MAINTAINING WEB APPLICATIONS INTEGRITY RUNNING ON RADIUM

Wasi Ur-Rehman

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

August 2015

APPROVED:

Bill P. Buckles, Major Professor
Robert Akl, Committee Member
Song Fu, Committee Member
Barrett Bryant, Chair of the Department of Computer Science and Engineering
Costas Tsatsoulis, Dean of the College of Engineering and Interim Dean of the Toulouse Graduate School
Ur-Rehman, Wasi. *Maintaining Web Applications Integrity running on RADIUM*. Master of Science (Computer Science), August 2015, 39 pp., 2 tables, 13 figures, 16 titled references.

Computer security attacks take place due to the presence of vulnerabilities and bugs in software applications. Bugs and vulnerabilities are the result of weak software architecture and lack of standard software development practices. Despite the fact that software companies are investing millions of dollars in the research and development of software designs security risks are still at large. In some cases software applications are found to carry vulnerabilities for many years before being identified. A recent such example is the popular Heart Bleed Bug in the Open SSL/TSL. In today’s world, where new software application are continuously being developed for a varied community of users; it’s highly unlikely to have software applications running without flaws. Attackers on computer system securities exploit these vulnerabilities and bugs and cause threat to privacy without leaving any trace. The most critical vulnerabilities are those which are related to the integrity of the software applications. Because integrity is directly linked to the credibility of software application and data it contains.

Here I am giving solution of maintaining web applications integrity running on RADIUM by using daikon. Daikon generates invariants, these invariants are used to maintain the integrity of the web application and also check the correct behavior of web application at run time on RADIUM architecture in case of any attack or malware. I used data invariants and program flow invariants in my solution to maintain the integrity of web-application against such attack or malware. I check the behavior of my proposed invariants at run-time using Lib-VMI/Volatility memory introspection tool. This is a novel approach and proof of concept toward maintaining web application integrity on RADIUM.
ACKNOWLEDGEMENTS

The work presented in this thesis was conducted as part of the Master’s program in the Department of Computer Science & Engineering at University of North Texas.

I would like to express the deepest appreciation to Dr. Bill Buckles and the rest of committee members Dr. Song Fu and Dr. Robert Akl. Beside my committee I would like to thank my previous advisor Dr. Mahadevan Gomathisankaran without his supervision and constant help this thesis wouldn’t have been possible and also would like to thank Dr. Krishna Kavi for all his support during my stay in his lab.

Last but not the least, I would like to thank my parents and siblings: who always provided me moral support and said “Education is a key you can open any door of opportunity”.

