2005

# A formal process for the testing of servers

Scott Hancock

# A Formal Process for the Testing of Servers

**By**

**Scott Hancock**

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Information Technology

**Rochester Institute of Technology**

**B. Thomas Golisano College**
**of**
**Computing and Information Sciences**

May 20, 2005

# Rochester Institute of Technology

## B. Thomas Golisano College of Computing and Information Sciences

## Master of Science in Information Technology

## Thesis Approval Form

Student Name: Scott Hancock

Thesis Title: A Formal Process for the Testing of Servers

Thesis Committee

| Name | Signature | Date |
|---|---|---|
| Dr. Yin Pan<br>Chair | | 5/20/05 |
| Dr. Swaminathan Natarajan<br>Committee Member | | 5/20/05 |
| Dr. Charles Border<br>Committee Member | | 5/20/05 |

# Thesis Reproduction Permission Form

## Rochester Institute of Technology

## B. Thomas Golisano College of Computing and Information Sciences

## Master of Science in Information Technology

# A Formal Process for the Testing of Servers



Date: 5/31/05 Signature of Author: Scott Hancock

# Acknowledgements

I would like to thank my wonderful wife, Dr. Stephanie Ludi, who encouraged me to continue my education and gave me the moral support I needed. I love you.

I would also like to give special thanks to Dr. Swaminathan Natarajan for being on my thesis committee, helping me to improve this paper over many versions and for helping me get through the thesis process.

Finally, I would like to thank the rest of my thesis committee Dr. Yin Pan and Dr. Charles Border.

# Table of Contents

# 1 Introduction

Making sure that servers are constantly up and running correctly is a core responsibility for system administrators. This requires that they avoid downtime by detecting early warning signals of potential problems and addressing them promptly. Server configuration changes are also notorious trouble spots, since there is always a possibility that the changes might break something that was working earlier, or that newly introduced software might interfere with the operation of the server. These requirements point to the need for constantly checking that all aspects of the system are operating correctly. This research presents a framework and methodology for the systematic testing of servers based on software engineering principles. Using software testing as a model, the framework describes the steps for the baselining and testing of servers.

This paper has its origins in a number of observations arising from the author's experience in system administration, and circumstantial evidence from the news media and security researchers (Kaner, Bach, & Pettichord, 2002; Naraine, 2003; Rescorla, 2002; Roberts, 2003). First, system administrators are not applying security patches to their servers in a timely manner and in some cases not at all (Rescorla, 2002). This leaves these servers open to hackers and worms that exploit previously known operating system and software vulnerabilities, even those that have had fixes available for as much as six months (Roberts, 2003).

Second, one of the reasons that system administrators do not apply security patches in a timely manner is that patches must be tested before they are used in production servers to make sure that they will not cause a malfunction in the server

(Naraine, 2003). System administrators see this testing as time consuming and not foolproof. Even if they do test, there is a possibility that the patch will break something that they did not test (Steve Beattie, 2002). Some administrators wait for a full release or service pack so they can test many patches all at once (TechRepublic, 2002). Unfortunately, while patches come out as needed, full versions or service packs may come out only once or twice a year.

Third, system administrators are not monitoring their servers closely enough. As a result, they are unable to detect changes in their servers. This has allowed hackers to break into systems and use them for their own enterprises undetected (Delio, 2001; Dizard III, 2003). It may be understandable for a system administrator to not detect the presence of hacker activity on their server when it is covert. However, when a server is used to distribute gigabytes of illegally copied movies, music and software (Kucher, 2004) it is hard to understand how a system administrator could not detect the change in storage and network use.

Currently, there is no generally accepted systematic methodology for the testing of servers. A review of course syllabi for System Administration courses (Couch, 2004), (Oslo University College, 2004) shows that most do not cover testing at all. A similar review of System Administration texts again (Limoncelli & Hogan, 2002), (Burgess, 2004) finds few if any references to testing. The few that do mention testing merely say that it should be done and refer the reader to Software Engineering texts. Thus, system administrators have no specific guidance on how to build test plans for their servers.

The aim of this research is to remedy this gap and provide guidance to system administrators on how to plan their server testing, design and set up automated periodic tests and analyze the results.

# 2 Objective

The objective of this project is to design a systematic methodology for system administrators to regularly test that their systems are operating correctly and to detect potential problems as early as possible. This objective directly leads to the evaluation criteria for the effectiveness of the framework:

- Ability to detect problems arising from software installations, upgrades
- Ability to detect problems arising during normal operation
- Ability to detect unauthorized activities and intrusions
- Costs of using the framework, including effort required and cost of acquiring tools

## *2.1 Motivation*

Currently, there is no generally accepted systematic methodology for the testing of servers. Most system administrators test their servers when they are built and anytime patches or upgrades are applied. Ideally, the system administrators have a server set aside for testing patches, new software and software upgrades before installing them to production systems (Howes, 2002). Unfortunately, many organizations cannot afford to dedicate one or more servers to testing. Whether or not they do, the testing process involves manually testing the functionality of the server to see if the software change has caused a problem. For instance, after applying a patch to the operating system of a web server, the system administrator may test that web applications on the server still work. In most cases there is no test plan and no test cases to guide the testing. The system

administrator just superficially tests that the web applications still run. Any problems outside of that, such as performance degradation or incompatibilities with other software on the server will not be found until the server is in production and then the problems will be found by users. With the large number of security patches being regularly released, there is less time for administrators to test them all thoroughly (Kawamoto, 2005).

As an example, in March of 2003, Windows system administrators began reporting that their systems would not reboot. The problem was traced to a security patch to fix a buffer overflow in Microsoft's Internet Information Server. The patch was incompatible with a previous patch that some system administrators had applied. As a result, when the system administrators attempted to reboot the machines, they crashed and could only be fixed by using the recovery console. Many system administrators, in a panic to patch the vulnerability before they were attacked by internet worms such as Code Red and Nimda, did not test the patch and were left with non-functional servers (Evers, 2003). A systematic server testing methodology with execution automation would avert such situations.

# 3 The Model

The framework and methodology developed in this paper draw heavily upon the field of software engineering, particularly software testing. The first part of this section provides background information on concepts in software engineering and software testing. This review is framed in such a way that the ideas are also applicable to system administration. The second part of the section addresses challenges in applying these ideas to system administration, thereby laying the foundation for the framework.

## 3.1 Software Engineering

Software Engineering is being used as the model for the framework because "In general, software engineers adopt a systematic and organized approach to their work as this is often the most effective way to produce high-quality software" (Sommerville, 2001). In order for system administration to go from a folk-art to an engineering-type profession, practitioners must adopt such systematic and organized approaches in their work.

Software engineering is defined as, "an engineering discipline which is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use" (Sommerville, 2001). "Computer software is the product that software engineers design and build. It encompasses programs that execute within a computer of any size and architecture, documents that encompass hard-copy and virtual forms, and data that combine numbers and text but also includes representations of pictorial, video, and audio information" (Pressman, 2001). Correspondingly, in system administration, the services provided by the system are the product, and it encompasses the hardware and the software used to provide those services.

Software engineering process is defined as, "...the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of *key process areas* (KPAs)(Paulk, 1993) that must be established for effective delivery of software engineering technology" (Pressman, 2001). The process defines a set of activities that are common to all software projects. The framework activities are made up of a collection of task sets comprised of

software engineering work tasks, project milestones, work products and quality assurance activities (Pressman, 2001). Testing is the cornerstone quality assurance activity.

## 3.2 Software Testing

Software testing involves creating an overall test plan and designing test cases that will exercise a program in ways that are most likely to uncover errors that need to be fixed (Pressman, 2001). Any engineered product (and most other things) can be tested in one of two ways: (1) Knowing the specific function that a product has been designed to perform, tests can be conducted to demonstrate that each function is fully operational while at the same time searching for errors in each function; (2) knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised. The first test approach is called black-box testing and the second, white-box testing" (Pressman, 2001).

White-box testing, also called glass-box testing, of software requires a knowledge of the inner workings of the source code of the software and "is predicated on close examination of the procedural detail. Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops. The 'status of the program' may be examined at various points to determine if the expected or asserted status corresponds to the actual status" (Pressman, 2001).

Black-box testing, also called behavioral testing, involves sending input to a program and verifying that the output matches what is expected (Sommerville, 2001). It "focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all

functional requirements for a program. ... Tests are designed to answer the following questions:

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

" (Pressman, 2001).

Black-box testing requires the tester to know the requirements of the software: "the requirements for a system are the descriptions of the services provided by the system and its operational constraints. These requirements reflect the needs of customers for a system that helps solve some problem such as controlling a device, placing an order or finding information" (Sommerville, 2001). Software requirements can be classified as functional or non-functional. Functional requirements "are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations" (Sommerville, 2001). Non-functional requirements "are requirements that are not directly concerned with the specific functions delivered by the system. They may relate to emergent system properties such as reliability, response time and store occupancy. Alternatively, they may define constraints on the system such as the capabilities of I/O devices and the data representations used in system interfaces" (Sommerville, 2001).

