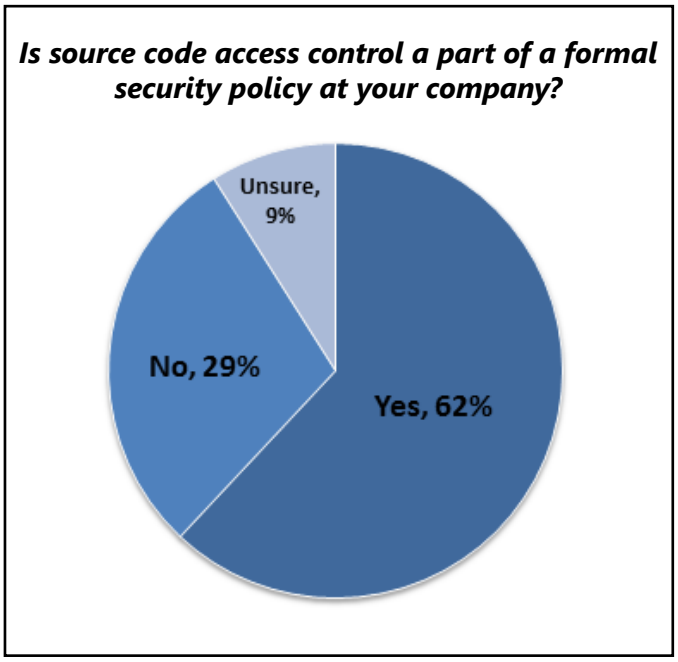
Figure II: Application Hardening Development Process

Figure II illustrates a modern approach to enterprise application hardening. The developer best practice is to have programmers indicate where protection transformations may or may not be appropriate. For example, reflection may confuse renaming transforms, high performance algorithms may call for less aggressive control flow settings or the elimination of string encryption, and the most sensitive code may call for the most aggressive protection settings. Obfuscation and hardening transformations are applied after the build and do not require access to source code. During the transformation, additional services such as compaction (stripping of unused code to reduce size) and linking (combining multiple DLLs into one to simplify distribution) can also be applied.

Further, all transformations should be captured (and previous transformations reused as appropriate) to support the many development scenarios outlined above (debugging, patch management, etc.). Again, all of this should be embedded within the integrated development environment to ensure continuous automation, transparency and quality.

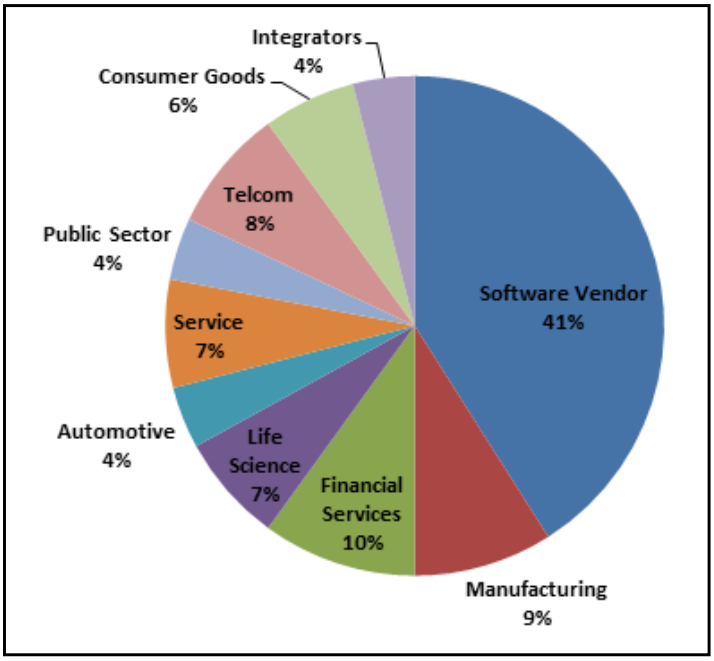
IT Controls and Governance, Risk and Compliance

Application hardening / obfuscation can be defined as a compensating[1], detective control to manage risks stemming from unmanaged source code distribution. These risks include an increased likelihood of system attack, theft of intellectual property, privacy violations and revenue loss through circumvention of usage and other metering enforcement. An obfuscation control includes the documentation of obfuscation processes, the specific risks that are being managed and the training/ communication activities that ensure obfuscation will be applied appropriately (e.g. not over- or under-used) and consistently (using approved technologies and practices).



Obfuscation is used to control the distribution of source code to communities that need access to binaries but not to source code. In environments like .NET and Java, where extraction of source code from intermediate code is simple and well-understood, obfuscation is a common antidote to this source code access control gap. A survey of 300 companies that included both SMB and global 2000 corporations reported three well defined risks connected to the uncontrolled distribution of source code. Intellectual property loss was cited slightly more often that the other two categories but revenue loss and the exposure of application security vulnerabilities were also clearly important.

Given the widely understood risks that stem from uncontrolled distribution of source code and the expanding adoption of .NET and Java platforms that dramatically lower the bar for source code extraction from binaries, it is no surprise that obfuscation is a common compensating control in these environments.



A second study looked at 2000 companies that had been identified as having adopted an obfuscation process within their application development processes.

Not surprisingly, the study found that obfuscation was most heavily adopted within industries that proportionately heavier reliance on software development. Software vendors, financial services, manufacturing telecommunications were the heaviest adopters. However, it is worth noting that obfuscation, like software development, cuts across all industries.

