

Software Build Analysis

Eliminate Production Problems
to Accelerate Development

Ben Chelf, CTO

Introduction

Software builds are a pulse of progress for any software development cycle. An agile, well-designed build system should seamlessly assemble software as it is being developed in the most efficient way possible. Today, too few organizations are able to bring this ideal to fruition. Many development projects are plagued with builds that are too slow, break often, or cannot easily be modified. These limitations constrict the ability of developers to incorporate new or acquired code into existing applications. This in turn slows the delivery of more value to end users and more revenue to the business.

Build systems are a common point of delay in software development because development managers often lack clear insight into the production process, creating dangerous uncertainty around exactly what code is being shipped to customers.

These software production failures threaten time-to-market by slowing the development cycle and introducing risk surrounding the delivery date of an application and the security of the final application itself. New technological breakthroughs now allow automated analysis to move beyond examining the source code and architecture of software to examining the actual build system with the intent of eliminating the traditional problems facing the software build process.

This paper reviews how development organizations can understand and improve the build process that is ultimately responsible for assembling software for delivery. To begin, this paper will discuss builds in the overall context of the software development process and review the known problems of existing build systems. Second, existing management technologies that support builds and the inherent limitations of these technologies with regard to build visibility will be explained. Finally, new technology that enables organizations to gain deep insight into the build system at a process level will be described, along with the role this technology can play in eliminating software production problems altogether.

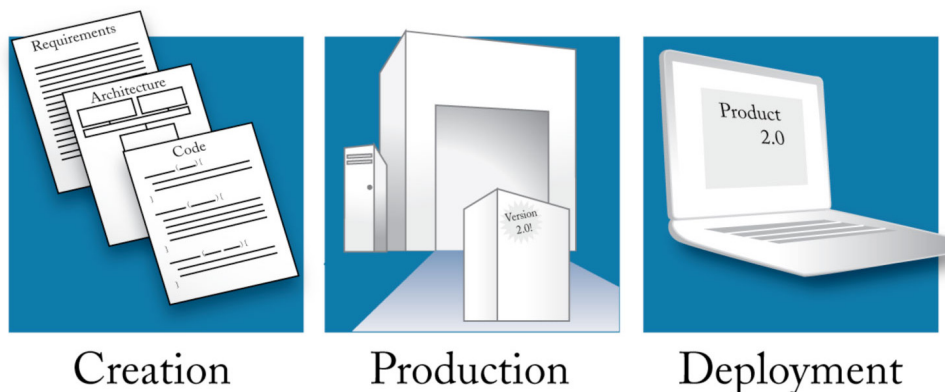
Software Development and “Builds”

Before defining software builds, it is helpful to first consider the overall software development process. As *Figure 1* illustrates, the creation phase of software development encompasses all work that a human must do to create an application. This includes writing requirements, designing the architecture of the system, and writing the code (the typical Requirements, Design, Code phases of Application Lifecycle Management). However, it also includes other activities such as test case creation, build script creation, documentation creation, and so on.

The second phase of the software development process is production. In this phase, machines execute processes to produce a product that the organization knows is fully functional and can be shipped with confidence. During production, all of the human-generated artifacts from the creation phase are put through the production process, which includes executing the build system, automated tests, packaging scripts, as well as any human-executed tests that must be performed to ensure that application requirements are properly implemented. You can think of the production phase as an assembly line for software.

The last phase of the software development process is deployment, where software is in the hands of end-users. This phase is outside the scope of this paper and is only mentioned to present a complete view of the full software development process.

Figure 1: The Software Development Process



It should be noted that the figure above does not dictate a strict ordering of software development processes. For example, all steps of the creation stage may not happen prior to production, or deployment may occur during the beta process before official release, etc.

