PreEmptive Solutions

Enterprise Obfuscation

Technology, Process and Control

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:

- A summary of obfuscation technical capabilities and considerations,
- Identification of common system configuration management dependencies,
- Application lifecycle workflow and process requirements,
- Development, quality assurance and support best practice,
- Guidance as to when enterprise obfuscation can be an effective IT control,
- Enterprise obfuscation evaluation criteria,
- Guidance for incorporating obfuscation into an Agile process,
- Extending obfuscation from reverse engineering prevention to tamper detection and notification, and
- Synergy with other post-build functionality including instrumentation, optimization and linking.

Table of Contents

The Three Dimensions of Enterprise Obfuscation

Reverse engineering has become a common practice for support, training, and debugging of .NET and Java applications. However, unmanaged access to application source code can also pose material risks including Intellectual Property (IP) theft, application vulnerability exposure, and software piracy. For those organizations where these risks must be managed, there is really only one safe option: obfuscation.

At its core, obfuscation is defined as a collection of transformations that are applied to compiled applications that make reverse engineering materially more difficult for people and machines but do not alter the behavior of the obfuscated application. However, the heightened emphasis on application security, compliance and development best practices has rendered this definition inadequate. A complete definition must have three dimensions: technological, as a development process, and as an IT control.

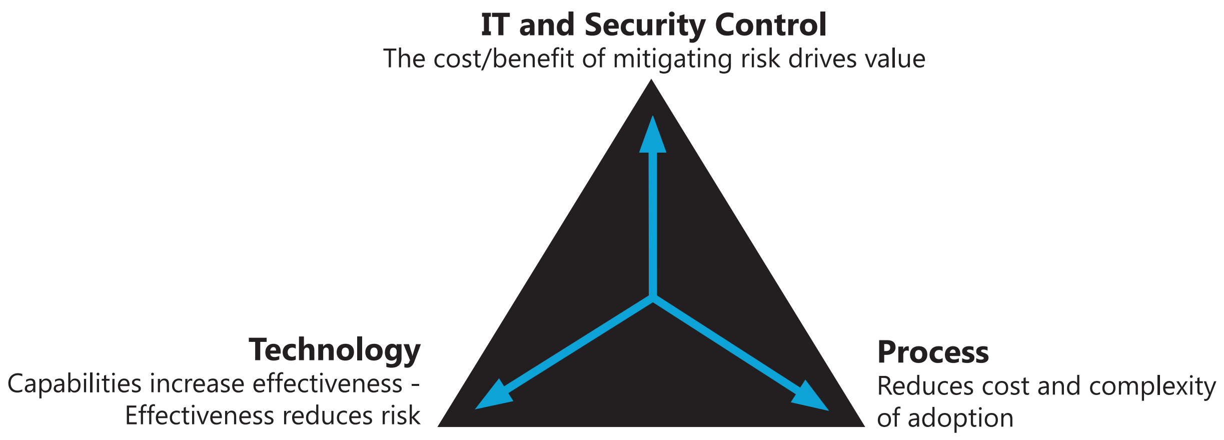

IT and Security Control
The cost/benefit of mitigating risk drives value
Technology
Capabilities increase effectiveness - Effectiveness reduces risk
Process
Reduces cost and complexity of adoption

Figure I: The three dimensions of enterprise obfuscation

Technology

Obfuscation 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.
- **Other:** there are numerous other techniques including metadata stripping, application watermarking, etc. that raise the bar for reverse engineering above what is required to reverse engineer native code.

Process

The process of obfuscation has emerged as the pivotal element in driving the viability and the value of obfuscation. Without a well-defined and integrated obfuscation process, the complexities and risks introduced by obfuscation may ultimately outweigh the perceived benefits that it promises. Obfuscation can complicate debugging, patch generation and management, distributed development practices and the reuse of libraries, components and web services. However, tight integration with development platforms (such as Visual Studio), the Enterprise Obfuscation: Technology, Process, and Control inclusion of tools and utilities that can unwind and/or reuse obfuscation transformations and, lastly, integration with operations management platforms can mitigate these potentially costly side-effects.

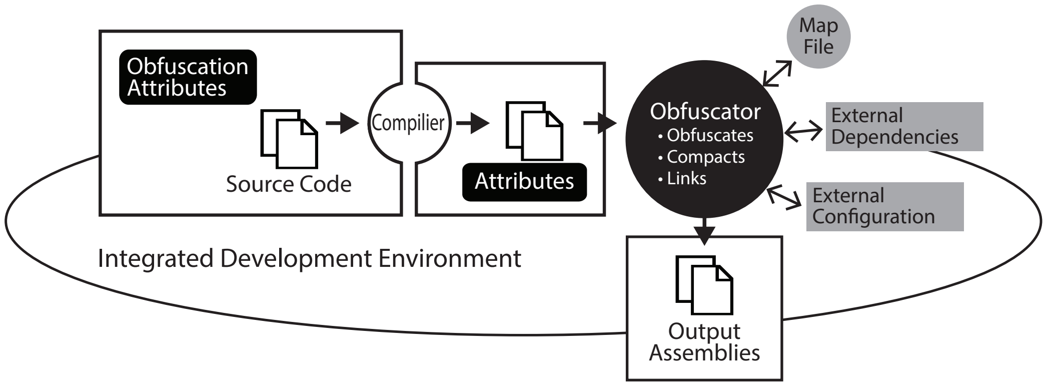


Figure II: Obfuscation development process

Figure II illustrates a modern approach to enterprise obfuscation. The developer best practice is to have programmers indicate where obfuscation 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 obfuscation settings. Obfuscation transformations are applied after the build and do not require access to source code. During the obfuscation 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.