Some researchers suggest that 90 percent of software in use has security vulnerabilities in the application layer. As a result, cybercriminals are exploiting apps to compromise corporate infrastructure, install malware and exfiltrate data, and new app security regulations are emerging to combat this new threat.

[1] "Compensating" refers to the fact that obfuscation is a control that compensates for a gap in a preexisting control, e.g. access control for source code.

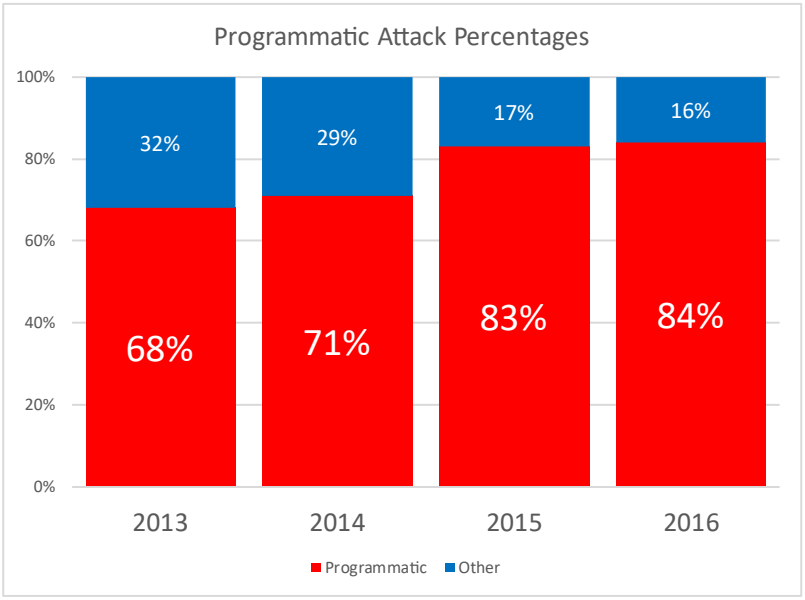
Here are just some of them:

- **OWASP** — The Open Web Application Security Project (OWASP) remains a go-to standard for application security. The OWASP Top 10 provides yearly reports on specific app threats, and encourages organizations to both compile a list of all third-party components used in software along with monitoring these components for potential breaches. In addition, OWASP's new resilience requirements mandate that mobile apps must be able to detect rooted or jailbroken devices, leverage multiple defense mechanisms and include multiple response types based on detected threats.
- **GDPR** — The new General Data Protection Regulation (GDPR) — covering all of EU and British citizen information, wherever it is stored — includes requirements for rapid provision of consumer data upon request by owners along with strict control of access, transmission and storage of this data. The result? Apps not built to reflect this standard could lead to sanctions, fines and serious damage to corporate reputation.
- **PCI DSS** — Payment Card Industry Data Security Standard (PCI DSS) also requires organizations to assess app security vulnerabilities using reputable outside sources and assign a severity ranking to these vulnerabilities. Best bet? Build in these collection capabilities from day one rather than trying to add-on security layers after the fact.
- **HITRUST CSF** — Health Information Trust Alliance Commom Security Framework (HITRUST CSF) — not only are commercial software products required to undergo security assessments before implementation, automated security controls are mandated for all applications.

Prevent Hackers From Using Their Favorite Tools

According to NIST's National Vulnerability Database, six vulnerability categories have grown from 68% to over 84% of the total number of reported vulnerabilities in just the past four years.

What these categories have in common are the tools hackers rely upon to probe, discover, and exploit these increasingly mainstream vulnerabilities. Specifically, hackers begin with application debuggers and reverse engineering tools to pick apart and modify applications. These "programmatic hacks" have led to many of today's most devastating application and data exploits.



Programmatic Hack
Bypass something
Code Execution
Gain Information
Gain Privileges
Memory Corruption
Overflow

Other
CSRF
Directory Traversal
DoS
File Inclusion
Http Response Splitting
SQL Injection
XSS

Sources: NIST National Vulnerability Database, Common Vulnerabilities and Exposures (CVE)

Stop Debugging

Anti-debugger controls can, when combined with code obfuscation (reverse engineering prevention), tamper defense, and other runtime checks, materially reduce application and data risk by impeding (if not outright preventing) the research typically required to identify and exploit application vulnbilities.

In each of the programmatic CVE categories listed above, a hacker likely began their attack by using some flavor of debugger to explore and manipulate a running instance of an application to bypass security, execute unauthorized code, elevate privileges, etc.

Effective anti-debugger controls mitigate these risks while minimizing potential development, quality, compliance, and/or performance side effects.

- **Debugger detection**: Debuggers come in a variety of flavors and packaging. An effective control will detect both managed and native debuggers.
- **Debugger defense**: Once an unauthorized debugger has been detected, a variety of pre-packaged real-time measures as well as application and runtime-specific tactics must be readily available for the developer to choose from. These can include throwing random exceptions, exiting the program, "bricking" the application permanently, generating custom log entries, etc.
- **Debugger notifications**: In addition to real-time defense and mitigation, it is valuable to emit an alert or notification that can initiate an operational response including isolating the device or even the local network running the compromised application.
- **Implementation**: Real-time counter measures and runtime reporting represent a new category of application behavior that must be specified, documented, and tested. Minimizing the amount and complexity of this incremental effort will often be the determining factor as to how consistently and effectively these controls are applied.
- **Quality and support**: The mission-critical nature of these controls mandate the highest levels of quality, transparency, and support to ensure that the controls do not create more risk than they mitigate.

