

OVERVIEW

West Coast Labs found the SIP implementation on the Mu-4000 to be extremely thorough, providing deep protocol coverage, as measured by the three dimensions of code coverage, in multiple protocol states, combined with finding, enumerating and identifying faults – including those of high severity. All SIP suites were executed with minimal effort and allowed for exact fault isolation, regression testing, and high levels of test automation.

Product Test Report

February 2008



Mu-4000 Analyzer

Reliability, Availability and Security
(including Negative) Testing Study

Vendor Address:

686 W. Maude Avenue, Suite #104
Sunnyvale, CA 94085, USA

Vendor Telephone Number: +1 408 329 6330

Product: Mu-4000 Analyzer

Test Laboratory Details

Test Laboratory Name:

West Coast Labs

Test Laboratory Address:

Unit 9 Oak Tree Court, Mulberry Drive,
Cardiff Gate Business Park, Cardiff, CF23 8RS
United Kingdom

Test Laboratory Telephone Number:

+44 (0) 2920 548 400

Date: 10th January 2008

Issue: 1.0

Author: Rob Tanner, Matt Garrad

Contact Points for Technical Queries on the Test Report

Contact Name: Matt Garrad

Contact Telephone Number:

+44 (0) 2920 548 400

TABLE OF CONTENTS

| | |
|------------------------------|----|
| Executive Summary | 4 |
| Test Overview and Objectives | 6 |
| Test Environment | 6 |
| Test Results | 8 |
| Summary | 10 |
| Appendices | 10 |
| West Coast Labs Disclaimer | 15 |

Executive Summary

Reliability, Availability and Security testing (including Negative testing) for software and hardware systems was in its infancy five years ago, but now is a rapidly maturing marketplace for ensuring higher quality software that provides the highest possible uptime for service providers and other users.

As such, it is important to understand two important points:

- 1) What is reliability testing for software?*
- 2) How can products that purport to do availability, security or negative testing be objectively compared?*

This report illustrates the latter by defining a set of test procedures that show how to do such an evaluation.

Reliability and Negative Testing: History and Evolution

Informal and manually-based negative testing for software has been around for nearly 20 years. Black box and negative testing is defined as "taking an external perspective of the test object to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid input and determines the correct output. There is no knowledge of the test object's internal structure. This method of test design is applicable to all levels of [software testing](#): [unit](#), [integration](#), functional testing, [system](#) and [acceptance](#)."¹

Load testing² (commonly known as performance testing) is a primitive form of negative testing. In a sense, what negative testing in general is all about is finding the edge of the envelope, and what performance testing is all about is overloading the target, usually with valid test cases that are designed to show scalability.

The primary difference between black box and load testing is in the possible universe of results. To really get under the skin of a product or application's network protocol implementation (e.g. Voice Over IP and Session Initiation Protocol <SIP>), black box and negative testing is required to dynamically and statefully interact with the target and send invalid messages in all portions of the state machine(s) of the target. The nature of well-written reliability and negative test cases is that in a sense the target implementation should find them to be unexpected, but a well-written target should handle any unexpected exceptions gracefully, with minimal or no impact on response time, and certainly without crashing or hanging.

Negative testing, including availability and reliability measurements, for software has a parallel with other forms of testing commonly performed in other fields of engineering, for example both the automotive and aviation industries construct their products with safety in mind, and in each case these systems are there to protect the passengers in the case of an unexpected scenario where damage could occur. Negative testing enables everyone from VoIP product developers to Triple Play reliability engineers to design mitigation for the unexpected. Models are built and intentionally destroyed to ensure that they meet or exceed design goals for protection of occupants in a range of extreme conditions.

¹ Wikipedia Black box Testing - http://en.wikipedia.org/wiki/Black_box_testing

² Wikipedia Load Testing - http://en.wikipedia.org/wiki/Load_testing

All forms of engineering, with the notable exception of software engineering, have a long history of negative testing. Networking hardware, software, Airplanes, Automobiles, Electrical Appliances and even clothing is tested by their respective industries for three to four times higher than the minimum respective safety or operating requirements. Autos are tested at high speeds just as performance testing applies high rates of traffic – usually *well beyond* the standard design requirements. Unfortunately, it is the unexpected conditions that any of these products encounters that truly need to make more reliable, available and secure under the most strenuous of conditions and performance testing does not expose these findings.