There are several methods for gathering the requirements for a system. Interviewing the customer is one method. The developer discusses what the potential users want the system to do. Another method is to analyze existing systems. If the new

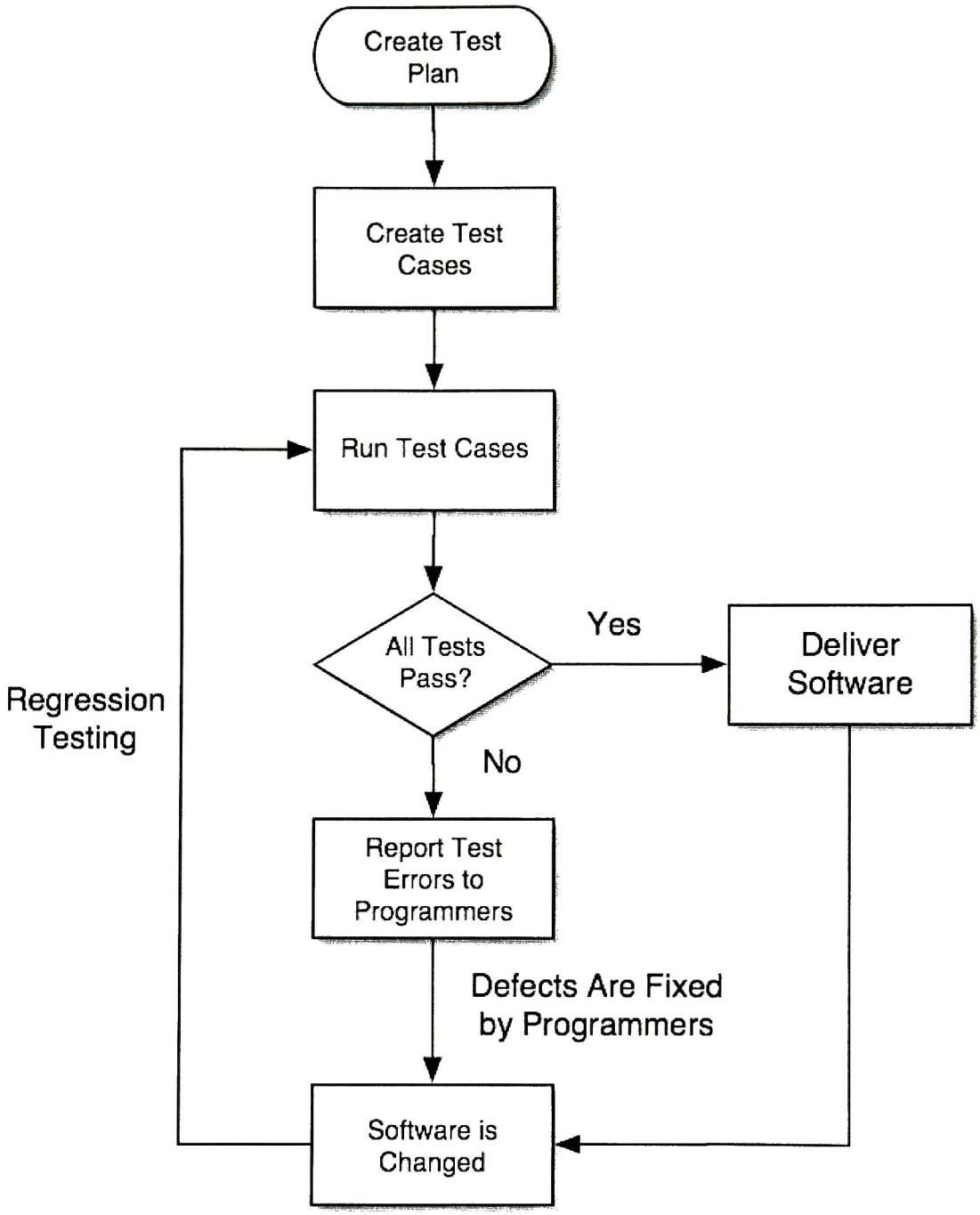
system is replacing an old one, the older system can be observed to see how users utilize it. One other method is task analysis, where the developer observes how the customer performs a task that will be replaced or enhanced by the software (Sommerville, 2001).

The testing process begins with a test plan (Black, 1999) which lays out the scope of the testing, how the testing will be carried out and how problems will be reported (Culbertson, Brown, & Cobb, 2002). Then the test cases are developed (Tamres, 2002). Each test case describes a test that will exercise one aspect of the program. The test case lists the steps to setup the test, run the test, and return the test system to its original state (Black, 1999). Problems discovered during testing are reported to the software developers who make the needed fixes to the software. After the software is fixed, it is then sent back for further testing.

Any time a change is made in the software, including fixing a defect found during testing, the software must be regression tested. "The purpose of Regression Testing is to ensure that the Application Under Test (AUT) still functions correctly following modification or extension of the system (such as user enhancements or upgrades). Typically, the modifications or extensions will be tested to ensure that they meet their requirements, after which a series of tests will be run to confirm that the existing functionality still meets its requirements" (Watkins, 2001).

The testing process continues until the software is ready for release and starts over whenever a change is made to the software after its release. The diagram below is an overview of the Software Testing Process:

**Figure 1: The Software Testing Process**




Planning and documenting the testing process makes testing more thorough and repeatable. To find the most errors in software "tests must be conducted systematically

and test cases must be designed using disciplined techniques"(Pressman, 2001). One method to use in deciding what software needs to be tested is risk analysis. "Most recently, the notion of defining testing in terms of risk has become increasingly popular. In this use, the term 'risk' relates to the possibility that the Application Under Test (AUT) will fail to be reliable or robust and may cause commercially damaging problems for the users"(Watkins, 2001).

## 3.3 Applying the Concepts to System Administration

When we try to apply these concepts to system administration, two challenges are immediately obvious:

- What requirements do we test against? The purpose of software testing is to ensure that the software meets the requirements that have been defined. In system administration, there is no requirements document against which to test.
- What does white-box testing mean in the context of system administration? In software engineering, programmers have access to the source code of the software and can then identify flaws in it's internal operation, but system administrators may not even have access to the code for the applications they run.

One of the challenges in software testing is identifying behavioral requirements to test against. The requirements document is generally written by the software designer who interviews the software's user to see what they want the software to do and how it should perform. The functional requirements describe what the software is supposed to do. An example for a word processor might be that it can spell check, print documents and automatically save the document. Non-functional requirements describe how the

software should perform. For example, a spreadsheet program may need to be able to do one hundred calculations per second or it may need to run on a computer with 128 MB of RAM and a 500 MHz processor. If the requirements document for the software is written well then it will have a complete list of functional and non-functional requirements and tests can be written to verify that the software does what the requirements document says that it should. However, if there are few or no documented requirements for the software then the tester must go to the user of the software to find out what the requirements are for the software. In addition, how users use the software may change over time, so the test plan must evolve to keep up with these changes. For instance, after a year the users may find that they need the spreadsheet to do at least two hundred calculations per second. Thus, the tester must keep in touch with the users in order to know if tests need to be changed or new ones added to reflect how the software is being used.

For system administration, the requirements are the system behaviors that the users expect. These requirements include both the services that the server should provide and performance requirements of the users. To be able to perform testing, system administrators must identify and document these user requirements. They must go through a requirements gathering process similar to that performed by software engineers above. These requirements must be kept updated as user needs change. They must cover the specific functionality and services to be delivered:

- The applications that must be available and operational.
- The processing, storage, communication and output services to be provided.
- The management policies to be supported.
- Services provided to external agencies e.g. mail routing and forwarding, nameserver services etc.

The requirements document must also cover the non-functional requirements i.e. user expectations of system behavior:

- The level of availability of the system and its services.
- Performance expectations.
- Security expectations: threat identification, threat management, data integrity and data confidentiality.

In system administration, black-box tests are performed to ensure that the system services meet these requirements. This includes testing that the applications and server are working correctly and delivering the expected behavior. This paper describes how tests can be set up to regularly check that the system is meeting all these requirements, and to detect and report problems as soon as they occur. These are the types of problems that users will find if they are not first detected by system administrators.

White-box testing tests the internal behavior of the system. The key to applying white-box software engineering principles to system administration is to realize that system administrators deal with the system, consisting of closely integrated software and hardware, as a whole, rather than with the internals of software. While software engineers look at the software at a high level (black-box testing) and a low level (white-box testing), system administrators are only interested in the software at the high level, to ensure that it is behaving correctly. They look at the low level of the system, which is the internal behavior of the server, specifically its performance and configuration parameters. These are the parts of the system that the user never sees but will impact them indirectly if there are any problems. Based on their knowledge of the system internals, system administrators can define a set of internal parameters that need to be tested e.g. level of CPU usage, storage usage levels, desired network throughput etc. They can define tests that monitor whether these parameters are within acceptable levels. This has exactly the

same flavor as white-box tests for software, in that if this internal behavior is not correct, the product as a whole will not behave correctly.

This in turn brings up the question of how system administrators can define what constitute "acceptable levels" for each parameter e.g. storage usage or CPU usage. These levels depend heavily on the specific hardware as well as the usage patterns of the users, and these change over time, hence it is hard to pre-define a specific level as being "acceptable".

The solution that this work presents is the concept of "server baselines". System usage typically follows established patterns, and it is changes in these patterns that indicate problems. As long as the system continues to behave as it has in the past, the administrator has reasonable confidence that it is functioning normally. When there are sudden drastic and unexpected changes in behavior, this might indicate that something has gone wrong. Hence system administrators can establish server baselines by observing the behavior patterns of internal system parameters, and then constantly test that the system is continuing to match these baseline behaviors. Whenever a significant change is detected, they can investigate to determine whether this change is due to a problem or merely due to a temporary or permanent change in the usage pattern. The system must be re-baselined periodically to track the evolution of usage patterns.