Prohibit Rooted Devices

Regulators, standards bodies and IT auditors have become increasingly likely to recommend complete prohibition of rooted Android devices in production environments. As the 2017 PCI Mobile Payment Acceptance Security Guidelines state, "Bypassing permissions can allow untrusted security decisions to be made, thus increasing the number of possible attack vectors."

It is only natural that the apps themselves rise up to act as a ubiquitous governance, risk, and compliance management layer – preventing, detecting, responding, and reporting on threats - including those posed by unauthorized rooted devices.

The PCI Mobile Payment Security Guidelines recommend the following (4) controls be in place:

Section 4.3 Prevent Escalation of Privileges

"Controls should exist to prevent the escalation of privileges on the device (e.g., root or group privileges). …

(1) the device should be monitored for activities that defeat operating system security controls (e.g., jailbreaking or rooting) and, when detected,

(2) the device should be quarantined by a solution that removes it from the network, removes the payment-acceptance application from the device, or

(3) disables the payment application.

(4) Offline jailbreak and root detection are key since some attackers may attempt to put the device in an offline state to further circumvent detection."

Therefore, hardened apps should enforce a policy where they will not run on a rooted device. They should trigger real-time responses to running on a rooted device including notifications, auto-exit, and even a permanent disabling of the app (a quarantine or bricking).

Stop Reverse Engineering

The goal of application hardening is to prevent a specific category of access control violation: the reverse engineering of binaries to gain unauthorized access to source code. It is designed to achieve this goal while accommodating the unique requirements that binaries present as a unique data type-requirements that create potentially serious problems for traditional access control strategies.

In order to be effective, hardening and obfuscation transforms must make reverse engineering materially more difficult without altering an application's behavior or access control profile. In practice, application hardening is a general term that refers to a collection of techniques that meet these dual sets of requirements.

Control	Implication for Binary Access
Restrict Read/Execute Access	• Cannot find or execute binary making this approach ineffective for groups that need to execute binaries • Role and evidenced based security structures may effectively restrict read/execute access in circumstances where the binary is controlled/not distributed
Encryption	• Effective for transit and storage only – binary must be decrypted to execute which returns binary to original state • Including native code to decrypt managed code at runtime makes validation of the application impossible, violating IT controls within many enterprises • Encryption offers a high degree of protection related to its objective
Hardening / Obfuscation	• Does not restrict execution of binaries because it is independent of access control settings • Is effective/persistent on disk, during transit and execution • Generates 100% valid intermediate/managed code • Effectiveness of techniques vary depending upon the nature of the transforms

The Obfuscation Approach

Obfuscation works by removing the context that humans and decompilers use to understand the functionality of a program. Consider the following two versions of the same message, responding to the question, "how many tickets should I buy?"

Email "in the clear"	Pseudo "Obfuscated" email
From: Mark To: Bill Subject: RE: how many tickets should I buy? Bill, I already bought 15, so all we need is 7 more. Looking forward to seeing all of you tonight as we agreed.	To: Bill From: Mark 7

Both messages can be delivered and answer Bill's question, but the "obfuscated" version has been stripped of its context. An unauthorized reader would not know that this is a reply to a question or that Bill is being asked to purchase 7 tickets or that 15 other tickets have already purchased and that the entire group will be meeting later that night.

In simple terms, this is how obfuscation works – by applying multiple transformations that strip away information and alter structure to make the code less understandable to humans and decompilers while preserving integrity and behavior.

App Hardening Side Effects, Implications and Best Practices

Obfuscation and app hardening transformations are, by design, intended to profoundly alter the way a binary appears. Effective obfuscation offers significant security and performance benefits. However, obfuscated binaries do not discriminate; they are as opaque to "good guys and good programs" as they are to "bad guys and bad programs."

Anticipating and accommodating the expected consequences of including obfuscated binaries into a complete application development lifecycle ensures that obfuscation benefits will be maximized and that any associated complexity or risk is minimized.

Debugging

The purpose of obfuscation is to make it as difficult as possible to connect a binary to its original source code. Without a well-defined and reliable "de-obfuscation" tool that enables development, QA and support to unwind obfuscated binaries, the debugging process can be complicated or even impeded by obfuscation.

Performance

Obfuscation has the potential to alter an application's performance footprint. Control flow obfuscation can, in some instances, degrade a finely tuned algorithm. Conversely, identifier renaming may reduce the size – and therefore the load time – of an entire application. Developers need a means to selectively test and apply obfuscation transformations at a fine grained level within an application to ensure that they can achieve the maximum benefits of obfuscation while eliminating the potential for unexpected side effects.

Size

Obfuscation can also alter the size of an application. The insertion of a string decryption method will increase the overall size of an application. However, the most common result of obfuscation is to reduce the size of an application. In addition to the shortening of identifier names, some obfuscators will also "prune" those portions of applications and third party libraries that are included but never called. Effective use of pruning and identifier renaming typically can reduce the size of an application by 30%-40% and sometimes as much as 70%.

