Best Practices Overview for Auditing and Securing Your Build Environment with

A White Paper and Best Practice overview of how to secure and trace builds using OpenMake Meister

Request-info@openmakesoftware.com

www.openmakesoftware.com

INTRODUCTION **4**

COMMON BUILD CHALLENGES **5**

THE OPENMAKE MEISTER SOLUTION **6**

FOOTPRINTING AND BUILD AUDIT REPORTING **7**

EXAMPLE BUILD AUDIT REPORT 9

DEPENDENCY DIRECTORIES **11**

BUILD TYPES **11**

PUBLIC BUILD JOBS **12**

MANAGING GROUPS AND USERS **12**

CONCLUSION **13**

COMPANY OVERVIEW **13**

Introduction

OpenMake Software, the leader in enterprise build management technology, is designed to directly benefit corporations by improving the ability to audit and trace build results by eliminating the need for imperative build scripting. OpenMake Meister supports a collaborative engineering process that allows developers to contribute to the build while ensuring standardization and consistency across all platforms, compilers and IDEs. This standardization creates an environment for builds to be repeatable and secure from all states in the life cycle including development, testing, production and emergency updates.

OpenMake Meister addresses the primary problem with the uncontrolled, enterprise build process – the imperative coding of make, maven, Gradle, and Ant build scripts. Manual scripts cannot provide the consistency and traceability that is required to fully audit the movement of a source code change from development to production. Manually coded solutions do not provide the ability to perform the necessary detailed dependency analysis and footprinting capabilities needed to confirm matching source to executables. In addition, manually coded scripts can be executed without any level of security checking. Even if a "scheduling" type tool is used to control the manual scripts, it does not prevent the scripts from being executed outside the control of the "scheduling" tool.

To make matters even more difficult for the enterprise relying on manual scripting, the detailed configuration components of a build, including the use of source code directories and compile flags, cannot be easily managed with the use of manually coded scripts. It is not unusual for large development organizations to have thousands of manually coded build scripts. Each of these scripts can be written using various syntax and logic flow. To control the content of the scripts, a large team of build script experts would need to be employed to validate the correctness of the scripts before each build was executed.

This whitepaper summarizes the OpenMake Meister features that are designed for build auditing and traceability. It covers best practices for setting up “Groups” and “Users” for managing access privileges on the OpenMake Meister build configuration components and on build execution.

Common Build Challenges

There are common factors that contribute to the reliability of any scripted build process. The first and foremost challenge is knowing which version of the source code, compiler libraries and third-party components are to be used in the build, not only for system level builds, but also for daily development compiles. The environment variables INCLUDE, LIB, VPATH and CLASSPATH are critical as they determine what versions of the files are to be included at build time. These environment variable settings are specific to each machine. Because of the individual settings of these environment variables, the build is dependent upon the build machine. It may be close to impossible for two machines to reproduce the same build results, since developers compile applications on their own workstation, and no two developers’ machines are likely to be configured in the same way.

When building applications using scripted build processes, the standardization of these environmental factors is almost impossible. Without the assurance of a non-scripted based build process, there is greater potential to introduce errors that may not even be noticed until the application is running in production or installed by the end user.

The OpenMake Meister Solution

OpenMake Meister provides a standardized method for creating and managing build control files that replace imperative manual scripting. Build Control files can be generated to build a single object, supporting developer daily compile activities, or generated for a complete application, containing hundreds of inter-dependent modules. Builds can be managed from an empty build directory pulling source code from a pre-defined Dependency Path, or by retrieving source code from a Version or Artifact Repositories. OpenMake Meister controls the environment variable settings such as LIB, INCLUDE and CLASSPATH so that regardless of the build machine, the build results are the same.

OpenMake Meister fully supports Ant Tasks for completing Java builds. Meister extends the use of Ant without the need for manually coding Ant based imperative scripts. In the place of hard-coded scripts, Meister includes a library of reusable PERL modules and a central Build Knowledge Base that serves as a build Operator. To call the Operator, each team declares their application specific information that is then passed to the central Operator to execute the build process. Declarations are done through *Target Definition* files, used in conjunction with the Meister internal Knowledge Base that references the templated PERL modules to generate the Build Control file specific to the application. Build Control files are completely portable regardless of application development tool or operating system.

