Guidelines for Effective Software Discovery

By Gareth Loy and Tom Gafford

Abstract

The legal system, by its nature, is typically behind the curve of technology, and the methods and procedures of discovery can be quite hampered by the limitations imposed by courts unfamiliar with what is actually required for competent and thorough software discovery. This article seeks to shed some light on this important subject.

Introduction

Courts refer to electronically stored information as ESI. Software source code in its native habitat is a form of ESI.

Bottom line: ***analysts performing software discovery need the same source code, tools, documents, and environment as the original developers of the software****.*

Effective software discovery requires a *transparent production*, an *optimal discovery environment* and *good working conditions*. It also requires *time*.

This document discusses what is required for effective software discovery.

Definitions

Source code: Human readable instructions, which are translated into machine readable instructions that comprise the shipped program in the accused device.

Build files, or “make” files: human-readable instructions that direct the translation of source code into executable code, including instructions to be executed by machines, and steps to be followed by persons responsible for creating the shipped product.

Software Production

The software production must include everything required to determine infringement/non-infringement, including source code, directory hierarchy, build rules, documentation, and suitable software development environment.

Source Code

Source code must be accessible in its native habitat. *Native habitat* means the electronic software development environment used by those who maintain and develop the software, NOT paper printouts, Concordance databases, TIFFs, or PDFs of source code. Source code files must be encoded in their native form, typically ASCII or Unicode.

Directory Hierarchy

Files must appear in their original directory hierarchies so that cross-references such as “include” statements and hyperlinks resolve to the correct target file. The file hierarchy must NOT be flattened into straight lists, as is common practice with document management tools like Concordance. Standard file systems include UNIX, Windows (FAT, NTFS), Macintosh, and Linux. What is produced must be what developers actually developed, in the electronic format that they developed it, not a transliteration of their work, not a subset of their work.

Software Development Environment

The software development environment must include or stipulate tools for reviewing, building, debugging, and preparing the software for delivery to customers. Depending on the product, software developers commonly employ commercial integrated development environments (IDE) such as Microsoft Visual Studio. The type and version of IDE must be stipulated. All files used by the IDE such as project description files, must be disclosed. Any required customizations or global preference settings for the IDE must be disclosed.

Developers may augment the IDE with lower-level software tools to perform specialized operations, such as building libraries, fetching appropriate software modules from remote servers, database construction, and preparation of software for installation. Such lower-level software systems are often based on tools supplied with the UNIX operating system, or emulated versions of these tools (such as CYGWIN). Required low-level tools must be stipulated or disclosed.

Sometimes developers will build their own custom software to develop, process, build, or install their software. All custom build software must be disclosed. All build files must be disclosed. All standard lower-level tools required to build the software must be disclosed or stipulated.

Software Target Release and Marketing Requirements Documents

Realistically, software that is developed and maintained over years typically is segmented into target releases that correspond in time with when the company wants to release a new feature. Internally, Engineering and Marketing departments negotiate over features and bug fixes, and draft a kind of internal company contract called a Marketing Requirements Document (MRD) to specify

the capabilities of the release. A convenient code name is given to define the project leading to the release. In practice, Marketing usually assigns a different name to the released product than is used internally. For example, a development effort internally named “Stoli” (after a famous hard liquor) might be released by Marketing as a completely separate product name, “Frobnitz 2.0”.

Oftentimes, the source code for each release will be saved as a unit for the purposes of manufacturing the product. Defendant must disclose relevant lists of product capabilities, internal names, years manufactured, and corresponding product names of all disclosed software so that analysts can focus on those releases most relevant to the time-span and capabilities of the disclosed software.

Conditional Compilation

Modern computer languages allow developers to conditionally include or exclude blocks of code in a particular source code file via *conditional compilation*. Instructions for which blocks of code are to be included or excluded are given in the build files that accompany the source code.

Conditional compilation can also be used to set global variables that determine the overall behavior of the software.

Therefore, build files must be disclosed so that it can be known which parts of each source code file are actually used to build the shipping software.

Object Code, Debugging, and Executable Code

The human-readable source code is converted on instructions of the build rules, using the build tools, to create object code, which is in turn linked and loaded to create the executable code that can then be shipped to customers.

