International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2015): 78.96 | Impact Factor (2015): 6.391

Container Based Intrusion Detection System in Multitier Web Applications

Nishigandha Shendkar

Department of Computer Engineering, Pune Institute of Computer Technology, Pune University, India

Abstract: ***Internet services and applications have become an inextricable part of daily life, enabling communication and the management of personal information from anywhere. To accommodate this increase in application and data complexity, web services have moved to a multitier design wherein the web server runs the application front-end logic and data are outsourced to a database or file server. In this paper, we present an IDS system that models the network behavior of user sessions across both the front-end web server and the back-end database. By monitoring both web and subsequent database requests, we are able to ferret out attacks that independent IDS would not be able to identify. Furthermore, we quantify the limitations of any multitier IDS in terms of training sessions and functionality coverage. We implemented using an Apache webserver with MySQL and lightweight virtualization.***

Keywords: Anomaly detection, virtualization, multitier web application

1. Introduction

Web delivered services and applications have increased in both popularity and complexity over the past few years. Daily tasks, such as banking, travel, and social networking, are all done via the web. Such services typically employ a web server front end that runs the application user interface logic, as well as a back-end server that consists of a database or file server. Due to their ubiquitous use for personal and corporate data, web services have always been the target of attacks. These attacks have recently become more diverse, as attention has shifted from attacking the front end to exploiting vulnerabilities of the web applications in order to corrupt the back-end database system (e.g., SQL injection attacks).

A plethora of Intrusion Detection Systems (IDSs) currently examine network packets individually within both the web server and the database system. However, there is very little work being performed on multitier Anomaly Detection (AD) systems that generate models of network behavior for both web and database network interactions.

In such multitier architectures, the back-end database server is often protected behind a firewall while the web servers are remotely accessible over the Internet. Unfortunately, though they are protected from direct remote attacks, the back-end systems are susceptible to attacks that use web requests as a means to exploit the back end. To protect multitier web services, Intrusion detection systems have been widely used to detect known attacks by matching misused traffic patterns or signatures.

Individually, the web IDS and the database IDS can detect abnormal network traffic sent to either of them. However, it is found that these IDSs cannot detect cases wherein normal traffic is used to attack the web server and the database server. For example, if an attacker with non admin privileges can log in to a Web server using normal-user access credentials, he/she can find a way to issue a privileged database query by exploiting vulnerabilities in the web server. Neither the web IDS nor the database IDS would detect this type of attack since the web IDS would merely see typical user login traffic and the database IDS would see only the normal traffic of a privileged user. This type of attack can be readily detected if the database IDS can identify that a privileged request from the web server is not associated with user privileged access.

Unfortunately, within the current multithreaded Web server architecture, it is not feasible to detect or profile such causal mapping between web server traffic and DB server traffic since traffic cannot be clearly attributed to user sessions. Our approach can create normality models of isolated user sessions that include both the web front-end (HTTP) and back-end (File or SQL) network transactions. To achieve this, we employ a lightweight virtualization technique to assign each user's web session to a dedicated container, an isolated virtual computing environment. The container ID is used to accurately associate the web request with the subsequent DB queries. Thus, it can build a causal mapping profile by taking both the Web server and DB traffic into account.

2. Related Work

2.1 History about multitier web application

The three tier Architecture may seem similar to the model-view-controller (MVC) concept; however, topologically they are different. A fundamental rule in three tier architecture is the client tier never communicates directly with the data tier; in a three-tier model all communication must pass through the middle tier called Web tier. Conceptually the three-tier architecture is linear. However, the MVC architecture is triangular: the view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model.

From a historical perspective the three-tier architecture concept emerged in the 1990s from observations of distributed systems (e.g., web applications) where the client, middle ware and data tiers ran on physically separate platforms. Whereas MVC comes from the previous decade (by work at Xerox PARC in the late 1970s and early 1980s) and is based on observations of applications that ran on a

single graphical workstation; MVC was applied to distributed applications later in its history. Today, MVC and similar model-view-presenter (MVP) are Separation of Concerns design patterns that apply exclusively to the presentation layer of a larger system. In simple scenarios MVC may represent the primary design of a system, reaching directly into the database; however, in most scenarios the Controller and Model in MVC have a loose dependency on either a Service or Data layer/tier [4].