Patch Management

Renaming decisions and other decisions that are potentially unique to each obfuscation process must be carried forward into subsequent builds to ensure compatibility. This also applies to patches and components that may be built and obfuscated at different times and locations. An effective integration into the IDE combined with the appropriate obfuscation utilities simplifies the management of this touch point within the development lifecycle.

Best Practices

The most effective approaches to assimilating obfuscation into the development lifecycle combine tool and process extensions with a modest amount of awareness. This section summarizes the best practices that sophisticated development organizations have found to be effective.

Declarative Obfuscation

Developers know their code better than anyone else. The most efficient and effective means to ensure that source code is obfuscated correctly without unintended or unexpected side effects is to provide a set of attributes that permit developers to specify (declare) which obfuscation transformations should (or should not) be applied to any portion of their code. Ideally, these attributes are recognized by their IDE and can be tracked, validated and shared. The benefits of this approach include:

- Individual developers of a module can specific where to apply specific transformations anywhere within their code, e.g. renaming, string encryption, control flow obfuscation, etc.
- Elimination of configuration files at build time
- Component providers and other developers of reusable components can embed obfuscation requirements into their code for others to use

Distributed and Secure Debugging

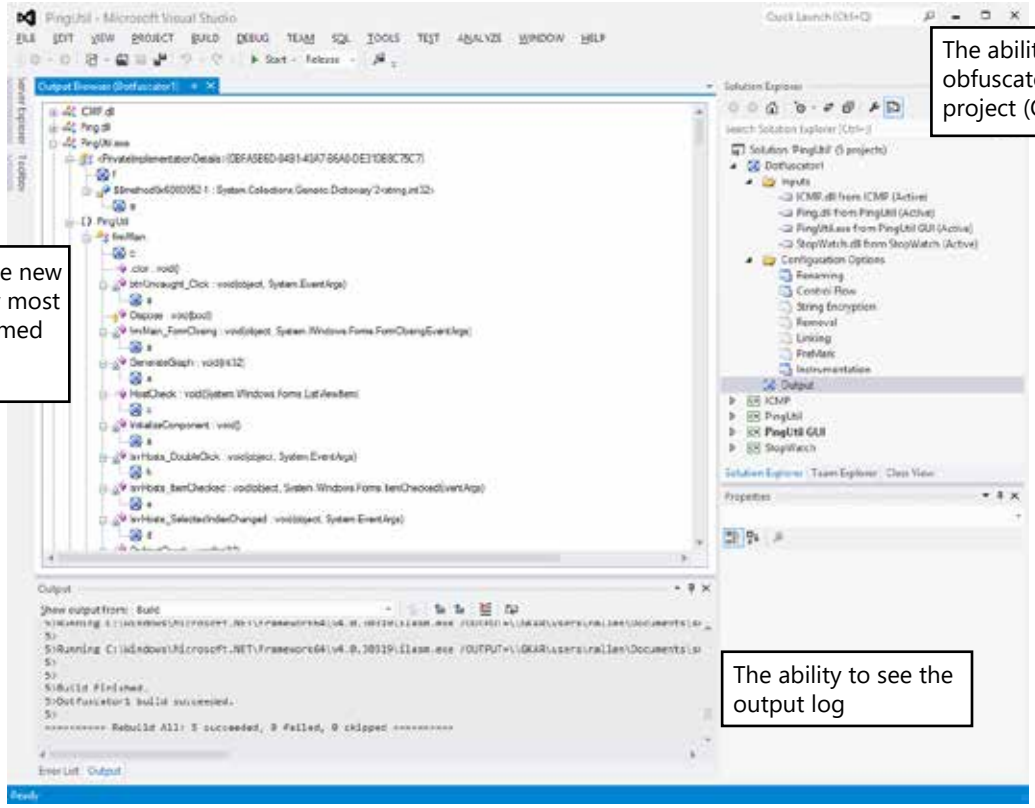
Developers, QA and support personnel all have occasion to debug binaries. Virtually the only stage in a development process that does not typically debug binaries is the manufacturing or build function. An obfuscation solution must provide a cost-effective, lightweight and low maintenance utility to support distributed debugging of obfuscated binaries without requiring the complete obfuscation solution or access to the build environment. A de-obfuscator should accept the debugging symbol files for obfuscated applications and use this information to “unwind” the obfuscated assembly to simplify the debugging process. The benefits to this approach include:

- Existing development, quality and support functions and processes are uninterrupted.
- Build environments are not compromised by debugging activities not normally supported.
- Heavy-weight obfuscation solutions do not need to be widely deployed or administered.

IDE Integration

An integrated obfuscator should work as part of the preferred integrated development environment. For example, a Visual Studio solution should be recognized as a native project type, accept input files from one or more other Visual Studio Projects (such as C# or VB.NET projects) and be able to resolve all dependencies and privileges automatically.

This screen capture illustrates an example of an IDE with integrated obfuscation capabilities. The following capabilities can be seen:



The benefits of IDE integration include:

- Reduced build errors
- Improved tracking of previous obfuscation maps to support debugging and patch management
- Increased automation, process transparency and quality

Patch Management

Patch management is of particular interest to enterprise development teams maintaining an integrated application environment. By generating name mapping records during an obfuscation run, obfuscated API names can be reapplied and preserved in successive runs. A partial build can be done with full expectation that its access points will be consistently renamed across builds. The benefits of this approach include:

- Patch generation and distribution processes are uninterrupted
- Modules that have not been modified and have been distributed do not need to be reobfuscated or redistributed

Continuous Integration - DevSecOps

Organizations that have incorporated a continuous integration development approach should include the hardening step for their developers doing unit testing. This is particularly true when developers are taking advantage of

declarative obfuscation as these developers may want to evaluate the impact of applying specific obfuscation transformation on their work. The benefits of this approach include:

- Developers can assess the impact and value of obfuscation on their code before checking their work in.
- Build, linking and other dependencies can be identified and resolved earlier in the development lifecycle.