Regression testing after making changes is also applicable to system administration. System administrators must test that applications continue to work as expected after making changes, such as installing new applications or new hardware, applying patches or making changes to the system configuration. They must watch for bugs in the new software as well as incompatibilities with other software that is already

running on the server. They must also watch for unexpected changes in system parameters that might indicate a problem with the change. If there are changes in the system parameters but these are found to be reasonable considering the modification that has been made, then they must re-baseline the system.

These ideas enable us to define a mapping of the software testing concepts to system administration. Other principles of software testing, such as test processes, planning, test automation and configuration management can also be applied to server testing. The next section describes a server testing framework that incorporates these concepts.

# 4 The Framework

This section describes the server testing framework and methodology. The framework utilizes functional, baseline and regression tests. It is general and can be applied to any server regardless of hardware, software or operating system differences. The core concepts of the framework, derived from the discussion in the previous section, are as follows:

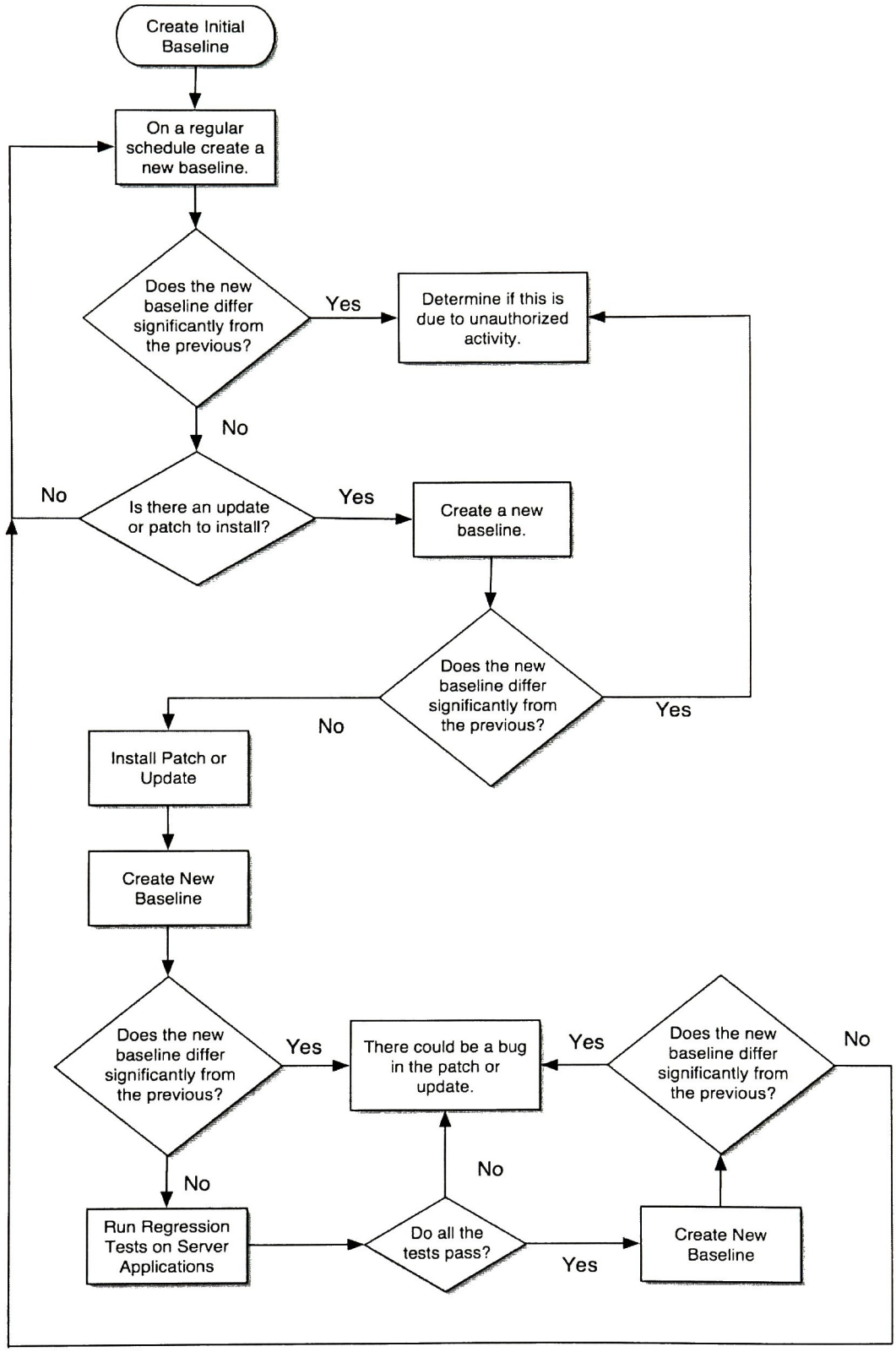
- Correct operation of servers is viewed as consisting of two parts: a set of functional services to be provided to users, and a set of non-functional characteristics to be satisfied.
- Server testing consists of a set of black-box tests that check that the systems are operating correctly, and a set of white-box tests that track various operational and configuration parameters of the system. These tests are run periodically to detect problems in system operation. If problems are found, they are fixed and the tests are run again.

- Sudden significant changes in system operational parameters are indicative of possible problems. They may be caused either by normal system usage or by unauthorized activities or system problems. When such changes are observed, it is a trigger for further analysis by the system administrator to determine the actual cause.
- Tracking of changes to server parameters is done through the use of baselines and comparison against baselines. Baselines are created periodically based on an observation of current system parameter values. The baseline may either be a single value, or a set of values that account for variations in pattern of system usage over a day / week or other patterns. Subsequently, each time a test is run, the results are compared against the baseline to check for significant deviations. Each new baseline is also compared with previous baselines to detect significant changes
- Whenever a significant change is made to the system, such as installing new applications, applying patches or changing the system configuration, the system parameters should be re-baselined. The parameters should also be compared before and after the change to detect possible problems with the change (conflicts between components, buggy patches). If no problems are detected, then the regression tests should be run.

Figure 2 shows the process associated with this framework.

The rest of this section discusses the methodology for performing the framework activities, including gathering the requirements, designing the test cases, executing the tests and analyzing the results.

**Figure 2 Server Testing Framework**



## 4.1 Test Design

In order to begin creating tests for the server, the system administrator must first make an inventory of the hardware and software on the server. This includes but is not limited to:

- Operating System
- Software Applications Installed
- Processor Type and Speed
- Amount of RAM installed
- Hard Disk Space
- RAID Configuration
- Network Connections
- Other Hardware Installed

### 4.1.1 Gathering Requirements

Second, the system administrator must determine the requirements for the server. Like software requirements, the server requirements are determined by the needs of the users. The requirements will include the services that the server should provide as well as performance and other non-functional user needs. As with software engineering, the system administrator can gather these requirements by:

- Customer Interviews
- Task Analysis
- Observing Existing Systems

For example, the system administrator may sit down with the users of the system and discuss their needs. They will identify what services the users need the server to provide. These may be things such as printing, a web store or a database. Next, they will determine performance needs by asking the users questions such as how many users will use the database at the same time and how quickly they need the database to respond to queries. By watching the users do their work, the system administrator may also discover other requirements for the system. Finally, if the new system is replacing an old one, then

the old system can be observed to see what services it provides and its performance requirements. Using these methods, the system administrator can determine the functional and non-functional requirements of the servers. Examples of functional requirements for the server include:

- The user can make a purchase with the shopping cart program.
- The user can print a document on a shared printer.
- The user view their past purchases in a web browser.
- The user can create and use tables in a database.
- Remote users can upload and download files.

Examples of non-functional requirements for the server are:

- The shopping cart program can handle at least 10 simultaneous users.
- The print spooler can handle at least 500 simultaneous print jobs.
- The web application can retrieve the user's past purchases in less than 30 seconds.
- The database server will handle at least 10 simultaneous users.
- The database server will respond to queries in no more than 5 seconds.
- Users will be able to send and receive Email with no more than a 2-minute delay due to this server.
- The server must have 90% availability.

The users' requirements will always be changing so it is important for the system administrator to keep in touch with them and to modify the tests as necessary. For example, an online shopping cart program may need to support ten simultaneous shoppers and the tests developed by the system administrator must verify that the system can handle that at a minimum. In six months, the business may have expanded, and the shopping cart program may need to support at least twenty simultaneous shoppers. The test must be modified to reflect this shift in the requirements. Once the system administrator has compiled an inventory of system components and the requirements for the server, then they can begin creating the baseline and regression tests.

### 4.1.2 Defining Baselines

Baseline tests, which are equivalent to white-box testing in software engineering, collect information about the state of the server. The tests should gather all pertinent information about the state of the server. The set of parameters that will be monitored must include those that are likely to be affected by unauthorized activity and problems arising on the server. Parameters that are likely to vary widely and unpredictably during the normal operation of the server must be avoided. While the parameters may vary widely based on the type and configuration of the server, the following list may be a useful starting point for choosing relevant parameters:

- Memory Usage
- Memory Paging
- Hard Disk Usage
- Hard Disk Reads in per Second
- Hard Disk Writes out per Second
- Open Network Ports
- Listening Network Ports
- Network Packets in per Second
- Network Packets Out per Second
- Total Network Packets In
- Total Network Packets Out
- Network Utilization
- Total Processor utilization
- Running Processes
- Process CPU Utilization
- Process Real Memory Utilization
- Process Virtual Memory Utilization
- Date and Time Process was Started
- Process ID
- Process Owner
- Number of Threads Used by Each Process

These properties will vary depending on the day and the hour so several baselines should be made of the server. For instance, the file server for a small business may see peak utilization from 9 am to 5 pm Monday through Friday and very little utilization from 6 pm to 7 am during the week and all day on the weekends. In this case, baselines

would be made during office hours, after hours and on the weekend for a total of three different baselines. It is up to the system administrator to choose the most appropriate times and days for which to make baselines. These tests will be repeated on a regular basis and the results will be compared with the previous baseline test results. If the results of the test differ significantly from the previous baseline, then this is an indication that there may be something amiss with the server. It may be a hardware or software problem, an indication of unauthorized activity or it may simply be a result of normal usage. The system administrator must determine what caused the change and if it is a problem, then fix it. If the change is not caused by a problem but by a rare or one time event then another baseline may need to be created. Otherwise, the baseline tests may need to be modified to avoid repeated false alarms.

The first baseline should be created when testing is first instituted. Ideally, this should be done on a trusted system such as one newly built from trusted media. The system should be re-baselined on a regular basis (such as once a week) and before and immediately after any significant change is made in the server. Significant changes include:

- Installation of New Software
- Installation of Patches
- Upgrading of Existing Software
- Installation of New Hardware
- Replacement of Hardware

This is done because any change to the system will affect the properties of the server, thus a new baseline must be created to reflect this. The new software may, for instance, increase the amount of memory or processor time used. The new baseline is then

compared against the previous. This will allow the system administrator to see if the new software is using too many resources and may be causing a problem.

### 4.1.3 Regression Tests

Server regression tests test the behavior of the server, and are equivalent to black-box testing in software engineering. They are run after any software change is made in the server. Changes include the installation of software upgrades, new software and patches. These tests could cover items like printing, database server, web server, web based applications, etc. They check to make sure that the new software works correctly and does not interfere with the existing software. The challenge is to determine what it means for the software to be working properly. This can be found in the requirements compiled by the system administrator. Using the functional and non-functional requirements of the system, tests can be developed that make sure that the software is both working correctly and meeting the performance needs of the users.

As new software is added to the server and the requirements change based on the users' needs, new tests will need to be added and old tests will need to be modified to reflect these changes.

### 4.1.4 Prioritizing the Test Cases

Operating systems come with a great deal of software. Exhaustively testing every piece of software and the operating system on a server every time new software is installed is not practical. The system administrator can use two methods to prioritize testing and to choose what to test.

Using the risk-based approach the system administrator must decide what software is essential to the server and to the business. If the failure of a piece of software

would have significant business impact and/or cause significant interruption to or interference with the work of employees, then it is essential. "Using a risk-based approach, the tester is involved in the analysis of the software to identify areas of high risk that need to be tested thoroughly to ensure the threat is not realized during operation of the system by the user" (Watkins, 2001). Software whose failure would have a high impact must be covered during every regression test to make certain that it is functioning properly, even if the patch or software installed has no apparent connection to this software.

The second method is for the system administrator to carefully analyze what a patch is designed to affect. Information about the patch may come from documentation, security alerts or by examining the files in the patch itself. While the patch may not adversely affect the software it is designed to patch, it may break another piece of software that is related to the first. For example, a patch for a web server service may not break the web server service, but it may break the ftp service that was part of the same package. Software that might have been affected by the patch should be tested after the patch is installed.

### 4.1.5 Automating Test Execution

Once it has been decided what to test, it must be determined how to perform the tests. Manually testing everything is one option, but that would take a great deal of time. It is a much better idea to automate as much of the testing as possible. For baseline testing and some regression testing, software such as Nagios (Galstad, 2004) and Big Brother (Quest, 2004) can automate the process of gathering performance data. They were developed to continuously monitor servers and network devices to alert system

administrators when predefined thresholds are reached or when certain events occur. They are capable of monitoring multiple devices and centralizing the reporting of information. Agents that monitor properties such as memory usage, processor time, and network utilization are installed on servers or network devices such as switches. This information is continuously collected in a central server, where it is compared against appropriate baseline values. If significant differences are observed, it triggers, a message to the system administrator. This may also include triggers on predefined thresholds or events. For example, the software may be set to give an alarm if the agent installed on a server detects that the hard disk usage on the server has reached 80%. The collected information is accessible through a web interface so that current status and historical data can be viewed.

The data collected by monitoring software can be used to create the baselines. The historical data can be analyzed to determine when the most appropriate times for baseline creation should be. These should be times that the system performance is consistent and representative of how the computer is expected to be used at that point. One of the benefits of constant monitoring is that the exact point that the performance of the system changed is recorded. This can be correlated with logs to determine what activities occurred at that time that may have caused the change. Thus automated periodic monitoring greatly facilitates causal analysis, in addition to reducing time lags in detecting problems.

The monitoring software can also be used to monitor the server services constantly and notify the system administrator if there is a problem with any of them. For instance, the monitoring software can be configured to test the web server every five

minutes by accessing a static web page or web application. If the web service does not respond, responds incorrectly or responds with a Page Not Found error, then the system administrator can be notified immediately that there is a problem with the web server.

Another way to automate testing is the create scripts that can be run manually or run by a task scheduling program such as cron. These scripts can use operating system utilities to measure the properties of the server and write them to a log, or even better, a database.

As described previously, the set of tests to be run after a change should include both high impact applications and those likely to have been affected. It is possible to group the tests into subsets based on risk and execution effort. Different subsets of tests can be run at different intervals. For example, the level 1 (critical and fully automated) tests can be run every hour. Level 2 tests that are less important or take more resources to run, but are can still be automated, may be run every day. Level 3 tests that require manual intervention may be run, for example, once a month by the administrator. Level 4 tests that require significant effort and comprehensively test all aspects may only be run occasionally after major changes to the system. Subsets of these level 4 tests could be run as regression tests after changes that are viewed as likely to affect the particular services or applications.

## *4.2 Implementation Issues*

There are several issues that can impact the implementaion of the framework. It may not be possible to automate some tests because they require human interaction. One example would be verifying that a printout has occured. The computer may show that a print job has completed but it cannot verify that the job actually printed and printed

properly. The only way to verify the printing is for someone to go to the printer and see the printout.

Time is also a factor, even after testing has been prioritized and automated, there may not be time to test every possible bit of functionality thoroughly. The system administrator must create tests that are the most likely to detect a problem.

Another issue that may arise is unexplainable, intermittent test failures. Some tests may fail but there is no explanation for it and the failure is not repeatable. In this case, the system administrator watches the logs of such failures for patterns that may eventually provide a clue as to causes. Correlations with another system event that also occurs whenever a failure occurs, with network loading patterns, with spikes in processor load etc may provide indicators of the source of the problem.

Change management is important to the success of server testing. The system administrator must be aware of any software that is legitamately installed on the server by other system administrators or users.

There may be certain applications or services which are particularly difficult to test automatically, either because they are not amenable to automation or because the problems leave no discernible footprint e.g. unauthorized intrusion through a loophole that does not cause large changes in any of the measured parameters. It is quite clear that automated testing will not catch such problems, just as software testing does not find every bug. However, the value of systematic testing is that it results in early detection of many common problems.

## 4.3 Analyzing Test Results

When the tests detect a problem or a change in the behavior of the system, it must be analyzed to determine the cause. Functional failures directly indicate the application that is not functioning, but they do not reveal the cause of the failure. Investigation will be needed to find the cause of the malfunction. System parameter changes such as an increase in network or CPU usage can be more difficult and require more analysis since there are so many factors that can affect them. The starting point can be a checklist of possible causes such as:

- User behavior change
- Software patch causes problem
- Incompatibility with new software
- Misconfiguration or other administration error
- Hardware failure (intermittent or permanent)
- Software failure due to application bug
- Runaway process
- Intrusion

When anomalies or failures are detected by the tests, there are several ways to track down the cause. One is to check the system and application logs on the server to see if they show activity that would explain the anomalies. Most operating systems have logs for security, the system, and applications. These logs record important events and errors by date and time. By cross referencing the time of the test failure with the logs, the system administrator can see if any events or errors were recorded that might explain the failure or it may point them to the cause. A few important events that can be found in logs are:

- User activity
- Network activity
- Software crashes
- Hardware errors
- Software errors

Some logs can be configured to increase the amount of information they record or the verbosity of the messages in the log. Configuring these options can make the logs even more useful in tracking down problems. System administrators, in attempting to track down a problem may increase the verbosity of logging temporarily to gain additional information.

Another is to analyze the system manually using system utilities. Many of the utilities that can be used for testing have verbose modes that give a great deal more information than the default mode and may aid in explaining the anomalies. Thus it is important for system administrators to be familiar with these types of utilities for their operating system so they can react quickly to problems.

A third technique is to look at recent changes to the system such as a configuration change or new software as the possible cause. Changes that are under suspicion of causing a conflict can be rolled back to a previous version or removed to see if it resolves the problem.

The most important aspect of analysis is that it must be performed every time there is a significant change in behavior. It is the continuity of observing behaviors and changes from the current trusted baseline that ensures that we can trust the current state to be reasonably free of problems. If analysis is performed sporadically, then a number of problems will escape notice until the consequences become serious.

## *4.4 Evaluating the Framework Against Its Objectives*

The preivous sections have described the framework, and how it is used: designing a set of server tests, running the tests and analyzing the results. This section now evaluates the framework against its original objectives.

The framework is being evaluated based on the following criteria:

- Ability to detect problems arising from software installations, upgrades
- Ability to detect problems arising during normal operation
- Ability to detect unauthorized activities and intrusions
- Cost of using the framework, both effort required and cost of acquiring tools

The first three criteria are the ability to detect different types of problems on the server. The framework is not able to detect all types of problems that may occur. The table below shows some of the types of software bugs, server problems and unauthorized activity that the framework will and will not detect.

Table 1: Problems and Unauthorized Activity Detection Matrix

| | Detect | Won't Detect |
|---|---|---|
| Bad Password Attempts | X | |
| Buffer Overflow | X | |
| Denial of Service Vulnerability | | X |
| Gradual change in Disk Space Usage | X | |
| Gradual Change in Memory Usage | X | |
| Gradual Change in Processor Usage | X | |
| Logic Error | | X |
| Memory Leak | X | |
| Port Scan | X | |
| Printer Malfunction | | X |
| Runaway Process | X | |
| Server Service Failure | X | |
| Security Vulnerability in Software | | X |
| Spike in Disk Space Usage | X | |
| Spike in Memory Usage | X | |
| Spike in Processor Use | X | |
| Unauthorized Files Loaded | X | |
| Unauthorized Network Connections | X | |
| Unauthorized New User Added | X | |
| Unauthorized Ports Opened | X | |
| Unauthorized Software Installed | X | |

As the table shows, the framework detects many common problems that can occur on a server. Some of the problems that it won't detect such as logic errors or security

vulnerabilities in software can only be found by experienced programmers with access to the source code. In short, the framework can only test what you can write a test for. For instance, right now there is no software that can automatically detect security vulnerabilities in software without access to the source code. If at some point software that can do this becomes available, then the framework would be able to detect security vulnerabilities in software.

The last criterion is the cost of using the framework. The approach we will use to determine the cost effectiveness of the framework is to compare the cost of the framework with the value it provides i.e. the cost of not using the framework. The primary cost of using the framework is the cost in staff hours to create, implement, and monitor the tests and to respond to anomalies or errors detected by them. Creating the tests involves designing the tests, writing and debugging them. The implementation of the tests includes installing the tests on the servers and maintaining the tests by making any necessary changes to them. The amount of time taken in creating and implementing the tests depends on how many are created and how complex they need to be. Most of the anomalies reported by the tests will be innocent and their source will be quickly identified. A few (2-3 a month) will be a real error that may take several hours to investigate.

The second cost is the cost of any software used in testing. Most operating systems already come with most of the necessary software utilities to do the tests and scripting engines to automate the testing. If the system administrator decides to use server monitoring software or needs other software to implement a test, then that cost will depend on whether they choose an open source or commercial product. In that case, the

cost can range from nothing to thousands of dollars. Automating the testing will dramatically reduce the amount of time needed to use the framework. The cost calculation for using the framework then is:

(Hours * Salary) + Cost of Software = Cost of Using the Framework

The costs of not using the framework can be calculated by looking at the immediate, short-term and long-term costs incurred by a company due to a server failure. (Cisco, 2002) (CounterStrike, 2004)

**Immediate:**
Cost to Fix Systems: Staff * (Salary*Time)
Disruption of Business: Staff * (Salary * Down Time)
Loss of Business: (Projected Sales per hour * Down Time)

**Short Term:**
Loss of New Business Opportunity: Avg. New Customers per year * Avg Customer Spending
Loss of New Customers: (Avg New Customers per hour * Down Time )*(Avg. Spent per customer per year).

**Long Term:**
Loss of Stature: Increase in Advertising spending to offset bad publicity.

While the actual numbers will depend on the specifics of the situation, it is clear that that the cost of using the framework is of the order of a few thousands of dollars one-time, plus a significant portion of the system administrator's time (to monitor test results and run manual tests) - perhaps another couple of thousand dollars a month. If the approach detects 2 or 3 problems a few hours earlier each month, and avoids a few hours of business downtime per month, it would easily pay for itself. This does not take into account the possibility of major failures and undetected intrusions, which could cost tens or hundreds of thousands of dollars per incident. It also does not take into account the significant increases in customer satisfaction due to the improvement in quality and

reliability of system performance. It is clear that the small cost of using the framework is greatly outweighed by the potential cost of not using it.

# 5 Validation of the Framework

This section describes the result of an experiment in using the framework. The experiment set up a typical small server configuration, and designed a set of tests for it according to the framework. These tests were implemented with scripts and run periodically, with results being compared against baselines. To determine its effectiveness in finding problems, a series of patches and updates were applied, and the framework was used to determine if these changes had created problems. This was then compared with the known list of issues with those patches, to check whether the framework indeed managed to find the problems. The methodology and results are detailed below.

The first step was to create a Server Baseline Test Plan (see Appendix A). The test plan described the overall purpose, scope, and configuration of the tests to be run. These tests recorded the disk space usage, network ports in use, running processes and memory usage on the server. These attributes were chosen as the most likely to indicate unauthorized activity and problems on this type of server. Together, the results of the tests were the Baseline for the server. The Server Baseline Tests were run on a regular basis as well as before and after any patches or updates were installed on the server.

The second step was to create the regression test plan called "Apache and MySQL Test Plan" (see Appendix A) for the software applications on the server. The test plan described the overall purpose, scope, and configuration of the tests to be run. These tests verified that the Apache Web Server and MySQL database server were functioning

properly. The Apache and MySQL tests were run immediately after any patch or update was installed on the server.

Next, test cases for both of the test plans were developed (see Appendix B, C and D). The test cases described the initial setup for the tests, the steps to run the tests and what the expected output should be from the test.

The server used in the experiment was a Sun Microsystems Blade 100. First, the Solaris 9 operating system was installed on the server. Second, Apache, an open source web server, was installed. Third, MySQL, an open source database server, was installed. Finally, several web pages and databases were installed to enable the testing of the web and database server software.

After the server operating system, web and database server software were installed, the entire system was tested to determine if all of the software was working properly. Once it had been determined that all of the software was functioning correctly, the Server Baseline Tests were run in order to create the initial baseline for the server. The test cases recorded:

- List of running processes
- Total amount of physical RAM being used
- List of all network connections to and from the server
- Total amount of hard disk space being used

Every twenty-four hours, the Server Baseline Tests were run. The results of these tests were compared to the previous baseline. If there were no significant differences between the two, then the new baseline test results became the new baseline for the server. If there were significant differences, then the reasons for them had to be tracked down.

The next step was to begin applying existing software upgrades and patches to the server. Each of the products' home websites were checked for new security patches and updates which were then downloaded. The patches and updates were applied to the server one at a time but before each patch or update was installed, a new baseline was created for the server. Then, after the patch or update was installed, the Apache and MySQL tests were run to verify that the software was unaffected by the updates or patches. If any problems had been found, they would have been fixed before proceeding, either by removing the update or patch or by the installing a fix for the problem from the software's publisher. Finally, another baseline was created since the update might have changed the properties of the server. The new baseline was compared to the previous. The differences in the two baselines were checked to make sure that there were no problems in the server caused by the updates and patches such as memory overuse or unwanted network ports being opened.

After all of the existing patches and upgrades were applied, the products' websites and the Computer Emergency Response Team (CERT) advisories were monitored for new security-related updates.

The results of the tests showed that the tests were indeed effective in finding problems (see Appendix E). The patches and updates applied to the server operating system and MySQL did not cause any problems. The tests did detect the changes in disk space that accompanied the installation of patches. The Apache test did detect a problem with the Apache update. A problem found only on the Solaris version of Apache caused the web server to stop responding to requests after the update was installed. The update was removed. A subsequent update fixed the problem. To validate the

comprehensiveness of problem identification, the support web pages for Solaris, MySQL, and Apache were checked for information about bugs that had been found in the updates and patches after they were released. Other than the Apache bug mentioned previously, no other bugs were found that the test cases should have identified.

Of course, this was not a real-world test of the framework. However, it does also enable us to analyze the operational aspects of using the framework, particularly the effort required for test creation and test execution. The tests each took an average of one and a half minutes to run (See Appendix E). The tests themselves are basic, using simple commands to gather the necessary information. Each of the tests took less than fifteen minutes each to design, write, test and debug. The results returned are also very basic, often just a number, which makes them easy to analyze. The exception to this is the running processes since normally there are more than twenty running processes on a server. The comparison of the results of the tests to the previous results took an average of one and a half minutes. The tests and the comparison of the output against previous tests could be automated to make the process even faster.

It was found that the test Memory Usage is not valuable on the Solaris platform. Solaris uses all available memory for file caching so the amount of available physical memory eventually goes to zero. Normally UNIX system administrators look at memory paging activity to monitor memory usage on their servers. This however, is a time consuming task with little benefit to the goal of the experiment. Thus, the Memory Usage test should be dropped from the test plan when using UNIX. On other platforms, such as Windows, it may be of benefit.

# 6 Putting the Framework Into Practice: An Example Scenario for the Guidance of Practitioners

The previous sections have described the concept of the framework and discussed its implementation in conceptual terms. However, some system administrators may find it easier to grasp the concepts and benefits in terms of actual operational scenarios. With that in mind, this narrative example has been included to show how the framework might be used in a real-world environment.

Bob Johnson is the senior system administrator for a medium sized company called ITP. Bob and his staff are responsible for the company's two dozen servers. These include departmental file servers, human resources and payroll application servers, enterprise email servers and web servers. Bob's department also has several development servers for their programmers to develop, test and maintain enterprise applications and testing servers used to test new software and patches before they are installed on production servers. In addition, the testing servers are used to test their backups and as replacement servers if one of the production servers becomes nonfunctional. The servers run a mixture of Linux and Windows 2000.

Bob leads a staff of three junior system administrators who are on call twenty-four hours a day, seven days a week year-round. Most of the company's enterprise applications need to be available only during working hours (Monday-Friday 8 A.M. - 8 P.M. excluding holidays). Some of the departments require their file server and web applications to be available 24 hours a day seven days a week except holidays so that sales representatives can access them while on the road. They all allow for a four hour

maintenance window once a week during off hours. However, these maintenance windows must be approved ahead of time.

ITP cannot afford for these servers to be down. Any unscheduled downtime costs ITP money, either in lost productivity of the company's employees or in lost revenue if current or potential customers cannot access their website.

The greatest challenge for Bob has been to keep up with all of the critical security patches for their servers. The patches must be tested and installed before a worm or a hacker can compromise the servers. At first, Bob would just put the patches on the servers and then take them off if they caused any problems. However, this caused too much disruption of business so he procured several servers to use for testing patches, updates and new software. Even with the test servers there were still problems. His testing process consisted of installing the software and then just trying various things on the server to see if they all worked. This process was not documented and was never the same. It depended on whatever he could think of at the time and how much time he had. Inevitably, problems were missed and ended up causing problems on production servers. Bob decided to develop test plans for the servers to make testing consistent and thorough.

First Bob made an inventory of all of the servers. This inventory included all of the hardware and software on the machines. It was regularly checked and updated any time software or hardware was added or removed from a server.

Next, he began making a list of requirements for the servers. The typical users were interviewed to determine how they used the services on the servers and what their needs were in terms of server application performance. The servers themselves were observed to see how they were used and how they and their software performed. All this

information was compiled into requirements documents for each server. The documents were regularly updated based on regular feedback from the users on their needs.

Then Bob was able to create a test plan, baseline tests and regression tests based on the information gathered. These were regularly updated to reflect newly added software and changes in requirements.

Bob subscribes to several security mailing lists and the security lists of his software and hardware vendors, so that he is immediately aware of any new security patches. Once a patch is made available, it is downloaded and he begins the process that any new software that will be installed on the servers goes through. First, a baseline is taken of the test server, which is an exact copy of the production server that the patch is to be installed on. The patch is then installed on the test server. The test server is then baselined again. This baseline is compared to the last to determine if the patch is negatively affecting the server's performance. The server is then tested according to the test plan, to make sure that all the applications and system services are still working. Once all of the tests have been passed and he is sufficiently sure that the patch does not interfere with the operation of the servers, it can be installed on the production servers. It is important that the patches be tested as quickly as possible so that they can be installed and the servers will be protected.

Baseline tests are run on the servers on a regular basis. These are the same tests that are used when new software is installed. However, the purpose of these tests are to detect unusual activity on the server which may indicate unauthorized activity or a problem with the server. Each baseline test is compared to the previous test at the same time. If there is a large difference in the results, this indicates that there may be a

problem. Bob determined the schedule based on usage patterns on the servers, which are as follows:

Monday-Friday
7:00 A.M. - 6:00 P.M. - Normal business hours
7:00 P.M. - 11:00 P.M. - Staff working from home. Sales reps downloading data.
1:00 A.M. - 2:00 A.M. - Batch processing of Financial and HR Data
2:00 A.M. - 4:00 A.M. - Backups of all servers.
5:00 A.M. - 7:00 A.M. - Maintenance windows.

Saturday-Sunday- Likely Hacker activity all day.
8:00 A.M. - 7:00 P.M. - Staff working from home occasionally.
2:00 A.M. - 4:00 A.M. - Backup of all servers.

Holidays
No activity.

Bob scheduled the baseline tests so that the results will be a good representation of the state of the servers throughout the day. He avoided times when the activity would be unusually high such as during batch processing and early on the workday, when people are getting to work, logging into the system and checking their email. Conversely, he also avoided times when server activity would be unusually low such as during lunchtime. He did however, chose off-hour times when hacker activity would be likely. The baseline tests are run at the following times.

Monday - Friday
7:00 A.M. - Minimal Activity
2:00 P.M. - Normal working conditions
12:00 A.M. - Minimal activity. Likely time for hacker activity

Saturday - Sunday and Holidays
6:00 A.M. - Should be minimal. Looking for unusual activity.
12:00 P.M. - Should be minimal. Looking for unusual activity.
6:00 P.M. - Should be minimal. Looking for unusual activity.
12:00 A.M. - Should be minimal. Looking for unusual activity.

In creating baseline tests, Bob had to determine what server properties he was interested in monitoring. For his tests, Bob chose the following properties:

- Memory Usage
- Hard Disk Space Used
- Running Services
- Running Programs
- Processor Activity

When Bob and his staff first began running baseline tests on the servers, the tests were run manually. This worked fine but it limited them to running tests only during working hours and Bob wanted to have baselines made more often. The solution was to automate the tests. Bob wrote scripts to run the various tests and record the results. He then used a task-scheduling program to run the tests at certain days and times.

Along with automating the baseline tests, Bob also created scripts to run some of the regression tests every 30 minutes. These tests verified that services such as the web server and database server were still running. If any of them failed, the system administrator on call was notified automatically so they could fix it.

On a Monday morning, when Bob compared the new baseline tests to the old, there was a ten-gigabyte increase in hard disk space usage on the Human Resources file server. By looking over the previous days baselines Bob was able to determine that the increase had happened suddenly over the weekend. He searched the server for files created over the weekend. Quickly he found that the server was being used to store and distribute pirated movies and video games. The account used to save the files was created that weekend and there was now a small FTP server running on the system. It was clear that the server had been hacked. It was also clear that other people were accessing the FTP server over the Internet and downloading the pirated goods. Already this was

causing a 20% increase in network activity and was affecting the performance of the server. Bob gathered as much information as possible, shut down the FTP server process, deleted the hacker's account and deleted the files. While he would have liked to make a forensic image of the machine, his main priority was keeping the file server running so that operations would not be interrupted. That evening, he took that file server down for closer examination and brought up a backup server in its place. After determining how the server was hacked, he made the necessary changes to the server and any others that had the same vulnerability.

In total, it took Bob twelve hours to determine that the server had been hacked and to fully remediate the system. The event did not cause any downtime, affect any users or impact business operations since the intrusion was detected early. Since it was a print server machine, Bob was relieved the server that had been compromised did not contain confidential data and that the hacker's intent seemed to have been just to use the server to distribute the pirated materials.

For Bob, the immediate cost of the intrusion was fairy straightforward to calculate. It was just the cost for him to investigate the incident and to fix the server. The time it took to deal with the intrusion was minimized since he already had an incident response plan in place. He also calculated how quickly the costs could have risen if the intrusion had not been detected early. As more hackers stored and retrieved files from the server, the extra activity would have slowed the network and the disk response time on the server. This would then impact the HR staff, batch process on the server and the time needed to run a backup on the server. If the hackers had decided to be malicious, they

could have easily caused the server to crash. These calculations can be found in Appendix F - Cost of Intrusion.

# 7 Conclusion

This initial work lays the foundation for applying software engineering testing principles to system administration. By providing a systematic process, the framework makes server testing easier, faster and repeatable. System administrators are encouraged to make server testing a regular part of ongoing maintenance so that servers can be made more secure and reliable.

The primary cost of using the framework is the cost in staff hours to create, implement, and monitor the tests and to respond to anomalies or errors detected by them. This is much less than the cost of an intrusion (which can be found in Appendix F).

Formalizing and documenting exactly what is to be tested, how to test it and what the expected results are, ensures that the testing is done uniformly by all system administrators throughout an organization. This means that an organization can be assured that all of their servers are being monitored and that all patches and updates are being thoroughly tested before they are put into production.

The test cases are documentation that the servers are being monitored on a regular basis for unauthorized activity. This documentation could be used in legal proceedings to help show that due care was used in the securing of the companies' systems. This documentation may also be important to companies for compliance with laws and regulations such as Sarbanes-Oxely (Langin, 2004).

Further work should include adding change management and risk management to the framework to make it more complete. Software could be developed to automate and centralize the testing of servers. The data could be automatically analyzed and alerts sent out to the appropriate individuals.

# 8 Appendix A – Test Plans

# Server Baseline Test Plan

**Overview**
The purpose of this plan is not to test for defects in the server operating system, it is to record a set of properties for the server. The properties that will be recorded are:

1. List of processes in RAM
2. Total amount of physical RAM being used
3. List of all network connections to and from the server
4. Total amount of hard disk space being used

The list of properties is called the baseline of the server. These tests will be run after any software is installed since the installation will change one or more of these properties. Then the tests will be run on a schedule. The results will then be checked against the baseline. If there are significant differences between the test results and the current baseline, it may indicate that there is unauthorized activity occurring on the server.

**Scope**
These tests are for recording properties of the server. They will not confirm the functionality of or identify defects in the server.

**Test Configurations**
The tests will be run directly on the server. The programs used in the tests directly on the server will be run from trusted removable media (such as a CD). This is to assure that the programs used have not been replaced in order to hide unauthorized activity.