While the software is under development, the build tools usually construct the executable code so that it can be debugged. Debugging means being able to inspect the software as it executes. Tables are placed in the executable code that show the corresponding location in source code for each step taken by the software. Developers can debug software by setting breakpoints in the code that halt execution when the breakpoints are triggered. When halted, developers can inspect the software to determine such things as how the software arrived at the breakpoint, what it will do next, and the values of data in memory.

Platform-Targeted Compilation

Using the development environment (IDE), developers can customize how software is linked for different platforms. For example, software to be executed on both Windows and Macintosh computers must include different libraries to interface with the different platform capabilities. Customizations for different

platforms must be disclosed.

Documentation, Comments, Revision History

Product documentation, code commenting, and revision history provide essential context for understanding the software.

It has been said that software is like quick-set cement: once it is written, it is difficult to change. At least part of the reason for this difficulty is that *it is easier to write code than to read it*. As a consequence, source code can become so complex that it is prohibitively expensive to maintain, unless good code hygiene is maintained: follow good coding conventions, name data structures and variables for what they actually do, comment the code, document the code, and keep documenting the code throughout its lifetime as it evolves.

These considerations also apply to the analyst's source code review: it is vital that all relevant documentation, comments, and revision histories are disclosed.

Documentation

Documentation of source code includes anything that helps understand how to develop, maintain, test, and ship software products. These include white papers, user guides, theory of operations, product specifications, standards documents, application programming interfaces (API), marketing requirements documents (MRD), engineering change orders (ECO), tutorials, electronic and other correspondence (email), marketing literature, dictionaries, internal and external web sites for developers/marketers/customers, help documentation for developers/marketers/customers, troubleshooting guides, audio/video materials, PowerPoint presentations, and the like.

It is generally not enough to disclose only product specification documents written as thought experiments at the beginning of a software development project. Follow-on documents are also required that show how the software development evolved, showing how good and bad ideas in the initial specification had to change to realize the shipping product, to respond to market forces, competition, equipment limitations, and so on.

Comments

The native software development habitat typically includes commented source code. Comments typically describe important data structures, characterize the theory of operation of the principal methods, and give insight as to how parts of the system cooperatively perform the required actions. If development engineers commented their code, comments must NOT be stripped from the produced source code.

If source code is developed by non-native English speakers, analysts may

require translation services.

Revision History

Source code revision history typically consists of a database of changes made by developers to the software source files. This provides historical information about when features were added and bugs were fixed. Revision history is typically managed by a revision control system (RCS) that automates storing, retrieval, logging, identification, and merging of revisions. Many commercial and noncommercial systems perform this function. In some, the revision history is stored in the individual source code files (RCS). Alternately, revision information is stored in a central repository (Microsoft Source Safe, CVS, or Subversion).

Revision history must be supplied with source code.

Discovery Environment

Defendants disclosing software often fear untoward disclosure of their trade secrets by competitors as a side-effect of analyst discovery. It is reasonable in these circumstances to restrict access to the software, but not to place galling limitations that would unduly limit discovery.

Analysts must be able to install software inspection tools of their choice on the computers containing discoverable software.

Analysts must have access to all needed services to perform discovery. If, for example, the discoverable software requires Internet access to operate, then the analyst must be allowed to debug the software while it is connected to the Internet. One solution to performing this step while protecting defendant's proprietary information is to enclose the discovery environment on a private network or on a subnet of the defendant's own network. That way, network traffic does not go across the Internet, thereby protecting the defendant's software from inadvertent disclosure to competitors.

Work Environment and Location

Work conditions should not adversely affect the work of the analyst, and should respect the human needs of the analyst.

A convenient location should be provided for analysts to optimally perform their work. Ideally, the work site is the analyst's own premises. Second best is a protected site within easy commuting distance from the analyst's premises. Arrangements requiring overnight travel put a significant burden on discovery.

The analyst should be allowed Internet access, to send/receive email and surf the web during discovery. Cell phone coverage should be available at the site. Natural light and air should be available.

Given the many hours required to perform discovery of all but the most trivial software, discovery hours should be generous. Provision to extend to nights and weekends should be available.

Fallbacks and Work-Arounds

The bottom line: anything less than the native software in its native environment renders the analyst's work more difficult.