The Enterprise Application Hardening Process

The app hardening process integrates obfuscation and runtime checks into the application development lifecycle specifically supporting the development, testing, integration, manufacturing, application support and patch management phases of development.

Application Lifecycle Integration

Organizations that do not distribute source code with binaries obfuscate those binaries to reduce the risk of system attacks, Intellectual Property theft and revenue loss due to metering circumvention.

Properly applied, obfuscation offers a means to reduce risk that arises from the uncontrolled distribution of source code.

The figure on this page illustrates the multiple touch points that a mature app hardening solution must maintain in order to reduce risk within a well defined application development process.

1. **Declarative obfuscation controls**: Developers know their code the best. Using the XML-defined attributes can specify which transformations to apply, how they should be applied and where within an application to apply them. Declarative obfuscation supports an arbitrary level of granularity enabling the maximum degree of obfuscation with the minimum degree of side effects.
2. **Configure & apply obfuscation transformations:** SCM or build managers configure and execute the final obfuscation transformations. These capabilities should be extensible to include support for unit testing, continuous integration processes.
3. **De-obfuscation for efficient debugging**: A distributed capability that can unwind obfuscated binaries providing an effective connection back to the original source code. Debugging is rarely effective when restricted to build machines.
4. **Incremental obfuscation transformation**: The ability to obfuscate patches that both prevent access to source while maintaining compatibility with previously obfuscated and distributed binaries.

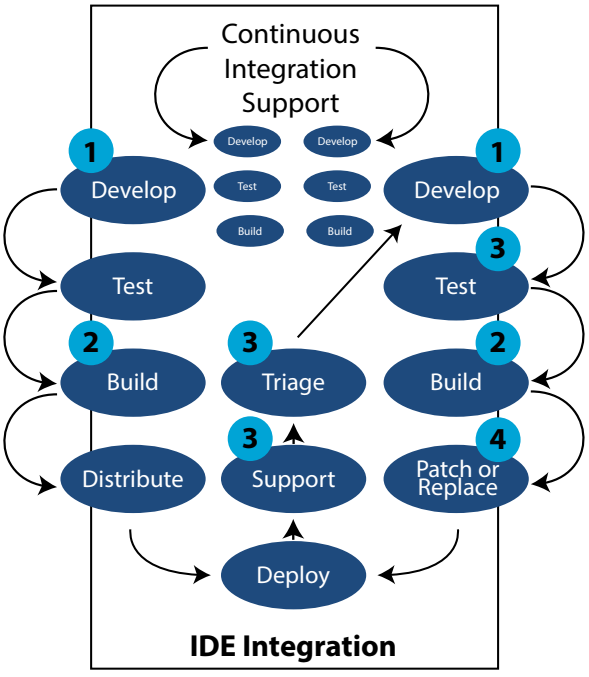


Figure III: Multiple touch points of app hardening

Integration with preferred IDE's and deployment flexibility to support today's distributed work environments are the final elements of an effective obfuscation process.

DevSecOps

The next step in establishing an effective application hardening control is to instantiate a consistent means of assessing how and when to apply hardening and obfuscation. Once the specific integration points between the development lifecycle and the hardening process have been identified (see the previous section), integration with a development methodology such as Agile software development can be used to ensure that this control is consistently and appropriately applied.

Agile is a conceptual framework for undertaking software engineering projects. Agile methods emphasize real time communication, preferably face-to-face. Figure IV illustrates how obfuscation may be integrated into "Iteration 0" – the planning stage for a development project.

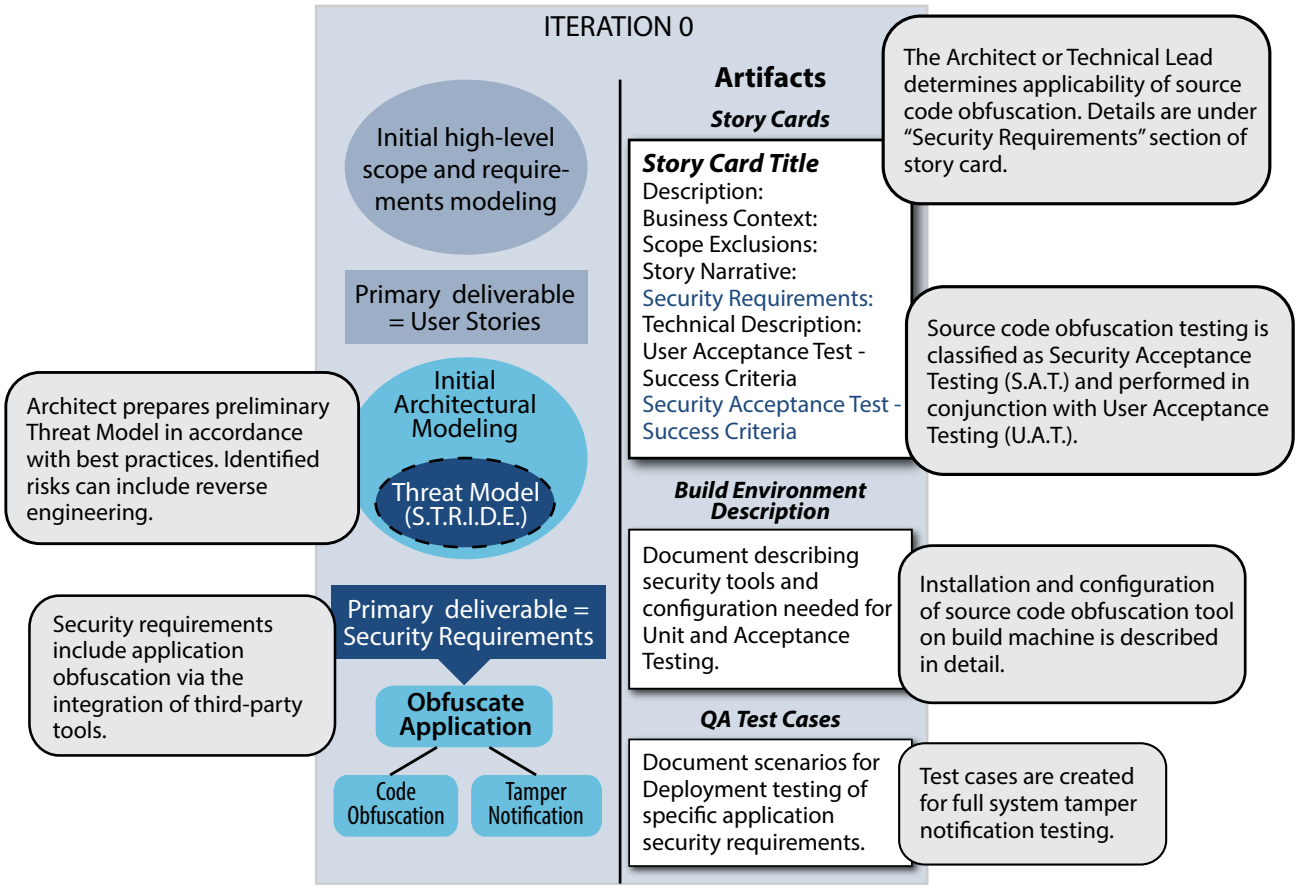


Figure IV: Integrating obfuscation into Iteration 0 of the Agile development process

In iteration 0 of the Agile approach, the development project is set up and basic planning is carried out. The risks from reverse engineering must be factored into the broader threat modeling and risk management frameworks that guide and inform design and development priorities.

Once the level of risk has been gauged, strategies to mitigate these risks including obfuscation and tamper notification can be included as development requirements. Further, the details on which tools, how they will be deployed and their effectiveness measured are all included in Iteration 0, the project planning phase.

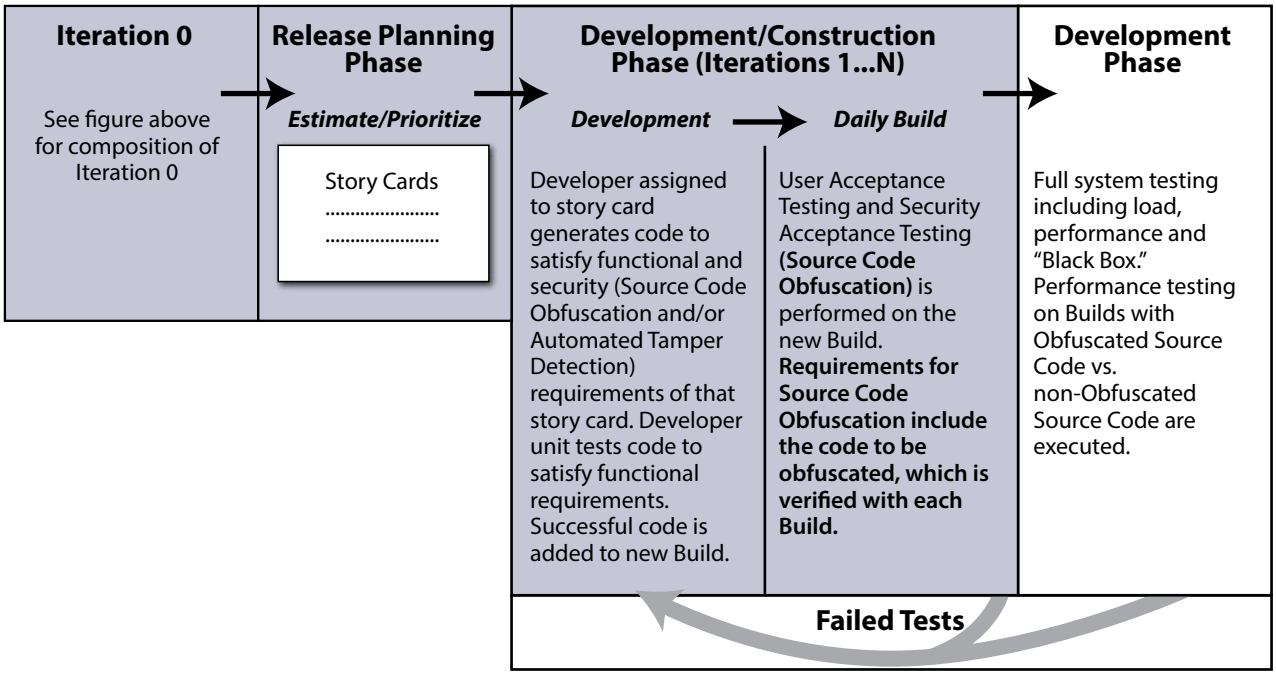


Figure V: Integrating obfuscation into Iterations 1 – N of the Agile development process.

In iteration 1, after additional planning for future cycles, the team carries out some development activity. In the subsequent iterations (2 to N-1), the team does all of this, plus tests the development work of previous iteration(s) and adjusts the goals and scope of the solution based on the feedback and testing. Finally, in iteration N, the team finishes testing and releases the solution.

Figure V illustrates how developers, QA and build managers incorporate, verify and assess the effectiveness and backward compatibility of obfuscated code. Further, coverage, e.g. the percentage of code that should be obfuscated versus the percentage of code that is actually obfuscated, can also be measured.

The net result of this integration into an established development process is that obfuscation (including its role as a detective control reporting on application tampering) can be reliably and efficiently incorporated into mature and scalable development environments. Documenting, automating and continuously improving the obfuscation process assures maximum risk mitigation while minimizing the potential to introduce its own risk and complexity.

Application Hardening as a Compensating Control

Organizations in every industry are incorporating app hardening and shielding into their IT control practices in order to satisfy the many regulatory, governance and other compliance obligations they face. The goal of obfuscation is to produce programs that are materially more difficult for unauthorized individuals to understand or modify. Obfuscation is a compensating control filling a well-understood access control gap for source code in Java and .NET environments.

Assessing the Need for App Hardening / Obfuscation

Who Needs to Know Your Source Code?

There is a near universal consensus[3] that access to information should be granted on a "need to know" basis as part of a "least privileged" access control governing policy. Given the ease of extracting source code from binary in contemporary environments like Java and .NET, the tautology of unprotected binary = source is now a fact. The question to evaluate is who has a "need to know" your source code and is that community smaller than the

[3] COBIT, ITIL and COSO control frameworks as well as guidance from the Institute of Internal Auditors

community that has read access to the associated binaries? One extreme use case is the open source community. Clearly, obfuscation holds no value as the source code is widely available by design. Conversely, many software vendors, financial service providers, manufacturers, defense contractors, etc. rely upon software to meter usage (access, billing, etc), protect privacy and ensure operational continuity and along the way they embed into their source proprietary business rules, programming logic, etc. All of these typically represent material value and their loss or dysfunction represents material risk.

A Layered Approach to Application Security

One hundred percent industry consensus around application security and shielding is impossible to achieve, but organizations like OWASP are trying. It recently released new protection guidelines around how mobile apps handle, store and protect sensitive information. For example, its Mobile Application Resilience Requirements now recommend that apps:

- Detect and respond to the presence of a jailbroken device
- Prevent or detect debugging attempts
- Include multiple defense mechanisms
- Leverage obfuscation and encryption

Implementing these guidelines starts with solid app development best practices such as writing secure code, only using authorized APIs and regularly testing apps prior to deployment. Application hardening and shielding, meanwhile, makes your application more resistant to intrusion, inspection, tampering and reverse engineering. In addition, it may also collect data to both identify attack vectors and help prevent future attacks. It is a critical link once applications go live in untrusted environments.

General app security is critical in a world consumed by software. Application security testing and vulnerability patching are a well-known step along the way. Application hardening and shielding, meanwhile, is another critical component for high value application that run in untrusted environments. These include any apps that access sensitive information, gate access to value, or contain intellectual property.

Demonstrating "Due Care" to Minimize Liability

Risk management and security programs offer more than a return on investment; they are part of a broader obligation to customers, employees and investors to operate a sound business. Neglecting that broader obligation can be interpreted as a breach of responsibility and carries with it a potentially serious liability. Auditors, regulators and the courts look to standards of behavior to measure how effective organizations are in preventing, remediating, monitoring and continuously improving compliance and security programs. Organizations that can demonstrate an effective and sustained program to mitigate risk, liability can be reduced and sometimes even avoided.

Due care is demonstrated by erecting an ascending series of barriers that

I. Discourage the opportunistic

II. Impede the malicious

III. Deter the skillful

IV. Identify and enable punishment of the successful

App Hardening is a way to accomplish (I) and (II) above. Failing to do that, it increases the difficulty of (III) and may help achieve (IV).

BUYER GUIDE FOR IN-APP PROTECTION

1 ASSESS RISK

- Does app have intellectual property?
- Does app gate access to value?
- Does app access private information?
- Is app subject to regulation?
- Does app run in an untrusted environment?

2 ESTABLISH OBJECTIVES

- Protect against IP theft
- Mask vulnerabilities
- Protect against data theft
- Hinder tampering and fraud
- Reduce piracy

3 IDENTIFY POTENTIAL SUPPLIERS

Evaluate fit to requirements by:

- Features
- Services
- Viability
- Focus

4 SELECT SUPPLIER CANDIDATES

Review:

- Longevity and viability of company
- Support offered
- Reputation
- Internal references
- External references

5 CANDIDATE DUE DILIGENCE

Evaluate:

- Features
- Support services
- Ease of integration
- Validation against requirements

6 PROCUREMENT AND IMPLEMENTATION

- Agree on licensing terms
- Add layers of protection based on requirements
- Plan and integrate into your DevSecOps process

7 ADOPTION

Create or review corporate-wide security policy for all apps worth protecting.

