Vulnerability Assessment Enhancement for Middleware[⋆]

Jairo Serrano, Elisa Heymann, Eduardo Cesar, Barton Miller[2]

Universitat Autònoma de Barcelona, Spain
University of Wisconsin-Madison[2], USA
jairodavid.serrano@uab.es, elisa.heymann@uab.es, eduardo.cesar@uab.es,
bart@cs.wisc.edu

Abstract. Security on Grid computing is often an afterthought. However assessing security of middleware systems is of the utmost importance because they manage critical resources owned by different organizations. To fulfill this objective we use First Principles Vulnerability Assessment (FPVA), an innovative analystic-centric (manual) methodology that goes beyond current automated vulnerability tools. FPVA involves several stages for characterizing the analyzed system and its components. Based on the evaluation of several middleware systems, we have found that there is a gap between the initial and the last stages of FPVA, which is filled with the security practitioner expertise. We claim that this expertise is likely to be systematically codified in order to be able to automatically indicate which, and why, components should be assessed. In this paper we introduce key elements of our approach: Vulnerability graphs, Vulnerability Graph Analyzer, and a Knowledge Base of security configurations.

Keywords: grid, middleware, security, vulnerability assessment, vulnerability graph

1 Introduction

Vulnerability assessment is a security task that is insufficiently addressed in most existing grid and cloud projects, even SCADA systems. Such projects use middleware software that often manages lots of critical resources, making them an attractive target for attackers and terrorism activities.

Usually supercomputing middleware bases its security on mechanisms such as authentication, authorization, and delegation to protect passwords, credentials, user files, databases, system access, storage, and so on. These mechanisms have been studied in depth and effectively control key resources, but are not enough to assure that all application's resources are safe. However, middleware systems usually do not undergo a thorough vulnerability assessment during their life cycle or after deployment, whereby security flaws may be overlooked. One possible solution would be to use existing automated tools such as Coverity Prevent [2] or

[⋆] This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974

Fortify Source Code Analyzer (SCA) [5], but even the best of these tools find only a small percentage of the serious vulnerabilities [13].

A thorough vulnerability assessment requires a systematic approach that focuses on the key resources to be protected and allows for a detailed analysis of those parts of the code related to those resources and their trust relationships. Consistently, *First Principles Vulnerability Assessment (FPVA)* [14] answers these requirements. FPVA had been successfully applied to several large and widely-used middleware systems, such as Condor high-throughput scheduling system [1], Storage Resource Broker (SRB) a data grid management system [9], Crossbroker a grid resource management for interactive and parallel applications [12], among others [6]. FPVA starts with an architectural analysis, identifying the key components in a middleware system. It then goes on identifying the resources associated with each component, and how the privilege level of each component is delegated. The results of these steps are documented in clear diagrams that provide a roadmap for the last stage of the analysis, the manual code inspection. This top-down, architecture-driven analysis, can also help to identify more complex vulnerabilities that are based on the interaction of multiple system components and are not amenable to local code analysis.

For all these systems analysts noticed that there is a gap between the three initial steps and the manual code inspection. The analyst should provide certain expertise about the kind of security problems that the systems may present. For example, depending on the language used the analyst should look for different kind of vulnerabilities. We have realized that this knowledge is similar to the one recorded on several available vulnerability classifications, suchs as CWE [4], Plover [10], and McGraw et al. [15], and that it can be codified in the form of rules to be applied automatically. In particular, we used the vulnerability assessment carried out on CrossBroker, which is based on gLite middleware, to sketch our initial ideas [16, 17]. We showed that FPVA clearly overcome the best current automatic tools available, and proposed an approach for systematically determining how the analyst expertise is used for deciding which middleware components are critical based on the FPVA artifacts. In addition, we also proposed a suitable representation for the information gathered in the initial steps of FPVA. The major contributions of this paper are a Vulnerability Graph definition, which is the first stage to the approach we are developing, a middleware and vulnerability taxonomy characterization, the Vulnerability Graph Analyzer, as well as a study case.