Reliability, Availability and Security Testing in Action

Negative testing for hardware and software is focused on isolating the unexpected condition, in that software needs to be forced to process inputs that are specifically designed to unearth common programming mistakes. What may be unexpected for a programmer can be systematically expected and implemented by the designers of the negative testing equipment. The media regularly publishes stories about how Internet software has failed in a variety of situations, and as such the average user is becoming more aware of these issues. It is well known that most software has bugs, and negative testing (hitting software, specifically protocol implementations, with unexpected inputs) finds these weaknesses and bugs before they manifest themselves in production systems and cause downtime. Protocol implementations are especially vulnerable forms of software since they cannot control their inputs. Protocol software must be able to “expect the unexpected.”

Software practitioners such as engineers, developers, QA engineers, and so on along with software users have not — until recently — had the commercial (or non-commercial) tools to perform negative testing using systematic, automatic, and scientific methods. A number of these tools are available in the marketplace at the time of writing; the Mu-4000 analyzer from Mu Dynamics is one such example, which has been shipping for more than two years.

Comparative Test Findings

West Coast Labs evaluated the Mu-4000’s stateful protocol fuzzing (negative testing) engine, and identified suitable metrics to make a meaningful comparison between different negative testing platforms. This objective was achieved by leveraging the SIP attack suite on the Mu-4000 to target Asterisk 1.4.0, an open source Voice over IP (VoIP) soft PBX application. West Coast Labs determined through a series of rigorous, repeatable tests that the commonly conceived metric of code coverage is an insufficient evaluation metric for comparing the protocol fuzzing component of any negative testing system.

West Coast Labs found code coverage data should be combined

with:

- 1) the number of unique faults found in an application (using an objective fault-detection criterion), as well as*
 - 2) the weighted severity of any such faults.*
- These three items (coverage, faults, severity) together form a more holistic picture of the critical capabilities of any negative testing platform under investigation.*

For a user's best deployment scenario, the protocol fuzzing engine and black box testing system platforms need to be evaluated holistically against a variety of target intrinsic or extrinsic metrics, and code coverage is just one of these. This is important especially since code coverage is only available when the target's source code is available to the tester, which is commonly not the case with many leading Service Providers, Carriers, Government agencies and other end users who purchase products from many vendors. Even for vendor developers, when source code is available, code coverage metrics do not provide a complete picture of the negative testing platform's effectiveness.

Test Overview

All West Coast Lab benchmark testing concentrated on the SIP protocol used in both VoIP and IP Multimedia Subsystem (IMS) deployments worldwide. Using the Asterisk VoIP application as a target for the Mu-4000 analyzer intelligent protocol fuzzing engine and comparative fuzzing tools, West Coast Labs compiled detailed code coverage data, using the gcov tool, generated on the target after each Mu-4000 SIP protocol test completed. All observed faults were analysed and the severity of each one was assessed.

Please refer to appendices A through E for a more detailed background on the tests and justifications for the selection of VoIP, SIP, and the open source Asterisk application, as well as for information on code coverage limitations, the gcov tool, and fault analysis observations.

Test Objectives

A key objective of this project was to provide the interested reader with the necessary details to recreate the Mu-4000 tests and SIP protocol fuzzing suites, as performed by West Coast Labs, simultaneously ensuring that the published results within this report are reproducible and verifiable. Moreover, interested parties can reliably compare the Mu-4000's integrated protocol fuzzing engine to other negative testing engines or comparable complete systems – especially important for evaluating latency sensitive real-time applications, which may include VoIP, streaming media, broadband internet provisioning, and mission-critical web service environments.

Test Environment

The test environment consisted of a single target machine serving Asterisk 1.4.0 – the latest version available at the outset of this project – which was compiled to work with the open source gcov code coverage tool. Asterisk was installed as a service within a VMware Workstation 5.5 Virtual Machine, running Linux Fedora Core 5 as both the guest and host operating system. VMware was configured to bridge the guest network connection to the underlying Ethernet interface of the host.