In addition to replacing the manual method of imperative build scripting, Meister provides strict control and auditability through the following features:

- Foot printing and Build Audit Reporting – providing specific details regarding the content of a build.
- Dependency Paths (VPATH) for all languages.
- Build Types.
- Public Build Jobs - allowing for pre and post build commands.
- User Groups and Privileges – allowing for the creation of build roles and responsibilities.
- Support for over 200 different compilers and development tools.

Details of the Meister features are described below.

Footprinting and Build Audit Reporting

Footprinting, or the use of a Build Audit Report, is critical in the ability to trace build results. Without a Footprint or a Build Audit Report, there is no way to guarantee matching source to executables, or even know what source was used to create and executable.

A Build Audit Report is a listing of the source code and intermediate binary objects that went into the build of a Target presented in a report format. It is the information "left behind" after a build has completed. A Footprint is the process of embedding this information into the binary itself. If the information is embedded into the built object, it is retrieved using the “omident” program. The Build Audit Report can be used as an alternative, if changing the size of the built object is a concern. Footprinting and/or Build Audit reporting are key ingredients for matching source to executables. The Build Audit Report and Footprint capture the following information:

- Source code information including the directory the file was found in and the files size, date and timestamp. If a versioning tool is in place, Meister can retrieve and report on the source version information allowing the user to trace the actual version of the source back to the versioning tool. Meister performs this activity through the real-time dependency scanning. As the build executes and dependencies are located, Meister saves the information for reporting purposes.

- A list of the machine specific environment variables set during the build which includes:

    - User Name
    - Machine ID
    - Compiler Path Settings
    - Meister Project and Dependency Directory Settings
    - Any environment variable “set” at the time of the build.

Example Build Audit Report

Below is an example Build Audit Report.

Bill of Materials Report for

C:\ PROGRA~1\OpenMake\examples\TestBld\bin\hello.exe

Project Variables:

Built on FRISBEE by USERNAME at 09/05/2002 13:41:42

Environment Variables:

APPL=BUILD DEMO

CLASSPATH=

.;c:\VisualCafe\BIN\COMPONENTS\SYMBEANS.JAR;c:\VisualCafe\BIN\COMPONEN TS\

COMPILER=C:\PROGRA~1\MICROS~2\VC98

COMPUTERNAME=FRISBEE

ComSpec=c:\winnt\system32\cmd.exe

EBU_HOME=C:\ORANT\OBACKUP

HOMEDRIVE=C:

HOMEPATH=\

LOGONSERVER=\\FRISBEE

MAKEFILEDIR=

MSDevDir=C:\ProgramFiles\MicrosoftVisualStudio\Common\MSDev98;

C:\ProgramFiles\De

NUMBER_OF_PROCESSORS=1

OPENMAKE_HOME=C:\openmake\bin

OS=Windows

Os2LibPath=c:\winnt\system32\os2\dll;

PATHEXT=.COM;.EXE;.BAT;.CMD

PROCESSOR_ARCHITECTURE=x86

PROCESSOR_IDENTIFIER=x86 Family 5 Model 8 Stepping 1, GenuineIntel

PROCESSOR_LEVEL=5

PROCESSOR_REVISION=0801

PROJECTVPATH=

.;$(REFDIR)/build demo/development;$(REFDIR)/build demo/ test;$(REFDIR)/

PROMPT=PG

PWD=C:\PROGRA~1\OpenMake\examples\TestBld

Path=

c:\Perl\bin;c:\winnt\system32;c:\winnt;C:\ORANT\BIN;C:\ORANT\OBACKUP\B IN;C:\PRO

REFDIR=C:\PROGRA~1\OpenMake\examples\REF

```
REF_DIR=p:
        STAGE=DEVELOPMENT
        SystemDrive=C:
        SystemRoot=c:\winnt
        USERDOMAIN=FRISBEE
        USERNAME=Administrator
        SERPROFILE=c:\winnt\Profiles\Administrator
        VPATH=
        .;$(REFDIR)/build demo/development;$(REFDIR)/build demo/ test;$(REFDIR)/build
         XVT_DSP_DIR=c:\openmake\xvtdsp45\w32_x86
        include=
                C:\Program Files\Microsoft Visual Studio\VC98\atl\include;C:\Program Files\
         lib=C:\ProgramFiles\MicrosoftVisualStudio\VC98\mfc\lib;C:\ProgramFiles \Microsoft
        windir=c:\winnt

Source Code Dependencies:
Date      Time    Size   Target Dependencies

09/05/1999 13:41:26 1968   bin\hello.res
08/31/1999 21:48:43 4450   C:\REF\build demo\release\hello\hello.rc

08/31/1999 21:48:43 529   C:\REF\build demo\release\hello\resource.h

08/31/1999 21:48:42 1078   C:\REF\build demo\release\hello\HELLO.ICO
09/05/1999 13:41:38 17794  bin\hello.obj
09/03/1999 16:28:57 3495   C:\REF\build demo\development\hello\hello.cpp

08/31/1999 21:48:43 469   C:\REF\build demo\release\hello\stdafx.h

08/31/1999 21:48:42 1495   C:\REF\build demo\release\hello\hello.h

09/05/1999 13:41:36 3359940 bin\hello.pch
08/31/1999 21:48:43 501   C:\REF\build demo\release\hello\stdafx.cpp
09/05/1999 13:41:40 1734   bin\hello.lib
09/05/1999 13:41:36 584   bin\stdafx.obj
08/31/1999 21:48:42 12    C:\REF\build demo\release\hello\hello.def
09/05/1999 13:41:40 538   bin\hello.exp
END: Bill of Materials Report for C:\PROGRA~1\OpenMake\examples\TestBld\bin\hello.exe.
```

Dependency Directories

Dependency Directories are the way in which the location of source code is controlled by Meister. Dependency Directories are passed to the compilers by Meister during the build. Dependency Directories contains a list of approved directories that can be used in the build. If the file is not found in the declared Dependency Directories, Meister displays an Error Message indicating that the source was not found.

Dependency Directories are important because they carefully control how source code gets into the build. Access to defining Dependency Directories can be controlled based on Group privileges. This means that only certain individuals can define the high-level directories that will be used during a build.

Build Types

Build Types are unique to Meister. Build Types control the entire build configuration components including compile flags, link flags and build parameters. For managing enterprise builds, Build Types are critical for build traceability since all configuration data used in the build is administered in a single location and documented. Unlike in the case with manual scripts, the build configuration components are controlled using Meister. Access to Build Types can be controlled based on Group privileges. This means that only certain individuals can define build configuration data that will be used during the build.

Public Build Jobs

Build Jobs are executed to perform a build on a particular endpoint, executing pre and post build commands. Build Jobs can be local to a developer, called a “Private” Build Job, or shared for all approved Users called a “Public” Build Job. Private Build Jobs can be promoted by a developer to a Public Build Job. When this occurs, the Public Build Job can be repeated anyone with access.

Public Build Jobs are important for Audit control over the build process because they provide the repeatability necessary for builds to move between a “Development” stage to a “Pre-Production Control” stage. More importantly, access to Public Build Jobs can be assigned to Groups. This means that only certain individuals can see and execute the Public Build Job. For environments where the use of Secure Remote Build servers are in place, the use of Public Build Jobs can streamline the Build process and control whom has access to perform the Builds.

Managing Groups and Users

Managing Users is a simple task performed by the Meister Administrator. Groups are defined by the Administrator to have access to Dependency Directories, Build Types and Public Build Jobs. Much of the work of managing Users is automatically handled based on default “User” and “Administrator” Groups. The access privileges defined to these default Groups can be modified, or custom Groups can be created.

Conclusion

Securing the software build process requires a disruption in the way we manage the software compile and link step of our CI/CD pipeline. Old ways of writing imperative build and package scripts are prone to errors and easy to hack. A better method is to adopt a declarative build process that can be secured and managed through a central control point. Securing the build process involves tightly controlling your software supply chain including both external and internal artifacts. The build step needs to include clear audit trails of all objects consumed during the process, down to the compiler. Meister allows you to achieve this level of control improving the consistency of builds and providing the guard rails need for securing builds in today's CI/CD pipeline automation.

Company Overview

OpenMake Software started the evolution of builds in 1995, serving mainly the financial community with the mission of delivering a 100% insulated build process that were also fast. The OpenMake Software team understood the ins and outs of software compiles and links, and how easily a build could be the bottleneck of the software delivery process and be easily compromised on accident or on purpose. With this mission in mind, OpenMake Meister was created and has been serving large enterprises for over 25 years, the longest serving solution in the DevOps ecosystem. Meister has been sold and distributed by Broadcom for over 20 years.

For more information reach out to us at request-info@openmakesoftware.com.