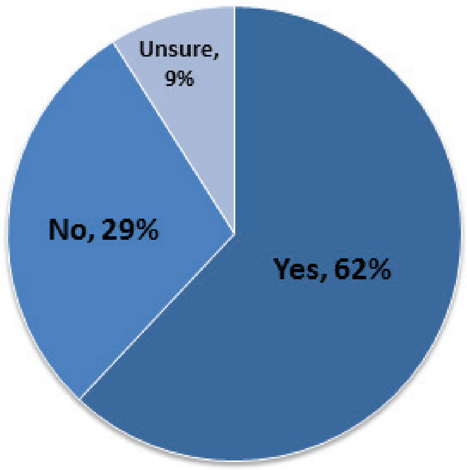
Further, all transformations should be captured (and previous transformations reused as appropriate) to support the many development scenarios outlined above (debugging, patch management, etc.). Again, all of this should be embedded within the integrated development environment to ensure continuous automation,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

IT Control

Obfuscation can be defined as a compensating[1], detective[2] 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 international companies that included both SMB and global 2000 corporations reported three well defined risks connected to the uncontrolled distribution of source code. Intellectual

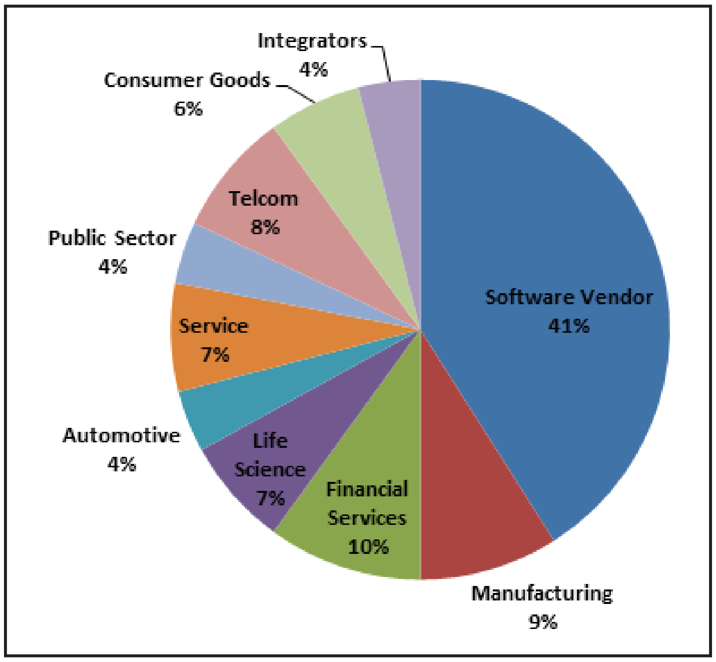
Is source code access control a part of a formal security policy at your company?



[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.

[2] "Preventative" refers to the fact that obfuscation is a control that prevents an incident (reverse engineering) from occurring. Controls either prevent or detect incidents.

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.

The following chapters explore the three dimensions of obfuscation with an eye towards helping organizations assess their need for obfuscation and to provide a consistent set criteria that can be used to effectively and efficiently reduce risk.

The Fourth Dimension of Enterprise Obfuscation: Tamper Notification

If you invest in fire prevention – don't you also want fire detection? So, if an organization cares enough to invest time and resources into reverse engineering prevention, wouldn't that same organization benefit from reverse engineering detection?

Four simple steps can extend obfuscation to provide tamper notification services enhancing obfuscated applications to be able to self-diagnose and report on tampering whenever and wherever it may occur. Figure III illustrates these steps.