A 100 Mbps Ethernet interface on the target machine was directly connected to a 10/100/1000 Mbps attack interface on the Mu-4000 platform, to form a completely isolated test network.

The Mu-4000 was running Release 3.0.2, with SIP protocol version 5047.

The following IP topology was configured:

- the Asterisk target was assigned an IP address of 126.9.1.111 and a 24 bit subnet mask
- the attack interface on the Mu-4000 was assigned an IP address of 126.9.1.110 and a 24 bit subnet mask

The SIP options on the Mu-4000 were:

- destination URI: 111@126.9.1.111
- source URI: 110@126.9.1.110

The following environment configuration details are included with this report:

- the Asterisk makefile, showing code coverage modifications
- the *asterisk.conf* file, showing the configuration of Asterisk inside the VMware virtual machine
- the Mu-4000 XML based analysis template, used throughout testing

To determine the thoroughness and effectiveness of the SIP attack suite on the Mu-4000, West Coast Labs measured:

- *the code coverage achieved by the Mu-4000 when targeting the Asterisk application*
- *the documented bugs found, while eliminating likely duplicates*
- *the resulting vulnerabilities and robustness issues ranked by severity*

Test Methodology

West Coast Labs designed the following straightforward test methodology and procedure, appropriate to each of the SIP test suites on the Mu-4000:

- prior to commencing each SIP test, stop the Asterisk service on the target virtual machine, delete the code coverage data (zero counters), and re-start the Asterisk service, as per the commands below,

```
service asterisk stop
lcov -d 'enter_Asterisk_channels_directory_path_here' -z
service asterisk start
```

- start a test run on the Mu-4000 for each SIP suite, using the pre-configured test template

- once the Mu-4000 test run for the current SIP suite has completed, stop the Asterisk service, generate the code coverage data, and save the data to an archive destination, as per the commands below,

```
service asterisk stop
lcov -d 'Asterisk_channels_directory_path' -c -o
'Asterisk_channels_directory_path'/coverage.info
genhtml Asterisk_channels_directory_path'/coverage.info -o
'Asterisk_channels_directory_path'/
mv
'Asterisk_channels_directory_path'/channels/chan_sip.c.gcov.htm
l 'destination_directory_path'/
```

```
mv 'Asterisk_channels_directory_path'/gcov.css
/'destination_directory_path'/
```

- view the chan_sip.c.gcov.htm and note the code coverage figure

Test Results

West Coast Labs determined that the SIP implementation on the Mu-4000 is organized into nine suites that exercise the ACK, BYE, CANCEL, INVITE, OPTIONS, PING and REGISTER methods, as well as the SIP Torture Test (as identified in RFC-4475), these suites are listed below:

- ACK (unsolicited)
- BYE (unsolicited)
- CANCEL (unsolicited)
- INVITE-ACK-BYE
- INVITE-CANCEL-ACK
- OPTIONS
- PING
- REGISTER
- TORTURE-TEST

West Coast Labs used these nine SIP suites on the Mu-4000 platform to target Asterisk, which produced the following code coverage results:

| | |
|-------------------------|-------|
| ACK (unsolicited) | 16.2% |
| BYE (unsolicited) | 15.6% |
| CANCEL (unsolicited) | 15.6% |
| INVITE-ACK-BYE | 26.3% |
| INVITE-CANCEL-ACK | 26.2% |
| OPTIONS | 15.6% |
| PING | 14.8% |
| REGISTER | n/a |
| TORTURE-TEST | 25.1% |
| All suites run together | 28.9% |

As depicted in the above table, code coverage for each suite is not additive. For instance, all the results include an 11.6% coverage that is generated simply by starting and stopping Asterisk – before any SIP calls have even been processed.

The INVITE method can be used in two ways, either for calling an endpoint that supports auto-answer, or for calling one that does not. The duality of that method appears to be why the Mu-4000 supports mutations to two different call flows that both start with INVITE. As West Coast Labs anticipated, the two call flows exercise different paths within the code, though the coarseness of the code coverage number does not make this immediately apparent or distinct.

