2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)

CoRuM: Collaborative Runtime Monitor Framework for Application Security

Saikat Das, Sajjan Shiva
Department of Computer Science
The University of Memphis
Memphis, USA-38152
{sdas1, sshiva} @memphis.edu

***Abstract*—We propose a framework based on Collaborative Runtime Monitors (CoRuM) for application-level security. CoRuM detects the abnormal behavior of an application by observing critical characteristics during program runtime. In this paper, we discuss the application's critical and essential characteristics to be monitored, the components of the framework, and its workflow on different use case scenarios. We provide experimental results on typical cyber-attacks and provide the throughput and detection accuracy measures. We also propose multidimensional preventive measures using honeypot and backup servers.**

Keywords—CoRuM; Collaborative Intrusion Detection System; Intrusion Detection System; Runtime Verification; Runtime Monitor

I. INTRODUCTION

Ensuring the security of modern applications is a challenging task. Although security experts have provided the mechanisms for the CIA security triad (confidentiality, integrity, and availability) over the last two decades, they have often failed due to the lack of knowledge of new attacks or unexpected vulnerabilities that occur at runtime. An Intrusion Detection System (IDS) is a software or hardware implementation that monitors a system (software, hardware, OS, network, etc.) for malicious activities and policy violations. IDSs can be classified based on their detection techniques, functionality, deployment approach, working mode (active/passive), placement, etc. Most IDSs have a security information database, an event management system (SIEM), and an administrator [1]. SIEM uses an alarm filtering technique that distinguishes a malicious activity or policy violation and reports to the administrator. The administrator on receiving the alarm, activates it's built-in action plan to prevent malicious activity and corresponding damages. However, detecting the system abnormality in real time is a challenging task due to unexpected system behavior.

A Runtime Monitor is a program designed to monitor another program's behavior during execution time. The robustness of the Runtime Monitor depends on its capability to monitor the application's crucial and vital characteristics.

A collaboration of Monitors can help to detect the multidimensional attacks. For a large-scale system, the robustness of system security and detection mechanism can be achieved by partitioning the system into smaller modules and monitoring them in a collaborative way. In this paper, we discuss the typical critical and essential characteristics of applications that are monitored. Then we propose a Collaborative Runtime Monitor (CoRuM) framework that aids the application security. We also provide experimental results with various attacks.

The rest of the paper is organized as follows. The related work is briefly discussed in Section II. Section III summarizes the concept of Runtime Monitor and discusses the typical crucial and critical characteristics of applications to be monitored in detail. Section IV provides the details of CoRuM framework. The experimental results and case studies are discussed in Section V. Section VI concludes the paper and provides the direction for future research.

II. RELATED WORK

Over the last few years, significant amount of research has been done and various types of runtime monitors have been proposed for different purposes.

Dharam et al. [15] identified legal execution paths of the application at the pre-deployment testing phase to feed those specifications onto their runtime framework. Their method can detect and prevent the SQLI attack by verifying the specification with the application's runtime behavior. The monitor can halt the program's execution in case of any unexpected program behavior. However, their proposed method can only handle a specific pattern of SQLI attack.

Bas et al. [8] proposed TesterLink, which uses collaboration of local monitors. The purpose of the collaboration was to verify system properties with a local view generated by each local monitor. They used Linear-time Temporal Logic (LTL) to specify and verify system properties. However, their proposed model has no specification of how they detected the anomalous activity and no precaution was taken for the prevention of attacks.

In Service-Oriented Architectures, a lot of research has been done in monitoring web services with an Active Service Broker. To enhance the dependability of web service related software, Bai, et al. [9] proposed a collaborative runtime monitoring approach based on their earlier work [12], where an Active Service Broker acts as both active and passive service provider. Sensors were instrumented into three different levels and can remotely communicate among themselves. However, their proposed architecture lacks identifying the specification of verified patterns and policies.

An efficient manager-based collaborative intrusion detection system (CIDS) was proposed by Wu et al. [10], where the manager is used to gather alarms from different IDSs. Graph and Bayesian's network-based decision maker are embedded in the

978-1-7281-0359-4/18/$31.00 ©2018 IEEE
DOI 10.1109/UCC-Companion.2018.00057

CIDS which facilitated the decision. However, their approach was not able to classify the different types of rule-based attack patterns and no preventive action was taken.