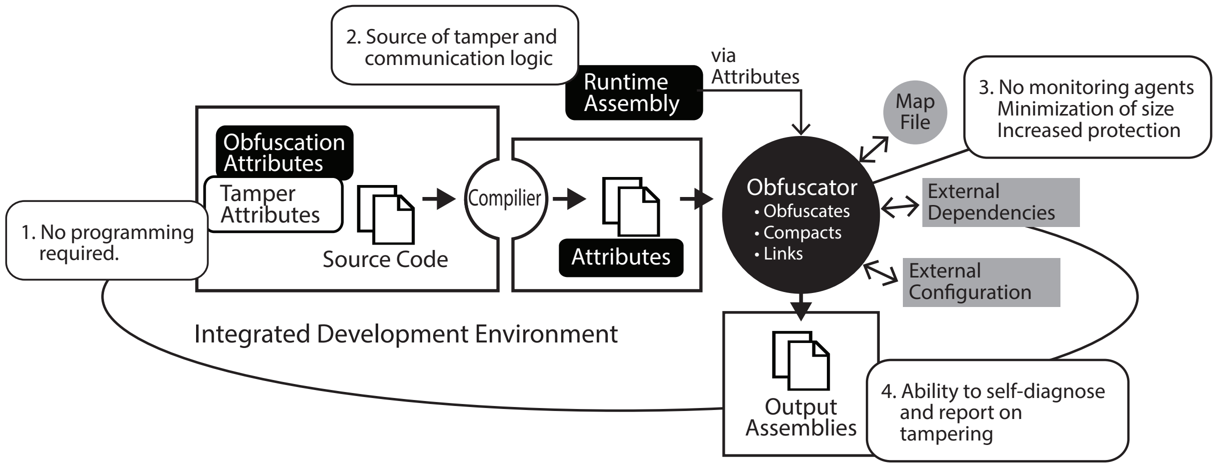

2. Source of tamper and communication logic
Runtime Assembly
via Attributes
Map File
3. No monitoring agents Minimization of size Increased protection
Obfuscation Attributes
Tamper Attributes
Compilier
Source Code
Attributes
1. No programming required.
Obfuscator
• Obfuscates
• Compacts
• Links
External Dependencies
External Configuration
Integrated Development Environment
Output Assemblies
4. Ability to self-diagnose and report on tampering

Figure III: 4D obfuscation development supporting Tamper Notification

Step 1: Extend custom attributes (or annotations) to enable developers to indicate program inflection points enabling the obfuscator to instrument the obfuscated binary, e.g. to insert additional logic to detect tampering. There is no programming required. This reduces the effort required to implement this service and reduces training overhead.

Step 2: Tamper detection and communication logic is delivered as a part of the obfuscator in a runtime library. This logic is then injected into the compiled and obfuscated application after the build. This improves both the quality and the consistency of transmitted data. How and where this logic is injected is constrained by the developers using the attributes/annotations in step 1.

Step 3: The obfuscator uses linking to weave the additional DLLs into the final binary and compaction to minimize any potential increase in application size that may result from the addition of Tamper.

Step 4: The result is a single, agent-less application that is able to diagnose its own tampering and report back to both its developers and its enterprise owners.

Those who lose sight of the fact that the organizing principle behind obfuscation is the requirement to manage risk also undervalue the importance of the obfuscation process. The result is a one dimensional view of obfuscation as a finite set of technology limited to the transformation of application binaries. However, the emergence of Service Oriented Architecture (SOA) and Software as a Service (SaaS) combined with a growing recognition that the most effective IT controls must bridge the development lifecycle and operations management has made it possible, for the first time, to prevent reverse engineering and to detect tampering when it should occur.

Obfuscation Technology

The goal of obfuscation is to prevent a specific category of access control violation: the reverse engineering of binaries to gain unauthorized access to source code. Obfuscation 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, obfuscation must make reverse engineering materially more difficult without altering an application's behavior or access control profile. In practice, obfuscation 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
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 obfuscation 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.

Representative Obfuscation Techniques

The following techniques are applied to compiled binaries – not to source code – to generate a new set of one or more binaries that exhibit equivalent behavior.

Identifier Renaming

Program identifiers including classes, interfaces, methods, fields, method parameters, generic type parameters, etc. are renamed to remove all meaning. "GetPayroll" becomes "a". Advanced renaming capabilities overload these new names on a massive scale by tracking the scope of each identifier and reusing names on multiple identifiers where there is no overlap. This increases security because every identifier must be individually analyzed since every use of "a" may in fact be an entirely different identifier renamed to the same new name.

Control Flow Obfuscation

High level source flow is compiled into a sequence of instructions that closely mirrors the original source. Control flow obfuscation transforms the original instruction sequence into a logically equivalent sequence with the intended side effect that the new sequence will not decompile back to the original source. Often, decompilers will output incorrect code, crash, or terminate with an exception.

Metadata Removal

Not all metadata is required to execute an application. For example, on the .NET platform, some properties, events, method parameter names, and custom attributes can be stripped, removing relevant information about the development process and developer intent without impacting execution.

String Encryption

While encrypting an entire application has well understood limitations, encrypting strings within an obfuscated binary at obfuscation time adds another effective layer to the obfuscation approach. Decryption is accomplished by embedding a decryption method call immediately after every string load instruction and replacing the encrypted string on the stack with the "clear" string at runtime. This provides the benefits of encryption while still offering significant protection during runtime.

System Configuration

The following components represent the basic building blocks of an enterprise obfuscation function 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.