In general, the SIP attack suites were designed to each statefully exercise the SIP target implementation, either in a state that is set up for the purpose of the test (such as the INVITE-*.*) suites), or in a situation where the suite uses a message alone – which is sometimes normal (such as OPTIONS or PING methods). However, at times the message is normally transmitted as part of a larger call flow, thus a legitimate interest exists as to whether receiving a packet in an improper state has any effect on the target.

West Coast Labs noted that the Mu-4000 Torture-Test is more extensive than RFC-4475 requires; there is a variant – a subset within the suite – that transmits the test cases exactly as stated in that RFC, and another variant that adjusts the test cases to use the local configuration options that were used for the other SIP tests (for example, the currently configured SIP URIs).

It is important to state that in the test configuration used by West Coast Labs, the Asterisk target was not configured for dynamic registration, thus it was not possible to test the REGISTER method. It is also worth noting that the Torture-Test code coverage data was incomplete as a consequence of the Mu-4000 crashing the target so thoroughly that the analysis failed – Asterisk could not be restarted without manual intervention. Therefore, the actual Torture-Test result may have been marginally higher had the test finished.

All test results are 100% reproducible and were captured with Mu-4000 analyzer settings of 250 ms inter-vector delay and 5000 ms throttling interval. The only faults that were isolated were instrumentation/vector faults – those rated as “!!!!” on the Mu-4000.

A total of 6 faults were reported to the development team for remediation at Digium - the open source Asterisk development hub – as a result of this investigation.

Summary

West Coast Labs determined that not all Asterisk code coverage is attributable to the operation of the Mu-4000. For instance, simply starting and stopping the Asterisk service generated 11.6% code coverage – from internal initialization procedures. Also, Asterisk had certain code for SIP methods that were not supported by the Mu-4000 at the time of writing (for example, SUBSCRIBE and NOTIFY). Therefore, code coverage alone is not a sufficient metric for comparing different negative testing platforms. Code coverage data should be supplemented by other metrics, including the number of unique faults found, as well as the severity of those faults; all three metrics should be used together for a more complete comparison of testing platforms.

West Coast Labs would strongly encourage any interested reader to adopt a similarly comprehensive test methodology to the one outlined in this paper, in order to compare the Mu-4000 results to any other protocol fuzzer in the marketplace. The key is to establish situations wherein meaningful comparisons between test tools give comparable results.

Appendices

Appendix A – VoIP and SIP

VoIP services are increasingly relevant in both service provider provisioning and enterprise deployment. A large number of equipment vendors also build VoIP capabilities into their diverse products. SIP was chosen for these tests simply because it is a complex protocol that is changing rapidly, and it highlights the ability of the Mu-4000 to generate deeply specific, stateful test cases.

The complexity of SIP relative to other protocols and its rapid pace of change increase the opportunities for undiscovered implementation flaws to be present in various SIP implementations. As there are widely available open-source implementations of SIP, West Coast Labs were able to use a popular instance (Asterisk) compiled with code coverage enabled, thereby providing the basis for the reported tests. Obviously, if a non-open-source SIP implementation had been chosen, code coverage analysis would be impossible since there would not be source code available to measure – and it would also be impossible for interested parties to verify the results contained in this document.

SIP was also chosen since it is a very stateful protocol, and it has a lot of messages and options that require numerous special-case and exception-handling routines. Finally, SIP was chosen in order to showcase the abilities of the Mu-4000 to generate deeply stateful test cases that illuminate nooks and crannies within the code of the target implementation. Even a simple implementation of SIP will have such nooks and crannies, and a negative testing platform that hits more of those will achieve measurably broader code coverage.

Appendix B – Asterisk

In order to best compare security analysis tools, one must benchmark such tools centered on how each one affected the target. It is important to select a reasonably complete protocol implementation as the target of any comparison. For instance, if a SIP implementation contained 500 lines of code and only implemented the OPTIONS method, then a test tool that implemented other methods would, in a sense, be wasting its time. A test tool could achieve 100% code coverage with little effort here, but again, that is not very meaningful unless the target is sufficiently complete. In the purported simple SIP implementation, any unimplemented method would likely be as good as any other at exercising the presumably limited-scope exception-handling routines in any such implementation.

By contrast, the SIP implementation within the Asterisk version used, contains 19,010 lines of code (the entire SIP implementation is located within one file named *chan_sip.c*) and supports a large number of SIP methods and configuration options. In addition, the popularity of the chosen target indicates that numerous different classes of users find it to be “good enough” (or “more than good enough”) for their purposes. An unpopular target might be unpopular because it is incomplete, but a market-leading implementation like Asterisk can only achieve that position if it has enough features to make it useful to a large number of different classes of user. The fact that it is free is only part of the reason for its popularity...if it didn’t work, no one would use it.

The final criterion that led to the selection of Asterisk as the target was the fact that it was open source, and thus West Coast Labs had access to the source code – which was necessary in order to compile it for code coverage (*gcov*). By using standard C compiler build directives, one can instrument any application binary to generate code coverage data that pertains to the most recent run(s) of the application.

Appendix C – Code Coverage Limitations

In the reliability, available or security testing and analysis fields, every vendor has different methods of generating test cases and it is difficult to compare these test platforms directly. All such platforms use proprietary techniques to generate test cases that are designed to uncover implementation flaws in various protocol implementations, such as SIP. A well-known aphorism in the software development and QA world is that “if it hasn’t been tested, it is broken.” Code coverage, though one accepted metric, is the bare minimum way to judge the effectiveness of a software-testing platform. With the growth of IP and product complexity, if a line of code has not been exercised at least once, no one knows if it has any bugs, therefore the presence of bugs must be assumed.

Even if a tester manages to test the implementation such that every line of code is executed one time, the tester has only established the behaviour of that code for one “state” of the system when that line of code was executed. However, within complex protocols the same code may be

executed in many different states. As a result, a negative testing platform must exercise the code more than once, in all valid states — both expected and unexpected.

If two tools yield similar code coverage, other metrics emerge as tiebreakers. A dynamic stateful fuzzer executes a given line of code a billion times but would get the same credit as a static fuzzer that only touched that line of code once. One way to measure the efficacy of the fuzzer is to also look at the number of unique faults as well as the severity of said faults.

Code coverage is expressed as a percentage: By dividing the number of lines executed into the number of lines that were instrumented for code coverage gives a — potentially very coarse — result. If two tools get 10% coverage, did each tool cover the same 10%? The number has no context, so it is rather meaningless when used in isolation. Even worse, code coverage only cares if a line of code was touched at least once. Thus from a code coverage perspective there is no difference between a test tool that is very stateful and touches certain lines of code thousands or millions of times, than another tool that touched the same lines of code only once — as each tool would get the same score.

Thus, West Coast Labs concluded that code coverage results of any test tool at best conveys only the breadth of its capacity for finding vulnerabilities, not the true depth of its coverage. A negative testing or protocol fuzzing engine could get excellent code coverage and yet be very shallow. Hence, additional metrics are required to compare test tools, and get a more complete picture on the effectiveness of each one.

Appendix D – the gcov/lcov tool

A program must be compiled with various `gcc` (the GNU C Compiler) command-line optional directives to cause the compiler to instrument the executable it produces for code coverage measurements. Once the program has been successfully compiled, it will write code coverage data alongside the source code files in the source tree. The `gcov` code coverage data files in the source code directory have a “.`gcda`” file name extension. The way `gcov` works is that any code coverage data (the `.gcda` files) that is (are) present when an instrumented program is run is accumulated with data from the previous run(s), rather than starting over from zero. The accumulation feature is actually extremely convenient.

The important caveat for code coverage analysis using `gcov` is that the code coverage data is only written to disk when the program exits normally. This fact ensures that if a test case exists which crashes the target, no code coverage data for that test case will be produced (or possibly, for any previous test cases since the last time the program exited normally). In order to use `gcov`-style code coverage to evaluate any negative testing tool, a user will need to be mindful of this limitation in `gcov` or risk losing substantial portions of code coverage data.

The *lcov* command is used to process the *.gcda* files and other associated code coverage data after the program has exited normally.

Appendix E – Fault Analysis

Two other ways to compare negative testing tools is by the number of faults that they find, and by ranking the severity of those faults. The fault count in a pure black box test equates loosely to the number of crashes that the test tool caused. Each fault must be examined to see if it is derivative in nature, and to see if the location of the crash is unique. After filtering the results, the remaining number of unique faults is a good “fingerprint” for the test tool when it is aimed at a particular target.