Trace matches were used in another CIDS proposed by Bodden, et al. [11]. They used regular expressions over the dynamic execution traces to formalize their program specifications. Their technique followed two partitioning: spatial and temporal. Spatial partitioning was used to monitor different segments of the program by different monitors and temporal partitioning was used to switch the program monitoring on/off.

CIDSs are being extended on large-scale distributed systems. Zhou et al. [13] proposed LarSID that runs on a decentralized architecture for a large-scale CIDS, which can defend system against an attack by sharing the potential evidence among its IDSs by using a Distributed Hash Table (DHT). However, they did not test their CIDS in a real-world large-scale system.

Vasilomanolakis, et al. [14] provided a taxonomy and survey of all existing CIDSs. They mainly classified CIDSs as centralized, decentralized, and distributed. They proposed the requirements and basic building blocks of traditional CIDS, compared various approaches, and showed the relationship among different attacks. Their analysis suggested a lack of CIDS implementation and quantitative evaluations.

Most of the CIDSs are implemented for specific purposes and a few of them are scalable and implemented in the distributed fashion. In addition, none of the studies identified the general characteristics of the application that should be monitored at runtime. Prevention and strategic actions are missing on most of the proposed frameworks. A comprehensive package of attack detection, prevention against it, honeypot concept, and moving target defense strategy has motivated our research.

III. RUNTIME MONITOR

. Runtime monitoring helps detect the abnormal behavior of the applications by extracting system information and analyzing the program behavior by comparing it to specifications. Any deviation of the behavior or policy violation is treated as an intrusion.

A monitor-able property is a characteristic of the application that can be used to check and analyze the violation of specific program specification during runtime. The monitor-able property can be identified by its impact on an application. Depending on the robustness of an application, the number of characteristics may vary. The following sections describe some critical and essential characteristics of an application [1, 2, 3, 4, and 15].

A. Threshold Level of Functional Parameters

For a complex system, the task of handling complexity of the whole system is minimized by refactoring it into smaller modules. Generally, a module refers to a single functionality of the complex system consisting of one or more functions (aka methods) depending on the system design. In programming, a named section of a program that performs a specific task by receiving from zero to more parameters and providing an output) is defined as a function or method or procedure. Depending on the programming language architecture, method and procedure may differ by design. However, the basic objective of a function/procedure is to perform a specific task. When the system requires performing a certain task, it commands the program to run a specific function that is responsible for accomplishing that task. Since a function is the smallest worker of the system, the parameters of the function are of great importance. Functional parameters/variables are the gateways of entering the system in a legal way without tampering the code. A functional parameter can be a critical variable when it communicates with the external user and the internal functional work. An external user can get information from the internal system/database or other storage through tampering or by compromising the functional parameter (i.e. API). Even though a system has enough security measures, a functional parameter can be used by an intruder through this gateway to create a threat. In addition, during the execution time of an application, the value of each functional parameter changes frequently. Therefore, it is mandatory to monitor those functional variables closely by considering each of them as a potential point of entry of an intruder to the system. Security experts now recommend using as few functional variables as possible in system design.

According to the system specification, the functional parameters should have a certain threshold range. As a monitor-able property of a system, it is necessary to monitor the functional parameters continuously whenever they reach their threshold limits. The threshold limit is pre-determined during system deployment phase. Exceeding the threshold limit is treated as an abnormal system behavior. Critical or functional variables can be very dangerous in a web application [1]. By injecting vulnerable script, an intruder can retrieve unintended information from the system. SQLIA is a type of attack that injects a script through an input field, which is eventually connected with a functional parameter of the system [1]. There are several ways to identify functional variables, such as by analyzing source code, security specification, scanning the software repository, and so on.

B. Threshold Level of Iteration Count

Looping is the most used coding concept on any programming language. A loop helps iterate a process in an increasing or decreasing order by using a variable of certain range limit. In general, a loop has three main components: an initial state, incremental or decremental fashion and a termination state. Using those three components, a loop and its iteration can be controlled. Most common loops used in programming languages are ‘for’ loop, ‘while’ loop, ‘do while’ loop. The iteration process continues infinitely if there is a cavity (changing loop variable) inside the loop. Therefore, a loop counter is as important as the functional parameters. A system malfunctions whenever an intruder changes the loop variables in such a way that makes a process run in an infinite loop, which might result in a denial-of-service attack. Thus, the loop variable is one of the vulnerable points of a system and monitoring the loop variable is mandatory.

C. Program Execution Path

A program has a set of valid execution paths. These are determined at the initial stages of software development lifecycle.

Data flow and basis path testing are the most common approaches to find the valid execution paths of a program. Data flow testing helps to identify all possible legal sub-paths. A sub-path is the direction flow (parameter passing through functions) of a functional or critical variable. A subset of all possible valid legal execution paths can be aggregated by closely analyzing the data flow of all functional variables. Basis path testing also helps to identify the minimum number of the valid execution path of a system. Glass box testing [2] is one way to identify the valid execution paths using the functional variable. Before the system goes into execution mode, glass box-testing finds and aggregates three types of paths: paths within a smaller unit, paths between the units when units are integrated, and path between the subprograms on system level testing. Besides those, control flow testing, code coverage, decision coverage, statement coverage, condition coverage, and prime path testing of a glass box testing also help to identify a minimum set of valid execution paths.

The minimum number of valid execution paths can be generated by the following:

- Execution paths from Data Flow testing (EPDF)
- Execution paths from Control Flow testing (EPCF)
- Execution paths from Basis Path (EPBP)
- Execution paths from Code Coverage (EPCDC)
- Execution paths from Condition Coverage (EPCNC)
- Execution paths from Decision Coverage (EPDC)
- Execution paths from Statement Coverage (EPSC)

$$\text{Minimum number of valid execution paths} = EPCF \cap EPDF \cap EPBP \cap EPCDC \cap EPCNC \cap EPDC \cap EPSC \quad (1)$$

A Runtime Monitor monitors the program execution paths and raises an alarm when the application diverts from the listed paths.

D. The sequence of Actions

A system consists of multiple subprograms or subroutines. An action is a collection of modules or a single module and each of the actions might have some dependencies (e.g.; data dependencies) on prior actions. Therefore, it is necessary to execute the prior actions before executing a dependent action.

The sequence of executing each action is listed on the specification. The Runtime Monitor monitors the execution of each action and matches the sequence with the specification. Any violation of a sequence of actions is a potential threat and treated as abnormal behavior. Kassem et al. [3] proposed a monitor that detects the potential irregularities. They used Quantified Event Automata (QEA) in their monitors to check the vital properties of the e-exam system. To run the exam module, they generated a sequence of following actions: course registration, checking the availability of exam time, submitting an exam registration, getting the exact exam time-period, accepting the exam, and submitting the exam. In the specification, they used e-exam's critical properties in the sequence that produce the automata. Their monitor continuously follows the automata and any violation from the automata is treated as anomalous behavior of the system.

E. File Access Permission

A system is more vulnerable if it has more publicly accessible files. Therefore, restricting the file access is a security concern of a system. By overriding the file access permission, an intruder can easily get the access to the file system that may contain confidential data or source code. Most of the secure systems strictly impose the file access permission restrictions over their file systems. A File Management System (FMS) solely depends on file operations and files can be accessible by different hierarchical roles such as super admin, admin, group user, normal user, etc. Making the system to malfunction, a normal user can override his access permission, resulting in getting access to the group user's files or admin user's file.

A Runtime Monitor helps detect malfunctioning by checking the file metadata and consistency of file operations. Sun et al. [4] proposed a Runtime Monitor, which checks the consistency of "UPDATE" operation of a file before it is committed to the destination. They used metadata interpretation on their monitor, which leads to minimizing the data loss. Their monitor checks metadata interpretation to ensure corruption of the file data structure. Runtime monitor can ensure the valid file operation by consistency checker and uncorrupted file transmission by checking file metadata interpretation.

IV. Collaborative Runtime Monitor (CoRuM) Framework

CoRuM aims to provide the security by collaboratively monitoring an application or a system at its runtime. In the following sections, we discuss our proposed Collaborative Runtime Monitor (CoRuM) framework in detail. Fig. 1. and Fig. 2. show the framework, which has three main components: Monitor, Application, and Honeypot or Backup Server.