iii
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS ....................................................................................................................</td>
</tr>
<tr>
<td>LIST OF TABLES .................................................................................................................................</td>
</tr>
<tr>
<td>LIST OF FIGURES ..............................................................................................................................</td>
</tr>
<tr>
<td>LIST OF ACRONYMS .............................................................................................................................</td>
</tr>
<tr>
<td>CHAPTER 1 INTRODUCTION ...................................................................................................................</td>
</tr>
<tr>
<td>1.1 Technology Revolution ................................................................................................................</td>
</tr>
<tr>
<td>1.2 Importance of Information ............................................................................................................</td>
</tr>
<tr>
<td>1.3 Recently Reported Viruses (Worms) ...........................................................................................</td>
</tr>
<tr>
<td>1.4 Trusted Computing .......................................................................................................................</td>
</tr>
<tr>
<td>1.4.1 Trusted Platform Module (TPM) ...............................................................................................</td>
</tr>
<tr>
<td>1.4.2 Root of Trust for Measurement ..............................................................................................</td>
</tr>
<tr>
<td>1.5 Thesis Scope ...............................................................................................................................</td>
</tr>
<tr>
<td>CHAPTER 2 RELATED WORK ..................................................................................................................</td>
</tr>
<tr>
<td>CHAPTER 3 DATA INVARIANTS .............................................................................................................</td>
</tr>
<tr>
<td>3.1 Definition ..................................................................................................................................</td>
</tr>
<tr>
<td>3.2 Types of Invariants .......................................................................................................................</td>
</tr>
<tr>
<td>3.2.1 Loop Invariant .........................................................................................................................</td>
</tr>
<tr>
<td>3.2.2 Class Invariant .........................................................................................................................</td>
</tr>
<tr>
<td>3.2.3 Redundant Invariants ...............................................................................................................</td>
</tr>
<tr>
<td>3.2.4 Data Invariants .........................................................................................................................</td>
</tr>
<tr>
<td>3.2.5 Structural Invariants ...............................................................................................................</td>
</tr>
<tr>
<td>3.2.6 Program Flow Invariants .........................................................................................................</td>
</tr>
<tr>
<td>3.3 Application of Invariants ...........................................................................................................</td>
</tr>
<tr>
<td>3.4 Working of Invariant Generation ...............................................................................................</td>
</tr>
<tr>
<td>CHAPTER 4 OVERVIEW OF RADIUM ARCHITECTURE ........................................................................</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Performance Comparison Analysis between RADIUM and other Trusted Computing System</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Web-integrity Invariants on Open Source Infected Web-Applications</td>
<td>32</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1.</td>
<td>Communication Between Step7 Software and PLC in the Industrial System</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.2.</td>
<td>Step7 and PLC Communication is Infected by the Stuxnet Virus</td>
<td>4</td>
</tr>
<tr>
<td>Figure 1.3.</td>
<td>Trusted Platform Module, with All Its Internal Components</td>
<td>6</td>
</tr>
<tr>
<td>Figure 2.1.</td>
<td>[5] Scoped Invariants Detection Architecture</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2.2.</td>
<td>READS Architecture</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.3.</td>
<td>Security Debugging Process with CBones</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2.4.</td>
<td>ELF (Executable and Linkable Format Program Runtime Memory Layout)</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3.1.</td>
<td>Dynamic Invariant Detection Using Daikon</td>
<td>18</td>
</tr>
<tr>
<td>Figure 4.1.</td>
<td>Schematic View of Radium Architecture</td>
<td>21</td>
</tr>
<tr>
<td>Figure 5.1.</td>
<td>Threat Model</td>
<td>26</td>
</tr>
<tr>
<td>Figure 5.2.</td>
<td>Invariant Generation in Daikon</td>
<td>27</td>
</tr>
<tr>
<td>Figure 5.3.</td>
<td>Workflow of My Proposed Algorithm for Web-application Integrity</td>
<td>28</td>
</tr>
<tr>
<td>Figure 7.1.</td>
<td>Invariant Generation in Daikon</td>
<td>35</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>ARTM</td>
<td>Asynchronous root of trust for measurement</td>
<td></td>
</tr>
<tr>
<td>BIOS</td>
<td>Basic input output systems</td>
<td></td>
</tr>
<tr>
<td>CLB</td>
<td>Check look a side buffer</td>
<td></td>
</tr>
<tr>
<td>DRTM</td>
<td>Dynamic root of trust of measurement</td>
<td></td>
</tr>
<tr>
<td>ELF</td>
<td>Executable and linkable format</td>
<td></td>
</tr>
<tr>
<td>LPC</td>
<td>Low pin count</td>
<td></td>
</tr>
<tr>
<td>PCR</td>
<td>Platform configuration registers</td>
<td></td>
</tr>
<tr>
<td>PLC</td>
<td>Programmable logic controller</td>
<td></td>
</tr>
<tr>
<td>RADIUM</td>
<td>Race-free on demand integrity measurement architecture</td>
<td></td>
</tr>
<tr>
<td>REDAS</td>
<td>Remote attestation to dynamic system properties</td>
<td></td>
</tr>
<tr>
<td>SCADA</td>
<td>Siemens industrial control systems</td>
<td></td>
</tr>
<tr>
<td>SRTM</td>
<td>Static root of trusts of measurement</td>
<td></td>
</tr>
<tr>
<td>TCG</td>
<td>Trusted computing group</td>
<td></td>
</tr>
<tr>
<td>TOCTOU</td>
<td>Time of check to time of use</td>
<td></td>
</tr>
<tr>
<td>TPM</td>
<td>Trusted platform module</td>
<td></td>
</tr>
<tr>
<td>VT</td>
<td>Virtualization technology</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Technology Revolution

The use of information technology has been expanding enormously over the last 20 years. We are witnessing an era where accessing information is easier than before, so many wirelessly connected electronic devices around us replace paper-based information with digital format or e-format information. Because of this technological revolution, we can now access our bank accounts, do shopping, buy tickets, schedule meetings or take online prescriptions, video calling and satellite communication on just one click. Besides getting all these benefits, there is always a concern how this digital information is stored or handled, because information is very sensitive, like credit card information, online banking, administrator password or user privileged and medical information.

According to Forbes [1] computer security attacks are increasing steadily and do not seem to be slowed down by current preventive measures. Akamai Technology [2] reported that attacks on websites went up to 75% in the final quarter 2014. The most important thing to notice about these attacks is that they are rising. According to a report published by Russian Digital Crime [3], computer attacks or cybercrime may cost huge monetary loss to large companies like Sony, Citi Group, Stratford and AT&T. In extreme cases, this monetary loss may go up to billions of dollars. According to 2014 MacAfee report, security attacks/cybercrime cost $400 billion to global economy [4]. Because of the huge amount of financial losses, companies started investing in for the security of their systems.
1.2 Importance of Information

Companies greatly depend on the information of personnel, business and commodities for their day to day operational work and decisions depend on the information. It also includes the user account information (i.e. roles or privilege), cryptographic keys and low-level system information. Therefore, it is of utmost importance to save this information from any unauthorized access, maintaining its integrity and confidentiality. In addition, maintaining the integrity of the components used in the networks; like routers, switches, servers, workstations and USB drives is also extremely important to avoid such attacks.

1.3 Recently Reported Viruses (Worms)

Recently industrial systems have encountered Stuxnet attacks which was propagated by USB devices; Stuxnet is a worm that attacks PLC (programmable logic controller) of the industries. It has been used to target Siemens industrial control systems (SCADA); these systems have Windows operating systems and Step7 industrial application. The main purpose of Stuxnet is to intercept information between software running on Windows machines and the PLC. To date, there is not a single effective remedy of Stuxnet.