The Role of Build in the Creation and Production Phases of Development

There are two primary aspects to a software build – the specification of a build and the execution of a build. To fully understand the role of builds in the overall software production process, both of these aspects must be considered in concert. Unfortunately, a large gap exists between the specification and execution of many builds. At present, there are some tools that help create builds, but these tools treat the execution of a build as essentially a black box that affords developers no visibility into the build itself. This is problematic because at their essence, builds are similar to any other software application. For example, consider the normal debugging task a developer must go through when software fails. First, the developer writes code, then that code is executed. If you asked a developer to diagnose the failure of executing code without providing any insight into how that execution tied back to the source code, they would have a very difficult time debugging the failure.

The primary issue facing builds today is the barrier between build specification (in the creation phase) and build execution (in the production phase). Specifically, the tools used to specify the build are disconnected from the tools that are used to manage the assembly process. As software systems continue to become more complex due to the use of third party code, outsourced code, aging legacy code and so on, this barrier can mean disaster for developer productivity and the overall integrity of the shipped application.

Existing Build Tools and Technologies

When creating software, a build engineer typically works with the development organization to specify exactly how the software system should be constructed. These build system specification packages can include `make`ⁱ, `Ant`ⁱⁱ, `Jam`ⁱⁱⁱ, `Visual Studio`^{iv}, `scons`^v and so on. There are a plethora of options to specify how a given software system should be assembled. Some development organizations also use the build system specification as a harness for their testing system. It is not uncommon to see a “make test” target within a Makefile in conjunction with other scripts that are executed to run the automated tests for an application.

During the production phase, the build itself is executed. The tools that development organizations use here have been traditionally home grown, but in the last few years, specific “build management” products have surfaced to help organizations automate the actual assembly process. Examples include commercial tools like `BuildForge`^{vi} and `Electric Commander`^{vii}, or open source tools such as `Hudson`^{viii} or `Cruise Control`^{ix}. Some of these tools are actually considered continuous integration technologies, meaning not only do they facilitate executing the build to assemble software, but they also manage the automated test function of the production phase of software development. `UrbanCode's Anthill`^x is one such technology.

Importantly, conventional build management tools and continuous integration technologies treat the software assembly process as a black box. So while the software will be assembled and tested when the management tool executes the correct commands, these tools do not analyze the link between the original source code, how it is supposed to be assembled, and the nature of the tests that are executed and how they relate to the source code. Because of this, they are limited in their ability to optimize of the overall software production process. They can execute build commands and test the result of a build, but they cannot see if what they are building and testing is correct.

There are some instances that run counter to this generalization. For example, some parallel build systems enable optimization (e.g., `make-j4`) or accelerate existing serial builds by adding parallelization. However, most of these tools simply view a build system and afford no analysis capability to aid developers during the production phase of the software development process. While there are some optimizations that can occur at this granularity (e.g., farming out multiple different builds and tests to different machines simultaneously), the potential benefits are limited in scope.

The Cost of Broken Builds

Software developers do not need to be reminded that builds cause an array of difficult, time-consuming problems since most development teams have faced broken or brittle builds. Generally, a build will fail in one of three ways:

- **Loud failure** - When a certain file doesn't compile, or the link step fails so the build comes to a halt.
- **Silent failure** - When the build system succeeds but does not produce the correct assembled software system because of a bug in the build system. For example, a file that changed was not recompiled, or an old library was linked into the executable.
- **Intentional failure** - When the build system has been compromised by a malicious user who has access to the build (such as an outsourced development team) and the resulting shipped product contains backdoors, etc., because the assembly line has not been controlled.

Of the build failure types above, silent failures merit further explanation because they are particularly damaging to the software production process. Unfortunately, silent failures must be addressed with traditional debugging techniques. If the development organization is lucky, the failure will occur in the test suite, but sometimes it slips into the field, making it even more costly to eliminate. To compound the challenge of silent failures, developers often take the assembly process for granted; making it one of the last areas to be investigated after the developer has exhausted investigating the source code itself. In other words, silent build failures can appear to be problems in the source code even though they may actually be problems in the assembly of that source code.

Unfortunately, these three failure types are not the only problems facing build systems today. Compliance and standardization is also a problem because there is no way to ensure that all files are compiled the same way. Similarly, there is no way to ensure that the development team knows exactly what is in the shipped product. A list of other potential build challenges is presented in *Table 1*.

Table 1: Potential Build Challenges

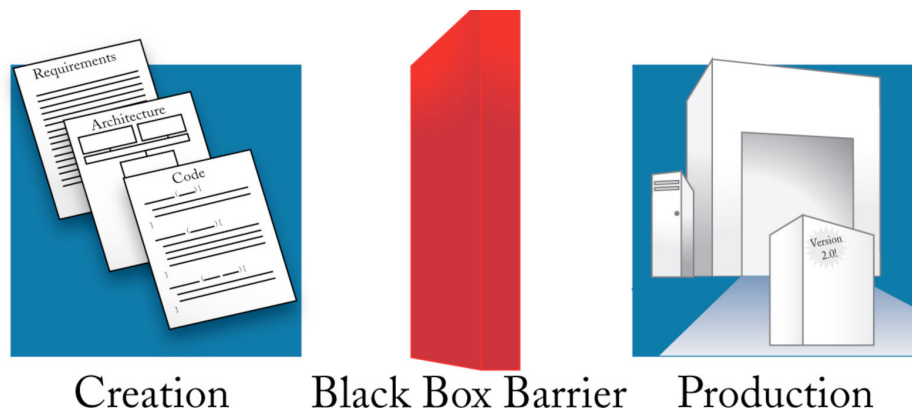
Standardization in assembly	Making sure that every single file is compiled the same way (e.g., same optimization level, same debugging flags, same defines, etc.)
Version assurance of shipping products	Making sure that every file that contributes to a shipped version of your product is the correct version of the file
Root-cause analysis for legacy failure	Reproducing old builds in exactly the same fashion from the original assembly so a debug version of that build can be used to track down a failure from a system that has long since shipped
Composition analysis	Accurately assessing the set of files that are necessary in creating your shipped product and knowing exactly where those files came from (e.g., third party software that you purchased, open source software that is licensed with the GPL, etc.) to confirm legal compliance
Hardware utilization	Keeping the hardware that you have dedicated to development tasks running processes instead of sitting idle

At most organizations, the build is a second class citizen in the development process—unlike almost every other industry, where the assembly line is understood to be as critical to the profitability of the business as any other part of the design and sales process. For example, an auto manufacturer will keep a detailed accounting of the cost of materials and the cost of assembly. The time between each step is measured precisely and kept as short as possible. If a problem occurs on the assembly line, it is tremendously expensive for that manufacturer so it is addressed immediately.

Yet, in software production, build systems remain an area of development that is notoriously understaffed. While such under-staffing may have been acceptable in the old world of software development with relatively small, self-contained software systems, in today's world, it is not sufficient. With so many software components coming from external sources such as third parties, open source and outsource vendors, development managers need to pay more attention to the software assembly process. As evidence of this, in an internal survey performed in the summer of 2008, more than 67% of Coverity customers report they have plans to address one or more of the above described problems within the next year.

Most of the problems above can be traced to the barrier that exists between build creation and production as illustrated in *Figure 2*.

Figure 2: The Build Barrier



This barrier between build creation and build production exists when a development organization specifies the build and test system, but after specifying it, treats it as a black box. In this common scenario, build management solutions deal with the build environment without understanding the nuts and bolts of each specific build. They have a macro-view, not the micro-view necessary to identify bottlenecks in build production, generate reports with a complete bill of materials for a software package, and debug build failures.

By this token, if a build takes ten hours to run, the build management facility may find that acceptable. This is because when builds are treated as a black box, there is no analysis to determine if the build should take one hour, five hours, ten hours or more. There is no fundamental understanding of whether a build is executed efficiently or not. The drain protracted builds take on a development organization includes lost developer time, slow development cycles, and the wasted use of equipment and energy. Only by opening the black box of software builds can development organizations solve their software production problems by identifying these types of bottlenecks while improving the overall integrity of their code.

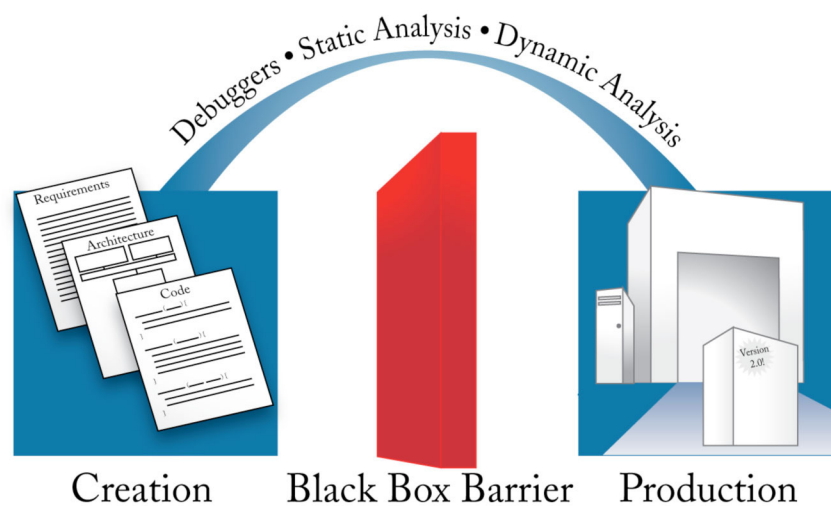
Build Analysis Defined

To bridge the gap between builds in the creation and production phases of software development, the ability to automatically observe builds at a very granular level is required. To be effective, this technology must monitor the precise processes that are executed to assemble and test applications. We define this automated observation as Build Analysis. In this section, we discuss innovative new technology from Coverity that solves the build analysis problem. We will also explain the ways that this technology can be applied in the future to solve many of the problems in build systems.

Historically, a number of bridges have existed between the creation and production phases of software as it pertains to source code. Classic debuggers such as `gdb`^{xi} take the executing system and help the developer tie what happens as the code runs to what happens in the source code. Dynamic analysis like Purify^{xii} and Coverity Dynamic Analyzer^{xiii} instrument a running program so that as it executes, additional memory and concurrency properties can be checked for correctness and linked back to the source code.

Static analysis like Coverity Prevent^{xiv} goes the other direction by analyzing the source code itself for defects that will certainly manifest themselves at runtime and shows the path through the code that leads to the error (Note: This is different than a coding standards checking tool or compiler warnings as those types of reports are completely contained within the creation phase of software development). These other bridges are illustrated in *Figure 3* to emphasize the need to tie the creation and production phases more tightly together.

Figure 3: Bridges between software creation and production



Implementing Build Analysis

The primary goal of build analysis is to create a map of all processes that are executed during software production. For each process, this map contains a complete set of all the input files to that process and all the output files that are generated by that process. This provides a complete dependency tree for the entire software production process with both file and process granularity.

There are some practical constraints to consider when discussing how to implement build analysis. To begin, it should be agnostic to existing build creation tools. Due to the heterogeneous nature of build creation tools, recording a build with respect to any particular build creation tool (e.g., make, ant, etc.) may produce a good result for that particular build but would not be useful for other types of builds.

Second, when recording a build, build analysis must not significantly impact the overhead required by the build itself. If the act of recording a build causes it to take ten times as long to execute, obviously any benefit from build analysis would be more than consumed by the extended build. While the overhead that is tolerated can vary widely from one organization to the next, Coverity's goal is to constrain overhead to ensure it has a minimal impact on performance.

Lastly, for build analysis to be effective, the information it captures must be complete. A recording of a build that is not semantically equivalent to the original build cannot provide sufficient value to development teams as they work to connect the creation and production of applications. In other words, effective build analysis must be able to re-execute a given build in its entirety, without relying on the existing specification of the build.

The Build Analysis Solution

The solution for build analysis lies in recognizing that every build system, regardless of how it is specified, must interact with the operating system in order to perform its task. Most modern day operating systems provide a facility to observe system calls as they are made, or a facility that intercepts calls to dynamically linked libraries. Coverity employs both these tactics to perform build analysis.

For example, on the *NIX operating systems, by leveraging the functionality of `ld.soxv` to intercept calls to `exec`, `read`, `write`, etc., Coverity's build analysis can see, for every process that is created by the build system, all the files that it reads and writes. The way this works is through the `LD_PRELOAD` environment variable. When this environment variable is set, the specified libraries are loaded before all others when executing an application. Since the system calls that need to be recorded are wrapped with calls from the dynamically linked `libc`, by re-implementing those functions as a record and pass-through, we can take note every time one of them is called. On Windows, there is a package call `Detoursxvi` that enables similar monitoring at the operating system level (as opposed to the `libc` level).

By keeping the intercepting code small, build analysis can deliver a very low overhead for this type of recording. Also, because it is happening at the dynamic library and operating system level, build analysis is completely agnostic to any build system and requires no configuration to start recording a build. Because build analysis is operating at this very granular level, its recording is not a build specific recording. This means build analysis can record any set of processes, including the execution of the test suite, even if that suite is implemented separately from the build system.

Challenges to Effective Build Analysis

To make the build analysis process complete, a number of significant technical challenges had to be overcome. This section will highlight a few of the more interesting cases, including file reuse, renames, the concurrent execution of child processes, temporary files, and completeness of interception. To track every file and process operation, there were over 80 different functions that had to be intercepted at the system level.

Files that change between builds are complex because they may have the same file name and location used to store two different files during the production process (e.g., the file is created, removed, and then recreated). While this problem is difficult, the problem of directory renames is even more difficult. Consider the following case:

```
gcc -c foo.c -o abc/foo.o
mv abc output
gcc main.c output/foo.o -o foo.exe
```

In this example, the “abc” directory can also have subdirectories with files that cause even deeper dependencies. The presence of renames can lead to missing dependencies. To account for this, build analysis must simulate a virtual file system and “replay” the build to resolve dependencies correctly (e.g., in the example above the second gcc command relies on the output of the first, even though the two never directly operate on the same file name). Symbolic links are a similar issue. For example, consider if one of the directories in a symbolic link chain is changed:

```
ln -s x86 arch
... compilations that access arch/machine.h
rm arch
ln -s solaris arch
... compilations that access arch/machine.h
```

The solution to the problem above is similar to the solution for renaming directories.

Programs like tar are relatively straightforward to record, though the resulting dependency graph can contain edges that seem spurious. For example, if subcomponent A is built and the results ‘tarred up’ then ‘untarred’ elsewhere and used by subcomponent B’s build, all processes in B will have a dependency on the tarball – which depends on all files in A. This is technically correct but a bit coarse.

The concurrent execution of child processes is another challenge to build analysis. For example, if process A spawns child processes B and C, they will all run together unless A waits for the completion of B before beginning C – and usually this is the case. Consider a scenario where a gcc wrapper can execute the preprocessor first, wait for it to finish and then execute the compiler. The problem here is that there are some cases where both child processes are active at the same time, such as the use of pipes. In these types of cases it is hard to determine what the file dependencies are, because there is no clear order in which the file accesses are performed.

Temporary files can also be a problem for build analysis because they tend to introduce false dependencies. If a temporary file name is recorded as part of the build, and at a later point some other process reuses that name, then the build analysis tool may think there is a need to rebuild when there is not. This problem is solved with build analysis by using heuristics such as a fixed set of temp directories, or an algorithm to detect files that are created and removed within the build.

Lastly, there is the issue of “rolling up” certain processes. Some processes invoke other processes, but should probably be considered atomic. Because the goal of build analysis is to record builds in a semantically equivalent fashion for the purpose of replay and analysis that is relevant to the user, build analysis must understand which processes should be considered atomic and which should not. For example, consider the processes that execute when calling a compiler:

```
gcc -o a.out foo.c bar.c baz.c
```

For some versions of the gcc compiler, this call executes other processes like cpp (the C preprocessor), cc1 (the compiler proper), as (the assembler) and ld (the linker). During this process, various temp files are used, and while fine granularity is important, it’s also important to know how far to take this concept. It is unlikely the development organization will think it important to know the many sub processes of some of the fundamental building block tools in their tool chain!

Business Impact of Build Analysis

The analysis of builds provides valuable new insight into existing software production processes. Once extracted, a map of build data can be analyzed in a multitude of ways to improve build efficiency and build integrity. Benefits of build analysis include validation of builds, verification of compilation options, true product composition, and build acceleration – each of which is explained below.

Validate Builds

The first area of build analysis that provides immediate value to development teams is checking and validating builds. Build analysis finds problems that have traditionally plagued build processes. One simple, but dangerous problem that it can detect is the correct and uniform use of a build's 'clean' function. Deep analysis of real processes detects whether cleaning has been uniformly used and all created objects and executables have been removed. This ensures no stale files exist when constructing an application, allowing teams to avoid the silent build failures mentioned above.

Verify Use of Compilation Options

Build analysis can also check and verify the uniform use of compilation options. Many organizations believe they have a standard policy for options used with certain build targets (e.g., DEBUG, RELEASE). However, many times these standard scripts degrade over time as they are recycled via 'cut and paste' in creating build scripts for new functions or applications. This degradation can result in significant performance and security vulnerabilities if left unchecked.

For example, in modern Microsoft compilers, there are specific security settings that control the code generation surrounding buffers in the program to reduce the risk of a buffer overflow being exploited by a malicious user. However, because it is a compiler option, it is the responsibility of the development organization to ensure that this is correctly set.

Another example is DEBUG compiler options. As part of the software development and testing cycle, software is built using DEBUG compiler options. These options help resolve software failures by adding extra debugging instructions or opening additional ports to assist with test automation. If these options are inadvertently left on when you ship software, they can have a devastating performance or security impact.

True Product Composition

Another valuable use of build analysis is to deliver a ‘true composition’ of the result of the build, your shippable product. Development teams understand the need to accurately build the latest changes in an application, so that verification and validation accurately tests changes, and those changes are delivered to the customer. This is even more critical at organizations with corporate audit or government regulations that require proof all the code in a delivered application is code that was actually tested. Many organizations control this information through the use of Software Configuration Management (SCM) systems that control access to and versions of the software.

The integration of SCM systems in the build process can provide an auditable list of files that comprise a given build. However, doing so creates risk for companies because there is no guarantee that the files that have been checked out are actually the files contained in a specific build. SCM systems also lack the means to verify if given file list represents the entire list of files that comprise a specific build. Therefore, it is impossible to know if a build script has included files from a hard coded local directory, instead of the correct checked out version of the source code. Only by analyzing what actually was assembled can teams identify these inconsistencies.

Build Acceleration

Once build processes are understood and build defects have been corrected, the same analysis can be utilized to improve build speed. The turnaround time for builds has historically been problematic at most organizations because they simply take too long. As discussed earlier, extended build times cause developer inefficiency and tax the resources of an organization, but they also limit the ability of teams to use agile methodologies which require faster, more frequent builds to maintain development momentum.

Because build analysis can understand the flow of processes, in particular the dependency map of processes to the files they use, it can be used to improve build speed. This is accomplished by using build map data to understand what processes can be clustered together and the order that those clusters can be built. This information can then be used to feed into build management systems that control clusters of machines, or to restructure the build process for use with things like the `make -j` option, or Load Sharing Facility (LSF) from platform computing^{xvii}.

Another valuable build acceleration option is the use of build map data to accurately create an incremental build environment. With the entirety of the build process known at the granularity of individual processes and file dependencies, the modification of any artifact in the build can be used to understand the impact of that change and also to rebuild only the necessary parts. This allows developers to improve both the speed and accuracy of builds. The gain in build accuracy is accomplished by ensuring build artifacts, such as library components or jars, are not unnecessarily rebuilt and delivered into testing environments that trigger excessive retesting.

The combined capabilities above can deliver the following benefits to development teams:

- **Improve Software Quality** – Automatically identify the source of defects that occur due to improper or accidental inclusion of wrong object files
- **Reduce Wasted Time** – Recapture productivity lost due to unnecessary build bottlenecks such as broken make files or redundant and long running processes
- **Prevent Security Risks** – Halt the introduction of malicious or unintentional vulnerabilities through software components or open source packages that may contain known security problems
- **Stop Compliance Violations** – Put an end to the creep of compliance violations caused by the lack of visibility in the assembly process with a comprehensive ‘bill of materials’ that confirms the version and origin of all internal and open source code in your product

Coverity Build Analysis is part of the Coverity Integrity Center, and is a natural complement to Coverity’s other precision software analysis capabilities which include Coverity Architecture Analysis, Coverity Dynamic Analysis, and Coverity Prevent, the industry-leading static analysis product. The combination of Coverity’s precision software analysis offerings enables customers to identify quality, performance and security flaws at each step in the software development process.

Conclusion

Software is becoming more complex to assemble. Previous generations of build tools are not sufficient to ensure high integrity software because they treat the production process as a black box, limiting what they can understand about this important phase of software development. Coverity recommends the use of existing build production management, composition analysis and optimization tools. We also recommend development teams consider augmenting these tools to address quality, performance and security issues that can compromise the integrity of your software – issues existing tools are simply not designed to address.

Build analysis can help development teams understand and eliminate the software production problems that worsen as software grows in magnitude. However, build analysis is only possible through automation due to the size and complexity of most software systems. Because of these factors, an automated solution that improves and corrects today's flawed software production processes can offer significant benefits to development teams, including reduced build failures, better utilization of existing hardware for faster production throughput, and complete knowledge of the shipped product. By understanding and advancing the science of software builds and the production process in general, you can fundamentally improve the way your software is produced.

Free Trial

Request a free Coverity trial and see first-hand how to rapidly detect and remediate serious defects and vulnerabilities. No changes to your code are necessary. There are no limitations on code size, and you will receive a complimentary report detailing actionable analysis results. Register for the on-site evaluation at www.coverity.com or call us at (800) 873-8193.

About Coverity

Coverity (www.coverity.com), the software integrity company, is the trusted standard for companies that have a zero tolerance policy for software failures, problems, and security breaches. Coverity's award winning portfolio of software integrity products helps customers prevent software problems throughout the application lifecycle. Over 100,000 developers and 600 companies rely on Coverity to help them ensure the delivery of high integrity. Coverity is a privately held company headquartered in San Francisco with offices in 6 countries and more than 130 employees.

References

- i http://www.gnu.org/software/make/manual/html_node/index.html
- ii <http://ant.apache.org/>
- iii <http://www.perforce.com/jam/jam.html>
- iv <http://msdn.microsoft.com/en-us/vstudio/products/default.aspx>
- v <http://www.scons.org/>
- vi <http://www-01.ibm.com/software/awdtools/buildforge/enterprise/>
- vii <http://www.electriccloud.com/>
- viii <http://hudson.gotdns.com/wiki/display/HUDSON/Meet+Hudson>
- ix <http://cruisecontrol.sourceforge.net/>
- x <http://www.anthillpro.com/html/default.html>
- xi <http://www.gnu.org/software/gdb/>
- xii <http://www-01.ibm.com/software/awdtools/purifyplus/>
- xiii <http://www.coverity.com/html/coverity-prevent-static-analysis.html>
- xiv <http://www.coverity.com/html/coverity-prevent-static-analysis.html>
- xv <http://unixhelp.ed.ac.uk/CGI/man-cgi?ld.so+8>
- xvi <http://research.microsoft.com/en-us/projects/detours/>
- xvii <http://www.platform.com/Products/platform-lsf>

Headquarters

185 Berry Street, Suite 2400
San Francisco, CA 94107
(800) 873-8193
<http://www.coverity.com>
sales@coverity.com

Boston

230 Congress Street
Suite 303
Boston, MA 02110
(617) 933-6500

UK

Coverity Limited
Magdalen Centre
Robert Robinson Avenue
The Oxford Science Park
Oxford OX4 4GA
England

Japan

Coverity Asia Pacific
Level 32, Shinjuku Nomura Bldg.
1-26-2 Nishi-Shinjuku, Shinjuku-ku
Tokyo 163-0532
Japan