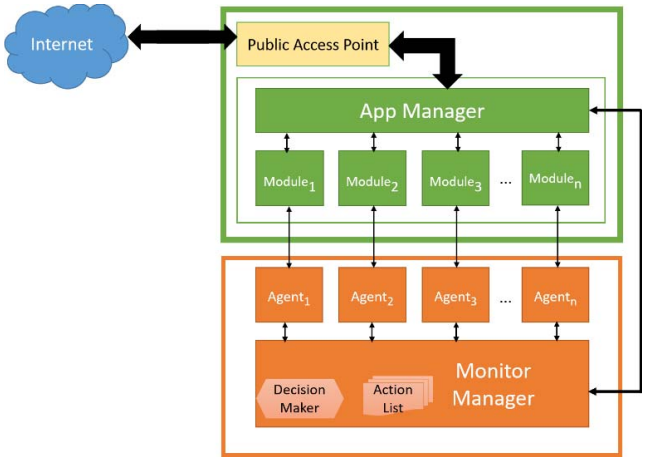

Fig. 1. CoRuM Framework

A. Monitor

The Monitor unit consists of two basic components: Agents and Monitor Manager.

Agents: An agent is a "Runtime Monitor" which is designed to monitor a module. Individual monitors are set up to monitor

the modules of the system. An agent fetches the corresponding module's specifications from the monitor manager, continuously monitors it during its runtime and sends back the monitoring information to the monitor manager. Any violation of the specification is treated as a potential threat. When the agent identifies a violation or deviation, it raises an alarm and reports to the manager for further action.

Monitor Manager: The monitor manager is the heart of the CoRuM architecture. It responds to agents' requests and sends the detail specifications to the corresponding module. The monitor manager aggregates monitoring information from all agents, decides on the presence of any suspicious activity, and takes necessary action against any potential threat. In our implementation, it has control over the full application as well as the subordinate agents.

Decision Maker: The monitor manager has a built-in decision maker, which consists of the logic and analytic ability to make an instant decision and verifies whether a set of activities is anomalous or not.

Action List: The action list is the list of action plans stored in the Monitor Manager. The manager finds an action from this action list that best fits the scenario at hand and takes an appropriate action.

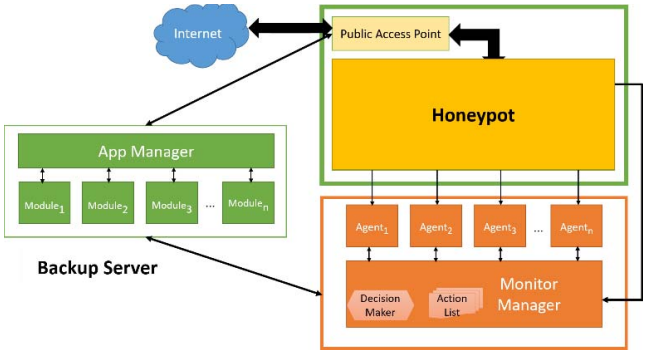

Fig. 2. After Backup Server and Honeypot Activation

B. Application

The application being monitored is configured as a set of modules with each module paired with an agent.

Public Access Point: The public access point is the public gateway through which users and other applications can interact with the monitored application.

Application Manager: To process a request, the public access point directs the packet to the application manager. The application manager has the ability and responsibility for distributing the workload among the modules in a parallel and distributed fashion. It acts as a master node of the application and assigns the module workers to complete specific tasks. The modules are responsible to accomplish the tasks and forward the output to the application manager. The manager aggregates the accomplished tasks and responds back to the requested IP address through the public access point. The goal of having an application manager is to distribute the workload among modules in a parallel way.

Modules: Each module is designed to accomplish a certain task assigned by the application manager. Modules submit their accomplished task to the application manager.

C. Honeypot and Backup Servers

CoRuM framework utilizes honeypot and backup servers to ensure the security of the system. A honeypot mimics original system with some dummy data. The honeypot is configured to analyze a suspected user to know his interaction with real data and real system. However, the user's accessible data never commits to the original system or database. In our framework, we also implement a backup server. The concept behind using the backup server is to divert the normal users to the backup service when the original system is affected by malicious activity or a service interruption due to malicious activity. Fig. 2 shows the backup server and honeypot implementation.

V. Experimental Results

A. Experimental Setup

We have implemented an e-commerce web application, which is integrated with the CoRuM. The application has several modules corresponding to functions such as product listing, search product, product recommendation, user management, cart, checkout, and payment gateway. For each module, a specific Monitor was assigned to monitor the activity and the monitor was responsible to report to the manager.