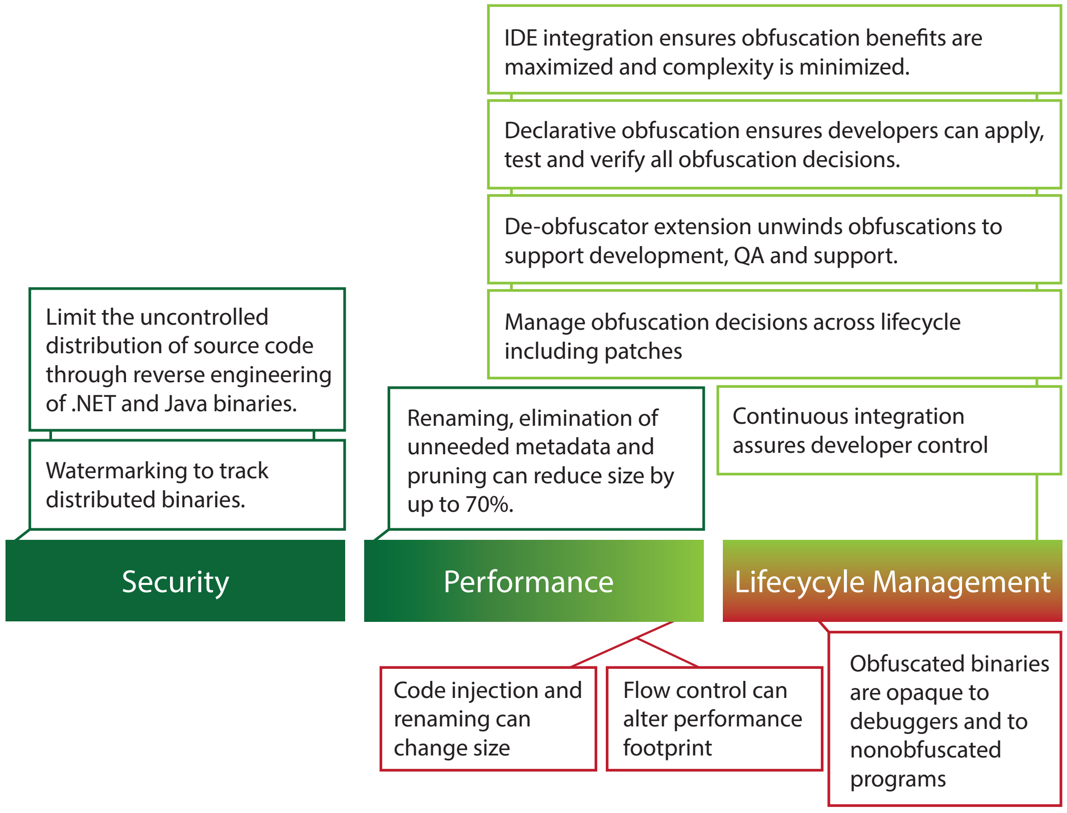
- **Input Programs** - Any .NET exe or DLL (MSIL)
- **External Dependencies** - Dependent DLLs
- **External Configuration** - XML configuration file
- **Obfuscator** - Post Compilation Protection System
- **Output Programs** - Smaller, harder to reverse engineer .NET exe or DLLs (MSIL)
- **Build Maps** - XML file containing information on how renaming was done and what attributes were applied.

Obfuscation Side Effects, Implications and Best Practices

Obfuscation 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.

The control options and risks associated with source code distribution and obfuscation are complex. To provide an overview, the following graphic summarizes the benefits, implications and potential side effects of obfuscation on security, performance and lifecycle management. Dark green indicates a positive feature or outcome, red highlights a caution that should be assessed before selecting or implementing a specific obfuscation solution and light green represents functionality that can compensate for and minimize the complexity and risk of incorporating obfuscation into the development lifecycle. The features and outcomes are discussed in the sections following the illustration.


IDE integration ensures obfuscation benefits are maximized and complexity is minimized.
Declarative obfuscation ensures developers can apply, test and verify all obfuscation decisions.
De-obfuscator extension unwinds obfuscations to support development, QA and support.
Manage obfuscation decisions across lifecycle including patches
Limit the uncontrolled distribution of source code through reverse engineering of .NET and Java binaries.
Watermarking to track distributed binaries.
Renaming, elimination of unneeded metadata and pruning can reduce size by up to 70%.
Continuous integration assures developer control
Security
Performance
Lifecycle Management
Code injection and renaming can change size
Flow control can alter performance footprint
Obfuscated binaries are opaque to debuggers and to nonobfuscated programs