Badtrans is another worm recently reported. It is distributed via emails. This worm exploited the vulnerabilities of Microsoft Outlook Express and Microsoft Outlook Program. This worm installs and executes on user machine as soon as the user opens his email to view it. Once the worm is executed, it starts sending messages to other contacts found on the host computer and install key logger. Key logger captures all the information typed by the user in the infected system. Famous email engines infected by Bradtrans are Excite, Yahoo, and IJustGotFired.com.
Figure 1.1 is about the normal operation of Step7 communication with programmable logic controller in the industrial systems. Step7 used in this figure is basic software tools, used for automating the process of the machinery used in the industrial systems.
In Figure 1.2, Stuxnet come in between the communication between Step7 software and the PLC. First installed (STUXNET) in the PLC and then start intercepting the communication between Step7 software and PLC. In this way it affects all the communication between different PLC’s running in the industrial systems.

1.4 Trusted Computing

To make a trusted computerized system (or information system), regardless of its size or the services it provides, we need to make sure that all the components of the computing system are capable of thwarting any outside attack and maintaining the trust of the system. Because every section of the system acts as a layer on the other section (like hardware, which is a lowest layer has BIOS (Basic Input Output Systems), system software and applications running on top of it). If any section of the computing system is compromised the whole system becomes
compromised. Here ‘trust’ of the system means the known good state of the system or the expected behavior of the system, which can never be changed. Any slight change in the behavior of the system means the system has been compromised and is no longer trustworthy. For measuring the trust of a computing system, there is an association of companies called Trusted Computing Group (TCG); they proposed TPM (trusted platform module).

1.4.1 Trusted Platform Module (TPM)

TPM is a chip installed on the motherboard of the system. It has 16 registers called platform configuration registers (PCR). These PCR’s are used to store measurements of the platform in the form of hashes. These PCR values are used to authenticate the platform state by matching with an already installed, known good state. TPM provides attestation service, endorsement service, and storage service. It communicates with the processor by low pin count (LPC) bus. TPM with BIOS together provide a core root of trust. Therefore it is necessary for the BIOS and the operating system to utilize TPM in order to assure platform integrity. Then the user can run their application on the platform.
1.4.2 Root of Trust Measurement

Core root of trust for measurement is the first piece of code or function that is executed at the computer boot time. It tells TPM the next software/firmware that is going to be executed after it. There are two basic approaches for root of trusts.

a. Static root of trusts of measurement (SRTM)

b. Dynamic root of trust of measurement (DRTM)

In this thesis, I used data invariance for maintaining the integrity of the web applications running on the RADIUM computing system. RADIUM proposed ARTM, which is a combination of VT (virtualization technology) and DRTM. Root of trust for RADIUM is hypervisor. One of the
major differences between RADIUM and other trusted computing is RADIUM gives on-demand integrity measurement of the application, which nullifies Time of check to time of use (TOCTOU) attacks which means there is no need to boot and launch the system to check the integrity. RADIUM is much better than the previous solutions where performance is concerned. Applications which can runs on RADIUM are any enterprise application, online banking application or distributed system application etc. According to www.owasp.org; one of the most common attacks on online portal or websites are SQL-injection and Cross-Site scripting. About 60% of application layer attacks are SQL injection.

1.5 Thesis Scope

The next chapter is about the related work, Chapter 3 includes fundamental concepts and a brief description of data invariants. Chapter 4 presents an overview of RADIUM computing systems. Chapter 5 is the threat model. In Chapter 6 I presented a novel solution against web-based attacks on the web-applications running on RADIUM. I used data invariance in my solution. These invariants are generated by Daikon to maintain the integrity of the web-application against these attacks mentioned in the threat-model. The behavior of these invariants monitored at run-time using LIBVMI/volatility plugin. The web-applications run on the target VM (virtual Machine) on the RADIUM architecture. I to date have not come across any such remedy of these attacks. This thesis is the first attempt at using data invariants in RADIUM for maintaining web application level integrity. The more detail explanation of my proposed solution discussed in Chapter 6. Conclusions and scope of future work from this thesis are included as chapter 7 and chapter 8.
Jinpeng Wei, proposed scope invariants for checking the integrity of XEN virtual machine. The values of these invariants depend upon their scope. Like global invariants, scope remains during the entire execution of the program while the heap invariants scopes remain till the allocation and freeing of heap memory. Local variable scope will remain in the function and in the block of the program. The researcher used QUEM tool to detect invariant values at the memory level to check whether the invariant is valid or not. Mathematical representation of scope invariants is:

\[ V(t) = k, t \in [s_1, s_2) \]

Value of the variable ‘v’ must be a specific value of ‘k’: where ‘s’ is the state of the system. A drawback of this approach is the limitation of dynamic profiling. Because some non-invariants may come into the invariants list and overcome the profiling would need to be run multiple times by setting different inputs. This will increase processing time and cost of detecting scope invariants.

Figure 2.1. [5] Scoped Invariants Detection Architecture

Chongkyung Kill gave a novel idea of Remote Attestation to Dynamic System Properties (REDAS) for integrity evidence. This evidence is then provided to other remote systems, especially
in the distributed environment. If any of the integrity evidences are modified or corrupted at runtime, that system will be separated from the distributed environment. The attestation process includes three steps: dynamic property collector, integrity measurement, and attestation service. Dynamic property collector extracts structural constraint from the binary file of the application. These properties are then used by the integrity measurement to monitor application at runtime on every system call. The drawback of this approach is that the integrity measurement by system call may not be suitable for heap data structure. Another drawback is that this approach only takes into account two dynamic properties and ignores the others. Lastly, dynamic attestation needs to access large numbers of dynamic objects repeatedly (because they are transient in nature), which consumes lots of system resources.