2.2 Literature Survey

A network Intrusion Detection System can be classified into two types: anomaly detection and misuse detection. Anomaly detection first requires the IDS to define and characterize the correct and acceptable static form and dynamic behavior of the system, which can then be used to detect abnormal changes or anomalous behaviors [5]. CLAMP [6] is architecture for preventing data leaks even in the presence of attacks. By isolating code at the Web server layer and data at the database layer by users, CLAMP guarantees that a user's sensitive data can only be accessed by code running on behalf of different users. In contrast, this system focuses on modeling the mapping patterns between HTTP requests and DB queries to detect malicious user sessions. There are additional differences between these two in terms of requirements and focus. CLAMP requires modification to the existing application code, and the Query Restrictor works as a proxy to mediate all database access requests. Container based system uses process isolation whereas CLAMP requires platform virtualization, and CLAMP provides more coarse-grained isolation than this system. However, this system would be ineffective at detecting attacks if it were to use the coarse grained isolation as used in CLAMP. Building the mapping model in this system would require a large number of isolated web stack instances so that mapping patterns would appear across different session instances.

Virtualization is used to isolate objects and enhance security performance. Full virtualization and para-virtualization are not the only approaches being taken. An alternative is a lightweight virtualization, such as OpenVZ [7]. In general, these are based on some sort of container concept. With containers, a group of processes still appears to have its own dedicated system, yet it is running in an isolated environment. On the other hand, lightweight containers can have considerable performance advantages over full virtualization or para-virtualization. Thousands of containers can run on a single physical host. Such virtualization techniques are commonly used for isolation and containment of attacks. However, in this system, we utilized the container ID to separate session traffic as a way of extracting and identifying causal relationships between web server requests and database query events.

3. Proposed Work

3.1 Container Architecture

Implementation of Intrusion detection System in multitier web application using container architecture as following: Container architecture basically detects intrusion in two sides that is web server side as well as database side. This architecture of Intrusion Detection System comes under two type of Intrusion detection system so we can also able to say, Implementation of Container Architecture Intrusion detection system is combination of behavioral IDS and Signature based IDS. That means it is Hybrid category of intrusion detection system. This is best approach for Intrusion Detection in multitier web application. An efficient system is proposed using container architecture that can detect the attacks in multi-tiered web services. It can create normality models of isolated user sessions that include both the web front-end (HTTP) and back-end (File or SQL) network transactions. To achieve this, a lightweight virtualization technique is employed to assign each user's web session to a dedicated container in an isolated virtual computing environment. The container ID is used to accurately associate the web request with the subsequent DB queries. Typical flow data particularly relevant to intrusion detection and prevention includes the following [2]:

1) Source and destination IP addresses.
2) Source and destination TCP or UDP ports or ICMP types and codes.
3) Number of packets and number of bytes transmitted in the session.
4) Timestamps for the start and end of the session.

In this prototype, each user session into a different container; however, this was a design decision. For instance, we can assign a new container per each new IP address of the client. In our implementation, containers were recycled based on events or when sessions time out. We were able to use the same session tracking mechanisms as implemented by the Apache server (cookies, mod, user track, etc.) because lightweight virtualization containers do not impose high memory and storage overhead. Thus, we could maintain a large number of parallel-running Apache instances similar to the Apache threads that the server would maintain in the scenario without containers. If a session timed out, the Apache instance was terminated along with its container. Consider, we used a 60-minute timeout due to resource constraints of our test server. However, this was not a limitation and could be removed for a production environment where long-running processes are required. Figure.1 depicts the architecture and session assignment of our prototype, where the host web server works as a dispatcher.

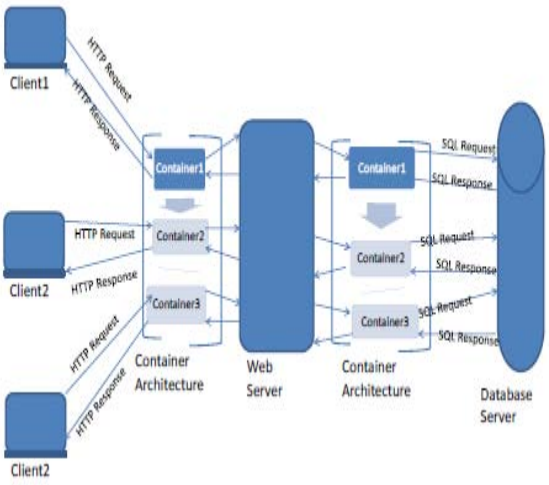

Figure 1: Container Architecture

Above figure1 shows container architecture [3]. This shows that how communications are categorized as sessions and how database transactions can be related to a corresponding sessions.