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

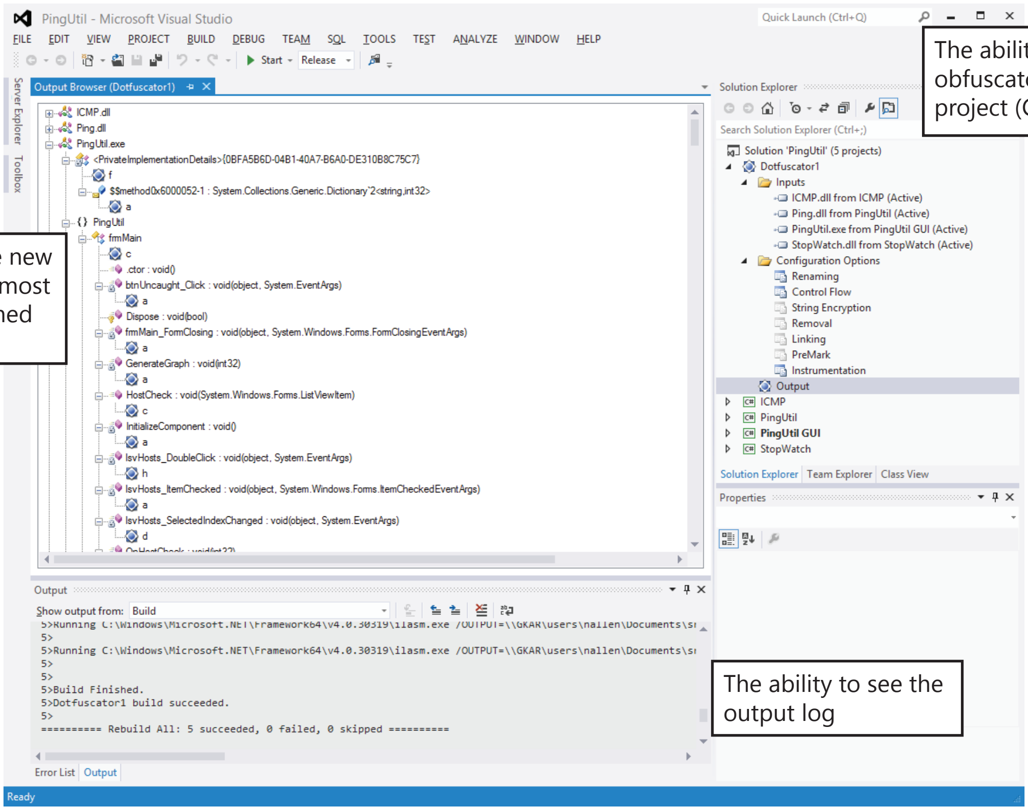
Developers, QA and support personnel all have occasion to debug binaries. Virtually the only stage in a development process that does not typically debug binaries is the manufacturing or build function. An obfuscation solution must provide a cost-effective, lightweight and low maintenance utility to 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.


PingUtil - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ANALYZE WINDOW HELP
Start
Release
Output Browser (Dotfuscator1)
Server Explorer
Toolbox
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution Explorer | Team Explorer | Class View
Properties
Output
Show output from: Build
Error List | Output
Ready

```
ICMP.dll
Ping.dll
PingUtil.exe
  <PrivateImplementationDetails>{0BFA5B6D-04B1-40A7-B6A0-DE310B8C75C7}
    f
    $$method0x6000052-1 : System.Collections.Generic.Dictionary`2<string,int32>
      a
  PingUtil
    frmMain
      c
      .ctor : void()
      btnUncaught_Click : void(object, System.EventArgs)
        a
      Dispose : void(bool)
      frmMain_FormClosing : void(object, System.Windows.Forms.FormClosingEventArgs)
        a
      GenerateGraph : void(int32)
        a
      HostCheck : void(System.Windows.Forms.ListViewItem)
        c
      InitializeComponent : void()
        a
      lsvHosts_DoubleClick : void(object, System.EventArgs)
        h
      lsvHosts_ItemChecked : void(object, System.Windows.Forms.ItemCheckedEventArgs)
        a
      lsvHosts_SelectedIndexChanged : void(object, System.EventArgs)
        d
```

```
Solution 'PingUtil' (5 projects)
  Dotfuscator1
    Inputs
      ICMP.dll from ICMP (Active)
      Ping.dll from PingUtil (Active)
      PingUtil.exe from PingUtil GUI (Active)
      StopWatch.dll from StopWatch (Active)
    Configuration Options
      Renaming
      Control Flow
      String Encryption
      Removal
      Linking
      PreMark
      Instrumentation
    Output
  ICMP
  PingUtil
  PingUtil GUI
  StopWatch
```

```
5>Running C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ilasm.exe /OUTPUT=\\GKAR\users\nallen\Documents\s
5>
5>Running C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ilasm.exe /OUTPUT=\\GKAR\users\nallen\Documents\s
5>
5>
5>Build Finished.
5>Dotfuscator1 build succeeded.
5>
========== Rebuild All: 5 succeeded, 0 failed, 0 skipped ==========
```

The ability to connect the obfuscator to the .NET project (C#, VB.NET, etc.)

The ability to see the new names - notice how most methods were renamed to "a"

The ability to see the output log

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

Organizations that have incorporated a continuous integration development approach should include the obfuscation 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 Obfuscation Process

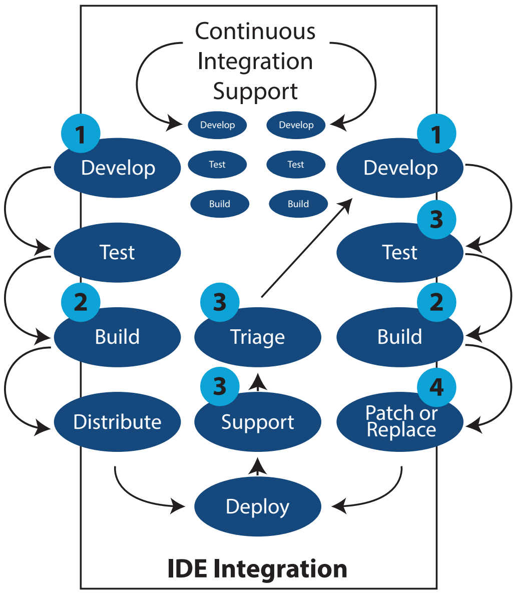
The obfuscation process integrates obfuscation functionality into the application development lifecycle specifically supporting the development, testing, integration, manufacturing, application support and patch management phases of development.