![Figure 2.2. READS Architecture](image)

[6]First part of Figure 2.2 shows the dynamic property collector extracting dynamic system properties from the program source code. These properties are sent to the integrity measurement component in the second part of Figure 2.2, to check the integrity violations. The integrity measurement component provides the integrity evidence to the attestation service. Attestation service response includes the nonce received from the challenger and the signature of TPM on PCR 8 register.
Chongkyung Kil proposed CBones which exploits the unknown vulnerabilities using the program structural constraints or program invariants without the instrumentation of source code (because they instrument program binaries) and with no additional hardware support. Program structural constraints are received from program binaries using static analysis of the compiled program executables. These invariants or structural constraints are verified during program execution against the structural properties of the memory objects using a dynamic monitoring agent. Performance and accuracy of CBones is quite impressive, with no false positives when tested against 12 different kinds of applications. CBones provides information as to how a security bug can be exploited.

![Figure 2.3. Security Debugging Process with CBones](image)

[6] In the above figure, constraints extractor takes program binary as input, extracts structural constraints from the program binary, and sends them to the monitoring agent with the .exe. The monitoring agent checks any structural constraint violations. If found the execution of the program will stop with error message and instruction responsible for the violations.
Figure 2.4. ELF (Executable and Linkable Format Program Runtime Memory Layout)

As shown in figure 2.4 code, [7] data heap and stack have fixed order in the memory layout. Stack has higher addresses than heap data structure and heap data structure has higher addresses than code and data segments. A limitation of CBones is that it only uses applications which are developed in C language. Modern day programming languages are not supported by CBones.

Sudheendra Hangal proposed Diduce. This dynamic invariant detector identifies the main cause of vulnerabilities by instrumenting and analyzing the application behavior at runtime. Diduce has a flexible invariant hypothesis, which means it is found in any condition which does not come under current invariant hypothesis. The main purpose of Diduce is to dynamically detect and check data invariants to help programmers debug major vulnerabilities. Diduce has two modes of operations: training and checking. Invariants in Diduce cover the definition of all programming points/expressions (e.g. Object, static variables, input parameters, and return values).

Phin Zou, proposed statistics based method called PC-based invariant detection. In which memory location is accessed via programming instructions. If we capture the set of invariants of these instructions which normally access a given memory location(variable), we
can detect the other unwanted suspicious instructions, which can cause different attacks like memory corruption, buffer overflow, stack smashing, or other memory-related bugs. The second contribution is the check look a side buffer (CLB), which reduce the monitoring overhead for software debugging in iwatcher framework. Then, he demonstrated the PC-based invariant detection approach using the Acc-Mon debugging tool. This tool had higher success rate and low overhead when compared to other invariant detection tool.

Tim Harris represented an idea using extended version of STM Haskell for introducing invariants in applications for atomic transactions. They propose expression (E) must be checked for invariant violation. For example, a list is supposed to be in order; but if any of its atomic values are unordered, Check E raises the invariant violation flag.

\[
\text{check:: STM a --> STM() ------ (i)}
\]

Equation (i) tests invariants and add them to the pool of invariants using STM computation. User transaction only commit if all the invariants from the pool are satisfied. In case of fail invariants, user transaction will roll back. The paper discussed four different types of invariants examples: range-limited Tvar, sorted list, invariant specific to a state, and ‘if’ invariant blocks. The crux of this work is that it extends the atomic transaction using invariants over a changing state and sees the dependency between the transaction and the state.

Arati Baliga, discussed the detection of rootkit malware at kernel level. The approach used for rootkit detection malware is divided into two steps: training phase and enforcement phase. Training phase creates invariants for kernel data structure, which covers both control and non-control data structure. Enforcement phase uses these invariants to check the integrity of the kernel against rootkit attack. Violation of these invariants indicates the presence of rootkit attack.
This method is implemented using Gibraltar on two separate machines. One machine acts as an observer, and another act as a target. These machines are connected through PCI card. This approach wouldn't work with our Radium architecture, because this approach has been implemented on two physically separated machines.

Christopher Kruegel used a static analysis technique at load time to detect rootkit attack in loadable kernel modules. Static analysis uses binaries of the loadable kernel modules. The first step is to check whether the module of the kernel is malicious, before load time, by using static analysis on the binary of the module. Whether the static analysis of these kernel modules declare them as being under rootkit attack depends upon two specifications/conditions: Kernel module written in illegal memory area or the use of forbidden kernel symbol reference to calculate address in kernel address space when writing operation. The Rootkit attack discussed in this paper targeted a critical data structure like system call table and kernel modules at load time.

Neelam Gupta, discussed the previous approaches for detecting invariants have issues. Randomly generated test suites were unable to cover all program instructions (poor coverage), and the grammar generated used a black box approach, which is very limited in terms of covering the implementation. So, the paper proposed augmentation of the test suites approach to increase the accuracy of detecting program invariants. Accuracy of dynamically discovering program invariants is directly proportional to the quality of test cases used. Therefore, the dynamic approach of detecting invariants is highly dependent on the quality of test cases. Gupta proposed two approaches to increase the accuracy of program invariants using existing test cases. First, before running the dynamic analysis of a program, the values of the invariants should
be known. Second, the dynamic analysis should be run to get the invariant value. These values are then used to augment the value of test cases.
CHAPTER 3
DATA INVARIENTS

3.1 Definition

An invariant is an attribute of a program that remains constant at certain points. The points in a program may include functions exit and entry points, return values, or any constant value. An invariant is a variable if its value can never be changed or remains constant for a certain period of time during program execution. However, a variable cannot be an invariant if its value changes more than once. In computer science an invariant is a property of a program that should be true during the life time of the program or during a certain point of execution.

3.2 Types of Invariants

i. Loop invariants
ii. Class invariants
iii. Redundant invariants
iv. Data invariants
v. Structural invariants
vi. Program flow invariants

3.2.1 Loop Invariant

Loop is a programming language statement. It’s a property that needs to be true before and after each loop iteration.
Example: Taken from (R1) reference, the invariant for this small code snippet is (i != n). At the start of every iteration, the values of ‘i’ not equal to ‘n’ need to be checked.

```cpp
while (i != n) {
    if (m < a[i])
        m = a[i];
    ++i;
}
return m;
```

3.2.2 Class Invariant

A class invariant remains valid to all instances of the class. A defined relationship between different attributes of the class, which remains valid before and after the execution of any method in the class (a combination of pre and post conditions).

3.2.3 Redundant Invariants

X is an invariant, with two relationships (4<x<7 and x!=0). Both these invariant statements refer to the same assertion. Therefore it’s a redundant invariant. These invariants are expensive in terms of memory usage and processing time, so they affect the efficiency of the application on large scale.

3.2.4 Data Invariants

This type of invariant is also called value-based invariant. It can be defined as properties/values of individual variables or a relation among a group of variables. Examples of these invariants are loop, equality, and inequality conditions.
3.2.5 Structural Invariants

These invariants are related to memory addresses. They include stack frame pointer which does not change during execution, and return addresses of a method. Heap and stack are should be in boundary range of memory addresses.

3.2.6 Program Flow Invariants

These invariants validate the flow of the program according to its requirements. For example, some applications have a certain order of program execution (execution order of methods).

3.3 Application of Invariants

Invariants are helpful in understanding program. They prevent the introduction of bugs into the program. They also verify the authenticity of the program by detecting vulnerabilities. Invariants can be helpful in writing efficient algorithms, (because algorithms have lots of relations among variables and equation.)
Example: Taken from (R1) reference, the invariant for this small code snippet is \(i \neq n\). At the start of every iteration, the values of ‘i’ not equal to ‘n’ need to be checked.

\[
\text{while (} i \neq n \text{) \{}
\text{if (} m < a[i] \text{)}
\text{m = a[i];}
++i;
\}
\text{return m;}
\]

3.2.2 Class Invariant

A class invariant remains valid to all instances of the class. A defined relationship between different attributes of the class, which remains valid before and after the execution of any method in the class (a combination of pre and post conditions).

3.2.3 Redundant Invariants

X is an invariant, with two relationships \(4 < x < 7 \text{ and } x \neq 0\). Both these invariant statements refer to the same assertion. Therefore it’s a redundant invariant. These invariants are expensive in terms of memory usage and processing time, so they affect the efficiency of the application on large scale.

3.2.4 Data Invariants

This type of invariant is also called value-based invariant. It can be defined as properties/values of individual variables or a relation among a group of variables. Examples of these invariants are loop, equality, and inequality conditions.
3.2.5 Structural Invariants

These invariants are related to memory addresses. They include stack frame pointer which does not change during execution, and return addresses of a method. Heap and stack are should be in boundary range of memory addresses.

3.2.6 Program Flow Invariants

These invariants validate the flow of the program according to its requirements. For example, some applications have a certain order of program execution (execution order of methods).

3.3 Application of Invariants

Invariants are helpful in understanding program. They prevent the introduction of bugs into the program. They also verify the authenticity of the program by detecting vulnerabilities. Invariants can be helpful in writing efficient algorithms, (because algorithms have lots of relations among variables and equation.)
3.4 Working of Invariant Generation

First, source code of the program is instrumented by the front end tool (kvasir in C/C++, and Chicory in java). These front end tools instrument the program to detect variables of interest over a period of time, then run this instrumented program with the set of test cases available in daikon. Then it checks variables for invariants by analyzing the behavior of these variables over a certain period of time. The method described here is for Daikon tool, which uses machine learning algorithm.
CHAPTER 4

OVERVIEW OF RADIUM ARCHITECTURE

This chapter covers the overview of radium architecture, which was developed by the members of the TSSL lab of University of North Texas, under the direct supervision of Dr. Mahadevan Gomathisankaran. I proposed a solution for maintaining the web-application integrity on RADIUM using data-invariance against SQL-injection and Cross Site scripting attacks. Detail description provided at the end of this chapter.

Before discussing the radium architecture, it is important to understand the concept of trusted computing. Trust is defined as a “known good state of the computing system (behaves as expected)”. Therefore, trusted computing is depends upon the trust of the computing system. To verify the trust of a computing system, a current state of that computing system is compared with its already existing/stored state. If both states of the systems match, the computing system is said to be trusted. A simple approach to check the trust of a computing system is to measure the state of each component of the computing system and match it to the already stored state of these components; this establishes a chain of trusts among all these components for any computing system. There are two existing approaches to measure the chain of trusts: SRTM and DRTM. Static Root of Trusts Measurement (SRTM) checks the trust of all components of the system at the boot time in sequence. Dynamic Root of Trusts Measurement (DRTM) can measure the components any time during system execution. The common assumption of these two approaches is that system components do not change after they are measured. RADIUM proposed Asynchronous Root of Trust for Measurement (ARTM) which is a combination of VT (virtualization Technology) and Dynamic Root of Trust measurement (DRTM). However ARTM
doesn’t suspend other program’s execution, as DRTM does. Instead it uses hypervisor, which provides hardware isolation. Therefore, root of trust for RADIUS is hypervisor. One of the major differences between RADIUS and other trusted computing solutions is; RADIUS gives on-demand integrity measurement of the applications which nullifies TOCTOU attack. In addition, there is no need to reboot or re-launch the system to check the integrity of the system.