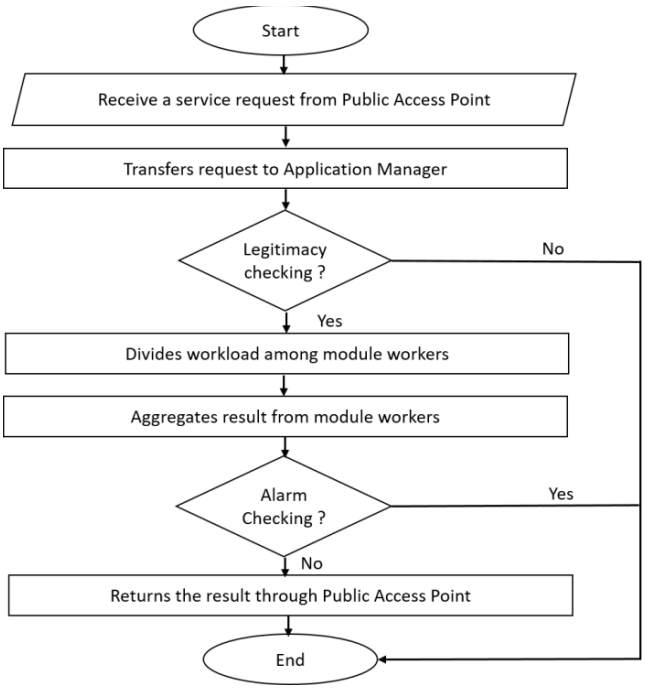

Fig. 3. Flowchart of a normal operation (From Application manager end)

The module/monitor pairs were located in different physical locations in this distributed implementation. Based on the characteristics analysis, we generated system specifications for the application. The sequence of actions during module run times were also listed on the specification. The specification was stored in an XML file where the details of an activity are

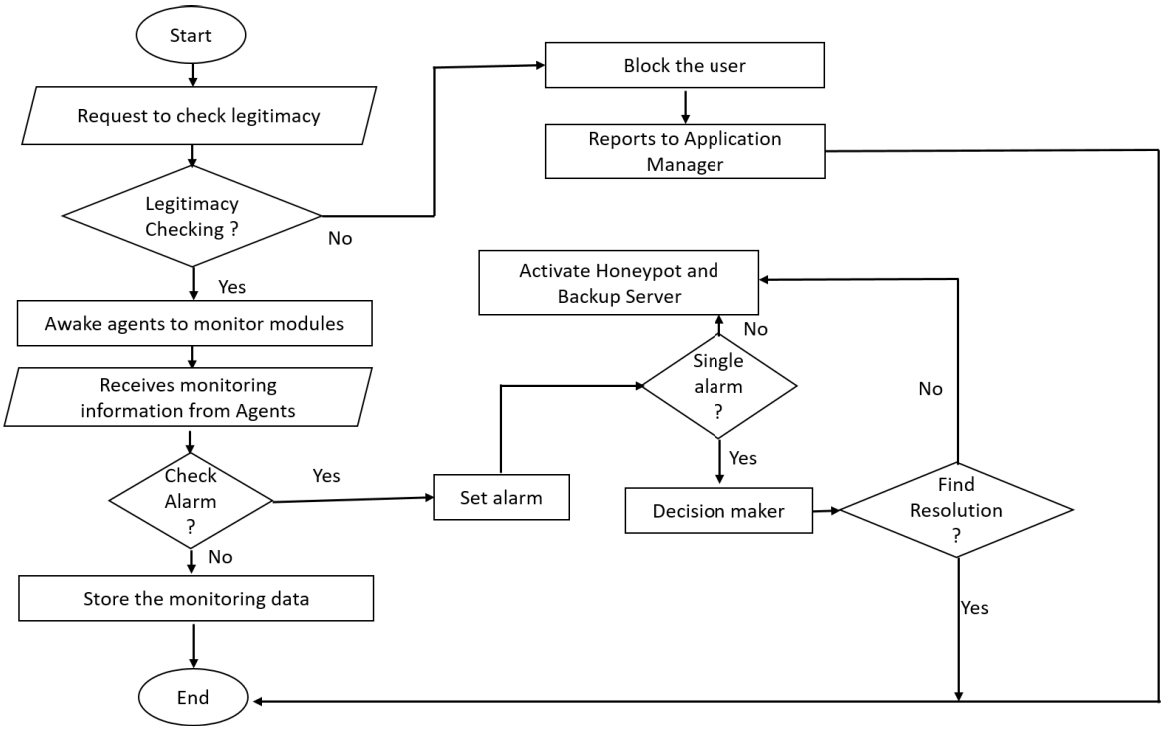

Fig. 4. Flowchart of an abnormal operation (From Monitor manager end)

identified with unique tags and each tag was enclosed with a parameter name and its value. After aggregating all specifications, we provided those to the monitor manager before it started monitoring the system. Each of the modules of the application runs with different actions and the corresponding agents were pre-loaded with specifications from the monitor manager. We generated different types of attacks on the system.

B. Case Studies

In general, there are two possibilities of system behavior: Normal and Abnormal. In this section, we discuss the workflow of our system operation for both the normal and abnormal cases. In addition, we show how our monitor operates the system when it finds the module is not working or has been compromised.

1) Normal Operation

Fig. 3. shows the normal scenario and corresponding workflow of our system. The application manager returns the result to the requested user, only after checking whether monitor raises an alarm. In this figure, we show the operational workflow from application manager's end and discard the monitor manager portion, as it is the normal case scenario.

2) Abnormal Operation

We see an abnormal operation in Fig. 4. An abnormal operation is indicated by the monitor manager raising an alarm. In this figure, we show the workflow from the monitor manager's end. Since it is an abnormal operation, the monitor manager takes control over all the components including the application manager.

After fixing the issue, monitor manager relinquishes the control and lets application manager proceed further. Therefore, in Fig. 4., we do not show any workflow from application manager's end. There is a branch marked as "Single Alarm" in the flowchart. That function (Single alarm checker) counts the alarms that have been reported by different agents for the same abnormality. Based on alarm count, we divide our monitoring strategy into two parts that are discussed below:

When single module compromises: For this scenario, when the monitor manager receives an alarm from any of its agents, it sends a special signal to all the agents, that instructs them to monitor the application more closely for that user and return the monitoring report more frequently. After analyzing all log reports, the decision maker finds an appropriate action plan from the action list database. Thus, the decision maker takes the action against that anomalous activity. Meanwhile, aggregating and analyzing all logs reports that are received from all agents, the monitor manager activates the backup server and honeypot to protect the system from further malicious intrusion.

In backup server and honeypot activation phase, the monitor manager assigns the current active server as a honeypot and stores/records the malicious user's activity through its agents for further analysis. In addition, it activates the backup server where a fresh copy of the original system is installed. To make this backup server running, monitor manager instructs application manager to maintain the connectivity with the backup server's module worker and accept the service for every service request except the one from the malicious user. Then the monitor manager observes the current malicious IP address more closely and frequently. After monitoring closely for a certain period, if all the monitors report a positive feedback for that malicious user, the manager changes suspected user from the blacklist to the whitelist. On the other hand, if the monitor manager does not receive positive feedback from all agents, then it keeps all agents monitoring and storing the information for further analysis. Until admin assistant takes further action, the monitor manager activates the honeypot and backup server.

When multiple modules compromise: For this scenario, when the monitor manager receives multiple alarms from multiple agents, it invokes decision maker. The decision maker finds a match from the action list. If it does not get any resolution, it activates the honeypot and backup server as in the previous case scenario. In addition, the decision maker blocks this IP address and adds it to the blacklist. For any further requests coming from that specific IP address, the application manager drops the packet and doesn't allow it to proceed further.

C. Experimental Results

We generated a Source Code Hijacking and Host Compromising attacks and applied them to our application to see how our monitor acts. Based on the generated attack scenario, we penetrated those attacks (by service request) from different computers such as lab computers as well as home, public computers having their different IP addresses and each service request details was stored to the admin panel log database. Once a week, we reviewed the log details from admin panel and determined whether CoRuM detected the attack or not, and counted, which is shown in TABLE I. In our experimentation, we tested more than 1000 service requests from different IP addresses shown in TABLE I. to test the compromising of single or multiple modules and for all the cases, our framework behaved as expected in approximately 98% test cases. In addition, the transition of activating the honeypot and backup servers was so smooth that a legitimate user did not feel interruption from the services. We divided the application into modules and hosted each module on different distributed locations and established a secure communication protocol to transfer the data between the application and monitoring system.

TABLE I. ATTACK DETECTION (SINGLE MODULE AFFECT VS MULTIPLE MODULES AFFECT) AGAINST SERVICE REQUESTS

Cyber Attacks	Single Module Compromised		Multiple Modules Compromised	
	Detected	Not detected	Detected	Not detected
Code Hijacking	284	7	318	9
Host Compromising	302	6	331	8

Finally, for the visualization of log data and the analysis of monitoring information purposes, a network admin panel was built where the admin could see blocked IP addresses and status of all servers. Moreover, the admin has the flexibility of analyzing the scenario manually, taking necessary actions to restore the original server as well as taking the actions against malicious IP addresses.

Compared to previous works, CoRuM has the ability to monitor the application in a generic way and is implemented over a distributed network. It not only can detect an anomaly but also can implement preventive measures. We have experimented in a smaller testbed (Application) but it can be scaled to a large system.

VI. CONCLUSION AND FUTURE WORK

We have proposed a Collaborative Runtime Monitor framework (CoRuM), which monitors an application at runtime and verifies the specifications for any violation. The specifications are generated from the critical and crucial characteristics (monitor-able property) of the application that are discussed here. In experimentation, the application was configured as a distributed system. Source Code Hijacking and Host Compromising attack were imposed on the application and the architecture defends system against the attacks spontaneously. Honeypot and backup server secure the system in a robust way and helps prevent multi-dimension attacks.

An extension of CoRuM framework to detect other types of attacks is underway. In addition, we plan to move our framework into the cloud. In the cloud network, the virtual machine (VM) is the smallest unit to store information. We plan to use multiple VMs to host our framework components that will be managed by OpenStack. By using machine learning and intelligent software agents, we plan to make our monitors intelligent enough to detect different types of attacks and learn to update themselves.

REFERENCES

[1] Dharam, Ramya, and Sajjan G. Shiva. "Runtime Monitoring Framework for SQL Injection Attacks." International Journal of Engineering and Technology 6.5 (2014): 392

[2] https://en.wikipedia.org/wiki/White-box_testing

[3] Kassem, Ali, Ylies Falcone, and Pascal Lafourcade. "Monitoring electronic exams." Runtime Verification. Springer International Publishing, 2015.

[4] Sun, Kuei, et al. "Robust consistency checking for modern filesystems."International Conference on Runtime Verification. Springer International Publishing, 2014.

[5] Chimento, Jesús Mauricio, et al. "StaRVOOrS: A tool for combined static and runtime verification of Java." Runtime Verification. Springer International Publishing, 2015.

[6] Pinisetty, Srinivas, et al. "TiPEX: a tool chain for Timed Property Enforcement during eXecution." Runtime Verification. Springer International Publishing, 2015.

[7] Colombo, C., Pace, G.J., Schneider, G.: LARVA - a tool for runtime monitoring of Java programs. In: SEFM 2009, pp. 33–37. IEEE Computer Society (2009)

[8] Testerink, Bas, Nils Bulling, and Mehdi Dastani. "A model for collaborative runtime verification." Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems. International Foundation for Autonomous Agents and Multiagent Systems, 2015.

[9] Bai, Xiaoying, et al. "Collaborative web services monitoring with active service broker." 2008 32nd Annual IEEE International Computer Software and Applications Conference. IEEE, 2008.

[10] Wu, Yu-Sung, et al. "Collaborative intrusion detection system (CIDS): a framework for accurate and efficient IDS." Computer Security Applications Conference, 2003. Proceedings. 19th Annual. IEEE, 2003.

[11] Bodden, Eric, et al. "Collaborative runtime verification with tracematches."International Workshop on Runtime Verification. Springer Berlin Heidelberg, 2007.

[12] Bai, Xiaoying, et al. "A multi-agent based framework for collaborative testing on web services." The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06). IEEE, 2006.

[13] Zhou, Chenfeng Vincent, Shanika Karunasekera, and Christopher Leckie. "Evaluation of a decentralized architecture for large scale collaborative intrusion detection." 2007 10th IFIP/IEEE International Symposium on Integrated Network Management. IEEE, 2007.

[14] Vasilomanolakis, Emmanouil, et al. "Taxonomy and survey of collaborative intrusion detection." ACM Computing Surveys (CSUR) 47.4 (2015): 55.

[15] Dharam, Ramya, and Sajjan G. Shiva. "Runtime monitoring technique to handle tautology based SQL injection attacks." International Journal of Cyber-Security and Digital Forensics (IJCSDF) 1.3 (2012): 189-203.