The remainder of this paper is structured as follows. In Section 2 we briefly describe the FPVA methodology. Section 3 introduce Vulnerability Graphs. Section 4 describes the Vulnerability Graph Analyzer approach. Section 5 discusses an example of VGA working on a vulnerability graph. The related work is introduced in Section 6. Finally conclusions and the work ahead before VGA can be applied are discussed in Section 7.

2 First Principles Vulnerability Assessment

FPVA proceeds in five stages: architectural, resource, privilege, and component analysis, and result dissemination. We provide a brief description of the first four

FPVA methodology stages, because the vulnerability graph is derived from the information gathered in these stages.

Architectural Analysis: This step identifies the major structural components of the system, including modules, threads, processes, and hosts. For each of these components, FPVA identifies the way they interact, both with each other and with users. Interactions are particularly important as they can provide a basis for understanding how trust is delegated through the system. The artifact produced at this stage is a document that diagrams the structure of the system and the interactions amongst the different components, and with the end users.

Resources Analysis: The second step identifies the key resources accessed by each component, and the operations supported on those resources. Resources include hosts, files, databases, logs, and devices. These resources are often the target of an exploit. For each resource, FPVA describes its value as an end target or as an intermediate target. The artifact produced at this stage is an annotation of the architectural diagrams with resources.

Privilege Analysis: The third step identifies the trust assumptions about each component, answering such questions as how are they protected and who can access them? The privilege level controls the extent of access for each component and, in the case of exploitation, the extent of damage that it can accomplish directly. A complex but crucial part of trust and privilege analysis is evaluating trust delegation. By combining the information from the first two steps, we determine what operations a component will execute on behalf of another component. The artifact produced at this stage is a further labeling of the basic diagrams with trust levels and labeling of interactions with delegation information.

Component Analysis: The fourth step examines each component in depth. For large systems, a line-by-line manual examination of the code is unworkable. In this step, FPVA is guided by information obtained in the first three steps, helping to prioritize the work so that the code relating to high value assets is evaluated first. The work in this step can be accelerated by automated scanning tools. While these tools can provide valuable information, they are subject to false positives, and even when they indicate real flaws, they often cannot tell whether the flaw is exploitable and, even if it is exploitable, the tools can not tell if it will allow serious damage. The artifacts produced by this step are vulnerability reports, which are provided to the software developers.

It can be seen that FPVA is focused in analysing the data and control flows among the system components looking for unsecure features. This orientation has led to the definition of the following concepts:

Attack Surface as the set of coordinates from which an attack might start, indeed it tells security practitioners where to start looking for the attacker's initial behaviour.

Impact Surface as the set of coordinates where exploits or vulnerabilities might be possible.

Attack Vector as the sequence of transformations that allows controlflow to go from a point in the attack surface to a point in the impact surface.

3 Vulnerability Graphs

With the objective of reducing the gap between the first stages of FPVA and the Component Analysis one, we have defined a structure called Vulnerability Graph for representing the results of these initial steps. Vulnerability graphs are aimed

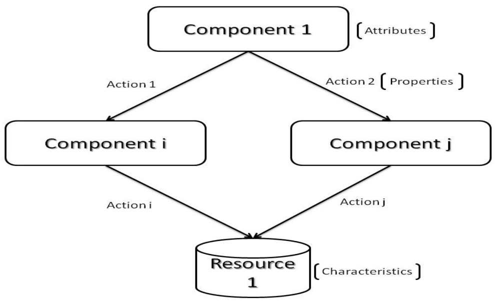

Fig. 1. First-approach vulnerability graph

at finding possible malicious patterns between middleware components and/or resources, following controlflow through their relationships. There are several elements in vulnerability graphs. Figure 1 shows a small vulnerability graph example. Here we can assume intuitively that Component 1 is part of the attack surface, and the Resource 1 might be a point on the impact surface. Based on the information present in Figure 1, we can potentially derive two different attack vectors, the first one includes Component 1, Component i, and Resource 1, the second includes Component 1, Component j, and Resource 1.

Formally, Vulnerability Graph is defined as:

Definition 1. A vulnerability graph $G = (V, E)$ is a tuple where,

- V represents the vulnerability graph nodes, a nonempty set of middleware components and resources.
- E represents the vulnerability graph edges, a nonempty set of actions that associate vulnerability graph nodes.

Definition 2. In security context,

- Vulnerability graph nodes representing components or resources that do not satisfy safety attributes, properties, or characteristics during vulnerability assessment might be considered vulnerable.
- Vulnerability graph edges can associate vulnerable nodes through actions with non-vulnerable nodes, which in turn may become vulnerable or exploitable.

The characterization proposal for middleware components, resources, and critical actions is based on the information that FPVA artifacts, developer teams, and documentation could provide. This characterization step is based on several FPVA artifacts, from six different middleware systems: Condor, SRB, MyProxy, gLExec, VOMS-admin, and CrossBroker. Table 1 shows the most relevant elements of the

Name	**Description**
c_id	An identifier for the component
c_host	The component hostname, where the component is actually running
c_suid	Is the sticky bit set up on the component?
c_priv	The component privileges, Unix style
c_cons	The component constraints is related to data, time, users, privs, and other restrictions
c_rel	The components and/or resources which are straight related to the component

Name	**Description**
r_id	An identifier for the resource
r_host	The resource hostname, where the resource is actually installed or shared
r_suid	Is the sticky bit set up on the resource?
r_priv	The resource privileges, Unix style
r_cons	The resource constraints is related to data, time, users, privs, and other restrictions

Name	**Description**
i_id	An identifier for the interaction
i_host	The interaction host specifies if the interaction happens in an unique host or more
i_stat	The interaction state, describes if interaction is active or passive between components and/or resources
i_type	The interaction type indicates a critical action which could be read, write, open, execute, query, etc
i_priv	The interaction privileges, specifies if interaction type runs as a privilege user
i_cons	The interaction constraints is related to data, time, users, privs, and other restrictions

Table 1. Middleware characterization proposal.

first middleware characterization approach of the vulnerability graph. First column contains the characterization items, and the second one a description of each item. The table is divided in three sections; the first is the components characterizaction; the second is the resources characterization; and the last one the interactions characterization. In our approach, we are going to use model checking techniques to analyze the safety attributes, properties, or characteristics in the vulnerability graph, along with the controlflow steps that allows to go from a point in the attack surface to a point in the impact surface.

4 Vulnerability Graph Analyzer

A manual vulnerability assessment following (FPVA) proceeds initially on architectural, resources, and privileges analysis, and then on a component analysis based on their results (i.e. the artifacts). However, which vulnerabilities are going to be searched in the selected components depend on the implementation details of each component and the analyst's expertise. Consequently, there is a gap between the artifacts generated on the first FPVA steps and the component analysis step that must be currently filled with knowledge of an external source. We claim that this knowledge can be found in several existing vulnerability classifications and that, in consequence, it can be systematically codified in order to be able to automatically indicate which components should be analyzed and why. To reach this objective we have defined the *Vulnerability Graph Analyzer (VGA)*.

VGA will traverse a vulnerability graph following the controlflow with the aim of finding potential malicious patterns or attack vectors, that might lead analists to determine where to search for a vulnerability. We know that most of the generated FPVA artifacts describe a particular operation of the middleware, such as submitting a job in CrossBroker, then starting and ending nodes belonging to the attack and impact surfaces can be clearly identified. In addition, the order in which the graph should be traversed is also quite clear because every edge is labeled with a number indicating when the interaction represented by the edge takes place. Finally, a characterization of a vulnerability taxonomy is required, to build a knowledge base where security configurations about possible malicious patterns are stored.

Ultimately, VGA outcomes are presented as security alerts, because we are not analysing the components code, nor the actual controlflow.