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 obfuscation 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.

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

Enterprise Obfuscation and Agile Software Development

The next step in establishing an effective obfuscation control is to instantiate a consistent means of assessing how and when to apply obfuscation. Once the specific integration points between the development lifecycle and the obfuscation 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.

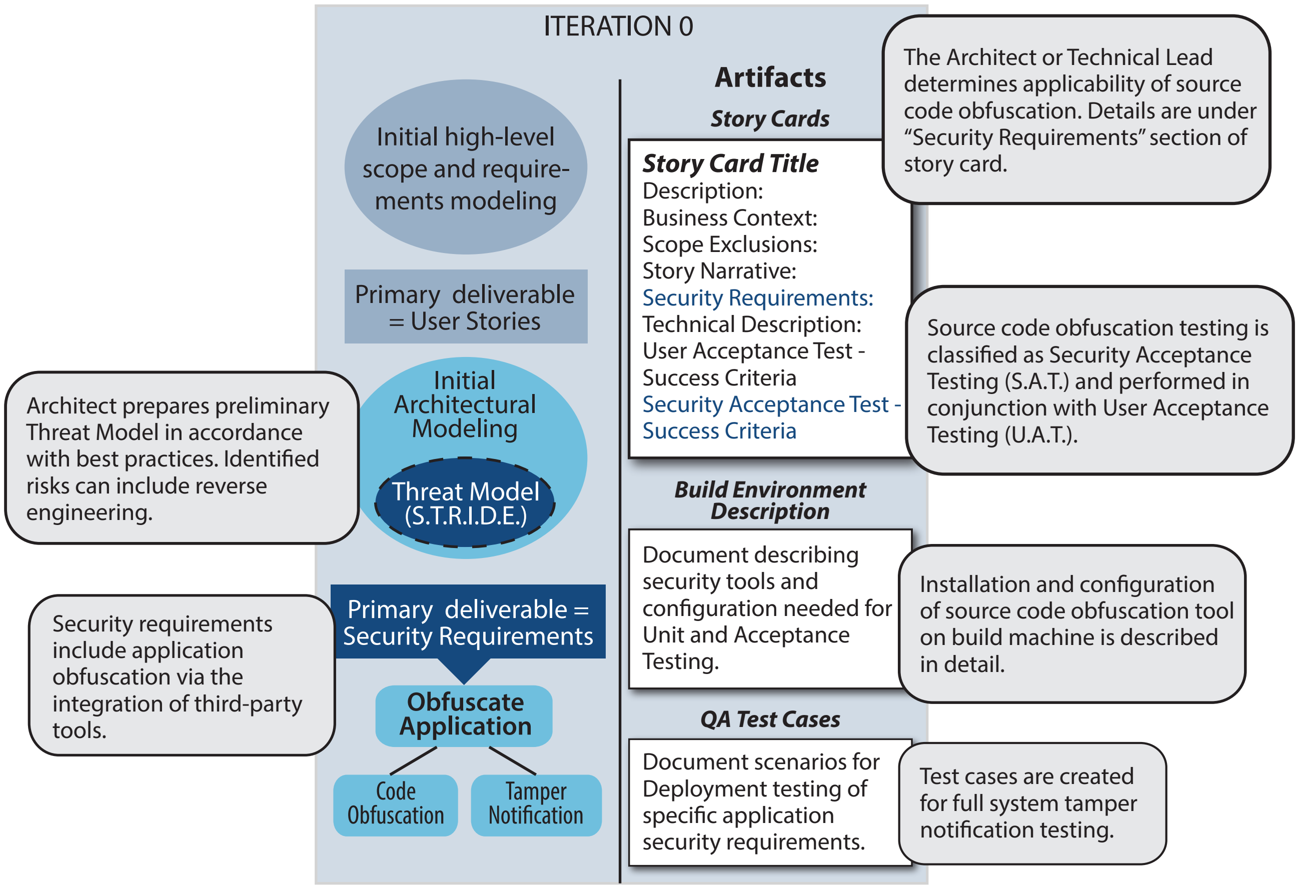

ITERATION 0
Initial high-level scope and require-ments modeling
Primary deliverable = User Stories
Initial Architectural Modeling
Threat Model (S.T.R.I.D.E.)
Primary deliverable = Security Requirements
Obfuscate Application
Code Obfuscation
Tamper Notification
Artifacts
Story Cards
Story Card Title
Description:
Business Context:
Scope Exclusions:
Story Narrative:
Security Requirements:
Technical Description:
User Acceptance Test - Success Criteria
Security Acceptance Test - Success Criteria
Build Environment Description
Document describing security tools and configuration needed for Unit and Acceptance Testing.
QA Test Cases
Document scenarios for Deployment testing of specific application security requirements.
The Architect or Technical Lead determines applicability of source code obfuscation. Details are under "Security Requirements" section of story card.
Source code obfuscation testing is classified as Security Acceptance Testing (S.A.T.) and performed in conjunction with User Acceptance Testing (U.A.T.).
Installation and configuration of source code obfuscation tool on build machine is described in detail.
Test cases are created for full system tamper notification testing.
Architect prepares preliminary Threat Model in accordance with best practices. Identified risks can include reverse engineering.
Security requirements include application obfuscation via the integration of third-party tools.