3.2 Behavioral approach in container architecture

According to figure 1, if client 2 is malicious and takes over the web server, all subsequent database transactions become suspects, and response to the client. But in figure 1, client 2 will only use the container 2 sessions and corresponding database transaction set T2 will be the only affected session of data within the database. A Container Architecture is a device or software application for intrusion detection that monitors network or system for malicious activities and produces reports to server.

The primary focus of Container Architecture is to identify possible incidents, logged information about them and produce report of attempts of an incident. Many organizations uses Container Architecture for other purposes like to identify the problems with policies of security, existing threats documentation etc. Nearly every organization uses the Container Architecture technique of intrusion detection for their security infrastructure.

3.3 Architecture and Confinement

All network traffic, from both legitimate users and adversaries, is received intermixed at the same web server. If an attacker compromises the web server, he/she can potentially affect all future sessions (i.e., session hijacking). Assigning each session to a dedicated web server is not a realistic option, as it will deplete the web server resources. To achieve similar confinement while maintaining a low performance and resource overhead, we use lightweight virtualization. In our design, we make use of lightweight process containers, referred to as "containers," as ephemeral, disposable servers for client sessions. It is possible to initialize thousands of containers on a single physical machine, and these virtualized containers can be discarded, reverted, or quickly reinitialized to serve new sessions. A single physical web server runs many containers, each one an exact copy of the original web server. Our approach dynamically generates new containers and recycles used ones. As a result, a single physical server can run continuously and serve all web requests. However, from a logical perspective, each session is assigned to a dedicated web server and isolated from other sessions. Since we initialize each virtualized container using a read-only clean template, we can guarantee that each session will be served with a clean web server instance at initialization. We choose to separate communications at the session level so that a single user always deals with the same web server. Sessions can represent different users to some extent, and we expect the communication of a single user to go to the same dedicated web server, thereby allowing us to identify suspect behavior by both session and user. If we detect abnormal behavior in a session, we will treat all traffic within this session as tainted. If an attacker compromises a vanilla web server, other sessions' communications can also be hijacked. In our system, an attacker can only stay within the web server containers that he/she is connected to, with no knowledge of the existence of other session communications.

We can thus ensure that legitimate sessions will not be compromised directly by an attacker.

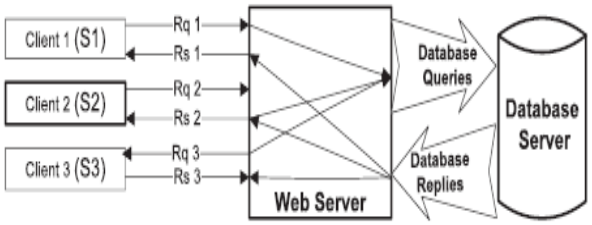

Figure 2: Classic three-tier model

Figure 2 illustrates the classic three-tier model. At the database side, we are unable to tell which transaction corresponds to which client request. The communication between the web server and the database server is not separated, and we can hardly understand the relationships among them. Figure 3 depicts how communications are categorized as sessions and how database transactions can be related to a corresponding session. According to Figure2, if Client 2 is malicious and takes over the web server, all subsequent database transactions become suspect, as well as the response to the client. By contrast, according to Figure3, Client 2 will only compromise the VE 2, and the corresponding database transaction set T2 will be the only affected section of data within the database.

3.4 Building the Normality Model

This container-based and session-separated web server architecture not only enhances the security performances but also provides us with the isolated information flows that are separated in each container session. It allows us to identify the mapping between the web server requests and the subsequent DB queries, and to utilize such a mapping model to detect abnormal behaviors on a session/client level. In typical three-tiered web server architecture, the web server receives HTTP requests from user clients and then issues SQL queries to the database server to retrieve and update data. These SQL queries are causally dependent on the web request hitting the web server. We want to model such causal mapping relationships of all legitimate traffic so as to detect abnormal/attack traffic.