There are four different areas where we can compare RADIUS with traditional trusted computing system in terms of performance. In a traditional trusted computing system, we have four parameters of performance analysis: 1) Measurement Time of Target VM, 2) Launch Time of Target VM, 3) PCR extended operation and 4) Quote operation. In Radium, VM measurement is done by measurement service, and there is no need to restart the VM. Still it's mandatory to check the state of measurement service before the measurement of VM. So the four areas of performance analysis in RADIUS are: 1) Time consumed in the measurement of measuring service, 2) Time consumed in launching the measuring service of the VM, 3) Measurement time of target VM and 4) PCR extended operation and quote operation. According to the experiment performance wise RADIUS is much better than other trusted computing systems. As displayed in table1. Radium takes less time to measure and launch the target VM compared to other trusted computing systems.
Table 4.1. Performance Comparison Analysis Between Radium and Other Trusted Computing System

<table>
<thead>
<tr>
<th>Traditional Trusted Computing System</th>
<th>Time(Seconds)</th>
<th>RADIUM</th>
<th>Time(Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurement Time Of Target VM</td>
<td>26.7</td>
<td>Measurement Time of Target VM.</td>
<td>1.7</td>
</tr>
<tr>
<td>Launch Time Of Target VM</td>
<td>35.8</td>
<td>Time consumed in launching the measuring service of the VM</td>
<td>11.1</td>
</tr>
<tr>
<td>PCR extended Operation</td>
<td></td>
<td>PCR extended Operation</td>
<td></td>
</tr>
<tr>
<td>Quote Operation</td>
<td></td>
<td>Quote operation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time Consumed by Radium in Measuring Measurement Service</td>
<td>27.3</td>
</tr>
<tr>
<td>Total Time</td>
<td>62.5</td>
<td>Total Time</td>
<td>40.1</td>
</tr>
</tbody>
</table>

Schematic in figure 4.1 shows the working of Radium. First, the hypervisor is launched using DRTM at system boot. The current state of hypervisor is then matched with the known good state of hypervisor, which is saved in the Trusted Platform Module (TPM). Later, measuring
service registers itself with the hypervisor and provides all rules to access the control policy module. These rules are used by the measuring service when it interacts with other environments. In addition, hypervisor stores the state of the measuring service in the TPM with cryptographic key for future verification. Before the execution of any virtual machine on hypervisor, it needs to register itself with hypervisor. The registration process includes access control policy rules and the (Universal Unique Identifier) UUID service of the measuring service. After registration of VM, it can instantiate. Hyper calls (hyper visor call like system calls in the operating system) are used by the measuring service to interact with the VM. The access control policy module checks that this particular measuring service has rights to access that particular VM. After the verification of the permission rights of measuring service. The module is allowed to access the memory pages of that particular VM and reports the current state of the VM to hypervisor, which is then stored, extending with already store value in the TPM.

The purpose of my research is to maintain the integrity of the web-applications running on RADIUM architecture against web-based attacks and malware. To accomplish this I developed a module, this module run on the target VM in the RADIUM architecture. First I proposed the threat model in chapter 5; my threat model covered 60%-70% of current web-based attacks happening these days. Then I proposed a novel solution for this threat model, using data invariance generated from daikon tool to maintain the integrity of the web-application at run time against these attacks. Daikon tool generated multiple invariants; I used only those invariants which suited my solution for maintaining web-application integrity. The criteria of selecting these invariants generated from daikon are described in detail in chapter 6. I also performed memory
analysis of these data invariance during runtime using LIBVMI/volatility to check their behavior against attacks or any malware.
CHAPTER 5
THREAT MODEL

Computer security attacks occur because of the presence of vulnerabilities and bugs in software applications. Bugs and vulnerabilities are the result of weak software architecture and lack of standard software development practices. Despite the fact that software companies are investing millions of dollars in the research and development of software designs security risks are still at large. The most common of these vulnerabilities are present in web applications, smart phone applications and also in operating systems. Attackers on computer system securities exploit these vulnerabilities and bugs and cause threat to privacy without leaving any trace. The most critical vulnerabilities are those which are related to the integrity of the software applications. Because integrity is directly linked to the credibility of software application and data it contains. Data is a sensitive thing it can be a credit card information, patient information, cryptographic key information, user password or access roles.

The objective of this threat model is an attacker by himself originating attacks or originating them with help of any malicious virus or worm use to steal information from the web-application or disrupt the normal flow of the application running on RADIUM server. I want to make sure that web-application running on RADIUM server can maintain their integrity against SQL-injection and cross-site scripting attacks. According to www.owasp.org, about 60% of web-application attacks are due to SQL-injection and cross site scripting. Here I draw the threat model for web-applications running on RADIUM target VM (Virtual Machine). The threat model covers above mentioned attacks. The applications running on RADUIIM would be any banking application, e-commerce application and medical application. The main
purpose of these attacks is to steal or update the database information in the database server and inject the malicious code in the web-application. The malicious code includes sequel queries, binary operators, comparison operator, special characters, client side script like html tags and java-script. Beside these attacks and viruses, there is also a possibility of software vulnerabilities like use of dynamic queries in place of parametrize queries. These malicious code not only update or steal the information in the database but also reveal the information to other non-authorize users by printing the sensitive database information on the webpages. Other aspect of these attacks is that, they disrupt the flow of the program by injecting html code in between the methods of the application. Suppose if user finishes his request by clicking on the event (button), these attacks take the user to their malicious page and ask him to enter his information. This way attacker get the information and also disrupt the flow of the application by injecting his code. In my threat model I consider all aspects of these attacks.
1) Inject Malicious Code in the application (‘1=1’, ‘--’, ‘”, script injected.
2) Exploit vulnerabilities in the application architecture. Dynamic queries generation with string concatenation.
3) No input validation.
4) Reveal sensitive information from database.
5) Update the information database.

Figure 5.1. Threat Model
CHAPTER 6
WEB APPLICATION INTEGRITY SOLUTION ON RADIUM

Here I am proposing a solution to thwart these attacks (sql injection & cross site scripting), using daikon invariant detector to maintain the run-time integrity of web-applications running on RADIUM, I called it web-integrity solution. Daikon introduced the invariants to make sure applications wouldn’t compromise against these web-attacks or malware.

![Diagram of invariant generation in Daikon](image)

*Figure 6.1. Invariant Generation in Daikon*

In figure 6.1, [13] observes the execution of the program and check the values which remains constant. We run the source code with instrumentation, which is Kvasir/Chicory. Kvasir/Chicory instrumentation of the source programs traces the variables of interest, running the instrumented program over a set of test cases, and then postulating and checking invariants over both the instrumented variables derived and the variables not manifest in the program. Daikon detects invariants at specific program points such as loop heads and procedure entries and exits; each program point is treated independently.
6.1 Working of My Proposed Algorithm

In the schematic of Figure 6.1, I described the flow of my proposed algorithm for web-application integrity on RADIUM. This proposed algorithm thwarts web-based attacks and maintains the integrity of web-application against attacks or malware on RADIUM. I used
invariants named as security invariants. These invariants are generated by daikon tool. Besides
invariants, I used buffer of black list characters (or malicious code, which include all special
characters and Boolean characters). Whenever users or attesters send requests, the algorithm
compare the input with the character in its black list of array. There is an invariant value depend
upon the matching of the character from the user with the already defined array of black list
characters. Then the value of this invariant variable is also check at the memory level using
volatility plugin. Second is the monitoring module, this module checks two condition. First the
invariant value it gets from the matching of black list of characters, second the other invariant
value which associates with every method call. This invariant value use to check the order of the
program. The order of the method mostly disturb in the presence of attacks and malware or virus.
Then verify this value of the invariant during the memory analysis at run time using Lib/VMI and
Volatility. RADIUM launches the measurement service from other VM, on top of that VM Lib/VMI
and Volatility is running to perform memory analysis of these invariants. The purpose of this
memory analysis is to check that invariant values remain constant during our proposed algorithm
execution or change in case of any attack or malware. After verifying all invariants in my proposed
web-integrity solution, application access is allowed to the requester. In case of any invariant
violation the threat detected and request of access will be denied.

6.2 Selection of Invariants

Daikon generates multiple invariants for the programs or applications. After running my
proposed algorithm many times with invariants and observing the behavior of these invariants, I
selected few of them which suit my solution. I used invariants in my solution which maintained
the flow of the program (order of the methods call) and the one which checked the presence of special characters in the user request. I excluded the invariants which are redundant in nature because redundant invariants increase the overhead of the computing system.

6.3 Excerpts of Invariants Generated By Daikon

Following is the excerpts of invariants generated by daikon. I have mentioned the selected invariants here, which suits my solution. Program flow invariant `dec-type int` with every method call. This invariant checks the flow of my proposed algorithm.

```plaintext
input-language Javadecl-version 2.0
 var-comparability
 ppt..main():::ENTER
 ppt-typeenter
 dec-type int
 ppt..main():::EXIT0
 ppt-typesubexit
 variable return
 var-kind variable
 rep-type int

dec-type int
 pptunsafeRecordRetrieve():::ENTER
 ppt-typeenter
 array_blacklist[20]

 variable conn
 dec-type int
 ppt ..unsafeRecordRetrieve():::EXIT0

 ppt ..safeRecordRetrieve():::ENTER
 ppt-type enter
 variable conn
 array_blacklist[20]
```
6.4 Memory Analysis on My Proposed Algorithm on RADIUM

The web-integrity module runs on target VM, and the measuring service running on DOM0 virtual machine. Measuring service performed on-demand measurement using Lib-VMI/volatility. Lib-VMI extracted the dumb ram memory and volatility performed the detail memory analysis during execution of web-application integrity module. In this analysis, I need to verify no malware bypassed my invariants by checking their values at runtime. In the meantime, I also calculated the overhead of invariants on the RADIUM system, which is negligible. Then I checked the robustness of my proposed algorithm against other open source infected web-application, the results are quite encouraging. I will discuss this in next section.

6.5 Results of Web-integrity Invariants on Open Source Infected Web-Applications

I got a few open source infected applications from www.exploitdb.com. These applications were infected by sqlinjection attacks and cross-site scripting attacks. I applied my proposed integrity solution on these open-source applications by introducing the invariants, which I used. I then ran these applications on RADIUM architecture and tried to perform those same attacks again and also perform the memory analysis of these applications at run-time using Lib-VMI/Volatility, to make sure that malware did not bypass the invariants. The results are very encouraging and have been mentioned in the below table. These results show how robust my proposed solution is for maintaining the web-integrity solution on RADIUM.
Table 6.1. Web-integrity Invariants on Open Source Infected Web-Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Vulnerability</th>
<th>CVE-ID</th>
<th>Invariants Introduced</th>
<th>Attacks Detected (False Positive)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library System</td>
<td>SQLI</td>
<td>2015-3314</td>
<td>Data Invariants</td>
<td>100% (0%)</td>
<td>70%</td>
</tr>
<tr>
<td>WP-Community</td>
<td>SQLI</td>
<td>2015-3313</td>
<td>Data Invariants</td>
<td>100% (0%)</td>
<td>90%</td>
</tr>
<tr>
<td>Add-Manager</td>
<td>SQLI</td>
<td>2015-2824</td>
<td>Data Invariants</td>
<td>100% (0%)</td>
<td>110%-80%</td>
</tr>
<tr>
<td>Classified</td>
<td>XSITE-SCRIPTING</td>
<td>2014-10013</td>
<td>Data Invariants</td>
<td>100% (0%)</td>
<td>70%</td>
</tr>
<tr>
<td>OTRS</td>
<td>XSITE-SCRIPTING</td>
<td>2014-1695</td>
<td>Data Invariants</td>
<td>100% (0%)</td>
<td>80%</td>
</tr>
</tbody>
</table>

The results I observe while testing the application is quite encouraging, I run my proposed module on target VM which has malware, the invariants detected the presence of the malware and also their value remain constant which was verified by the volatility plugin at runtime. Second I run all the available combinations of malicious code on these applications, and it detected all the attempts, also during all these attempts the values of the invariants again remain constant. Again this was verified by the volatility plugin at run-time. Besides running all these cases I also calculated the over-head of verifying the invariants at run-time, which is quite negligible compare to other trusted computing system. Now on the basis of the results in table 6.1 I can claim the proposed solution (module) for maintain the web-application integrity on RADIUM architecture has 0% false positive.
CHAPTER 7

CONCLUSION

In Chapter 1 I discussed the growing number of security attacks and the financial damages they cause to the large companies like Citi, AT&T and Stratford. Then highlighted the reason why security attacks do happen or how it exploited by the attacker. Then discuss few famous security attacks. Chapter 2 gives the literature survey about the usage of data-invariance in different kinds of distributed and cloud computing systems. Chapter 3 gives the brief description of invariants and its types. In Chapter 4, I had given the overview of RADIUM architecture, in the end of the chapter I given the introduction of my work/contribution. In Chapter 5, I presented my threat model and its brief description. In this Chapter I provided two ways by which integrity of the web-application system can compromise. One by the attacker run the malicious code or second due to malware/virus. Chapter 6 has the detail description of my contribution, which is the extension of RADIUM architecture; I provided the proof of concept for web application integrity during run-time on RADIUM. I used DATA invariants in my proposed solution/module for measuring the integrity of web-application. These invariants are generated by Daikon. Then perform the verification of these invariants value during run-time against attack or malware. The verification is done by LIBVMI/Volatility. Then I tested my propose solution to other open-source infected applications available on www.exploit-db, which are listed in table 6.1. The results are very encouraging because my propose solution of web-application integrity thwart all the attacks, and the value of my invariants remain constant.
CHAPTER 8

HIGH LEVEL VIEW OF FUTURE WORK

My future work checks the integrity of the application on RADIUM architecture using “signature-invariants”. Signature-invariant means the combination of different type of invariants use to check the integrity of the application at runtime during loading of the application. Here I am giving the example of apache webserver, in apache the web-pages are store in web-container. Servlets are used to deliver the pages at the client side. Each servlet request has its own separate life cycle to fulfill the request of the client. Life cycle of servlet consist of the three main methods, which are init(), service() and destroy(). All these methods are load in the same order; here the order of execution of these methods is our invariants (program flow invariant). Second invariant is the binary file of the web application. So our signature invariants are the combination of order of the execution of servlet life cycle methods (program flow invariant) and the binary of the web-application. The mathematical representation of signature invariants is.

8.1 Mathematical Representation of Signature Invariants

\[ \text{Signature Invariants} = \text{Flow of the methods of servlet life cycle (init()+service()+destroy())} \]

\[ \text{n} \quad \text{Binary of the web application} \]
Figure 8.1.
REFERENCES


7- Chongkyung Kil, Emre Can SezerPeng Ningand Xiaolan Zhang "Automated Security Debugging Using Program Structural Constraints", Department of Computer Science North Carolina State University, IBM T. J. Watson Research Center, U.S. Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316 and by the National Science Foundation under Grant No. CAREER-0447761.


13- Neelam Gupta, "Generating Test Data for Dynamically Discovering Likely Program Invariants", Department of Computer Science University of Arizona.