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.

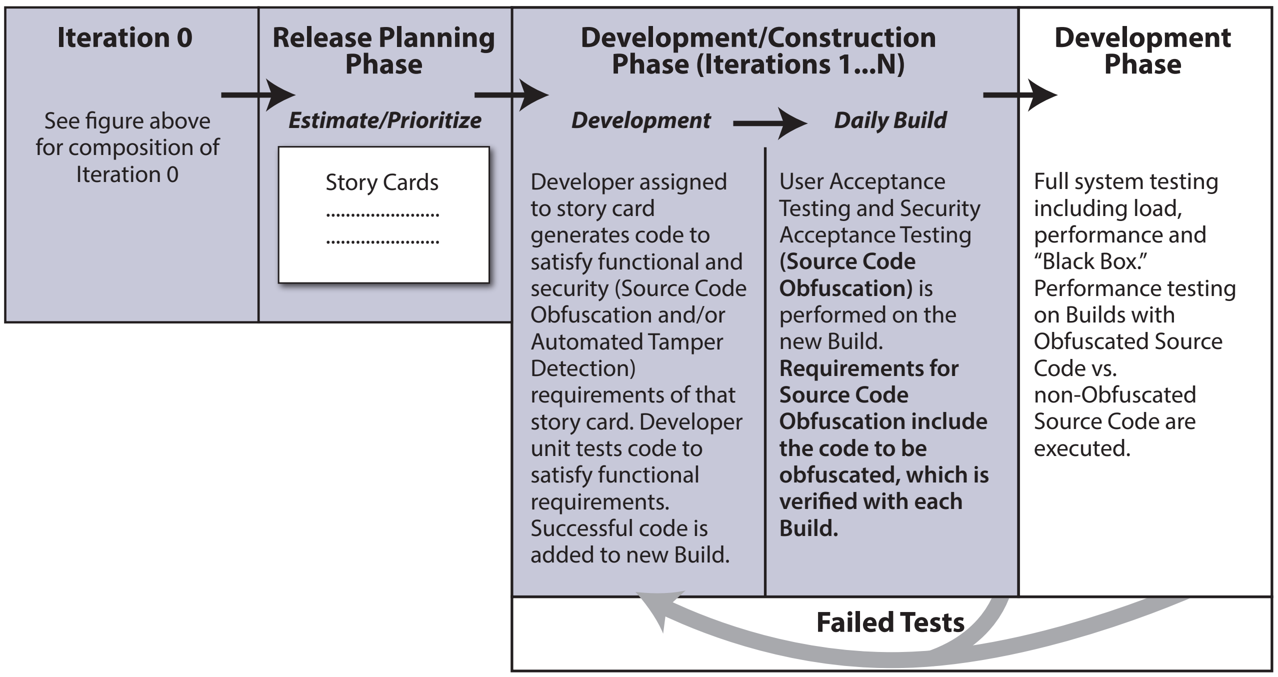

Iteration 0
See figure above for composition of Iteration 0
Release Planning Phase
Estimate/Prioritize
Story Cards
Development/Construction Phase (Iterations 1...N)
Development
Daily Build
Developer assigned to story card generates code to satisfy functional and security (Source Code Obfuscation and/or Automated Tamper Detection) requirements of that story card. Developer unit tests code to satisfy functional requirements. Successful code is added to new Build.
User Acceptance Testing and Security Acceptance Testing (**Source Code Obfuscation**) is performed on the new Build. **Requirements for Source Code Obfuscation include the code to be obfuscated, which is verified with each Build.**
Development Phase
Full system testing including load, performance and "Black Box." Performance testing on Builds with Obfuscated Source Code vs. non-Obfuscated Source Code are executed.
Failed Tests

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.

Obfuscation as a Compensating Control

Organizations in every industry are incorporating obfuscation 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 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 source=binary 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 community that has

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

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

If binaries are distributed widely, application vulnerabilities such as susceptibility to SQL injection, buffer overflows, etc. are a growing concern.

While defensive programming is probably the most effective control (eliminate vulnerabilities where possible), obfuscation is often used as an additional security layer to shield any unknown (or undetected) vulnerabilities. Layered security is a widely accepted best practice. Banks continue to secure their vaults within a building, behind locked doors, guarded by professionals and monitored by the police.

Identifying Specific Risks that Stem from Uncontrolled Distribution of Source Code

Source code access controls are designed to mitigate risks that stem from

- Exposure of coding practices that increase likelihood of successful system attacks, information theft and privacy violations
- Publication of intellectual property resulting in reduced company value, increased competition and an increase in legal defense expenses
- Application modifications “in the field” to circumvent metering functions that govern products and services reducing revenue and profitability.

From an ROI perspective, a control is justified if the difference between the cost and risk of implementing a control is meaningfully lower than the cost and risk of not implementing the control. The potential damage and the likelihood of occurrence of these or other risks must be established on a case by case basis.

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

Obfuscation is a way to accomplish (I) above, and failing to do that, it increases the difficulty of (II) through (IV).

Why include an obfuscation control?

- A layered approach to security is an acknowledged best practice,
- The cost of implementing an obfuscation 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 Obfuscation 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. This increased awareness has also produced a more comprehensive understanding of what is required to effectively use obfuscation 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 an obfuscation 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.

Additional Capabilities and Benefits

The obfuscation function requires a sophisticated static analysis of binaries. The techniques required for effective obfuscation also have applications to a number of other capabilities that can be applied concurrently with obfuscation. These include:

Pruning

Pruning is the identification and removal of unused code. As part of the obfuscation analysis, it is also possible to determine unneeded elements. The compaction or pruning process can then remove unused classes, methods, instance variables, design time metadata, and actual MSIL to produce a much smaller application. Size reduction caused by pruning can be significant with many applications showing up to a 70% size reduction. A typical size reduction falls between 30% and 40%.

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.

Linking and Packaging

An obfuscator can optionally merge simplifying distribution and installation.

Instrumentation

Obfuscation transforms existing code; Instrumentation supplements existing code through injection. Instrumentation can be further optimized in combination with pruning and linking to increase the likelihood that the original application format is preserved. Examples of instrumentation include Tamper (see The Fourth Dimension of Enterprise Obfuscation: Tamper Notification), Shelf Life (application inventory control), and Analytics.

Shelf Life

Shelf Life defines an application's enabling an application to self monitor and retire itself using business appropriate behavior.

Application Analytics

Application analytics are used to govern IT applications and to track feature usage customer experience improvement programs, and the tracking of evaluation activity are all examples of where application analytics can add significant value.

Note: Contact PreEmptive Solutions for detailed information on commercial instrumentation solutions in the solution categories mentioned above.

Conclusions: Obfuscators and Smoke Detectors

The empirical evidence is indisputable; tens of thousands of development groups have made the decision to include 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, .NET and Java continue to be among the fastest growing development platforms of our time.

Obfuscation is Like the Smoke Detector in Your Home

The answer lies in the likelihood of occurrence – not in a dispute over the potential damage. A calamitous threat that has a low potential of occurrence is not a new phenomenon and there are proven approaches to manage this category of risk. The low cost, low maintenance and ubiquitous smoke detector is an every day example of how to effectively manage a high cost/low likelihood risk.

A home fire threatens both life and property but is a relatively rare event. A smoke detector's value proposition combines a proven capability to reduce loss with a low cost and minimal maintenance process that is commensurate with the relatively low likelihood of occurrence.

The cost of an obfuscation solution represents a tiny fraction of the potential loss that it protects against.

While most people may be able to live their entire lives without a smoke detector, society is far better served by mitigating this risk in a broad and consistent fashion; that is why it is often illegal to sell a home without proof of an effective smoke detection system.

Obfuscation of Java and .NET binaries should be a staple of any application development process where source code is not widely distributed. Hopefully, for any given development team, they will never have needed the protection that obfuscation afforded them – but we also know that some teams most definitely will benefit and for those that skipped this simple step – the cost may be quite severe indeed.

It would be irrational to avoid living in a home to avoid the risk of a fire, but it is also irresponsible, and in some jurisdictions illegal, to not take simple precautions to minimize known risks.

.NET and Java environments offer tremendous advantages that far outweigh the easily managed risk of source code extraction from binaries.

Of course, hotels and restaurants have both a higher potential for loss and an increased likelihood of occurrence. The resulting requirements and penalties are proportionately higher.

Uncontrolled source code distribution holds varying degrees of risk that can be segmented by industry segment. Software vendors, financial service providers, telecommunications companies, manufacturers and other businesses that rely on applications to generate revenue, assure business continuity and whose applications represent unique intellectual property have a greater risk with proportionately higher requirements and more severe penalties for failure.

About PreEmptive Solutions

PreEmptive Solutions provides software and managed services that monitor, measure, and protect applications across on-premises, cloud, internet, and mobile platforms. With software on millions of developer desktops, PreEmptive Solutions is the recognized leader in application analytics, intellectual property protection, and anti-tampering software. Leveraging PreEmptive Solutions' application analytics and protection solutions, development organizations materially improve application quality, user satisfaction and development ROI across today's distributed and increasingly heterogeneous computing architectures. To learn more, email solutions@preemptive.com or call 440.443.7200.

PreEmptive Solutions
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: