VENKATAKRISHNAN, ROOPAK. Redundancy-Based Detection of Security Anomalies in Web-Server Environments. (Under the direction of Dr. Mladen A. Vouk.)

When developing software, the aim should be to make it secure. Failing that, detecting and preventing attacks before they compromise a system is a good alternative. Attacks and breaches can be detected using acceptance testing, redundancy-based mechanisms, and using external consistency checking such as watchdog processes. Acceptance testing typically works only in situations where we have some prior information about the attack patterns, categories, and models. Acceptance testing is used quite frequently in the security context. It comes in the form of virus identification patterns, white and black-lists, firewalls, and similar. When properly implemented, acceptance tests can detect certain types of zero-day attacks.

Redundancy-based mechanisms, on the other hand, are a step towards an oracle based on the expected behavior of a healthy system. That approach is also able to detect zero-day attacks. It is part of a technique where we use functionally equivalent, but in some way diverse components, and we compare their outputs and reactions for a given input vector. The full implementation involves voting followed by mitigation. Diversity, at the right level, provides an ability to avoid fault and failure correlation. It also helps detect unusual behaviors and inconsistencies not observed before, and through that detect attacks. A special case is back-to-back testing which is focused on anomaly detection only. Redundancy is not used often in the security context beyond diversification of code memory locations. In part this has to do with the potential cost, and another concern is the effectiveness of the approach.

In this thesis we examine the practical viability of back-to-back testing for detection of security attacks through a) an analysis in the context of recent attacks on web-servers, b) examination of the theory of back-to-back testing in this context and c) an exploration of what a viable practical diversity-based cyber-attack resilient system may entail. As a part of this, we examine the theory of n-version programming in the context of security vulnerabilities. We show that a number of recent difficult-to-detect attacks on some popular web-servers are detectable through the use of commercial-off-the-shelf redundancy and diversity. This finding applies to both direct attacks in real-time (e.g., via port 80 or 443), detection of latent vulnerabilities, and vulnerabilities injected through another method than direct attacks. Finally, we discuss the practicality of implementing a back-to-back testing approach.
Redundancy-Based Detection of Security Anomalies in Web-Server Environments

by
Roopak Venkatakrishnan

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2014

APPROVED BY:

Dr. Laurie Williams

Dr. William Enck

Dr. Mladen A. Vouk
Chair of Advisory Committee
DEDICATION

To my parents.
BIOGRAPHY

Roopak Venkatakrishnan was born in India. He received his Bachelors degree in Computer Science from Anna University, Chennai in June 2012. He joined North Carolina State University for the masters program in computer science in Fall 2012. He is expected to be a Software Engineer at Twitter starting July 2014.

Apart from coding and gadgets, Roopak has variety of interests which range from cooking, trekking, cycling, all the way to training dogs. He and his brothers have spent a lot of time teaching his dog Razor to perform some interesting tricks.
ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Mladen Vouk for his guidance through my graduate program at NC state. Working with Dr. Vouk has always been both a fun and learning experience which I have enjoyed. I would like to thank Dr. Laurie Williams for introducing me to the world of Software Security with a memorable course and Dr. William Enck for graciously agreeing to be on my committee.

I would like to take this opportunity to thank my parents, my friends Vinayasree, Aparna, Sarah, Naren, Nikhil and Anant for their support and encouragement. I would also like to thank Ashwin Shashidharan for his guidance and inspiration when I was writing my thesis. I would especially like to thank Roshan George who took time to look through my thesis and gave me some valuable inputs.

This work is supported in part through NSF grants 0910767 and 1330553, the U.S. Army Research Office (ARO) grant W911NF-08-1-0105 managed by the NCSU Science of Security Initiative and the NSA Science of Security Lablet, and by the IBM Share University Research and Fellowships program funding.
# TABLE OF CONTENTS

## LIST OF TABLES

| v7 |

## LIST OF FIGURES

| viii |

## Chapter 1 Introduction

| 1.1 Motivation | 1 |
| 1.2 Attacks | 1 |
| 1.3 Detecting Attacks | 2 |
| 1.4 Redundancy | 3 |
| 1.5 Web-Servers | 5 |
| 1.6 Classifying anomalies | 5 |

## Chapter 2 Related Work

| 2.1 Faults, Failures, Vulnerabilities and Exploits | 9 |
| 2.2 Fault Tolerance | 10 |
| 2.2.1 Redundancy | 11 |
| 2.3 Effectiveness of Redundancy | 14 |
| 2.3.1 Effectiveness | 15 |
| 2.3.2 Effectiveness - An experimental study | 18 |
| 2.3.3 Another look at theory | 19 |
| 2.3.4 Total Cost | 21 |
| 2.3.5 System Structure | 21 |
| 2.4 Detecting Security Anomalies in Web Browsers using Diversity | 23 |
| 2.4.1 The Algorithm | 24 |

## Chapter 3 Case Studies

| 3.1 Research Methodology | 25 |
| 3.2 Linux/Cdorked.A Attack 2013 | 26 |
| 3.2.1 What Happened? | 27 |
| 3.2.2 Could back-to-back testing have helped? | 27 |
| 3.2.3 Other Similar attacks | 28 |
| 3.3 An Nginx Vulnerability (CVE-2013-4547) | 28 |
| 3.3.1 What Happened? | 28 |
| 3.3.2 How a comparison based could have helped | 29 |
| 3.4 ColdFusion IIS Attack - (CVE-2013-0629) | 30 |
| 3.4.1 What Happened? | 30 |
| 3.4.2 How they got in? | 30 |
| 3.4.3 What did it do? | 31 |
| 3.4.4 Using back-to-back Comparisons to detect this malware | 32 |
| 3.5 Zendirect Redirect - Feb 2014 | 32 |
| 3.5.1 How did it work | 32 |
| 3.5.2 How back-to-back Testing could have found this | 33 |
| 3.6 Apache Tomcat Possible DoS Attack (CVE-2014-0050) | 34 |
LIST OF TABLES

Table 1.1 Number of web-servers of different types Nov 2013 - Source:Netcraft(http://www.netcraft.com) ........................................... 6

Table 3.1 Time taken to detect some recent web-server compromises. .................. 37
LIST OF FIGURES

Figure 1.1 Web-Server Distribution (Nov 2013) - Source:Netcraft(http://www.netcraft.com) 5
Figure 1.2 Types of Anomalies 6
Figure 1.3 Direct attacks to the web-server 7
Figure 1.4 Lateral compromise of a Web-Server 8

Figure 2.1 Errors, Faults, Defects, Failures and Anomalies - NCSU CSC547 class slides[59] 10
Figure 2.2 Log of Short Burst of Attacks - (logs analyzed using Loggly) 16
Figure 2.3 Probability of detecting a failure with change in probability of common mode failure 20
Figure 2.4 Structure of a simple voting system 22

Figure 3.1 Voting could be used to detect the Nginx vulnerability and decide on the correct response 29
Figure 3.2 CFML passed to Adobe ColdFusion to exploit a vulnerability 31
Figure 3.3 An example of a redirect sent by the Zencart malware[62] 33
Figure 3.4 Example of Multipart boundary in form post (as seen in Chrome debugger) 34
Figure 3.5 Remote Code Execution on Struts 2 - https://www.mandiant.com 36

Figure 4.1 General Architecture with replicated stack 41
Figure 4.2 System Architecture with acceptance testing and watchdog software 45
Figure 4.3 Real Time detection using Cloud based voting System 47

Figure A.1 Vulnerability in nginx being exploited 62
Chapter 1

Introduction

1.1 Motivation

The number of web-based attacks has increased by over 30% from 2012 to 2013 [63] and this number continues to increase. In practice, security of a system before it is released can only be improved to a certain degree. After that, we have to work toward detecting attacks and intrusions and finding a way to mitigate and manage them (preferably pro-actively).

Most of the solutions in place today look for and identify threats which we already know or attacks that we can expect. Identifying zero day attacks (vulnerabilities that we haven’t seen before) is an open challenge. We believe that using a security-adapted “N-Version Programming” (or NVP) is one way of doing that. In “classical” software reliability engineering (SRE), NVP is a technique that uses redundancy to identify and mitigate potential run-time failures[42, 31, 32]. This thesis focuses on evaluation of its testing variant, back-to-back testing, as a security anomaly detection mechanism. The specific domain of interest is the security of web-server core.

1.2 Attacks

According to the Kaspersky Security Bulletin [28], the number of attacks which are web based (i.e. that use the web as a medium to propagate) increased to 1.7 billion in 2013, with over 90% of these trying to employ an attack embedded in the HTTP / HTTPS traffic. And this is only for the companies which participated in Kaspersky’s study. It is safe to say that the number of web based attacks is growing alarmingly every year.

The use of malicious URLs is the most common method of attack. The report also states that most malicious URLs detected were for sites containing a vulnerability and sites which redirect users to exploits. This is interesting because this suggests that many web-servers are vulnerable
to attacks and may have been compromised. Another thing to note is that the current methods of
detection can be improved. In a number of recent situations, compromises took a long time to be
detected. It is unfortunate that in many situations administrators were the last ones to find that
their server had been compromised. As an administrator, being aware of the different attacks
discovered all the time is not as easy monitoring a database. Attacks are discovered by various
different security groups, most of whom maintain a blog where they post their findings. Usually
one can be confident that almost all major attacks are reported to the CVE Mitre database.

Staying up to date with this however is not as easy following a blog. Some well known blogs
which report major breaches/attacks/vulnerabilities in the industry include ArsTechnica[4],
Techcrunch[2], and it almost always comes up on HackerNews[73]. Finally staying up to date
with the CVE top 25[39] and the OWASP top 10[44] is some thing all web admins should do.

This thesis focuses on detection of attacks on, and intrusions into, web-servers using the
concepts of diversity and redundancy.

1.3 Detecting Attacks

There are essentially three ways of detection attacks and intrusions at run-time. They are appli-
cation level acceptance testing based on prior knowledge (e.g., signatures, patterns), watching
for anomalies from the outside(external consistency checking), and oracle tests not based on
prior knowledge but on behavioral tests such as voting.

Acceptance Testing - Pro-active acceptance testing is a technique where the input to the
system is validated against conditions given by the programmer. An example is a white-
list. Any input passed to the system first passes through an acceptance test and only
if it passes all the conditions of the test does it even get sent to the system. A reactive
acceptance test can detect attacks and some intrusions but only after the fact. It examines
the behavior (output) of a system, and verifies it against some acceptance criteria. If the
test is negative, an alarm is sounded, and mitigation is initiated. The pro-active system
would be perfect if we had the optimal white-list. That is never the case, and usually the
white-list is updated and augmented with black-lists (explicitly barred requests, IP num-
bers, users, etc.) as more attacks and vulnerabilities are found. A good start to obtaining
a white list would be to get the coverage-based operational profile of the system. The two
ways of approaching this are a) learning, and b) static analysis with dynamic updates.

Let us consider a web-server. Let this server run for a while, and then examine the
access log of the server to extract the operational profile of the server by computing the
frequency of requests for a given signature (a signature is a unique HTTP/HTTPS based
request to the server, e.g., GET /CC/index.html). If we then sort the content and pick out
the unique values, we get a white-list which essentially contains inputs accepted by the system. The downside of dynamic profile generation is that one of the requests may hit a vulnerability and result in an exploit. Plus, there may be some rare legitimate requests that do not appear in such a list. A better, but more expensive, approach is to first create a draft coverage profile by traversing the web-site content tree, making sure all outputs are correct. This works well if the site is static. If the site changes, or has dynamic elements (e.g., a webapp using PHP or a CGI based application), some dynamic profile building may be needed. This operational coverage profile can now be used to reject requests not on the white-list before they reach the server. The false alarm rate may be an issue and needs to be monitored.

Acceptance testing will work quite well in catching requests that violate a white-list, or an attack explicitly covered in a black-list. Unfortunately, all too often a seemingly harmless request could cause problems. Knowing what to look for and what to put into the table is the problem. This means that this type of acceptance testing may not be able to detect zero-day vulnerabilities.

**Consistency Checking** - This technique uses not only the information available to the system but also some extra information which is usually not available to the system internally, at run-time. Such information may be obtainable during system testing (e.g., through fault or vulnerability injection, and checks to see if defensive mechanisms can detect it), or at run-time through external observation of behavior (e.g., is processing exceeding allowed time?). In practice run-time consistency checks are often not granular enough to be useful except in one situation: diversity based redundancy, or run-time back-to-back testing.

1.4 **Redundancy**

In this thesis and in its context we use the following terminology: A human (or machine) makes an error that results in a root fault (of omission or commission) in the software or its artifacts (e.g., documentation). That fault may propagate as different defects through different stages of the software development, implementation and maintenance. If a defect is encountered at run-time, software may enter an error-state. In the case that the error-state results in an observable anomaly, we may have encountered a failure. If the fault, or one of its defects, is a security vulnerability, then its failure in the field is a security breach and possibly, an exploit. An input vector (data, environmental variables, etc.) that is intended to probe for a vulnerability, or exploit it and cause a breach, is an attack case. Attack cases can be administered during testing phases of the software, or may be received as part of true attacks in the field. Furthermore, changes in the environment, policies and regulations for which the software was designed can
result in exposure, usage profiles that are unusual and can end up breaking the software (or lead to compromised security). Incorrect use of the software (use-error) can also result in a security problem (e.g., using too short a password).

We are always looking for ways to improve the security of systems we create. This can be done in three ways: through fault-avoidance (e.g., perfect design, program proofs, and similar), fault-elimination (e.g., inspections, testing) and run-time fault-tolerance (techniques which recognize run-time error-states and then mitigate the situation through forward or backward recovery).

One such fault-tolerance approach is N-version programming. In N-version programming several versions of the software are developed in such a way that they are functionally equivalent. Under the hood these versions are desired to have diverse designs, perhaps diverse developers, and diverse implementations. A good example are different web-servers. While differing is details and in capabilities, at the core (delivery of HTTP or HTTPS requested streams, pages, etc.) they are functionally equivalent and in principle, conform to WWW standards. The conceptual idea behind N-version programming is that the same input is given to multiple functionally-equivalent versions of the software they then process it to give an output. If the essence of the output is the same for all servers, we say that they agree. Our assumption is that the probability that multiple functionally equivalent but diverse versions have a similar vulnerability that results in an identical and wrong response is very low. This means that comparison of the results obtained from such functionally equivalent but diverse software may be an excellent detector (through disagreement) of possible issues in one or more of the versions. Such comparisons are sometimes called back-to-back testing, and sometimes voting.

Detecting security anomalies at run time on the basis of design diversity, is potentially a good solution. However, we need to look into the various issues which might be a roadblock for using a comparison based system. These include, common mode failures, dealing with dynamic applications, identifying what properties to compare, and how to vote if it is desired to identify the “correct” response. The simplest form of voting is majority voting, but other voting techniques are possible (e.g., Consensus voting, Two-out-of-N voting, etc.[8, 19, 35, 36])

Detection of issues using comparisons, or voting, is one side of the problem. Mitigating or managing detected problems is another issue. In this work we limit the scope to detection. Also, our focus, in terms of real-world environments from which we draw case-studies, is on web-servers. One of the reasons is that functionally equivalent software is expensive, and in general, not easy to come by. Web-servers, through their standard WWW compliant cores, offer this in the commercial-off-the-shelf (COTS) space, and because of their importance, they are a continued target of attacks.
1.5 Web-Servers

When focusing on web-servers it helps to understand the distribution of the various kinds of web-servers across the Internet. Over 80% of the websites in the world use either Apache, Nginx or a Microsoft (IIS) server[43]. Another web-server which powers a sizable percentage of websites is Google’s custom web-server. Figure 1.1 and table 1.1 show the web-server distribution according to Netcraft[43] as of Nov 2013. In practice, most of the attacks directed at web-servers will probably target one of the top 3 servers than target the lesser known ones.

![Web server developers: Market share of all sites](http://www.netcraft.com)

Figure 1.1: Web-Server Distribution (Nov 2013) - Source:Netcraft(http://www.netcraft.com)

1.6 Classifying anomalies

As already mentioned, we focus on attacks that target the webserver core rather than faults in the web applications themselves or third party add-ons like scripting languages(e.g PHP) and databases(e.g. MySQL). Of special interest is an assessment of the ability of comparison-based approach to detect zero-day attacks. In classifying anomalies and their detection we only consider those that manifest through the HTTP/HTTPS port of the web-server, i.e., through its
Table 1.1: Number of web-servers of different types Nov 2013 - Source: Netcraft (http://www.netcraft.com)

<table>
<thead>
<tr>
<th>Developer</th>
<th>October 2013</th>
<th>Percent</th>
<th>November 2013</th>
<th>Percent</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>344,408,387</td>
<td>44.89%</td>
<td>348,159,702</td>
<td>44.33%</td>
<td>-0.55</td>
</tr>
<tr>
<td>Microsoft</td>
<td>177,216,296</td>
<td>23.10%</td>
<td>190,451,702</td>
<td>24.25%</td>
<td>1.15</td>
</tr>
<tr>
<td>Nginx</td>
<td>123,114,800</td>
<td>16.05%</td>
<td>190,968,564</td>
<td>14.00%</td>
<td>-2.04</td>
</tr>
<tr>
<td>Google</td>
<td>34,127,482</td>
<td>4.45%</td>
<td>37,748,743</td>
<td>4.81%</td>
<td>0.36</td>
</tr>
</tbody>
</table>

output, not its logs (although those can help if acceptance testing is added into the equation), and not its interactions with the host system or other applications. We also distinguish between a) webservers which may have already been compromised through other means than requests / attacks via its primary communication port (e.g., port 80) - lateral compromise, and b) those that are attacked through web requests and are compromised through a vulnerability in the server and via its primary communications port - direct compromise. In the latter category we also distinguish between vulnerabilities in the server core, and those in third-party “add-ons” (e.g., PHP, databases, etc.) or web applications.

While detection of a direct attack in real-time is important, so is reliable and timely detection of anomalies that may arise from successful indirect attacks on the webserver host system or a lateral compromise.

Figure 1.2: Types of Anomalies
While we would like to detect all attacks as they come in, this may not be possible in practice. A fall-back is to improve the number we catch before they cause harm. Certain attacks are silent, and don’t manifest in a perceivable manner. In such cases we may find that the malware / worm / virus remains dormant (in most cases collecting data) waiting for the right time to “explode” causing havoc. There have been a number of such cases in the recent times, and unfortunately they were noticed or discovered more by chance than deliberate action. Attacks that crash a server or cause an obvious end-user visible anomaly are relatively easy to detect. However the newer breed of malware tries to be as stealthy as possible, all the while collecting information. In some of those cases unanticipated side-effects of the intrusion are discoverable through diversity-based comparisons.

Figure 1.3 illustrates web-servers that are attacked directly. An attack can result in:

- A modified response but not a breach
- The server can be exploited and as the result of that its behavior on its primary principal communication channel is modified
- Despite being compromised the server seems to respond normally

![Figure 1.3: Direct attacks to the web-server](image)

Figure 1.4, illustrates the situation where the origin of the attack could be any other component which is a part of the system (e.g. ssh server, FTP etc.), a lateral compromise. In the case that the attack targeted the web-server core and modified it or any of its components in a malicious way, and the anomaly manifests some sort of change to the functioning of the web-server (e.g in terms of output), diversity can be used as a mechanism to detect this exploit. Some such attacks are studied in chapter 3. Examples of modifications which can be detected include: injection of a hidden iframe in responses, or a method for the attacker to retrieve data from the system along the web-server primary communication ports.
When looking to use diversity as a means for detecting anomalies, we have to address how effective this solution could be. We must also account for the fact that diversity requires multiple functionally equivalent modules to exist or be developed (which would increase costs). Another concern is the impact of exploits in third party add-ons up the stack (e.g. databases) or in the web applications. Finally, this will only be an attractive solution if it is cost effective and has a low false positive rate.

The rest of this thesis is set as follows: Chapter 2 discusses the related work in this field and how some of these concepts can be modified / extended to detect security attacks. In Chapter 3, we look at some of the recent major attacks on web-servers. Also we analyze how having a back-to-back comparison in place could have helped prevent these attacks in certain cases, and detect an intrusion in the remaining cases. Chapter 4 discusses practical implementation of a voting system, and how we would go about this. Finally, Chapter 5 concludes this thesis and discusses possible future work.
Chapter 2

Related Work

The idea of using redundancy to provide fault tolerance and detect problems has been around for some time[70, 8, 32]. Explicit investigations of the value of redundancy in tackling security issues and problems are relatively more recent[30]. However if vulnerabilities, or security faults, are considered a subset of the general class of faults, many aspects of the already developed body of fault tolerance science apply. There are many factors that have to be taken into consideration when considering an attack-tolerant system. One of the first things to understand are the differences between non-security faults and vulnerabilities, and the related assumptions. This section reviews the state-of-the-art in the diversity-based fault tolerance with a special focus on security.

2.1 Faults, Failures, Vulnerabilities and Exploits

As already mentioned non-security and security software problems have many similarities, but also some key differences. The latter include consequences of these problems turning into failures during field use, and the profile of the actions that lead to such failures - i.e., attacks. In turn, this dictates both pro-active and reactive processes and approaches that can be used to defend against such actions and consequences.

**Error** - An error is a mistake made by either human or machine. An error implies there is a difference between what is expected and what actually occurs. An error leads to a physical or policy fault.

**Fault** - Caused by an error, a fault propagates as a defect through the stages of software development, implementation and maintenance in a program.

**Defect** - If a defect or defective policy is executed / encountered at run time, the software enters an error-state.
**Vulnerability** - In terms of software security, CVE\[41\] defines a vulnerability as a mistake in a software that can be directly exploited by a hacker to gain access to a system. A mistake is a vulnerability if the hacker can use it to violate a security policy (excluding an open policy, where all users are trusted).

**Failure** - An error-state containing an observable anomaly results in a failure. If the failure is caused by a fault or defect which is a security vulnerability, the failure in that case implies a possible security breach brought about by an exploit.

**Exploit** - An exploit is a piece of code, or set of actions that can take advantage of an existing vulnerability in a system to gain extended access, privileges, or violate security policies.

### 2.2 Fault Tolerance

There are only two ways that guarantee that a software is completely free of faults, exhaustive testing and proof of correctness. In practice, neither is feasible for complex systems. Therefore the aim is to develop software which can work reliably despite lack of perfection.

There are many approaches taken to developing fault tolerant software. In all situations this
involves a way of detecting that there is a problem, and then mitigating the problem. In pro-
active fault tolerance failure is assumed and mitigation is always applied (e.g., error coding
based forward recovery such as convolutional coding), in reactive fault tolerance mitigation is
applied only if an actual failure is suspected (e.g., if memory checksum fails, some action is
taken).

Classical fault tolerant software uses methods like acceptance testing, voting or external
consistency checking to determine if the result produced is correct. As already mentioned,
acceptance testing is an approach where the software compares its output against a set of
known and valid output or internal consistency relationships. The acceptance conditions could
be defined by simple tests, boundary conditions, or even a white-list. Acceptance tests are
designed specifically for a system or purpose. Only information available to the program during
run-time is used. Voting and back-to-back testing, on the other hand is a step towards a more
environment independent “oracle” where behaviors of multiple functionally equivalent product
versions are given the same input/request and are then compared to see if they arrive at the
same answer, and if they do not some corrective action is taken. External Consistency checking
is a method that checks information available not only from internal but also external sources to
ascertain consistency of the computations, data, outputs or behaviors. Often such information
is not available at run-time to individual components of a system, but may be provided through
system overview functionality (e.g by watchdog software, an example are time-outs where it is
assumed something is wrong if a component exceeds certain amount of time not communicating
or not visibly progressing towards completion of its tasks), or only at verification and validation
time where known conditions are created, faults are injected, etc.

When, at run time, software encounters a defect it may end up in an error state which may
be perceived by the user as an anomaly. If the error state is identified, steps can be taken to
recover.

- **Backward Recovery** - In this mode, the state of the software is repeatedly saved. Each
  of these “save-points” is a recovery point. When a software has to recover from an error-
  state, it rolls back to the last recovery point, and then restarts from there.

- **Forward Recovery** - In this case, software masks or compensates for an internal error-
  state by deliberately entering a new state in which it can operate, either in a degraded
  mode with less features (like a “safe-mode”), or in an error compensation mode which
typically uses redundancy to find the correct answer or mask the error.

### 2.2.1 Redundancy

The idea of using similar software modules to improve the quality of software came from
hardware[16]. Hardware can randomly fail as it gets older, and thus using identical backup
components in case one piece of hardware failed, is a sensible approach. This idea was extended
to the software space\cite{8, 9, 46}, but the problem was that software faults are usually due to
design flaws or implementation errors, so replication either does not guard against failures, or
does so under special circumstances. For example separate groups of developers would develop
functionally equivalent components\cite{10, 48, 74}. When two groups of people are given identical
specifications to come up with software it is assumed that they would not use the exact same
design and implementation details and algorithms. It is also assumed that the faults left in these
software components will not be the same, i.e., common-mode or correlated, meaning that the
two versions would not fail on the same input and if the software failed, it would not fail with
identical and wrong responses. Diversity is the key here. This means that everything from the
language it is written in, to the operating system it runs on, to the run-time memory locations
can be used to make its run time internals more diverse, and still keep its functional behavior
the same. The idea is to make functionally equivalent software different in the right way, so
that the probability that two pieces of functionally equivalent code the outputs of which will
be compared fail in a correlated fashion is as close to zero as possible.

For fault tolerance that manages inherent faults that fail randomly during software operation
has been studied in great detail over the years \cite{33, 31, 32, 29}, and it appears to work well so
long as certain conditions (such as independence of failures) are satisfied. In fact, redundant
functionally equivalent software is used today to fly aircraft \cite{74}, and guide trains \cite{27, 37}.
When we try to extend the idea of using fault tolerance to security, it is necessary to take into
account that we have versions diverse enough to detect an attack. We want to make sure that
an incoming attack is identified as an attack and noticed as soon as possible.

In both acceptance based algorithms and in redundancy based error-state detection algo-
rithms one has to decide whether response(s) of a system is (are) acceptable. In the first case it
is done against known patterns and behaviors, in the second case it is done against the behavior
(or output) of the redundant versions. The algorithm that does the comparison and decides on
this is known as an ‘adjudicator’. There are many algorithms that are used for this purpose right
from simple acceptance tests (e.g., is the answer larger than X), to majority voting, to median
selection, to more complex decision analysis. We distinguish between anomaly detection, and
actions that may follow adjudication.

A correlated failure is one where two or more components fail on the same input. If the com-
ponents have the exact same response, the event is known as an identical-and-wrong response
event. Detection of such events is difficult using techniques which rely on the comparison of the
outputs alone. A point to note is that such a failure does not necessarily imply that the versions
failed on a “common cause” fault. A common cause fault is one where multiple versions fail
because they may share a faulty piece of code or component. The output of different versions
in the case of a common cause failure could be similar but need not necessarily be the same.
Adjudication Strategies based on Voting

In situations where one is not satisfied just with a finding that that there is a difference among the answers coming from several functionally equivalent versions that are given identical inputs, it may be necessary to decide which of the responses is the “correct” response. Voting is a common method of finding that there is an issue, and in the mitigation state what the most appropriate or correct response might be. For example, Lyu[33] and McAllister et al.[36] review some of the more commonly used methods. A point to note here is that, the output chosen by the voting-based adjudication method is only the agreed upon output. Also, an output could be a single number or character, a range of numbers or symbols, a vector with agreement measured through a distance between vectors, etc. It is not necessarily the correct output. Following are the adjudication strategies used in classical fault tolerance. Some of these could be used when working with security vulnerabilities.

Simple Majority Voting As the name suggests, this method looks at all the outputs and tries to pick the one with at least $m$ out of the $N$ outputs it received. Usually $N$ has a value of 3, and in some cases is greater. and $m$ is given by $\lceil (N+1)/2 \rceil$. Here $\lceil \rceil$ represents the ceiling function.

Two-out-of-N voting Scott et al.[52] showed that given a reasonably large set out outputs it would not be necessary to ever have an $m$ greater than 2. Thus with $N$ outputs to choose from, any output having at least 2 similar outputs from a group of N can be chosen as the agreed output. This method may be particularly suitable in the context of cloud technologies where a large number of diverse functionally equivalent software may be available. In fact, in this context location diversity may be sufficient to detect issues since a only one location may be compromised.

Median Voting This technique works when the number of possible outputs is a range. It would also require that the outputs can be arranged in some order (i.e preferably they are numerical). The median voting algorithm picks the middle output from the set of outputs. The reason this may be better than an algorithm that calculates the average is that it won’t be subject to a bias. Further in the case of discrete valued output a median voting algorithm will always succeed whereas the same cannot be said about the mean(or average).

Consensus Voting[36] It is a modified version of majority voting. The algorithm works as follows:

- If $m$ outputs agree, where $m$ accounts for more than half of the total number of outputs, then this answer is chosen.
• If there is a unique set of $m$ outputs which are all the same but, $m$ is not half of the total outputs, this answer is chosen. Only if $m$ is the maximum number of agreeing outputs. If there are two outputs, each with $m$ then another decisions step is needed.

• If there are two or more sets of outputs which agree each with $m$ outputs, then the following is used to pick one.
  
  – If consensus voting is used with N-Version Programming, any one of the groups is chosen using a random selection strategy.
  
  – Else If consensus voting is used with recovery block mode[46, 47], an acceptance test is used to pick one of the outputs.

2.3 Effectiveness of Redundancy

In the context of this thesis, we will say that a method is more effective than another method if it achieves a goal, such as identification of attacks, with higher probability than another method. On the other hand a method is more efficient than another method if it achieves the same goal with less resources (or lower cost).

A development-phase variant of N-version programming[8], again possibly quite suitable for cloud environments, is known as back-to-back testing[70]. The various versions of the software are developed by different groups of people independently in order to avoid the same faults in all versions. These software versions are built to satisfy the same requirements and hence their output should be the same to some tolerance level for the same input. In the cloud context, it may mean testing software against cloud-based services that already are functionally equivalent.

In the case of back-to-back testing one is interested in identification of issues (or disagreements) rather than automatic adjudication of correctness. This may be all that is needed in the case of security attacks - a warning bell. We now examine theoretical effectiveness of this approach.

There are several potential issues with back-to-back testing, and its run-time equivalent N-version programming. They include efficiency and costs since development of multiple versions can be expensive. However, now in the day of clouds and massive functional and resource redundancies, this may not be an issue anymore. They also include electiveness, the ability to detect issues, and as always the structure of architecture of the system. Again, as clouds proliferate, a side-benefit may well be automated back-to-back testing and pro-active redundancy-based run-time fault tolerance.

In this context, and for the purposes of this thesis we define the following terms

**Effectiveness** - The ability of a system (algorithm, solution) to detect an anomaly, especially a security anomaly.
Efficiency - The ability of a system to be effective with minimal expenditure of resources (e.g., time, test cases, redundant versions) and meeting its other requirements and performance parameters.

Total cost - The cost incurred for a system to deliver effectiveness and efficiency. This includes accounting for hidden-costs, development costs, operational costs, etc. For example, a proprietary system may be effective and efficient, but the cost of licenses, hardware and operation may be so high that it is beyond the price-point affordable by an organization.

2.3.1 Effectiveness

How effective is N-version programming or back-to-back testing in detecting attacks and vulnerabilities. Since attacks are coming from the outside, and a security failure is a very undesirable event, the focus should be on detection of attack probes and attack cases or input vectors before they impact the combined services offered by the redundant systems. To do that, we follow the reasoning discussed in the back-to-back testing papers by Vouk[70, 71]. However, assumptions are modified to reflect attack profiles and vulnerability characteristics. In principle if we have enough versions all designed such that they are diverse and will never produce an identical and wrong response, we would be able to detect any attack the first time it comes in, this includes zero-day attacks. Obviously when trying to implement this we come across some road blocks.

So what is the difference between non-security faults and failures, and security faults and failures or vulnerabilities and exploits? To start with there are many more non-security faults than vulnerabilities. According to Anbalagan[1] and Subramani[60] security problems in open source software are between 0.02 and 5% of the total number of reported problems. Furthermore, only a fraction of known vulnerabilities (not more than a few percent) are actually exploited in the field[18, 7]. This means that security events tend to be rare events, although potentially very high-risk rare events [49, 66, 7, 56]. This also means that rare-event statistics may apply.

The process of discovery of security problems or faults under normal operation does not appear to differ too much from what is happening with non-security faults. This allows prediction of the security properties of a software under normal operating conditions[1, 60]. Under normal operation software has an operational profile and frequency of operations/requests/inputs that reflects its use by non-malicious everyday users[42].

When an attack occurs, the situation changes. There are two extremes in that domain - slow background attacks, and short intense burst attacks. In both cases, sequences of requests and inputs that are sent to the victim system are deliberately modified and biased to maximize the chances, on part of the attacker, of detecting an exploiting a vulnerability in the software. Figure 2.2 shows a rapid burst of attack requests (in fact probes) intended to find out if the server under attack is operating a possibly vulnerable implementation/configuration of PHP.
This burst is mixed in with normal input cases. On the average, for that particular server, such attacks are only about 5% of the overall request traffic. From the defensive perspective, it means that we need to be looking at detection (and mitigation) of anomalies that may be true attacks among the 5% of the traffic. While real attacks may come in bursts, on the average, for this server, an estimate of the probability of an attack (that includes probes, as well exploits) is 0.05 or less. In the case of this web-server example, none of the attacks have been successful so far. Only very few of the attacks were sophisticated enough to escape immediate detection based on the existence of the requested web-page or service (404 of 403 error codes) - the rate was about 2 or 3 in 100,000 requests.

How effective is multi-version software in discovering that? To understand that we make some explicit assumptions about the attack streams (or attack profiles), attack ranges (attack surface), the processes attacks follow, and the properties of the attacked software.

Let requirements be implemented in multiple versions \( V_1, V_2, \ldots, V_i, \ldots, V_N \), \( i = 1, \ldots, N \) so that operational software is functionally equivalent, but has some diversity property. Let \( T_j \) be the \( j^{th} \) input vector (which may have a number of dimensions) that is received by all versions. Let there be \( n \) input vectors to a version (e.g., web requests to web-server). Let the probability that version \( V_i \) either fails (i.e., is compromised) or reacts to a security attack related probe in an anomalous fashion be \( p_{A_i}(T_j) \). Let, for simplicity for all \( i \), \( p_{A_i}(T_j) = p \). Let’s also assume that version failures or anomalous reactions are independent, i.e., \( P(\text{version i fails} | \text{version k has failed}) = P(\text{version k fails} | \text{version i has failed}) \) where \( i \neq k \).

In this case the probability of a server giving a normal response would be \( 1 - p \). The probability that all \( N \) versions respond as expected is then given by \( (1 - p)^N \). Following from this, it can be shown that the probability of \( N \) versions of the web-server detecting an attack

---

**Figure 2.2: Log of Short Burst of Attacks - (logs analyzed using Loggly)**
probe or an anomaly after running n inputs is

\[ P_D(n) = 1 - (1 - p)^{Nn} \]  \hspace{1cm} (2.1)

This works, and will even detect zero-day attacks, so long as the output vectors from failed or flagged versions are not identical. Depending on what the output space looks like, this may happen more often than one would be comfortable with. For instance, if all failures result in error code 404 and only that code is used to make comparisons, than it would indeed be difficult to distinguish a security failure or anomaly just based on comparisons. Of course, in this simplistic example, the fact that the code is 404 is already a warning (acceptance test), makes detection of such an anomaly automatic. It is less clear what to do if the returned code by all versions is 200 (OK), but we actually have an issue. Therefore, it is important for the output vector used for comparisons to include more information about the output and through that increase the cardinality of the failure output space. Comparisons in the of web-service status code space only is in fact a problem akin to voting in small output space [35]. This problem has been recognized by Totel et al.[69] in the context of intrusion detection for web-servers, and they offer some suggestions on how to increase the cardinality of the output space in which comparisons are being made. We discuss this paper further later in this section.

The effect of failure dependence is illustrated using a system where a single identical vulnerability is present in all N versions, i.e., the fault has span of N. Then, when one component fails, all fail. As the vulnerability has span N, for this vulnerability the effective behavior of the N-tuple is equivalent to that of a single version. If all versions fail with an identical answer, then this failure cannot be detected. If the answers are not identical, there is a finite probability that the failures stemming from this vulnerability or vulnerability probe will be detected. It can be shown [70] that in the first approximation, the probability that such a N-tuple detects such an attack is

\[ P_D = 1 - [\gamma_N p + (1 - p)]^n \]  \hspace{1cm} (2.2)

where, \( \gamma_N \) is the conditional probability that the N output vectors from the versions are accepted and are identical given that all versions have failed. A practical question is how often do functionally equivalent products, particularly the ones developed by different manufacturers, have identical vulnerabilities that could not be detected by increasing the cardinality of the space used for comparisons. A more helpful metric in the context of security is the conditional probability, \( \Gamma_1(N) \), that “all N versions agree on an answer, given that one or more of them have actually failed”. The probability that comparison detects a failure is
\[ P_D = 1 - \Gamma_1(N)P(\text{one or more of the } N \text{ versions fail coincidentally}) - P(\text{all } N \text{ versions return correct answer}) \quad (2.3) \]

where \( P(\ldots) \) denotes the probability of the event \( \ldots \). If \( F_u(N) \) represents the experimental frequency of undetected failure events, and \( F_D(N) \) the frequency of failures that have been detected, then the ratio

\[ \Gamma_1(N) = \frac{F_u(N)}{F_D(N) + F_u(N)} \quad (2.4) \]

is the fraction of failures not detected by comparisons, and it estimates the above conditional probability. Failure detection effectiveness of attack detection also depends on the number of output variables, or system states, that are monitored for agreement. Earlier work on non-security faults by Vouk confirms that\[70\].

2.3.2 Effectiveness - An experimental study

Section 2.3.1 looks at the theory of effectiveness of diversity. A study by Jin Han and others\[21\] from Singapore Management University consisted of classifying various Real-World vulnerabilities and looking as to what portion of them would be detectable using diversity.

The authors assumed that the chances of a common mode failure is low. This makes sense, as the chances of two different applications developed by two completely distinct groups with no shared modules failing on the same input vector in an identical manner is low. The paper tries to answer questions like,

- If a software was vulnerable, did it have substitutes in the market?, and whether they could be exploited using the same attack
- How diverse the software was? Were there substitutes which would run on different operating systems? If one ran on multiple operating systems was it vulnerable to the same attack on different OSes?

If these questions were answered one could talk about how effective diversity could be as a mechanism for detecting attacks. The authors analyzed about 6000 vulnerabilities during the year of 2007 to find the answer to these questions.

They classified vulnerable software into different categories such as application software, web script modules, operating systems, and languages and libs. In the context of this thesis we are specifically interested in application software and web script modules. Web script vulnerabilities comprised of about 46% of the total vulnerabilities, while applications were about 42%.
Application Vulnerabilities

The authors concluded that 45.7\% of vulnerable applications run on multiple operating systems, and in these 84.7\% have the vulnerabilities across operating systems. When the vulnerability represents itself across operating system there are at-least two factors like memory, or machine instructions etc, which make it hard for the exploit to happen on two OSes concurrently. This would mean a fairly decent amount of diversity just in terms of operating system diversity.

Web Script Vulnerabilities

Web script vulnerabilities are split up into cross-site scripting, SQL injection, remote file inclusion (RFI), directory traversal and others. The authors discuss each category and how it could potentially be identified using diversity. XSS can be caught by using diversity on the client side as the code is usually handled slightly differently by each web browser. SQL injection, can definitely be caught using diversity, but masking the differences might be an overhead cost while developing the application or the setup. RFI is considered out of scope as this can be disabled completely at the server end. Directory traversal however can be caught when using different operating systems, e.g., unix based vs windows.

The authors finally conclude that more than 98.5\% of the vulnerable applications have substitutes and that the chance of a common mode failure is extremely low. They also discuss that though the versions of the same software on different operating systems has about 80\% chance of failing on the same vulnerability, the attack code for OS distributions is different. They finally talk about the drawbacks of their analysis which, is that a lot of manual work is required to categorize attacks and how some are not categorizable easily and involve human judgment. The conclusion is that diversity can be used effectively to a certain extent to detect attacks.

The work discussed in this thesis complements the work in[21] by investigating some recent difficult-to-detect web-server vulnerabilities and attacks, and discussing the ability of redundancy and diversity to detect these attacks and vulnerabilities.

2.3.3 Another look at theory

An analysis of one of the web-servers on campus here at NCSU showed that roughly 5\% of the requests to a web-server are attacks. By looking at statistics from reports like Imperva’s Web Application Attack Report[25] and WhiteHats report[54], we can speculate that this value varies between 2 to 10\% depending on the type of website, its popularity, and what it is worth if breached. This is essentially the value $p$.

The paper by Han et al.[21] states that the chance of a common mode failure is low. But there isn't a number to go with this statement. The paper by Eckhardt[16], gives that the
probability of a coincident failure varies from $7 \times 10^{-5}$ to 0.03. Anbalagan states[1] that roughly 0.05% to 5% of the total number of problems are security problems. Assuming that this scales proportionally, we can say that the chances of a common mode security failure($\gamma_N$) ranges between $3.5 \times 10^{-8}$ to $1.5 \times 10^{-3}$. Figure 2.3 shows a plot of Equation (2.2) considering $p$ to be 5% and $\gamma_N = 0$ and $\gamma_N = 0.9$. The value $\gamma_N = 0.9$ is shown for reference. We have to remember that as stated above $\gamma_N$ represents the value that all N versions have not only failed, but also have an identical and wrong answer. We take a conservative estimate that if the N versions fail, they fail with an identical and wrong answer. This is an upper bound on the value of $\gamma_N$, and in any case its value will not be higher. This means that we calculate the lower bound for the probability of detection, and this value would improve depending on the value of $\gamma_N$ in practice.

The graph shows that for very low values of $\gamma_N$, attacks can be detected even with just about 100 or so inputs. As $\gamma_N$ increases we can see that the number of input cases required increases significantly. These curves have an effective value of N as 1, i.e., in case of a common cause fault.
2.3.4 Total Cost

One of the major things to keep in mind when using redundancy, whether to find the correct answer or security flaws, is that multiple versions have to be developed, multiple versions have to be run for their results to be compared, one has to compare appropriate output vectors, etc. This definitely means higher costs, from developing multiple versions all the way to resources required to run such a system.

An analysis of the cost of redundancy can be found in the paper on back-to-back testing by Vouk[70]. Studies done in the context of “classical” software fault tolerance[33, 31, 32] show that developing redundant diverse software is not always a multiple of the number of components, but it may grow sub-linearly. In the case of using redundancy for security, we propose to use Components off the shelf (COTS). The idea is that for commercially used software like web-servers which we focus on, there are already multiple versions of the same software available in the market. They are almost readily available functionally equivalent components, because they are software developed to serve the same specification by independent groups. They satisfy the requirements of using them as versions for back-to-back testing or N-version programming. However where the standard or specification is not clear or is open, each of them may do what they assume is right. So some masking of their design differences has to be done before the output vectors can be compared directly.

One of the reasons that makes redundancy-based systems worth it is the fact that they can be used to catch zero-day vulnerabilities.

2.3.5 System Structure

A fault tolerant system can be constructed in several ways[36, 33, 32]. One of the methods is recovery block(RB). In this technique, the adjudication strategy is Acceptance Testing(AT). Initially the output of the first module is passed through the acceptance test, if it is not acceptable, it rolls back and tries it on the second module and continues serially trying all versions till it succeeds or fails. It is a reactive approach, that may not suitable in security situations. If the first version is compromised (and that is detectable and manageable), and the second is sound, then there may be some value in the approach, but the key here is an appropriate acceptance test - recognition of unusual behavior.

Another commonly used architecture is N-Version Programming. In this case the input is passed to all N versions of the software as shown in Figure 2.4. The outputs along with reactions are passed to a “voter” which looks at all the data it receives and predicts the “correct output”. However in our case it could be modified to help identify a security anomaly. This architecture has variations which are used for different purposes.

Consensus Recovery block - This is basically N-version programming(NVP) followed by
Recovery Block. Here initially NVP is used and if that fails RB is used with the same N versions.

**Acceptance voting** - Here the normal N-version programming is modified such that the output of each version is first passed through an acceptance test and only those that pass the acceptance test are sent to the voting system. This technique will fail if none of the versions produce an output which passes the acceptance test.

**N self-checking programming** - Versions are grouped into pairs, and the output from each pair is compared, only if they agree is this passed on. This is repeated till the system either fails or produces an output.

It is important to note that in the case of security the emphasis is on reliability (or “not get-
ting compromised” first) and therefore a conservative and pro-active approach to detection and prevention of attacks is preferred to “classical” increased availability emphasis where a failure followed by a quick recovery is acceptable. Therefore some approaches such as the Consensus Recovery Block, might be less useful, while the other two methods could be potentially used. Acceptance voting may give us the best of both worlds.

2.4 Detecting Security Anomalies in Web Browsers using Diversity

A paper by Eric Totel et al. [69], talks about intrusion detection systems, and how design diversity can be used to detect anomalies. This paper focuses specifically on web-servers and how voting could be of use here. They recommend the use of COTS to provide redundancy. This thesis builds on the findings of the Totel et al paper by a) focusing on anomaly detection as it relates to some difficult-to-detect recent attacks, and b) examining engineering extensions needed to implement a redundant and diverse web-server solution.

Replicating web-services to balance load or provide fail-over in the case of random outages is a standard procedure. Using diversity and redundancy to combat cyber-attacks is far from routine yet. Achieving design diversity is expensive, as this would involve employing multiple groups to design the same software. This is where the idea of COTS is very beneficial. The authors point out that services available via the web have multiple components off the shelf that could be used. Hence using COTS could be a good idea from an economic perspective. However, the authors point out that there are bound to be differences where the specification is incomplete or open.

Their architecture has a proxy, an Intrusion Detection System (IDS) which is typically used to detect a successful attack on a system [50], and three or more web-servers. Every input to the system is received by the proxy and then forwarded to each of the web-servers. Similarly it also forwards the final response by the system to the client. The IDS compares the reactions of the web-servers to the given input and finalizes the output of the entire system. In the case the reactions of the web-servers vary, the IDS decides if any of the servers has been compromised, and correspondingly raises an alarm.

The web-servers are simple but diverse COTS, which could vary in terms of software, operating systems and hardware too. This variation helps reduce correlated failures.

Identifying design differences from actual anomalies and masking them is needed to allow output vector comparisons. This reduces false positives. Then an algorithm is defined specifically for web-servers which looks at status codes, headers, and the response itself to try and find a majority. In case this does not succeed an alert is raised, which means that this input might be
The authors ran about one month’s requests (about 800,000) to their system. These requests had originally come in to one of their web-servers and a previous analysis showed that 1.4% of these were harmful requests (or attacks). Experiments showed that, when one months requests where passed to this system, it raised an alert for about 0.37% of the requests. Of these raised alerts about 22% of them were thought to be false positives (these include failed attack attempts) and hence the 22% false alarm rate appears to be very high. It is possible to do much better by running acceptance tests via white and black-list filters. Also, most of the web-sites today are not static. The authors recognize this limitation.

2.4.1 The Algorithm

The algorithm described by Totel et. al[69], initially modifies all requests to mask design differences. Masking design differences involves a manual effort initially which tries to list all the possible differences between the functionally equally components. The next step involves partitioning the responses based on the status codes (HTTP Status codes). If a majority doesn’t exist, the algorithm fails and exits. Otherwise the next step is to check if the majority status code was an error code. If so, once again the algorithm terminates after raising an error for the versions that don’t agree. If the majority wasn’t an error code, the responses which are a part of the majority are partitioned on the basis of their headers. At this point as long there exists a majority, the algorithm continues and checks that there is a majority when the body of the entire responses are compared. While doing so, any server not having the same body has an alert raised saying it has to be checked.
Chapter 3

Case Studies

This thesis is focused on an assessment of security anomaly detection capabilities of what is sometimes known as back-to-back testing or comparison of responses from functionally equivalent software. We have seen that the problem has been studied to some extent before. In this Chapter we will examine some recent attacks on web-servers to establish whether back-to-back anomaly detection would have been able to recognize the attacks.

There are two kinds of anomalies that our system could help with. First is where it helps identify an oncoming attack. The second is the ability to identify an attack that has already taken place and has compromised a system. Identifying oncoming attacks is often done quite successfully by other methods such as acceptance tests either at the networking level, operating system or application level. However, acceptance tests do not work unless we have identified patterns or relationships that describe an observable characteristic of the attack case, or its behavioral impacts. The “front end” of acceptance testing usually means that we need to experience an attack before we can defend against it.

The problem of the detection of an existing compromise or hidden (perhaps already exploited) vulnerability is a different one. One way is through comparative analysis of the system behavior.

3.1 Research Methodology

In order to assess the effectiveness of diversity as possible mechanism of detecting security anomalies, this thesis looks at the science behind diversity, and also examines attacks on some popular web-servers in the last five years as case studies. All the attacks published as vulnerabilities by Apache httpd, Microsoft IIS, and Nginx over the last five years were studied. The vulnerabilities examined included all those published in the CVE database over the last five years pertinent to these servers. This consisted of about 60 published attacks. Apart from this,
recent attacks published in various security blogs (e.g. FireEye, Sucuri, Symantec etc.) which included a vulnerability in the web-server or a lateral compromise affecting targeting web-server were also studied.

Each vulnerability was examined to see if a back-to-back testing based approach could be used to detect an exploit. This involved studying the details of the vulnerability and assessing how back-to-back testing could play a role in detecting this attack. In the case that a preliminary scan suggested that diversity could work, if the vulnerability was replicable, a manual comparison of the output vectors from diverse components including the vulnerable version was performed to verify that a diversity based approach would work. In certain cases, especially in the cases of lateral exploits, replicating the vulnerability or the exploit was not possible. In such a case the analysis of the vulnerability was a thought process which involved looking at the reports by various security experts and determining if back-to-back testing would have caught such an attack.

The manual tests performed involved trying to replicate an exploit in diverse environments. The aim was to see if the output vectors from the diverse components showed a difference which could be identified by monitoring the output of the web-server. The tests also aimed to identify if there were correlated problems, and if that was the case, whether increasing the cardinality of the output vector would help identify the exploit. Initially only the status codes were looked at, but as more use cases were studied it was found that observing at least three or more fields in the output vector (such as status code, response size, last modified etc) would reduce the chance of an exploit being passed as a correlated failure. These tests showed that for the use cases studied, diversity would potentially help catch difficult to detect attacks.

3.2 Linux/Cdorked.A Attack 2013

This is the situation where a system, in this case a web-server, has been compromised is some fashion, but that exploit has not been detected yet and the compromised system was put into service. We will call this type of vulnerability / exploit an independently injected exploit. It is not a native vulnerability of the original software (in this case a web-server) and it did not happen through http request based exploitation of such, but through other means. This particular attack was called the most sophisticated attack on Apache like web-servers[17]. It is known to affect Apache and Nginx. It was unclear how the attackers were able to deploy a modified binary. The recent Windingo report[18], says that Cdorked was part of a larger operation, and a previous exploit on openSSH known as Ebury was used to deploy Cdorked. What is interesting is that the malware was written specifically for each of Apache, Nginx and Lighthttpd. Though much of the code was re-used the hooks for each version was different based on the server it was targeting.
3.2.1 What Happened?

Intrusion to inject Cdorked.A not appear to leave any trace in the logs of the affected system. Of course, Cdorked.A has its code hidden in the system, and this extra code is detectable if one knows what one is looking for (or has a way of checking uncompromised installation logs). It always operates only on memory. Thus a simple restart destroys all evidence of its operation. Initially the speculation was that a flawed installation of add-ons from a compromised(fake) installer was allowed. The recently published Windingo report[18] states the actual method used by the attackers. A lateral compromise which successfully broke into openSSH gave the attackers access to deploy Cdorked.

What made things worse is that the attack was designed in a way that made it very hard to detect it by admins of the machines. The infected web-servers had a plugin / module which sent the user a URL redirect based on many conditions which included a random factor as well. Some of these conditions included a check to see if the URL was from an admin like page, and if it was to not, redirected the user. Since operation of the malware was limited to in-memory operation, detection of Cdorked.A was a challenge. Cdorked.A made sure that no information were stored in the logs.

3.2.2 Could back-to-back testing have helped?

Our analysis, shows that detection of this malware would have been much easier using back-to-back comparisons. The reason is that this particular malware affects only Apache, and a small number of Nginx server versions. This means that a functionally equivalent but diverse server such as IIS etc. remained unaffected.

A triply redundant diverse system based on the Apache, Nginx, and IIS could have identified the compromise automatically by comparing results of relatively simple system inputs and outputs. Unfortunately the attack has a random factor based on which the redirect was sent. This means that passing the same input query to multiple infected servers would not mean that all of them would generate the malicious redirect. Therefore even without the use of diverse (different) servers, redundancy would be effective in detecting unusual behavior. Use of different servers adds an additional advantage. Simple web-server status codes and response sizes could have been used to construct a back-to-back testing mechanism, and would have raised an alert provided a sufficient number of requests were given to the system.

Obviously in this particular case, it is possible to get false alarms, but multiple such alerts should point someone to the fact that something is wrong. From then on it would be digging into what was going on. This is better than what was actually experienced with this malware. Web-server administrators did not know their servers had been compromised in some cases for over 6 months[7]. Cdorked.A had been initially noticed in August 2012, but without much
idea of what exactly it was. ESET and Sucuri analyzed and wrote a script to detect this rogue module in early February 2013 when the attack gained momentum[18]. Several thousands of servers were infected even after this, as it was only on March 20th and later that anti-virus scanners added this to their databases.

This particular attack, affected both Apache and Nginx. There is a possibility that an attack could affect all the types of servers our diversity approach uses. A point to be noted here is that in such a case even the use of 3 compromised Apache and Nginx servers could potentially detect the attack. The trigger for redirection includes a random factor which gives us the chance of detecting this particular attack with even compromised servers so long as random events to not coincide.

3.2.3 Other Similar attacks

The Cdorked.A malware is an example of a whole category of “on-installation” malware that hides and when invoked operates in memory. There have been other attacks which are similar and could have been detected using back-to-back testing methods as described for Cdorked.A. They are listed in table 3.1.

One specific example is Effusion is a malware which is similar in many ways to Cdorked.A. It also could have been found using a very similar approach[12]. It resided on RAM, lived on the modules of both apache and Nginx. Attacking Apache and Nginx using modules seems to be a new trend when trying to exploit / break into web-servers. This particular malware is reportedly being sold for $2500 for one copy.

3.3 An Nginx Vulnerability (CVE-2013-4547)

In November 2013, an engineer from Google discovered a vulnerability in Nginx[15]. This vulnerability could have allowed an attacker to bypass security restrictions among other things. This is a different category of security faults. It is a native fault in the software, and it is exploited through specially crafted server communication port requests. Again a comparison based system could help detect this attack as it happens.

3.3.1 What Happened?

When a request is passed to a vulnerable nginx (versions 0.8.41 through 1.4.3 and 1.5.x before 1.5.7[15]), certain checks were not performed on unescaped space characters. The space character is an invalid character according to the HTTP protocol[45, 23, 24]. This had been allowed for compatibility reasons. This allowed for certain security restrictions to be bypassed in certain cases. This made it possible to access a directory with permissions set to “deny-all”.

28
location /protected/ {
    deny all;
}

For example, if there exists a directory “folderone ” with a trailing space, a request like the following could be passed which ended up granting access to any file within “protected”, which should never be the case.

GET /folderone /../protected/secret_file.txt

This attack could only be exploited using tools like telnet which send the HTTP request as is without modifying or escaping characters.

3.3.2 How a comparison based could have helped

According to the error report, this flaw has existed since version 0.8.41, which released sometime in 2010. This is quite a while, considering that millions of web-servers through the world use Nginx. This particular vulnerability however may not have been exploitable too many times because of its pre-requisite requirements.

Having a real-time voting system in place could have detected this attack if it was tried. Consider a very simple voting system. This system makes decisions based on simple majority. Let us say this system has 3 web-servers. let one of these servers be Nginx, and that the other two be Apache, and IIS.

![Diagram](image)

Figure 3.1: Voting could be used to detect the Nginx vulnerability and decide on the correct response.

Figure 3.1 shows the structure of a simple voting system that could have been used in this...
case. The input to the system is the attack web request. All web-servers are passed the same request. In this case, only Nginx is vulnerable and it returns a 200 status code (OK) and the response itself. However, the other two servers return a 403 (Forbidden) as they should. Let us say we have a simple majority based voting system which looks only at the status codes. In this case Nginx automatically causes a disagreement, and hence an alert. The only problem with voting in this context is that typically one would like to accept the “correct” answer. Is majority rejection of an input the correct answer, or is the OK the correct answer?

So we can see that a comparison-based (back-to-back testing) based system could have identified this potential zero day vulnerability, which took a lot of time to identify otherwise, and might have gone unnoticed in many cases.

3.4 ColdFusion IIS Attack - (CVE-2013-0629)

In December 2013, there was an attack which utilized a vulnerability in ColdFusion to attack IIS[55]. The attack wasn’t widespread, however it started gaining a lot of popularity among hackers due to the fact that although Adobe released a patch, site administrators were slow to patch their site. This attack falls into the category request exploited native vulnerability.

3.4.1 What Happened?

The attackers used a vulnerability in ColdFusion to install some rogue DLLs on servers running IIS. The malware runs in “stealth mode”. The modified IIS Module silently looks through the POST data of all requests. This can be configured by the attacker to be just one specific set of URLs or more. All examined data is stored or recorded in log files as plain text. Encryption and decryption is not even a worry since the data is retrieved right from IIS itself.

Detecting this kind of attack is hard. This is mainly because there aren’t many symptoms. The attack just causes the server to perform certain actions which are seemingly harmless. For quite a while most anti-virus providers could not detect this module. However over time they were able to identify some characteristics of the module which included signatures of the rogue module.

3.4.2 How they got in?

Trustwave SpiderLabs, the people who caught the attack, also did an analysis of how the attackers got access to the system[57].

Adobe Coldfusion, had a module called RDS, a security add-on to allow administrators to remotely manage databases and HTTP access to files. One of the ColdFusion vulnerabilities allowed hackers to access certain restricted directories as noted in the CVE database[14]. As in
Figure 3.2 The attackers used this technique to pass a file with their own Cold Fusion Markup Language (CFML) which created a webshell back-door. They then used their file to upload their rogue IIS module.

Once the module had been uploaded, the attackers used the CMD feature of the webshell back-door, to activate the malicious module.

The DLL could then be controlled by sending it commands using GET parameters which it was designed to look for.

3.4.3 What did it do?

Once the module was installed, it would listen for commands from the attacker on port 80 in the form of GET requests. During the installation process the attacker can specify the path for the DLL, the log, the name for the log file etc. After this has been done it looks for keywords in the QUERY_STRING of requests that come in[56].

In general, once the list of URLs to listen for data from had been provided by the attacker, it took the plain text version of all post data that came in and inserted all of this into a log file. This means that plain text version of everything from credit-card info all the way to passwords of banking info could be recorded in the log. The attacker would typically let the log file collect as much of this info as he needed. He could then grab a copy of this log file from the server. The module had some keywords which had some basic actions associated with them. It specifically looked for the following keywords:

- isn_getlog  return the contents of the isn.log file
- isn_logdel  delete the isn.log file
- isn_logpath return the path of the isn.log file
The malware module was not widespread. However since its functionality is very dangerous and its detection rate is very low the threat caused by this module is pretty high.

3.4.4 Using back-to-back Comparisons to detect this malware

As mentioned before this kind of malware is hard to detect, and the only Spiderlabs came across it was based on heuristic guesses. However, if we were to use a back-to-back comparison system, this attack would not have gone undetected for such a long period of time. In fact as the attacker sends a request to download the log file, we would have seen that something unusual was going on. In our setup, a request is sent to at least 3 different servers which vote on the responses, one of which would be the compromised IIS server. This utilizes a very simple comparison system which looks at the size of the responses with which we should be able to notice that something was up with the IIS server.

One server, namely the infected IIS server alone, will return a log file of presumably large size which will be much more than the empty response sent by the remaining servers. This can be used to trigger an alert which needs to be looked at by an admin.

3.5 Zencart Redirect - Feb 2014

Another recent attack[61], reported by Sucuri, redirected some (a portion of) users to a malicious site. This particular attack went unnoticed for over 6 to 8 months because of the way it hides itself. This is again a case of an independently injected exploit. Zencart, the software in question had a malicious payload which was inserted in one of its source PHP files.

3.5.1 How did it work

The malware, affected only a portion of the visitors who visited the website. Initially it was deemed as only affecting the “www” users and not the “non-www” users. later it was found that this was not the case. It triggered only on certain User-Agents and Referrers. It is a conditional malware.

When the malware was noticed, several steps were taken to try clean up the system. This included manual inspection of the files and cleaning up of the databases. None of this caught the malware. Upon looking at the requests, it was seen that a USERID field of a cookie was populated upon a new request to the website before redirection occurred, as seen in Figure 3.3. The security research team traced the code back to the headers of the page, and found a payload which was base 64 encoded. This payload caused the redirects to happen. The attack was because the malicious payload was injected into the application (running PHP) rather than attacking the webserver itself.
Here is how the redirect worked.

1. On the first visit the site checks if the user is neither Chinese nor originating from Hong Kong.

2. The user should also come from either of Bing, Google, Facebook, or Yahoo

3. If the conditions were satisfied, the USERID field of the cookie was set as “shine-check” otherwise it was set to “twotime”.

4. On the next visit, the site looks at the cookie, and those with “twotime” are not redirected, while those with “shine-check” are. There is one other conditional eligibility test performed at this point. A user with “shine-check” is redirected with a 40% probability. If the user is not being redirected, the cookie value for USERID is changed to “twotime”. This means that a user who wasn’t redirected the first time will not be redirected again.

This particular malware, is hard to detect and current practices don’t allow for such malware to be caught easily.

### 3.5.2 How back-to-back Testing could have found this

Though our focus is on catching attacks aimed at the webserver. We can catch more than just that. In this case, the last factor determining whether the response should contain a redirect is conditional. Assuming we get a request which satisfies the first two conditions of not being from China and having one of the specific referrers, there is a 40% chance it will return a redirect. This means that if we forward an eligible request to 3 servers, irrespective of whether they have all been compromised or not, pretty soon we will notice a difference in their response just by looking at the status codes of their response.

This provides a compelling case for how back-to-back testing based systems could help identify attacks that are currently going undetected for long periods of time, usually until some one notices that something is off(by chance).
3.6 Apache Tomcat Possible DoS Attack (CVE-2014-0050)

Early in February 2014, an internal email from the Apache Tomcat community was leaked[58]. This email detailed an attack which could take down websites served using certain versions of Apache Tomcat. This issue has since been patched.

3.6.1 Understanding the Vulnerability

![Example of Multipart boundary in form post](image)

Figure 3.4: Example of Multipart boundary in form post (as seen in Chrome debugger)

When binary data or files are uploaded to a server using HTTP, there is boundary data which separates the file and various form fields. This boundary field is randomly generated in such a way that this boundary data is not present within any of data fields themselves[22]. An example of the boundary field in a post is shown in Figure 3.4. Certain versions of Apache Tomcat had an inherent vulnerability, wherein if the boundary data was 4092 characters or longer, it could cause one of the loops in Tomcat to run endlessly. The exit condition for this loop was not strict enough to catch this particular case, and if this case is ever entered, Tomcat
starts using a high amount of CPU resources until the process is terminated. This results in a possibility of a DoS attack on web-servers running Apache Tomcat (versions 7.0-7.0.50 or 8.0-8.0.1) and using Apache Commons FileUpload[58].

3.6.2 Detecting the attack utilizing diversity

Though such an attack can be detected by monitoring CPU usage of a web-server, utilizing a diversity based approach might trigger an alert earlier than usual. If a back-to-back testing system using different versions of Apache Tomcat were to be setup, and a watchdog process was to monitor the different web-server processes, we could very easily notice a higher CPU utilization on the vulnerable version of the server as compared to other versions. We could even use different web-servers (not just different versions of Tomcat) and compare the change in percentage of CPU utilization by each the servers. This would further improve the diversity in the system.

3.7 The Heartbleed Vulnerability - CVE-2014-0160

A vulnerability in OpenSSL was reported on April 7, 2014 by Neel Mehta at Google. This particular vulnerability had a huge amount of repercussions and is thought to have left two-thirds of the servers on the internet vulnerable[6]. Caused due to a simple coding error this particular bug leaves at least half a million servers exposed as of 04-10-2014(3 days after the exploit was disclosed), does not leave a trace of being exploited, and gives out information including private keys, or even user account passwords[5, 51].

3.7.1 Understanding the exploit

In SSL, an extension called heartbeat was introduce to keep a session alive if both parties wanted to keep the session alive even though they currently have no data to exchange. It consists of a payload and a matching response to make sure that the connection is functional in a proper manner. The payload is technically supposed to be limited to 16KB according to RFC 6520[53]. However, the vulnerable implementations had a missing bound check. Since the response is supposed to contain the original payload sent to the server, the server copies this back for the response. If a small payload say 1 byte was sent, but the length of the payload was set to 65535 bytes, the server did not check and copied this data back to the response sent to the client. This means the library copies from memory segments it was not meant to read. It has been shown that this has allowed attackers to get passwords, private key information and researchers are still trying to figure out what else might have been compromised[5].
3.7.2 Diversity in the use of detecting Heartbleed

This is a case of an inherent attack, but a problem with a third party add-on and not directly with the web-server. Diversity could have been used to detect the if the heartbeat vulnerability if it was exploited. All the information exchange takes place along port 443 (or the primary port for communication of a web-server when using HTTPS). This being the case, if a diversity based setup were to inspect all the traffic along this port, and not just basic request and response (not just the GET and POST), this attack would be very simply caught. An improper implementation of SSL would return 64KB of data whereas any other implementation would fail without a response. In this case diversity would have to involve servers not using OpenSSL. Microsoft services utilize SChannel as opposed to openSSL and thus anything using a Microsoft based stack remains unaffected[75].

3.8 What still escapes?

When looking for attacks at the web-server level, anything which involves common vulnerabilities in the extensions and add-on application middle-ware such as the database or the scripting languages etc., will continue to slip through the cracks. Another exploit which will go unnoticed is inherent design flaws when using frameworks. Of course, if the output vector is not chosen appropriately, it may not be sensitive to changes in the behavior and thus will not display an anomaly that is detectable by comparison. However, the aim here is to detect attacks at the web-server core. The following is an example of attacks which cannot be detected using diversity based techniques at the level of the server core comparisons.

```
1.1.1.1 -- [17/Jul/2013:00:00:00 -0000] "GET /mywebsite/index.actionredirect:##7B%23a=(new%20java.lang.ProcessBuilder(new%20java.lang.String%5B%5D%7B%22cat%2C%2F/etc/passwd%22%7D)).start(),%23b=%23a.getInputStream(),%23c=new %20java.io.InputStreamReader(%23b),%23d=new%20java.io.BufferedReader(%23c),%23e=new%20char%5B%5D%5D,%23d.read(%23e),%23f=%%23e%23context.get('com.opensymphony.xwork2.dispatcher.HttpServletResponse'),%23g=%%23f.getWriter().println(%23c),%23h=%%23f.getWriter().flush (),%23i=%%23f.getWriter().close()" HTTP/1.1 302 5
```

Figure 3.5: Remote Code Execution on Struts 2 - https://www.mandiant.com

In 2013, a vulnerability was detected in Apache Struts 2[40], which allowed an attacker to execute code remotely on the server. This was because the wildcard matching of actions was not handled properly. The Apache security bulletin for Struts 2 explains how the vulnerability can be exploited to execute a simple sum of numbers[13]. Mandiant delves into how this attack can be taken further to execute operating system level commands[34]. Figure 3.5 shows one
of the requests passed to the web-server to execute a command on the server. Now the Struts framework is developed by Apache, and is deployed on Java based web-servers like Apache Tomcat, JBoss, Websphere etc. Though we achieve diversity with web-servers, the fault lies in the design of the struts framework, where there are no other equivalent versions developed by other groups. Hence all deployments of the vulnerable versions of Struts 2, irrespective of the web-server used will be compromised when a request in the form an attack is sent as an input. This is one of the scenario’s where diversity will not be of any help.

3.9 Attack Detection Times

A motivation for use of double or triple redundancy in the case of web-servers is the fact that many more sophisticated compromises otherwise remain undiscovered for many months, perhaps years. If that happens on a high-assurance site that manages / delivers sensitive information, damage may be irreparable. The cost of implementing and maintaining redundant system is in those situations far cheaper than the damage control and recovery that needs to happen after the fact.

Table 3.1 shows the time taken to detect the two attack / vulnerability categories we discussed in the previous section a number of relatively recent similar attacks. In case of inherent vulnerabilities the time taken to detect is simply the time that an exploitable version has been out in the open without a patch available. All the vulnerabilities listed can be detected using back-to-back testing or diversity. An investigation of a number of “difficult-to-detect” web-server vulnerabilities and attacks disclosed over the last five years seems to indicate that on an average it takes about 3 and a half years to even detect such attacks. By the time they are found, a lot of harm can potentially be caused. Usually an attack is detected only when it starts to affect a large number of public machines, and its effects are noticed because of extensive exposure.

<table>
<thead>
<tr>
<th>Attack</th>
<th>Type of Anomaly</th>
<th>Time Taken to Detect</th>
<th>Start Date / Detected</th>
<th>Impact (No.of Servers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heartbleed Vulnerability (CVE-2014-0160)</td>
<td>Inherent (Addon)</td>
<td>2 years and 1 months</td>
<td>Mar 2012 / Apr 2014</td>
<td>600,000+ servers[51]</td>
</tr>
<tr>
<td>Apache httpd (CVE-2014-0098) (malformed cookie)</td>
<td>Inherent</td>
<td>8 years and 3 months</td>
<td>Dec 2005 / Mar 2014</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 3.1: Time taken to detect some recent web-server compromises.
<table>
<thead>
<tr>
<th>Attack</th>
<th>Type of Anomaly</th>
<th>Time Taken to Detect</th>
<th>Start Date / Detected</th>
<th>Impact (No.of Servers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zencart Redirect - Feb 2014</td>
<td>Inherent (Addon)</td>
<td>1 year and 8 months</td>
<td>Jun 2012 / Feb 2014</td>
<td>Unknown</td>
</tr>
<tr>
<td>Apache Tomcat DoS Attack (CVE-2014-0050)</td>
<td>Inherent</td>
<td>3 years and 9 months</td>
<td>Jun 2010 / Feb 2014</td>
<td>NA</td>
</tr>
<tr>
<td>Effusion - Modified Nginx Module 2013</td>
<td>Lateral</td>
<td>Possibly 6+ years</td>
<td>Unknown / Dec 2013</td>
<td>Unknown</td>
</tr>
<tr>
<td>Nginx Vulnerability (CVE-2013-4547)</td>
<td>Inherent</td>
<td>3 years and 5 months</td>
<td>Jun 2010 / Nov 2013</td>
<td>NA</td>
</tr>
<tr>
<td>Java.TomDep On Apache Tomcat 2013</td>
<td>Lateral</td>
<td>Unknown</td>
<td>Unknown / Oct 2013</td>
<td>0-49[64]</td>
</tr>
<tr>
<td>Linux/Cdorked.A Attack 2013</td>
<td>Lateral</td>
<td>5 months</td>
<td>Dec 2012 / Apr 2013</td>
<td>20,000 websites[7]</td>
</tr>
<tr>
<td>ColdFusion IIS Attack - (CVE-2013-0629)</td>
<td>Inherent (Addon)</td>
<td>3 years and 3 months</td>
<td>Oct 2009 / Jan 2013</td>
<td>Unknown</td>
</tr>
<tr>
<td>Apache httpd (CVE-2011-3348) (malformed request)</td>
<td>Inherent</td>
<td>2 years and 2 month</td>
<td>Jul 2009 / Sep 2011</td>
<td>NA</td>
</tr>
<tr>
<td>Trojan.Apmod for Apache httpd 2011</td>
<td>Lateral</td>
<td>Unknown</td>
<td>Unknown / Apr 2011</td>
<td>0-49[65]</td>
</tr>
<tr>
<td>IIS Bypass Authentication (CVE-2010-2731)</td>
<td>Inherent</td>
<td>2 years and 5 months</td>
<td>Apr 2008 / Sep 2010</td>
<td>NA</td>
</tr>
<tr>
<td>Apache httpd (CVE-2010-2068) (information disclosure)</td>
<td>Inherent</td>
<td>2 years</td>
<td>Jun 2008 / Jun 2010</td>
<td>NA</td>
</tr>
<tr>
<td>Nginx Windows Source Code (CVE-2010-2263)</td>
<td>Inherent</td>
<td>1 year and 1 month</td>
<td>Apr 2009 / Jun 2010</td>
<td>NA</td>
</tr>
</tbody>
</table>
We see that the number of inherent attacks appears to be much higher than that of lateral
(or indirect) compromises. One possibility is that lateral compromises may not use the main
communication channel (e.g., port 8) for propagating the compromise of further attacks. In that
case, monitoring that channel may not reveal an indirectly injected compromise. However later-
ally injected vulnerabilities appear to pose a bigger threat as they are usually more dangerous.
An attacker takes the trouble to break into a system probably because it is inherently a more
profitable channel of attack. When innate vulnerabilities do not leave a trace when exploited,
(which is frequently the case), they could be exploited for quite some time by an attacker who
discovers them early.
Chapter 4

Engineering

The science behind the methods proposed so far can be used to detect security vulnerabilities with high probability, perhaps even with guarantees. The problem is that a number of issues that we mentioned along the way - correlated failures, inadequate output vector, common-cause faults, etc. need to be addressed before the methodology is ready for field use. An example of voting-based attack detection and mitigation use with web-servers was discussed by Totel et al.[69]. There are areas which haven’t been addressed, and with some work, the algorithm proposed by Totel and team can be modified such that it can work in more practical conditions.

4.1 Modeling the system

The system has to be modeled in such a way that we are able to detect all kinds of attacks including those that are detected using only acceptance testing and watchdog software. Such a system, which uses multiple methods of attack detection will have a higher chance of catching the attack and possibly pro-actively defending against it. Since both acceptance testing and watchdog software are already in use in the field today, it is easier to plug these in as components of a diversity based attack detection system.

4.1.1 The Diversity aspect

The voting based algorithm and system proposed by Totel et al.[69] can deal with static content. They mention that in order to work with dynamic content, such as scripts and databases, their architecture would have to be replicated at each level and each component would be dealt with individually.

In the case of a web application, the property of being dynamic is brought about through add-ons such as databases or web applications containing scripts. A request to web-app could store information, or could retrieve information, or do both simultaneously. In this work we are
focusing on vulnerabilities only at the web-server core level and not anywhere else in the stack, so we do not create diverse components at other levels. The architecture we focus on therefore involves replicating the rest of the stack above the web-server core. Our future work might look at replication of the whole stack.

Figure 4.1: General Architecture with replicated stack

We modify the architecture to look like Figure 4.1, where the diversity and redundancy is only with the web-server. The rest of the stack could be, but need not be the same. For this to work, we have to ensure that when the same input is passed to each server, we should be able to expect the rest of the stack to function the same way on each server. Ideally any difference in output should be caused only by the web-server. This would make the detection of flaws or security breaches in the web-servers easier. Of course, any flaws in the stack above the core may not be detected.
One way to ensure that the stack above the web-server always gives the same output for a given input is to use the same stack in all environments. However this is not necessarily possible. For example as MySQL in windows is not the same as MySQL in Linux. Even a difference in architecture would be a potential problem. We have to work to mask these design differences. The specification for databases is usually so strict that outputs do not vary for a given query between architectures or even operating systems. Similarly scripts like PHP, Perl, etc. usually function fairly consistently across platforms. So we can consider that the stack can be replicated across environments. This alone is not enough to ensure that the rest of the stack above the web-server always produces the same output for a given input. We also have to consider some other factors such as those listed below.

**Randomization** - If the application in question, uses some randomization functions, e.g. `rand()` function in PHP, to perform some actions or to find random numbers, we are not assured that all ‘N’ versions of the stack will produce the same output. This is where the concept of Pseudo Random Number Generators (PRNG) are useful. A PRNG generates random numbers based on an initial seed with which it was initialized. Thus in a simple scenario if we initialize all our PRNGs with the same seed. In a practical scenario, somethings we have to consider includes platform (32 bit vs 64 bit), and operating systems. Usually a 64 bit system can be forced to perform calculations as if it was a 32 bit system, this would solve some of the potential problems we face.

To verify that this would not necessarily be a problem random numbers were generated using python across operating systems and architectures. The random numbers which were generated were similar on the different platforms. Thus we can be sure that if we take precautions to have the same versions of the script environments installed we can deal with the random functions.

**States** - Web based applications have states for all current online users. When a user disconnects from the server or “logs-out” the state if required is written to a database / file or discarded. However while the user is interacting with the system, the state is essentially the session details, which contains information about his / her interaction with the system during this session. Though HTTP and HTTPS on their own are stateless the session info stored voids this property. The reason this is important is that if we are working with ‘N’ web-servers, and at any point in time if one of them goes down, to keep our N-version system up we would have to take some steps. As discussed in the paper on high assurance systems by Vouk[72], we have two options, backward recovery and forward recovery. Backward recovery would mean saving the state of the web-servers as checkpoints and when one of the server fails, going back to the last checkpoint. Forward recovery involves...
transitioing the server to a safe-mode, this means that externally no signs of interrup-

tion are shown, but the server now functions in a degraded mode with lower number of

features.

Masking Design Differences

When multiple versions of a software are designed to do the same thing, they are all designed

according to a specification. There will always be some parts where the specifications fail to

mention what needs to be done. At this point it is usually up the developers to decide what

seems fit and go ahead with something logical. Now in our case we have multiple COTS which

have been developed by various groups. This means that each group of developers would have

implemented this portion of the software in a different manner. To compare the outputs of our

various COTS software, we have to mask the design differences.

A simple example would be the status codes on web-servers. Any URL terminating with a

‘/’ is treated differently on different web-servers. If a directory is requested, a terminating ‘/’
doens’t make a difference in case of the request. However they are handled in a slightly different

manner by the different web-servers.

So before the request is passed on to each web-server, steps have to be taken to mask the
design differences. This might have to be done in two stages, one before the request is passed
to the web-server and there may be also be cases where design masking has to be done after
the response is received from the web-server.

A Simple Detection Algorithm

Algorithm 4.1 uses status codes and response sizes to look for something off. If so an alert is
raised. Here we have set a 5% tolerance on the response size based on the average sizes of the
responses. This algorithm does not focus on mitigation techniques or even at finding the correct
response to a given request but rather at determining if the request was of a malicious nature.
However we have to keep in mind that a system without a mitigation strategy would not be
very effective for use in the field, and some techniques are discussed later on.

Understanding with an example

Let us look at how diversity could be used to detect an attack which is coming in via the web-
server core. We shall consider our Nginx case study to see what would have happened in that
scenario. Let us assume that we have 3 web-servers, Apache, Nginx, and IIS. There was native
fault in some versions of Nginx[15]. Let us assume that we had an affected version. Appendix A
shows a setup where an exploit of this vulnerability is possible. If a request trying to exploit this
vulnerability is sent to a diversity based detection system, it will be caught. Initially the proxy
Algorithm 4.1 Detecting security Vulnerabilities using Voting

**Info:** N is the number of versions, in this case web-servers, R is the set of responses, where $R_i$ denotes the response from the $i^{th}$ web-server for a given Request

**Precondition:** Request Req sent to different servers after masking design differences

1: function DetectSecurityAnomaly
2: if $\exists R_i.StatusCode \neq R_j.StatusCode|(0 < i, j \leq N) \& (i \neq j)$ then
3: Throw Alert
4: Exit
5: else
6: if $R_k.StatusCode \in 2XX|0 < k < N$ then
7: $\text{avgSize} \leftarrow \text{mean}(R_j.ResponseSize)$ where $0 < j < N$
8: if $\exists R_j.ResponseSize \notin \text{avgSize}$ with 5% tolerance then
9: Raise alert
10: end if
11: end if
12: end if
13: end function

would pass this on to all 3 servers, Apache, IIS and Nginx. Only Nginx is vulnerable in this case, and a request for a file from a forbidden directory returns a status code 200 along with the contents of the file. However both Apache and IIS return a status code of 403 forbidden. This throws an alert according to our simple algorithm.

At this point we are able to determine that the request is possibly an attack. However at this point we don’t know what the correct response to the query is, or which of our servers has been compromised.

### 4.1.2 System Overview

This section looks at how acceptance testing, watchdog software as well as diversity can be integrated into one system to detect and mitigate attacks. We have to keep in mind that a layered approach will not work. This might end up with us losing the advantage of either of the detection methods. For example, let us assume that we first perform an acceptance test, and if it fails we do a diversity test. Let us also assume that one of the servers was laterally compromised with the Linux Cdeored malware discussed earlier. Now in this case all requests are legitimate therefore acceptance testing always succeeds and hence never gives way for diversity to detect that the system has been compromised. Thus our system cannot have a layered approach and we would have to perform all kinds of detection all at once.

Figure 4.2 shows a modified architecture which includes both acceptance testing and watchdog software. Initially when the request comes in, it goes directly to a proxy. This proxy is
designed such that it has upstreams to more than one server. We have 3 upstreams, but in a general case there are ‘N’ upstreams to our ‘N’ versions. Each of these versions has a web-server and a corresponding stack on top of it. The stacks essentially perform similarly for all the servers. We assume that the servers in this case are functionally equivalent components. If using COTS type web-servers, we have to take efforts to mask the design differences. We maintain a spare for each of the servers and their stack. The request is also tested by our acceptance testing mechanism. The web-servers process the inputs they receive. Now at this stage we have to vote on the outputs if a response is to be sent back. Voting is done based on the responses from the servers as well as their logs. All the web-servers are monitored using a watchdog environment which includes a logging database[72].

Each of the web-servers processes the input and a voting system is given their “reactions”. The system decides if anything anomalous seems to be going on. This can be done using a very simple majority voting based on just the status codes of the responses from the server, or a slightly more complex method can be developed based on the application of the system. The voting system(called the Service IDS in Figure 4.2) makes a decision on what the “correct” output is. Once this decision has been made the user can be sent the response from one of the servers. Using hot-swappable spares along with a check-pointing mechanism would be a good
way to approach this problem. Each spare is maintained with a lag in the checkpoint. Each time a new checkpoint is created on a server, its corresponding spare is brought up to the previous checkpoint. In case any of the web-servers is compromised, it is replaced with its spare, and the system continues to function as usual. In the meanwhile a new spare is brought up for the server. Using cloud computing means bringing up new instances is relatively cheap and easy. The IDS also gets information from the watchdog environment, and in case of unusual activity on web-server, the decision algorithm could be modified to say give that server’s output less weight-age.

**Real Time detection**

When trying to detect attacks in real time, one of the ways to construct this system would be to have the usual set-up where there is a web-server which responds to client requests. Our entire architecture could be set up on the cloud, since it would be cheaper and easier to provision new web-servers with their stacks in case we need to if one server is compromised. All requests which come to the server is also passed to our setup. One out of N servers needs to be the same version as the one used by the actual web-server which is responding to requests. If our system detects something, a mitigation strategy can be applied. This way the web-server functions without an extra load while all the attacks are still being analyzed for potential attacks as shown in Figure 4.3.

**4.1.3 Mitigation**

Once a potential attack is identified, steps have to be taken to ensure that the system is not compromised, and that it can continue functioning and also detecting attacks. There are various ways this can be done once an attack is identified. As discussed earlier backward recovery could be explored. This approach involves using recovery points to save the state of the servers constantly and repeatedly. In the case that an attack is identified, the compromised server is rolled back functioning can be continued from that point. It is important to note here that, rolling forward with the attacks since the recovery point will end up compromising the server again. These particular request(s) should be avoided till the vulnerability can be patched, or alternative (non-vulnerable) server is used. In case the error was transient, rolling forward from the recovery point would also be an option.

One technique of using diversity to mitigate attacks was proposed by Just and others[26]. It focuses on a method to improve intrusion tolerance using COTS (Components off the Shelf). The authors propose a system which can help prevent intrusion to a system. This system however is deemed as an expensive solution by the authors, and they also state that the system cannot be a general-purpose server connected to the Internet.
The system, termed as HACQIT (Hierarchical Adaptive Control of QoS for Intrusion Tolerance), works by having a primary server and an on-line spare which work together, apart from these two it has a backup which functionally equivalent. The moment the primary is compromised, or the back up is, it is automatically taken off line and and a replacement is brought up in its place. The back up server is always different from the primary, this is to make sure that if a vulnerability was found, the chance it can be exploited again immediately is pretty low. Since a dynamic swap of servers might have to happen the system needs to be able to resynchronize data whenever required.

Diversity is used in the HACQIT system to make the attack of a system repeatedly a hard to achieve system. If when say IIS was attacked, if it was replaced by another IIS server, the attacker needs to just quickly, infect the server again. Assuming the attacker figures out a quick way to repeat this process over and over, could essentially enter the system over and over, and reduce the uptime of the system. However, replacing the primary with a different backup means that the attacker can’t hopefully use the same exploit.
4.2 Costs

Even if a system can detect and prevent 100% of the attacks that comes its way, it would be nonviable solution unless it is affordable. Such a system, which involves redundancy, requires the use of a lot of extra resources. Multiple web-servers, databases, scripting engines etc, compared to having one of each. This means an increased requirement of resources and therefore a higher cost. We can make use of technologies like cloud computing, and make this approach viable both cost wise, as well as implementation wise. In the cloud it is very easy to spin up an instance and bring down an instance as required. Hence in the case we detect an attack, and we know which server has been compromised, the system can be brought back to a safe state and the detection process restarted. This would typically involve bringing up a new VM, rolling forward so that all the web-servers are in sync and in the same state. This process is achieved very easily in the cloud.

The article on Cloud Computing by Ambrust et al.[3], discusses how cloud computing makes resources available cheaper. The recent price cuts by various cloud service providers[20, 67, 11, 68, 38] further goes to show that prices are continually dropping. In fact Google pointed out that the pricing of cloud computing has not followed Moore’s law[20]. They state that though hardware costs fell by 20 to 30% in the last 5 years but the prices of cloud services dropped by only about 8%. Their announcement was followed by both Amazon and Microsoft cutting their prices as well. This definitely works to our advantage, and the cost of our architecture will be much more affordable. The capability of quickly scaling as per our need and releasing resources all easily all while paying a nominal price for the resources is what makes cloud infrastructures an attractive platform for this system.
Chapter 5

Conclusion and Future Work

5.1 Summary

The focus of this thesis on an assessment of the possibility of using diversity and redundancy as method to pro-actively identify security anomalies in software. An analysis of two categories of difficult-to-detect attacks on web-services indicates that the approach could definitely help detect security anomalies. In fact, diversity may help identify attacks which cannot be detected easily at the moment, including quite a few zero-day attacks.

We note that attacks and compromises can be detected only if they exhibit some sort of change along the principal output vector we are observing, e.g. on the main communications channel. Detecting inherent (or software design) vulnerabilities can be done successfully in many cases. One is if the N versions are diverse enough to exhibit differences in their output vectors when an attack or exploit occurs. For example, Nginx had a number of vulnerabilities which were for its windows variant only, and not for any other platform. On the other hand, detecting an attack where there is no scope for diversity, e.g. in the Struts case, is not possible.

A system needs to have a low false positive rate to be cost effective. For example, if an anomaly detection system, identified 30% of the inputs to be an attack when the actual number of attacks is only about 2 or 3%, the detection system in question would not be very useful. In the case of a diversity based approach a system can be constructed to have a low false positive rate by tweaking the tolerance it has for difference in output vectors between the different versions.

The time taken to detect an attack using a diversity based approach will definitely be better than what we have today, which is mostly detection by chance or when stumbles upon a vulnerability. Currently, in the case of inherent vulnerabilities, the attacker may continue to exploit a vulnerability repeatedly till it is some how identified by a system admin/ security researcher. With lateral compromises, even though the time to detection seems shorter than
inherent vulnerabilities, it should be noted that the impact is much higher. Though it is hard to quantify that time for detection improves when using a diversity based mechanism, we can definitely say that an alert to a vulnerability is triggered the moment it is first exploited. i.e if an attacker tries to exploit a system we will know. This should significantly improve attack detection times.

Finally the system can be implemented only if easy access to cheap resources is available and this is where the idea of cloud computing comes in.

5.2 Future Work

This thesis has focused on the detection of security anomalies. Validating the detection methods experimentally and finding data to analyze the effectiveness of using diversity to detect security anomalies would be the next step. Another important thing to work on would be to use diversity to find the correct response and in the case of an attack, mitigation, when detected. Currently we identify the attack, and don’t go much further. The system can be improved such that on detection of an attack the system fails gracefully, and can be brought back to a state where it can continue to function normally and detect attacks.
REFERENCES


APPENDIX
Appendix A

Understanding the Nginx Vulnerability

In order to understand the Nginx vulnerability and test if detection was possible using a diversity based approach, one of the vulnerable versions of Nginx was setup and tested.

The setup had Nginx version 1.5.6, which is vulnerable to the particular attack[15].

A.1 Directory Structure

```
/var/www
|-- ‘625 attack ’
|-- donotallow
|   |-- abcd
|   |-- test.html
|-- hstart
|   |-- simple.xml
|   |-- start.htm
|-- index.html
```
A.2 Permissions on directories

```apache
server {
    listen 90;  ## listen for ipv4;

    root /var/www;
    index index.php index.html index.htm;

    server_name localhost;

    location / {
        try_files $uri $uri/ =404;
    }
    location /donotallow/ {
        deny all;
    }

    location ~ \.(\wp)\$ {
        fastcgi_split_path_info ^\.(.+\wp)(/\.)\$;
        fastcgi_pass 127.0.0.1:9000;
        fastcgi_index index.php;
        include fastcgi_params;
    }
}
```

A.3 Exploiting the vulnerability

Any request for a file within the donotallow directory should return a 403 forbidden. However since the illegal space(' ') character was un-escaped in certain versions it lead to wrong parsing of a request. The following request should return forbidden however it ends up returning the contents of file abcd.

```
GET /625attack /../donotallow/abcd
```

It is important to note that this works only if using a client like telnet or ncat, and not when using a web-browser. This is because browsers never pass the space character as is, and re-encode to a ‘%20’ before the request is sent. if re-encoded it becomes a usual request and is processed correctly.
A command that can be used to retrieve the contents of the abcd file in Linux is:

```
echo -e "GET /625attack ../../../donotallow/abcd" | ncat server_ip port | less
```

Figure A.1: Vulnerability in nginx being exploited