**Resources**
Server
Model: Sun Blade 100
Operating System: Solaris 9
Software: df, netstat, top, ps

**Change History**
No changes.

**Reference Documents**
None.

## Apache and MySQL Test Plan

### Overview

The purpose of this plan is to test that the Apache and MySQL software are functioning properly.

Scope

These tests are used to confirm that the software continues to function properly after it has been patched or upgraded. The tests will not identify pre-existing defects in the software. The tests will not determine if the updates have fixed the security issues they are designed to.

### Test Configurations

The tests will be run directly on the server and from a remote computer workstation. The server and the workstation will be connected over a local area network (LAN). The programs used in the tests directly on the server will be run from trusted removable media (such as a CD) and from a workstation. This is to assure that the programs used have not been replaced in order to hide unauthorized activity.

### Resources

Server
Model: Sun Blade 100
Operating System: Solaris 9
Software: Apache 1.3, MySQL 4.0

Workstation
Model: Any UNIX compatible
Operating System: Any UNIX compatible
Software: MySQL 4.0, a web browser

### Change History

No changes.

### Reference Documents

None

# 9 Appendix B – Server Baseline Test Cases

## 9.1 Server Baseline – Disk Usage

| Test Case ID | SBDF | Test Date | |
|---|---|---|---|
| Software | Solaris | Version | 9 |
| Test Start Time | | Test End Time | |

| Purpose |
|---|
| Determine the amount of disk space in use on the server. |
| Initial Setup<br>1. Server idle.<br>2. A trusted copy of the program is run from CDROM or other trusted media.<br>3. The program is run from a terminal window or the console. |
| Input<br>1. Type ./df k |
| Expected Results<br>Pass/Fail |

## 9.2 Server Baseline – Memory Usage

| **Test Case ID** | SBMU | **Test Date** | |
|---|---|---|---|
| **Software** | Solaris | **Version** | 9 |
| **Test Start Time** | | **Test End Time** | |

| Purpose |
|---|
| Determine the amount of RAM in use on the server. |
| Initial Setup<br>1. Server idle.<br>2. A trusted copy of the program is run from CDROM or other trusted media.<br>3. The program is run from a terminal window or the console. |
| Input<br>1. Type ./top |
| Expected Results<br>Pass/Fail |

## 9.3 Server Baseline – Network Usage

| **Test Case ID** | SBNU | **Test Date** | |
|---|---|---|---|
| **Software** | Solaris | **Version** | 9 |
| **Test Start Time** | | **Test End Time** | |

| Purpose |
|---|
| Determine the network ports in use on the server. |
| Initial Setup<br>1. Server idle.<br>2. A trusted copy of the program is run from CDROM or other trusted media.<br>3. The program is run from a terminal window or the console. |
| Input<br>1. Type ./netstat a |
| Expected Results<br>Pass/Fail |

## 9.4 Server Baseline – Process Inventory

| **Test Case ID** | PIMU | **Test Date** | |
|---|---|---|---|
| **Software** | Solaris | **Version** | 9 |
| **Test Start Time** | | **Test End Time** | |

| Purpose |
|---|
| Determine the processes running on the server. |
| Initial Setup<br>1. Server idle.<br>2. A trusted copy of the program is run from CDROM or other trusted media.<br>3. The program is run from a terminal window or the console. |
| Input<br>1. Type ps aux |
| Expected Results<br>Pass/Fail |

# 10 Appendix C – Apache Test Cases

## *10.1 Apache – Web Page*

| **Test Case ID** | APWP | **Test Date** | |
|---|---|---|---|
| **Software** | Apache | **Version** | 1.3 |
| **Test Start Time** | | **Test End Time** | |

| Purpose<br><br>Verify that Apache is serving web pages. |
|---|
| Initial Setup<br><br>1. Launch a web browser on a remote computer. |
| Input<br><br>1. Type http://server address/test.html |
| Expected Results<br><br>Web page is returned.<br><br>Pass/Fail |

# 11 Appendix D – MySQL Test Cases

## *11.1 MySQL – Add Table*

| **Test Case ID** | MSQA | **Test Date** | |
|---|---|---|---|
| **Software** | MySQL | **Version** | 4.0 |
| **Test Start Time** | | **Test End Time** | |

| Purpose |
|---|
| Verify that a database can be added and dropped. |
| Initial Setup<br><br>1. Open a terminal window or run from console.<br>2. Switch to mysql directory. |
| Input<br><br>1. Type ./bin/mysql -u root -p<br>2. Type MySQL root password.<br>3. Type create database test;<br>4. Type show databases;<br>5. Type drop database scott; |
| Expected Results<br><br>Pass/Fail |

## 11.2 MySQL – Select Data

| **Test Case ID** | MSQS | **Test Date** | |
|---|---|---|---|
| **Software** | MySQL | **Version** | 4.0 |
| **Test Start Time** | | **Test End Time** | |

| Purpose |
|---|
| Verify that data can be returned from the database. |
| Initial Setup<br>1. Open a terminal window or run from console.<br>2. Switch to mysql directory. |
| Input<br>1. Type ./bin/mysql -u root -p<br>2. Type MySQL root password<br>3. Type use mysql;<br>4. Type select host, name from user; |
| Expected Results<br>Pass/Fail |

## 11.3 MySQL – Remote Select Data

| Test Case ID | MSQR | Test Date | |
|---|---|---|---|
| Software | MySQL | Version | 4.0 |
| Test Start Time | | Test End Time | |

| Purpose |
|---|
| Verify that data can be returned from the database from a remote computer. |
| Initial Setup<br><br>1. On a remote computer, open a terminal window or run from console.<br>2. Switch to mysql directory. |
| Input<br><br>1. Type ./bin/mysql -u root -p<br>2. Type MySQL root password<br>3. Type use mysql;<br>4. Type select host, name from user; |
| Expected Results<br><br>Pass/Fail |

# 12 Appendix E – Test Results

| Test Case ID | Date | Software | Version | Start Time | End Time | Elapsed Time (min) | Results | Pass/Fail | Notes |
|---|---|---|---|---|---|---|---|---|---|
| SBPI1 | 7/3/03 | Solaris | 9 | 4:37 PM | 4:38 PM | 1 | na | Pass | Initial Baseline |
| SBNU1 | 7/3/03 | Solaris | 9 | 4:35 PM | 4:36 PM | 2 | na | Pass | Initial Baseline |
| SBMU1 | 7/3/03 | Solaris | 9 | 4:26 PM | 4:27 PM | 2 | na | Pass | Initial Baseline |
| SBDF1 | 7/3/03 | Solaris | 9 | 4:03 PM | 4:04 PM | 1 | na | Pass | Initial Baseline |
| MSQS1 | 7/3/03 | MySQL | 4 | 8:42 PM | 8:43 PM | 1 | retrieve table | Pass | Initial test |
| MSQR1 | 7/3/03 | MySQL | 4 | 8:39 PM | 8:40 PM | 1 | it responded | Pass | Initial test |
| MSQA1 | 7/3/03 | MySQL | 4 | 4:48 PM | 4:50 PM | 2 | showed table | Pass | Initial test |
| APWP1 | 7/3/03 | Apache | 1.3 | 4:46 PM | 4:48 PM | 2 | showed web page | Pass | Initial test |
| SBPI2 | 7/4/03 | Solaris | 9 | 2:22 PM | 2:24 PM | 1 | processes | Pass | Install Solaris Update Package |
| SBNU2 | 7/4/03 | Solaris | 9 | 2:18 PM | 2:20 PM | 2 | ports | Pass | Install Solaris Update Package |
| SBMU2 | 7/4/03 | Solaris | 9 | 2:15 PM | 2:16 PM | 2 | memory | Pass | Install Solaris Update Package |
| SBDF2 | 7/4/03 | Solaris | 9 | 2:12 PM | 2:13 PM | 1 | disk usage | Pass | Install Solaris Update Package |
| MSQS2 | 7/4/03 | MySQL | 4 | 2:09 PM | 2:10 PM | 1 | retrieve table | Pass | Install Solaris Update Package |
| MSQR2 | 7/4/03 | MySQL | 4 | 2:06 PM | 2:07 PM | 1 | it responded | Pass | Install Solaris Update |