Why Include An app Hardening Control?

- A layered approach to security is an acknowledged best practice,
- The cost of implementing an app hardening solution represents a tiny fraction of the investments into application development and
- The operational, financial and regulatory risks that stem from system attacks, IP theft and revenue loss can be catastrophic.

A well implemented obfuscation process offers a low cost, low maintenance control for material risks stemming from access to source through Java and .NET binaries.

Enterprise Application Hardening Evaluation Criteria

Organizations in virtually every industry are incorporating obfuscation into their application development lifecycle process. Application security, IT Governance and risk management programs have created a greater awareness of the need to protect against the uncontrolled distribution of source code and the unrestricted use of tools hackers rely upon to probe, discover, and exploit these increasingly mainstream vulnerabilities. This increased awareness has also produced a more comprehensive understanding of what is required to effectively use application hardening within broader security, IT governance and risk-based initiatives.

Every organization must develop and maintain an appropriate set of process and controls to manage these issues. While this broad requirement is essentially universal, there is no "one size fits all" solution for application security. Every organization must assess their specific needs and risks in order to settle on an appropriate set of processes, controls and technologies. The following five high-level questions can be used to capture the essential characteristics of an obfuscation solution to ensure a proper and effective fit with the needs and requirements of a business.

Questions to Ask an Obfuscation Solution Provider

1. How does their obfuscation solution fit into your company's application lifecycle and continuous integration framework?
   - Can developers markup code to ensure maximum obfuscation without loss of control?
   - What is the solution for debugging and managing patch releases for obfuscated binaries?
   - Is their a distributed, yet secure, deployment model that can accommodate support, operations and QA as well as your collaborative development processes?
2. What specific obfuscation, compaction and watermarking technologies are available?
   - Are they unique? Patented?
3. Is there support for more than one platform?
   - Is .NET and Java supported? Is the technology integrated into your IDE?
4. Can the solution be configured to accommodate your organization (size and distribution)?
5. What kind of support, upgrades and quality assurance is guaranteed?

The "right" answers to many of these questions are obviously dependent upon how and where the solution is going to be deployed. Still other responses will be assigned more or less weight for the very same reasons. This variability of acceptance criteria and value applies both across organizations and within a single organization over a prolonged period of time.

Given the potential lifespan of applications and the distributed and heterogeneous development practices that are emerging, it is particularly critical that a protection solution provider must develop their products, organize their business and align their strategy to ensure sustained technology leadership, flexible process and control capabilities and reliable support infrastructure. See the following graphic for an representative buyer's journey.

Building Best Practices for Protecting your Apps and Data

It's one thing to know that software and applications security regulations are starting to standardize and solidify — and another to build out best practices that deliver natively secure applications. Start with the big questions when you're designing new applications: Can security requirements be tested? Reliably? Can the results be accurately measured? Are security outcomes and processes clear and unambiguous? Do they meet relevant requirements for distribution? For example, if you're designing an application to handle global, retail mobile traffic does it comply with PCI-DSS and GDPR expectations?

This means building in automated security controls capable of detecting common threat vectors such as SQL injection, session hijacking and credential spoofing, in addition to reporting potential breach incidents in real-time. Other strong options include the use of post-compile injection to provide anti-root, anti-tamper and runtime checks — these are especially useful for critical software already deployed across corporate networks, since you aren't on the hook to recode critical functions from the ground up.

Today, application hardening and layered security measures are recognized as a critical feature of overall IT compliance. The result? Regulation is on the rise. Make sure you are compliant: Get familiar with applicable standards and implement app development best practices to boost basic security for all your apps worth protecting.

The empirical evidence is indisputable; tens of thousands of development groups have made the decision to include application hardening / obfuscation in their development process. Loss of revenue, operational disruption and loss of intellectual property are real threats that can arise from the uncontrolled distribution of source code. This begs the question as to why we don't hear more about this risk and why, if the potential damage can be so great.

About PreEmptive

Through a combination of binary hardening, detection, defense and alert controls, PreEmptive offers material risk mitigation against the following threats:

- Intellectual Property Theft
- Revenue Loss
- Trust and Brand Damage
- Fraud and Unauthorized Access
- Confidential Data Theft

PreEmptive's Protection tools hit the sweet spot between cost, convenience and functionality by helping you protect and secure your apps in a smarter way. Our layered security and obfuscation protection is directly infused into your .NET, Java, Android and iOS applications and that means we do not require an agent or changes to your end user's computer/device or network.

PreEmptive has thousands of corporate clients in over 100 countries spanning virtually every industry As a result, PreEmptive has established itself as the clear choice for any organization serious about hardening and protecting their desktop, mobile, server and embedded applications while insuring that their build processes, debugging efforts and app performance are not adversely impacted.

To learn more, email solutions@preemptive.com or call 440.443.7200.

PreEmptive
Visit us at www.preemptive.com

Worldwide Headquarters
767 Beta Drive, Suite A
Mayfield Village, OH 44143
Phone +1 440.443.7200
solutions@preemptive.com

European Headquarters
140 bis rue de Rennes
75006 Paris, France
Phone +33 01.83.64.34.74
eurosolutions@preemptive.com

Find us: