Connect

Assessing and Managing Security Risks Unique to Java and .NET

By Sebastian Holst

This article enumerates specific risks unique to managed code, guidance on assessing organizational materiality of these risks, and an inventory of broadly recognized risk-mitigation technologies and practices.

Abstract

Reverse engineering and modification of managed code (most notably .NET and Java applications) are well-understood and common practices. Legitimate scenarios include software debugging, technical support, and developer training. However, these same practices can present material organizational risk, including intellectual property theft, operational disruption, software piracy, and data loss. This article enumerates specific risks unique to managed code, guidance on assessing organizational materiality of these risks, and an inventory of broadly recognized risk-mitigation technologies and practices.

The rise of managed code – most everyone uses it somewhere

Managed code can be contrasted with "machine code" (or "native code"). An application distributed as machine code is comprised of low-level instructions that execute directly on the CPU where the application is running. Since each CPU type has its own instruction set, a developer must build a distinct executable for every type of CPU that he wants to support.

Conversely, an application distributed as managed code runs on a "virtual machine" rather than directly on the native CPU. This level of execution abstraction allows developers to build one executable that will run everywhere versus the labor-intensive alternative of building CPU-specific machine-code binaries. In order to make this approach efficient, a "just in time" compiler or *JIT* sits inside each virtual machine and generates the site-specific machine code as the application is being run.

One consequence of this approach is that managed code must include additional information for the JIT to do its job as compared to the machine code alternative. This additional information is also what makes managed code materially easier to understand, reverse engineer, and to modify. It is the just-in-time compilation that delivers all of the benefits of managed code, and it is why managed code presents unique risks.

The Java platform and the .NET framework are the two most widely adopted examples of managed code systems. Today, virtually every organization and individual relies on Java and .NET software in some fashion, either at the place of work, navigating the Internet, or using mobile phones.

Reverse engineering managed code – everyone does it

Today, there are literally dozens of freely available decompilers, and reverse engineering is a standard and common practice among developers. The reverse engineering of an application that might have taken weeks with native code can now be accomplished in seconds. Not only is reverse engineering easy, it is also an effective technique for a variety of legitimate development activities including:

- Debugging
- Developer training
- Ensuring interoperability
- Replacing lost source code
- Technical support

Modifying managed executables – it's a common practice

The modification of managed code is only modestly more complex that reverse engineering. In fact, the relative ease of managed code modification has made practical aspect-oriented programming, an entirely new programming approach that uses this ability as one of its core technologies. As was the case with reverse engineering, there are legitimate scenarios where the post-compile modification of managed code is both a common and a widely accepted practice. These include:

- Application hardening against vulnerability exploitation
- Auditing and logging
- Debugging
- Performance profiling
- Testing

The unique risk profile of managed code – it's well understood

Unauthorized reverse engineering of software is not new, but the risk profile for managed code is quite distinct and merits closer study. Reverse engineering has historically been a difficult and time-consuming undertaking. As such, it was a relatively uncommon occurrence, undertaken predominantly by organizations with a strong commercial or military incentive; legal remedies[1] could be relied upon to serve as a control, a disincentive, and a reasonable incident response mechanism. Clearly, managed code has turned this worldview upside down.

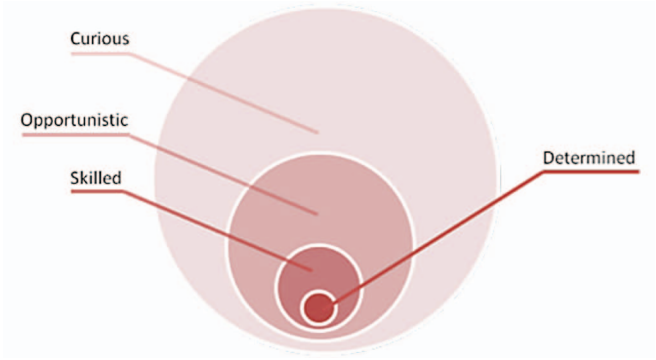

Figure 1: Changing risk profile as applications migrate from machine code to managed code

These capabilities are no longer restricted to the most skilled and determined (Figure 1). Today, virtually every developer, regardless of skill or motivation, can readily view and modify managed code. The use of intermediate languages that are compiled "just in time," the emergence of a wide array of legitimate use cases, and free developer tools have simplified reverse engineering and code injection to such an extent that a developer's idle curiosity may be the only motivation required to start playing with your managed code.

Risks stemming from reverse engineering and code injection

With easy access to reverse engineering tools and code injection technology, unauthorized access and modification of managed code can pose a broad spectrum of risks (Figure 2).

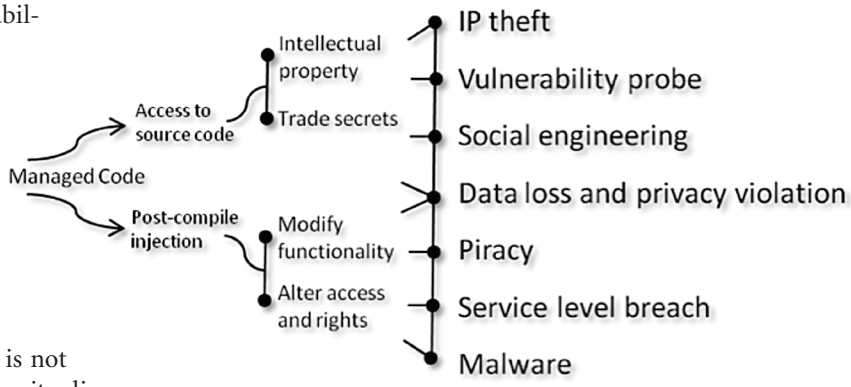

Figure 2: Risks stemming from unauthorized access to source code via reverse engineering and post-compile code injection.

Unauthorized access to source code through reverse engineering can lead to:

- **Intellectual property theft:** This is probably the most widely recognized risk that stems directly from unauthorized access to source code. Possession of functioning source code provides transparent access to any IP that may be encoded therein.
- **Vulnerability probe:** Classic application vulnerabilities such as SQL injection or cross-site scripting do not require access to source code to detect. However, source code analysis can reveal the presence of hardcoded credentials, data encryption conventions (or lack thereof) as well as a other operational vulnerabilities.
- **Social engineering:** Auditors have flagged managed code plug-ins inside online banking applications as revealing information that identity thieves could re-use as they pose as bank employees surveying customer experience. After convincing a bank's customers that they are with the bank and earning their trust, collecting personal information becomes a straightforward exercise.

Modification of applications after compilation can lead to:

- **Piracy:** This is probably the most widely recognized risk from direct manipulation of a commercial application. A common "cracking" strategy is to "stub out" license verification method calls or overwrite the verification result leading to unauthorized access to the application and/or other assets it may govern. NOTE: Piracy is not limited to application piracy. Software is often a "gateway" to access

1 Reverse engineering and application tampering is not inherently illegal. However, while laws vary, usually the circumvention of "copyright protection systems" is widely prohibited by law. Even here, exceptions are often made for interoperability, security, and privacy considerations. Further, many jurisdictions explicitly permit reverse engineering of lawfully acquired products as a legitimate means to extract trade secrets not otherwise protected by copyright or patents.

equipment and content. Telecommunications, medical, and automotive devices are all commonly managed and configured by software. Premium content is also commonly managed by software. Breaking software controls can lead to pirated equipment and content.

- **Service level breach:** This is a particular issue for organizations that monetize support for specific configurations of open source applications. Individuals within end-user organizations may attempt to fix or extend a supported configuration.
- **Malware:** Classic malware attacks rarely target managed applications. This likely explanation for this is simply that using Internet applications to introduce malware is far more efficient. However, this tactic may prove effective as a strategy to gain access to specific information or to destabilize an organization's operations.
- **Data loss and privacy violation:** Depending on the context, this may be the most serious risk. PCI, HIPPA, FISMA, and other regulations recognize that data cannot be effectively managed without control over the systems that create, manage, and distribute that data. Insight into the internal workings of an application as well as the ability to alter an application's behavior can be used to access, alter, and distribute private data.

Materiality – so what?

Given the awareness and ease of reverse engineering and code injection capabilities with managed code, one must assume that if an individual has access to a managed executable, he also has access to the source code and the ability to modify its behavior. Of course, this begs the question, so what? If your applications are open source, why should you care if your applications are reverse engineered? If the only users of your application are privileged to such an extent that if they are malicious, they will have the ability to do far more damage in other ways, then code injection is the least of your worries. As with all potential risks, assessing the likelihood of an event in combination with the materiality (degree of risk) should the event occur must be measured against your organization's tolerance (or appetite) for risk. Without this perspective, there is no reliable means to evaluate and prioritize controls and risk mitigation options (that come with their own costs and risks).

Reverse engineering and code injection may or may not pose a material risk to your organization. A full treatment of relevant risk factors and how to measure them is beyond the scope of this article. However, the following provides a broad overview of potentially relevant considerations:

Application considerations

- **Role or use of applications within critical business functions**: The extent to which managed applications play a critical role in material operations or offer competitive advantage will increase the materiality risk.

Applications that rely upon distributed components and services offer additional opportunities to compromise an application's integrity.

- **Application access to sensitive information:** The materiality of risk increases proportionately with the potential that a compromised managed application might lead to information loss or privacy violations.
- **Application architecture:** Applications that rely upon distributed components and services offer additional opportunities to compromise an application's integrity. This includes distributed computing models such as SOA, rich Internet application technologies such as Silverlight, and those designed to run on mobile devices such as J2ME.
- **IT infrastructure:** Applications that are run inside a single secure environment (inside an enterprise or security hosted) are more secure than those that are distributed to partners, clients, or publicly available for use or evaluation.
- **Authorship:** Applications developed by third parties present their own risk profiles. Under certain circumstances, enterprise consumers may want to mandate that their suppliers harden their commercial products.

Organizational considerations

- **Revenue or value associated with the applications' source code**: The materiality of risk increases when IP theft, software piracy, or the theft of other products and services occurs.
- **Size, complexity, and distribution of application development:** The likelihood that a malicious party will have access to your compiled application, that vulnerabilities may be introduced, and/or that artifacts within applications may hold unrecognized potential for vulnerability exploitation is increased in proportion to the complexity, scale, and distribution of your (or your suppliers') development activities.
- **Degree of regulatory obligations and oversight:** The materiality of risk increases proportionately with the likelihood that a compromised managed application might lead to regulatory violations or material incidents due to operational disruption, information loss or privacy violations. Regulatory violations amplify risk as they add fines, damage to reputation, etc.

User considerations

- **Inside or outside:** Users that are well-known and accountable reduce risk.

- **Technical skill:** Some small degree of technical training is required.[2]

Managing risk: technology is required but not sufficient

Obfuscation, the first technology designed specifically to prevent reverse engineering of managed code, emerged almost as soon as the Java platform was released in 1996. However, in recent years, there have been many advances in obfuscation as well as the emergence of a variety of other techniques that, with obfuscation, fall under the umbrella term "Application Hardening and Shielding," defined as " a set of technologies used to add or inject security functionality within applications specifically for the detection and prevention of application-level intrusions."[3]

Technology: application hardening and shielding

The following technologies can be applied in various combinations to mitigate some or all of the risks that stem from reverse engineering and code injection.

Anti-reverse engineering

- **Obfuscation:** 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. The most common and effective obfuscation transformations include:
  - **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:** logic and flow are re-expressed, preventing translation back into valid C# (or any other high level language).
  - **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.
- **Ahead of time compilation:** Techniques that apply the Just-In-Time compiler before distribution to convert managed code to native code have the potential to undermine the rationale for moving to managed code in the first place.
- **Packing:** Techniques that encrypt and compress managed code and include a native code component to "unpack" and execute the components at runtime.
- **Secure Virtual Machine (SVM):** A virtual machine that executes a form of encrypted intermediate language (IL). This is different than standard encryption that requires decryption of IL before it can be executed.

2 Neil MacDonald and Joseph Feiman, "Hype Cycle for Data and Application Security," Gartner, Inc., July 17, 2009.

3 Ibid.

Technology	Strengths	Weaknesses	Side-effects
Obfuscation: Renaming	Defeats human inspection. Performance neutral.	Decompiled source can be recompiled.	Can break reflection and other indirect method calls.
Obfuscation: Control flow	Defeats machine translation.	Method names are left intact.	Can impact performance for computationally intensive code.
Obfuscation: String encryption	Hides embedded strings such as queries and other strings.	Weak protection. Decryption must happen on the client; therefore, an attacker can observe and defeat it .	Can impact performance for computationally intensive code.
Ahead of time compilation	Easy to implement. Defeats machine reverse engineering.	Software is no longer managed code.	Loses platform independence.
Packing	Easy to implement. Defeats machine reverse engineering.	Easy to reverse in a very short amount of time.	May not PEverify. May lose platform independence.
Secure virtual machine	Extremely secure.	Debugging and patches cannot be supported.	Significant performance impact up to 1000X.
Code signing	Extremely difficult to defeat.	Assemblies can be "re-signed." No defense or notification. Always fee-based.	None.
Tamper defense	Easy to implement. Custom behavior can be injected for real-time defense and custom notification.	None.	May moderately increase program size.
Usage monitoring	Can detect unauthorized users and installation. Can detect suspicious usage patterns.	Does not defend or provide real-time response.	May moderately increase program size, require opt-in logic to be included, and impact application performance.

Table 1 – Application hardening technology characteristics

Anti-tampering

- **Code signing:** Code signing uses a digital signature to validate the integrity of the binary. In the case of the .NET framework, the CLR (Common Language Runtime) will by default validate the "strong name" of an assembly when loading it.
- **Tamper detection and defense:** Typically, tamper detection uses a checksum against specific regions of an executable to determine if that application has been modified since it was originally compiled. Once tamper has been detected, a defense can be invoked. The defense can be a simple abort or as sophisticated as a custom response such as the de-installation of the application. Additionally, a notification or alert can be transmitted to one or more locations to alert stakeholders that an attempt has been made to execute a tampered application (inside or outside of the runtime environment).

Anti-abuse

- **Usage monitoring:** Injecting logic to stream application adoption, method usage, user behaviors, system configurations, and data values to one or more end-points. This runtime data is processed for analysis and further integration into additional data sources such as user profiles, service level agreements, and business performance metrics.

Technology evaluation criteria

Each technology has strengths, weaknesses, and the potential to introduce side-effects into the applications that it is protecting. As with most security practices, a portfolio or layered approach is often the most effective. Table 1 on the previous page provides a high-level summary of application hardening technology characteristics.

The three dimensions of effective risk mitigation

Technology alone cannot mitigate risk. Technology must be wrapped inside a process that will be governed by a policy to ensure that the process is applied appropriately and consistently (Figure 3). These considerations are particularly important when evaluating the practicality of implementing potential technologies and specific commercial options.

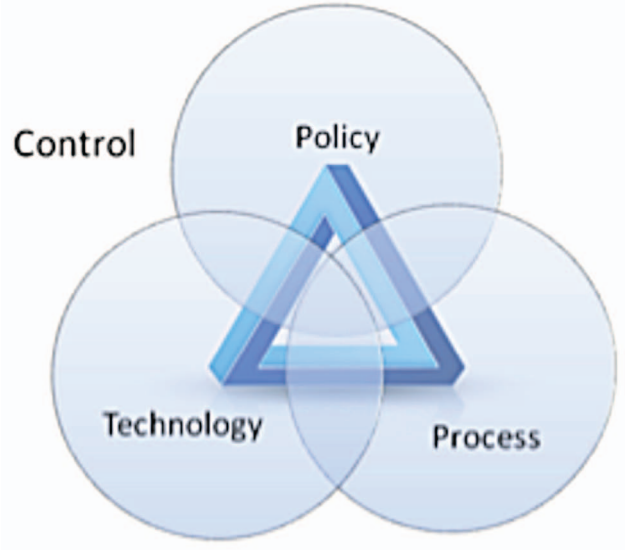

Figure 3: The three dimensions of an effective application security control

Commercial evaluation criteria

Application hardening and shielding technologies are often intrusive, are typically applied in the very last stages of the development cycle, and have the potential to materially alter the behavior of an application. Commercial-grade implementations address these issues with a variety of strategies. The following criteria can help to ensure that the application hardening "cure" is not worse than the disease.

- **Software development lifecycle support:** Identify how debugging, patch management, and support activities will be impacted. Identify to what extent distributed development and shared components can be accommodated.
- **Platform and framework support:** Ensure that current and potential future target platforms are supported. 64-bit, Silverlight, J2ME, compact framework, and the Azure platform are all examples of runtime environments that may pose special challenges. Different versions of the .NET framework and the Java SDK may also pose additional constraints.
- **IDE integration:** Integration into build processes, code editors, and application lifecycle management solutions reduces the level of effort, quality risk, and implementation complexity.
- **Support and vendor viability:** Given the level of integration that these technologies require, this common-sense practice is especially relevant.

Adoption, patterns, and practices

Independent research on the adoption and best practices to mitigate managed code risks are available, but are still relatively scarce.

Analyst coverage

Gartner, Inc. estimates application hardening and shielding market adoption to be between 5% and 20% of its "target market."[4] Gartner writes:

> *"Code obfuscation is the more widely adopted and more-mature method of protecting applications, but estimated adoption rates are still in the single digits, because most organizations are unaware of its benefits until they directly experience the theft of IP or an attack from an application compromise. Furthermore, for application protection techniques that rely on the insertion of code, development organizations may be reluctant to allow the injection of new code into an application from a source other than a developer."*[5]

4 Ibid.

5 Ibid.

Developer platform support

Microsoft is the only large software development platform provider to include obfuscation and has done so since the release of Visual Studio 2003. Through Visual Studio 2008, the obfuscation tool, Dotfuscator Community Edition, included only renaming obfuscation.

However, in October of 2008, Microsoft announced that its next major release, Visual Studio 2010, would extend the embedded application hardening and shielding capabilities to include:

- Tamper detection and defense
- Feature monitoring

This is noteworthy for two reasons. First, in providing these capabilities as a standard component of every Visual Studio user's desktop,[6] Microsoft is acknowledging the breadth of these requirements and their functional evolution. Second, at least for the .NET framework development community, they are establishing a development practice convention, if not a de facto standard, for the injection technique and monitoring data model. These new capabilities have been included in the early releases of Visual Studio 2010 beginning with the CTP release in October of 2008.[7]

Java development organizations can look to a number of open source obfuscators for their own "free" anti-reverse engineering alternatives.

Use case examples

Commercial software developers, wanting to protect their commercial products, are the most obvious scenarios for application hardening and shielding. In fact, equipment manufacturers, financial institutions, and a host of other scenarios where managed code plays a central operational role often face even greater risk. Here are three publicly available examples.

Customer relationship management

West at Home, a provider of home-based customer contact solutions, hardened and shielded their client-side software as an element of a broader strategy to manage their clients' risk. This is notable because, while it is common for software developers to invest in reducing their own risk, e.g., IP theft and piracy, West at Home also includes application hardening as part of their strategy to manage client risk.[8]

Medical device equipment manufacturing

Full Spectrum Software, a software development and testing provider for the medical and scientific industries, has published a best practices white paper focusing on the need for protecting and shielding managed code inside medical devices.[9]

Electronic gaming

Applied Concepts, an electronic bingo solutions provider, recently announced the pivotal role of managed software to their rapidly changing industry and the importance of integrating application hardening and shielding with activity monitoring for both security and business performance management.[10]

Conclusion

The unique characteristics of managed code that make reverse engineering and tampering almost trivial are well understood and are, in fact, a common best practice among the legitimate development community. As with all powerful technologies, reverse engineering and application modification can be used productively and also for mischief. A portfolio of technologies has emerged and matured over the past decade and is in use across industries. As with any risk-based strategy, there is no "one size fits all" approach. However, what may prove difficult to defend should an incident arise is no policy at all.

References

Free anti-reverse engineering tools:

- **Java platform:** Open Source Obfuscators in Java[11]
- **.NET framework:** Dotfuscator Community Edition[12]

About the Author

Sebastian Holst is currently CMO at Pre-Emptive Solutions. He has held executive product and strategy positions in both public and private software companies, focusing on application security, risk management, and content management. In addition, Sebastian has contributed at board and advisory committee levels to the W3C, IDEAlliance, the Open Compliance and Ethics Group, and Qifense, LLC. He may be reached at Sebastian@preemptive.com.

6 These capabilities are included in every Visual Studio SKU other than Express.

7 http://www.microsoft.com/Presspass/press/2008/oct08/10-27PreEmptivePR.mspx.

8 http://www.marketwire.com/press-release/West-Corporation-1002664.html.

9 http://www.fullspectrumsoftware.com/docs/codeobfuscation.pdf.

10 http://www.preemptive.com/applied-concepts-inc-selects-runtime-intelligence-service-as-part-of-its-bold-strategy-to-launch-a-new-software-service-and-change-an-industry.html.

11 http://java-source.net/open-source/obfuscators.

12 http://msdn.microsoft.com/en-us/library/ms227240.aspx.