Defendant produces paper copy of source

There is a special circle in Hell reserved for such defendants. If paper printouts are all that is available, printing quality must be sufficient to allow source code to be OCR'ed with 100% accuracy. This means no watermarks, no copies of copies, no dropouts, a serif font face that distinguishes I (uppercase i) from l (lowercase L). For any realistically large software effort, paper disclosure puts the plaintiff at a serious disadvantage because of the difficulty analysts face of navigating stacks and stacks of paper.

Defendant produces unsearchable PDFs or TIFFs of the source

These defendants end up in the same circle in Hell described above. The same criteria apply.

Defendant produces searchable PDFs of the source

Why did they go to all this trouble instead of just providing electronic sources? At least the period for discovery should be extended by the time it takes analysts to convert these files back to source code and reconstitute the directory hierarchy.

Defendant produces flat file list

The directory hierarchy of the software provides important information to analysts about the way in which the product is built and maintained. Likewise, file names provide insight as to what is contained, and also provide the targets for references from other source code, or from hyperlinked documents. Tools like Concordance that flatten the directory hierarchy to a numbered list of files throw away this information. While it can sometimes be reconstituted by analysts, it takes time and effort that is then not being spent on discovery.

Defendant produces more or less than is required

A common technique of uncooperative defendants is to provide more or less software than is required to build the accused product.

If too much code is produced, analysts depend upon the software build scripts and installation scripts to separate the wheat from the chaff, which adds time to discovery. This is the "needle-in-a-haystack" defensive strategy.

If too little code is produced, or build scripts are not produced, or there are missing libraries, or missing source code, time is wasted while analysts attempt to identify and request the missing elements.

Add time to discovery if defendant provides more or less than the exact source used to generate the product.

Defendant does not produce debuggable executable

If the defendant does not produce debuggable executable, then the analyst must use static software analysis to determine the operation of the software. See below.

Badly written software

If the software is particularly badly written, the time required for the analysts to understand it can be much longer than if the code is well written.

Even good code can go bad, just like things left in the refrigerator too long. For example, though the code has been changed, the comment describing how it used to work is not revised to reflect current reality. The analyst is thrown off, requiring more time to sort out what is actually happening.

Old code may be left around like a sponge left in a patient after surgery. If it's deemed by management to be too costly to remove, and wastes little time in the running program, some companies will just leave it in. Such code can act like a red herring to analysts.

Bad programming practices such as go-to statements may be used, leading to dreaded "spaghetti code". Code modules may be duplicated by "copy/paste" editing and modified slightly to perform a related function rather than using object oriented inheritance methods.

Software written by non-English speakers can be especially challenging, and may require translation services to understand.

Static Code Analysis vs. Dynamic Code Analysis

Dynamic software analysis is used when debuggable executable software is available. The analyst can compare the operation of the software directly to the statements in the source code, inspect data, and more naturally follow the train of thought of the software developers. Dynamic analysis is generally preferred over static analysis, but sometimes static analysis is also required, as when considering how a part of the program works that cannot be exercised because the right conditions cannot be triggered.

Dynamic analysis typically requires more than just a debuggable executable, however. It typically also requires appropriate software installation, so that the services of the underlying operating system are available; other processes to execute that may perform required auxiliary functions; databases initialized with appropriate settings to allow the software to function; relevant input files, appropriate directory hierarchies for output files; a network; other computers on the network performing specific functions such as network services; and so on.

Static software analysis is used when debuggable executable software is not available. The analyst must then understand the operation of the software by thinking about how it would operate if he could debug it. This can be exponentially more difficult than dynamic analysis for realistically-scaled software projects, for at least these reasons: 1) each branch in the program is potentially a different behavior of the software that must be evaluated; 2) values of data that affect program operation must be calculated by the analyst.

Static analysis is sometimes required, as when debuggable software is not available, or when software cannot be induced to dynamically evoke a particular response under investigation.

If non-debuggable software is provided, experience has shown that it is often worthwhile for the analyst to overcome whatever obstacles are in the way to make it debuggable.

Conclusion

Careful consideration of the above points can drastically affect software discovery.

Bottom line: ***analysts performing software discovery need the same source code, tools, documents, and environment as the original developers of the software****.*