Within the list of filtered faults a ranking of the severity of each one can also be listed – as with the Mu-4000. At the coarsest level, faults may be categorized as to whether or not they do permanent damage to the target. For example, does the fault prevent the target software from being re-initialized, possibly by causing the target application to corrupt its configuration file? Other indications of severity include faults that open access to the stack (such as a buffer overflow) over which arbitrary code could be loaded and executed, or that deeply affect and/or corrupt the target system – for example, a test case could cause a stack overflow resulting from unpacking a recursive data structure, potentially leading to a crash of the entire machine.

One might assume that it would be necessary to carefully document and eliminate the test cases that cause crashes before trying to collect code coverage data (since a crash means that no coverage data is collected), however the automatic fault isolation of the Mu-4000 makes this step unnecessary. West Coast Labs configured the Mu-4000 to perform a “*service asterisk restart*” operation on the target every time the Mu-4000 ran its valid test case instrumentation. Any time it saw a crash it would lose – at most – code coverage data for the most recent vector range (a maximum of 16 test cases).

The clever part happens when the Mu-4000 does a range re-run, which restarts Asterisk, and since the code coverage data was written to disk at the beginning of the most recent successfully handled vector range (via the *service asterisk restart* command). The Mu-4000 will cause *gcov* to accumulate code coverage data again once it gets to the single-stepping phase of fault isolation; up to the last test case before the one that causes the crash (this action presumes that the vector range rerun sees another crash). In summary, the fault isolation logic within the Mu-4000 will get complete *gcov* code coverage data for all the properly handled test cases and will automatically exclude just those test cases that cause crashes.

The Mu-4000 runs instrumentation after every test case in single-stepping mode, and as a consequence of West Coast Labs configuring the analysis to restart Asterisk every time instrumentation is run. Thus, when the Mu-

4000 finally encounters the specific test case that caused a crash, it will have automatically eliminated just the contribution of this single test case to the code coverage data, without corrupting the code coverage data itself.

After the analysis is complete, *gcov* will have accumulated all the necessary code coverage data and will have automatically eliminated the failing test cases without needing to do anything exceptional, such as doing a potentially time consuming “pre-test” in order to eliminate test cases that cause faults before running the actual test to collect code coverage data.

It is worth noting that in the event of the fault in the vector range rerun not being repeatable (meaning that the Mu-4000 never enters single-stepping mode), code coverage data for only those 16 test cases will be lost, out of potentially hundreds of thousands, or even millions of test cases.

West Coast Labs Disclaimer

While West Cost Labs is dedicated to ensuring the highest standard of security product testing in the industry, it is not always possible within the scope of any given test to completely and exhaustively validate every variation of the security capabilities and / or functionality of any particular product tested and / or guarantee that any particular product tested is fit for any given purpose.

Therefore, the test results published within any given report should not be taken and accepted in isolation. Potential customers interested in deploying any particular product tested by West Coast Labs are recommended to seek further confirmation that the said product will meet their individual requirement, technical infrastructure and specific security considerations.

All test results represent a snapshot of security capability at one point in time and are not a guarantee of future product effectiveness and security capability. West Coast Labs provide test results for any particular product tested, most relevant at the time of testing and within the specifies scope of testing and relative to the specific test hardware, equipment, infrastructure, configurations and tolls used during the specific test process.

West Coast Labs is unable to directly endorse or certify the overall worthiness and reliability of any particular product tested for any given situation or deployment.

Revision History

| Issue | Description of Changes | Date Issued |
|-------|--|-------------------------------|
| 1.0 | Mu-4000 Security Analyser Negative Testing Study | 10 th January 2008 |
| | | |
| | | |
| | | |



US SALES

T +1 (717) 243 5575

EUROPE SALES

T +44 (0) 2920 548 400

GLOBAL HEADQUARTERS

West Coast Labs,
Unit 9 Oak Tree Court,
Mulberry Drive,
Cardiff Gate Business Park
Cardiff CF23 8RS, UK

T +44 (0) 2920 548 400

F +44 (0) 2920 548 401

E info@westcoast.com

W www.westcoastlabs.com