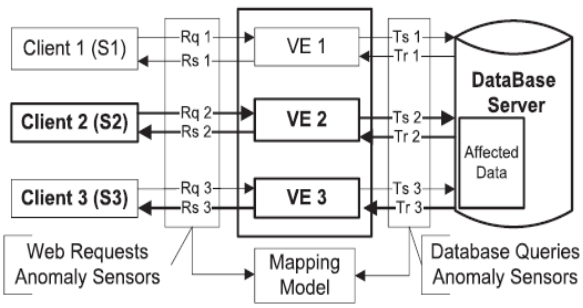

Figure 3: Web server instances running in containers

In practice, we are unable to build such mapping under a classic three-tier setup. Although the web server can distinguish sessions from different clients, the SQL queries

are mixed and all from the same web server. It is impossible for a database server to determine which SQL queries are the results of which web requests, much less to find out the relationship between them. Even if we knew the application logic of the web server and were to build a correct model, it would be impossible to use such a model to detect attacks within huge amounts of concurrent real traffic unless we had a mechanism to identify the pair of the HTTP request and SQL queries that are causally generated by the HTTP request. However, within our container-based web servers, it is a straightforward matter to identify the causal pairs of web requests and resulting SQL queries in a given session. Moreover, as traffic can easily be separated by session, it becomes possible for us to compare and analyze the request and queries across different sessions. Section 4 further discusses how to build the mapping by profiling session traffics.

To that end, we put sensors at both sides of the servers. At the web server, our sensors are deployed on the host system and cannot be attacked directly since only the virtualized containers are exposed to attackers. Our sensors will not be attacked at the database server either, as we assume that the attacker cannot completely take control of the database server. In fact, we assume that our sensors cannot be attacked and can always capture correct traffic information at both ends. Figure 3 shows the locations of our sensors. Once we build the mapping model, it can be used to detect abnormal behaviors. Both the web request and the database queries within each session should be in accordance with the model. If there exists any request or query that violates the normality model within a session, then the session will be treated as a possible attack.

3.5 Attack Scenarios

Our system is effective at capturing the following types of attacks

3.5.1 Privilege Escalation Attack:

Let's assume that the website serves both regular users and administrators. For a regular user, the web request r_u will trigger the set of SQL queries Q_u; for an administrator, the request r_a will trigger the set of admin level queries Q_a. Now suppose that an attacker logs into the web server as a normal user, upgrades his/her privileges, and triggers admin queries so as to obtain an administrator's data. This attack can never be detected by either the web server IDS or the database IDS since both r_u and Q_a are legitimate requests and queries. Our approach, however, can detect this type of attack since the DB query Qa does not match the request r_u, according to our mapping model. Figure 4 shows how a normal user may use admin queries to obtain privileged information.

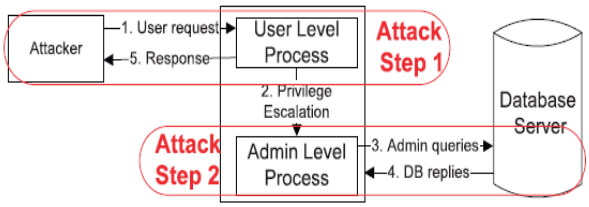

Figure 4: Privilege escalation attack

3.5.2 Hijack Future Session Attack:

This class of attacks is mainly aimed at the webserver side. An attacker usually takes over the webserver and therefore hijacks all subsequent legitimate user sessions to launch attacks. For instance, by hijacking other user sessions, the attacker can eavesdrop, send spoofed replies, and/or drop user requests. A session-hijacking attack can be further categorized as a Spoofing/Man-in-the-Middle attack, an Ex filtration Attack, a Denial-of-Service/Packet Drop attack, or a Replay attack. Figure 5 illustrates a scenario wherein a compromised webserver can harm all the Hijack Future Sessions by not generating any DB queries for normal-user requests. According to the mapping model, the web request should invoke some database queries, then the abnormal situation can be detected. However, neither a conventional webserver IDS nor a database IDS can detect such an attack by itself. Fortunately, the isolation property of our container based webserver architecture can also prevent this type of attack. As each user's web requests are isolated into a separate container, an attacker can never break into other users' sessions.

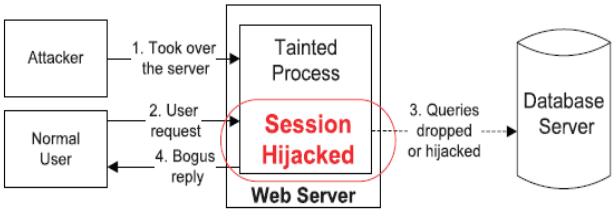

Figure 5: Hijack Future Session Attack

3.5.3 Injection Attack

Attacks such as SQL injection do not require compromising the webserver. Attackers can use existing vulnerabilities in the webserver logic to inject the data or string content that contains the exploits and then use the webserver to relay these exploits to attack the back-end database. Since our approach provides a two-tier detection, even if the exploits are accepted by the webserver, the relayed contents to the DB server would not be able to take on the expected structure for the given webserver request. For instance, since the SQL injection attack changes the structure of the SQL queries, even if the injected data were to go through the webserver side, it would generate SQL queries in a different structure that could be detected as a deviation from the SQL query structure that would normally follow such a web request. Figure 6 illustrates the scenario of a SQL injection attack.

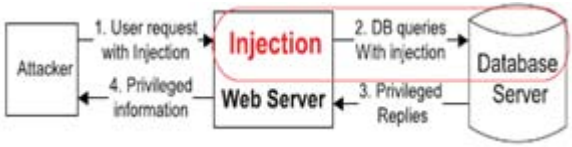

Figure 6: Injection Attack

3.5.4 Direct DB Attack

It is possible for an attacker to bypass the web server or firewalls and connect directly to the database. An attacker could also have already taken over the web server and be submitting such queries from the web server without sending web requests. Without matched web requests for such

queries, a web server IDS could detect neither. Furthermore, if these DB queries were within the set of allowed queries, then the database IDS itself would not detect it either. However, this type of attack can be caught with our approach since we cannot match any web requests with these queries. Figure 7 illustrates the scenario wherein an attacker bypasses the web server to directly query the database.

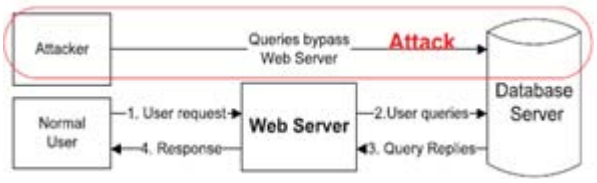

Figure 7: Direct DB Attack

3.6 Limitations

In this section, we discuss the operational and detection limitations of container based intrusion detection system.

1. Vulnerabilities Due To Improper Input Processing
Cross Site Scripting is a typical attack method where in attackers embedding malicious client scripts via legitimate user inputs. In this system, the entire user input values are normalized so as to build a mapping model based on the structures of HTTP requests and DB queries. Once the malicious user inputs are normalized, it cannot detect attacks hidden in the values. These attacks can occur even without the databases. It offers a complementary approach to those research approaches of detecting web attacks based on the characterization of input values.

3.7 Mapping Relations

In this system, it is classified into four possible mapping patterns. Since the request is at the origin of the data treat each request as the mapping source. In other word, the mappings in the model are always in the form of one request to a query set TO Q_n.

3.7.1 Deterministic Mapping
This is the most common and perfectly matched pattern. That is to say that web request r_m appears in all with the SQL queries set Qn. For any session in the testing phase with the request r_m, the absence of a query set Q_n matching the request indicates a possible intrusion. On the other hand, if Q_n is present in the session without the corresponding rm, this may also be the sign of an intrusion. In websites this type of mapping comprises the majority of cases since the same results should be returned for each time a user visits the same link.

3.7.2 Empty Query Set
In special cases, the SQL query set may be the empty set. This implies that the web request neither causes nor generates any database queries. For example, when a web request for retrieving an image GIF from the same web server is made, a mapping relationship does not exist because only the web requests are observed. This type of mapping is called r_m assign empty. During the testing phase, we keep these web requests together in the set EQS.

3.7.3 No Matched Request
In some cases, the web server may periodically submit queries to the database server in order to conduct some scheduled tasks, such as jobs for archiving or backup. This is not driven by any web request, similar to the reverse case of the Empty Query Set mapping pattern. These queries can't match up with any web requests, and we keep these unmatched queries in a set NMR. During the testing phase, any query within set NMR is considered legitimate. The size depends on web server logic, but it is typically small.

3.7.4 Non Deterministic Mapping
The same web request may result in different SQL query sets based on input parameters or the status of the webpage at the time the web request is received. In fact, these different SQL query sets do not appear randomly, and there exists a candidate pool of query sets. Each time that the same type of web request arrives, it always matches up with one (and only one) of the query sets in the pool. It is difficult to identify that matches this pattern. This happens only within dynamic websites.

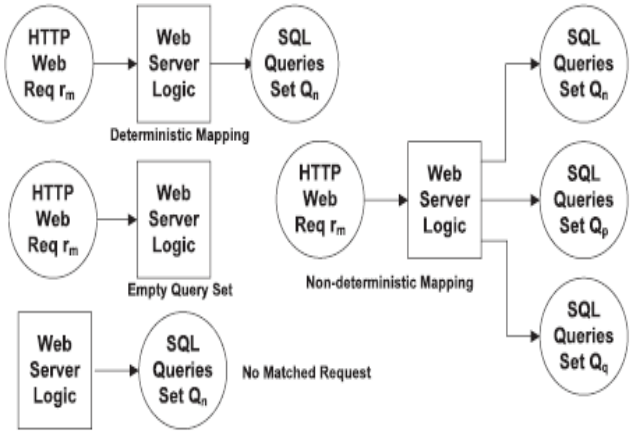

Figure 8: Overall Representation of mapping patterns

3.8 Performance Evaluation

We implemented a prototype by using a Webserver with a back-end DB. We also set up two testing websites, one static and the other dynamic. To evaluate the detection results for our system, we analyzed four classes of attacks and measured the false positive rate for each of the two websites.

3.9 Implementation

Initially, we deployed a static testing website using the Joomla [10] Content Management System. In this static website, updates can only be made via the back end management interface. This was deployed as part of our center website in production environment and served 52 unique web pages. For our analysis, we collected real traffic to this website for more than two weeks and obtained 1,172 user sessions. To test our system in a dynamic website scenario, we set up a dynamic Blog using the Word press [3] blogging software. In our deployment, site visitors were allowed to read, post, and comment on articles. All models for the received frontend and back-end traffic were generated using these data. We discuss performance overhead, which is common for both static and dynamic models, in the

following section. In our analysis, we did not take into consideration the potential for caching expensive requests to further reduce the end-to-end latency; this we left for future study.

4. Conclusion

We presented an Intrusion Detection System that builds models of normal behavior for multitier web applications from both front-end web (HTTP) requests and back-end database (SQL) queries. It forms container based IDS with multiple input streams to produce alerts. We have shown that such correlation of input streams provides a better characterization of the system for anomaly detection because the intrusion sensor has a more precise normality model that detects a wider range of threats. Furthermore, we quantified the detection accuracy of our approach when we attempted to model static and dynamic web requests with the back-end file system and database queries.

References

[1] Meixing Le, Angelos Stavrou, Brent ByungHoon Kang,"Double Guard: Detecting Intrusions in Multitier Web Applications", IEEE Transactions on dependable and secure computing, vol. 9, no. 4, July/august 2012.
[2] Manoj E. Patil, Rakesh D. More, "Survey of Intrusion Detection System in Multitier Web Application", International Journal of Emerging Technology and Advanced Engineering, vol. 2, Issue 10, October 2012.
[3] http: //www.omnisecu.com/security/infrastructure-and-emailsecurity/types-of-intrusion- detection-systems.html.
[4] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An Approach for the Anomaly-Based Detection of State Violations in Web Applications," Proc. Int'l Symp. Recent Advances in Intrusion Detection (RAID '07), 2007.
[5] H. Debar, M. Dacier, and A. Wespi, "Towards a Taxonomy of Intrusion-Detection Systems," Computer Networks, vol. 31, no. 9, pp. 805-822, 1999.
[6] B. Parno, J.M. McCune, D. Wendlandt, D.G. Andersen, and A. Perrig, "CLAMP: Practical Prevention of Large-Scale Data Leaks," Proc. IEEE Symp. Security and Privacy, 2009.
[7] Openvz, http://wiki.openvz.org, 2011.