| | | | | | | | | | Package |
|---|---|---|---|---|---|---|---|---|---|
| MSQA2 | 7/4/03 | MySQL | 4 | 2:03 PM | 2:04 PM | 2 | showed table | Pass | Install Solaris Update Package |
| APWP2 | 7/4/03 | Apache | 1.3 | 2:00 PM | 2:01 PM | 2 | showed web page | Pass | Install Solaris Update Package |
| SBPI3 | 7/5/03 | Solaris | 9 | 2:22 PM | 2:24 PM | 1 | processes | Pass | Apache Update 1 |
| SBNU3 | 7/5/03 | Solaris | 9 | 2:18 PM | 2:20 PM | 2 | ports | Pass | Apache Update 1 |
| SBMU3 | 7/5/03 | Solaris | 9 | 2:15 PM | 2:16 PM | 2 | memory | Pass | Apache Update 1 |
| SBDF3 | 7/5/03 | Solaris | 9 | 2:12 PM | 2:13 PM | 1 | disk usage | Pass | Apache Update 1 |
| MSQS3 | 7/5/03 | MySQL | 4 | 2:09 PM | 2:10 PM | 1 | retrieve table | Pass | Apache Update 1 |
| MSQR3 | 7/5/03 | MySQL | 4 | 2:06 PM | 2:07 PM | 1 | it responded | Pass | Apache Update 1 |
| MSQA3 | 7/5/03 | MySQL | 4 | 2:03 PM | 2:04 PM | 2 | showed table | Pass | Apache Update 1 |
| APWP3 | 7/5/03 | Apache | 1.3 | 2:00 PM | 2:01 PM | 2 | No response | Fail | Apache Update 1 |
| SBPI3 | 7/6/03 | Solaris | 9 | 2:22 PM | 2:24 PM | 1 | processes | Pass | MySQL Update 1 |
| SBNU3 | 7/6/03 | Solaris | 9 | 2:18 PM | 2:20 PM | 2 | ports | Pass | MySQL Update 1 |
| SBMU3 | 7/6/03 | Solaris | 9 | 2:15 PM | 2:16 PM | 2 | memory | Pass | MySQL Update 1 |
| SBDF3 | 7/6/03 | Solaris | 9 | 2:12 PM | 2:13 PM | 1 | disk usage | Pass | MySQL Update 1 |
| MSQS3 | 7/6/03 | MySQL | 4 | 2:09 | 2:10 | 1 | retrieve | Pass | MySQL |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | PM | PM | | table | | Update 1 |
| MSQR3 | 7/6/03 | MySQL | 4 | 2:06 PM | 2:07 PM | 1 | it responded | Pass | MySQL Update 1 |
| MSQA3 | 7/6/03 | MySQL | 4 | 2:03 PM | 2:04 PM | 2 | showed table | Pass | MySQL Update 1 |
| APWP3 | 7/6/03 | Apache | 1.3 | 2:00 PM | 2:01 PM | 2 | showed web page | Pass | MySQL Update 1 |
| SBPI4 | 7/10/03 | Solaris | 9 | 3:00 PM | 3:01 PM | 1 | processes | Pass | Baseline Test 1 |
| SBNU4 | 7/10/03 | Solaris | 9 | 3:03 PM | 3:04 PM | 2 | ports | Pass | Baseline Test 1 |
| SBMU4 | 7/10/03 | Solaris | 9 | 3:06 PM | 3:07 PM | 2 | memory | Pass | Baseline Test 1 |
| SBDF4 | 7/10/03 | Solaris | 9 | 3:09 PM | 3:10 PM | 1 | disk usage | Pass | Baseline Test 1 |
| SBPI4 | 7/11/03 | Solaris | 9 | 3:00 PM | 3:01 PM | 1 | processes | Pass | Baseline Test 2 |
| SBNU4 | 7/11/03 | Solaris | 9 | 3:03 PM | 3:04 PM | 2 | ports | Pass | Baseline Test 2 |
| SBMU4 | 7/11/03 | Solaris | 9 | 3:06 PM | 3:07 PM | 2 | memory | Pass | Baseline Test 2 |
| SBDF4 | 7/11/03 | Solaris | 9 | 3:09 PM | 3:10 PM | 1 | disk usage | Pass | Baseline Test 2 |
| | | | | | | | | | |
| SBPI3 | 7/12/03 | Solaris | 9 | 2:22 PM | 2:24 PM | 1 | processes | Pass | Apache Update 2 |
| SBNU3 | 7/12/03 | Solaris | 9 | 2:18 PM | 2:20 PM | 2 | ports | Pass | Apache Update 2 |
| SBMU3 | 7/12/03 | Solaris | 9 | 2:15 PM | 2:16 PM | 2 | memory | Pass | Apache Update 2 |
| SBDF3 | 7/12/03 | Solaris | 9 | 2:12 PM | 2:13 PM | 1 | disk usage | Pass | Apache Update 2 |
| MSQS3 | 7/12/03 | MySQL | 4 | 2:09 PM | 2:10 PM | 1 | retrieve table | Pass | Apache Update 2 |
| MSQR3 | 7/12/03 | MySQL | 4 | 2:06 PM | 2:07 PM | 1 | it responded | Pass | Apache Update 2 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| MSQA3 | 7/12/03 | MySQL | 4 | 2:03 PM | 2:04 PM | 2 | showed table | Pass | Apache Update 2 |
| APWP3 | 7/12/03 | Apache | 1.3 | 2:00 PM | 2:01 PM | 2 | showed web page | Pass | Apache Update 2 |

# 13 Appendix F - Cost of Intrusion

The costs of the intrusion can be calculated by looking at the immediate, short-term and long-term costs incurred by a company due to the intrusion.(Cisco, 2002) (CounterStrike, 2004)

**Immediate:**
Cost to Fix Systems: Staff * (Salary*Time)
Disruption of Business: Staff * (Salary * Down Time)
Loss of Business: (Projected Sales per hour * Down Time)
Cost to Notify Customers of Exposure of Personal Info: Mailing Cost * No. of Customers

**Short Term:**
Loss of New Business Opportunity: Avg. New Customers per year * Avg Customer Spending
Loss of New Customers: (Avg New Customers per hour * Down Time )*(Avg. Spent per customer per year).

**Long Term:**
Loss of Stature: Increase in Advertising spending to offset bad publicity.

# 14 References


Black, R. (1999). *Managing the testing process.* Redmond, WA: Microsoft Press.

Burgess, M. (2004). *Analytical network and system administration : managing human-computer networks.* Chichester, West Sussex, England ; Hoboken, NJ: John Wiley & Sons.

Cisco. (2002). *The Return on Investment for Network Security.* Retrieved 7/2004, 2004

Couch, A. L. (2004). *Comp150NET Network Administration.* Retrieved 5/2005, 2005, from http://www.cs.tufts.edu/comp/150NET/

CounterStrike. (2004). *Computer Network Security - System Shield Cost Justification.* Retrieved 7/2004, 2004, from http://www.counterstrike.com/sscostjs.html

Culbertson, R., Brown, C., & Cobb, G. (2002). *Rapid testing.* Upper Saddle River, NJ: Prentice Hall PTR.

Delio, M. (2001, June 23, 2001). *Hoosier Favorite Hack Victim?* Retrieved July 23, 2003, 2003, from http://www.wired.com/news/culture/0,1284,44501,00.html

Dizard III, W. P. (2003, July, 30, 2003). *Kentucky Shakes Up Systems After Large-scale Hacking.* Retrieved August 18, 2003, 2003, from http://www.gcn.com/vol1_no1/daily-updates/22965-1.html

Evers, J. (2003). *Latest Windows Patch Poses Problems.* Retrieved February 2005, 2005, from http://www.pcworld.com/news/article/0,aid,109877,00.asp

Galstad, E. (2004, December 14, 2004). *Nagios.* Retrieved June 2004, 2004, from www.nagios.org

Howes, T. (2002, October 2002). Winning the Cybersecurity War. *Computer Technology Review, 22,* 2.

Kaner, C., Bach, J., & Pettichord, B. (2002). *Lessons learned in software testing : a context-driven approach.* New York: Wiley.

Kawamoto, D. (2005). *Patching Up Problems.* Retrieved January 2005, 2005, from http://news.com.com/Patching+up+problems/2100-7347_3-5553945.html?tag=nefd.lede

Kucher, K. (2004, March 17, 2004). *SDSU Says Computer Server Was Infiltrated.* Retrieved May 9, 2004, 2004, from

http://www.signonsandiego.com/news/computing/20040317-9999-news_7m17hacker.html

Langin, D. J. (2004, 2004). *Darning SOX: Technology and Corporate Governance Elements of Sarbanes-Oxley*. Retrieved December 2004, 2004, from http://www.tripwire.com/files/literature/white_papers/Tripwire_SOX_WP.pdf

Limoncelli, T., & Hogan, C. (2002). *The practice of system and network administration*. Boston: Addison-Wesley.

Naraine, R. (2003, March 31, 2003). *When Patches Aren't Applied*. Retrieved March 26, 2003, 2003, from http://www.cioupdate.com/reports/article.php/2172051

Oslo University College, O. (2004, Thu, Aug 19, 2004). *Course plan for masters degree in network administration/network and system administration at Oslo University College:*. Retrieved 3/2005, from http://www.iu.hio.no/data/msc-course.html

Paulk, M. e. a. (1993). *Capability Maturity Model for Software*. Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

Pressman, R. S. (2001). *Software engineering : a practitioner's approach* (5th ed.). Boston, Mass. ; London: McGraw-Hill.

Quest, S. (2004). *Big Brother System and Network Monitor*. Retrieved February 2005, 2005, from www.bb4.org

Rescorla, E. (2002, November 15, 2002). *Security Holes.. Who cares?* Retrieved April 26, 2003, 2003, from http://www.rtfm.com/upgrade.html

Roberts, P. (2003, January 28, 2003). *Microsoft Slammed by Its Own Vulnerability*. Retrieved March 26, 2003, 2003, from Software giant says that unpatched machines on its network were hit by the Slammer worm.

Sommerville, I. (2001). *Software engineering* (6th ed.). Harlow, England ; New York: Addison-Wesley.

Steve Beattie, S. A., Crispin Cowan, Perry Wagle, Chris Wright. (2002, November 2002). *Timing the Application of Security Patches for Optimal Uptime*. Paper presented at the LISA '02: Sixteenth System Administration Conference, Philadelphia, PA.

Tamres, L. (2002). *Introducing software testing*. London ; Boston, MA: Addison-Wesley.

TechRepublic. (2002). Most admins patch Windows monthly or quarterly.

Watkins, J. (2001). *Testing IT : An Off-the-Shelf Software Testing Process*. Cambridge ; New York: Cambridge University Press.