4.1 VGA sketch

A visual representation of the vulnerability graph analyzer is shown in Figure 2. It contains the FPVA Artifacts, the Characterization Proposal, the Knowledge Base, the Graph Analyzer Engine, and Security Alerts. The main component of VGA is

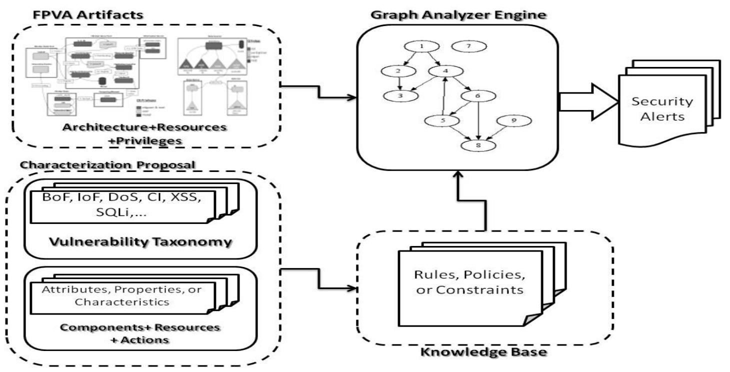
FPVA Artifacts
Architecture+Resources +Privileges
Graph Analyzer Engine
Security Alerts
Characterization Proposal
BoF, IoF, DoS, CI, XSS, SQLi,...
Vulnerability Taxonomy
Attributes, Properties, or Characteristics
Components+ Resources + Actions
Rules, Policies, or Constraints
Knowledge Base

Fig. 2. Vulnerability graph analyzer

the graph analyzer engine, which receives two inputs, and then it calculates the possible attack vectors to be analyzed. The first input is the Vulnerability Graph, which includes the set of components, resources, and critical actions from FPVA artifacts, translated accordingly to our characterization proposal. The second input is a knowledge base of potential and generic attack vectors. VGA basically consists in a instantiation process between the specific vulnerability graph representation and the generic attack vectors. This process generates a security alert each time that a generic attack vector can be instantiated with the information in the vulnerability graph.

4.2 Vulnerability Taxonomy Characterization

A vulnerability taxonomy characterization will provide the vulnerability graph analyzer with the knowledge about the different existing vulnerabilities that the security practitioner applies when he does the component analysis, this knowledge is in turn used by the graph analyzer engine in order to know how the vulnerabilities might be related to middleware elements and attributes during the instantiation process. We started classifying 51 vulnerabilities found using FPVA, publicly listed on [6], with two different taxonomies. In addition we introduce the CWE taxonomy. The 51 vulnerabilities belong to six different middleware systems: Condor, SRB, MyProxy, gLExec, CrossBroker, and VOMS-Admin.

The seven kingdoms taxonomy is the vulnerability classification from McGraw et al. [15], which has been supported by Fortify Software Security Research Group. The taxonomy includes seven general categories: 1) Input Validation and Representation, 2) API abuse, 3) Security features, 4) Time and State, 5) Error handling, 6) Code quality, 7) Encapsulation, in addition to an extra category called Environment.

The whole taxonomy includes 86 different vulnerabilities. In this case, the classification has shown that using McGraw's taxonomy is neither easy nor clear enough to properly fit the 51 vulnerabilities because we have found that nine vulnerabilities belong to two different categories, two vulnerabilities belong to more than two different categories, and 35 vulnerabilities are not thoroughly ambiguous. Also no vulnerabilities fit the last two categories, *Encapsulation* and *Environment*, which are related to specific language or framework programming (e.g. J2EE, ASP.net).

PLOVER is the preliminary list of vulnerability examples for researchers, from Mitre Corporation [7]. Table 2 shows the plover taxonomy. PLOVER taxonomy includes around 300 vulnerabilities categorized in 28 classes, thus the likelihood of properly fitting the 51 vulnerabilities increases considerably. The classification of our vulnerabilities with PLOVER showed that 32 vulnerabilities belong to two different classes, three vulnerabilities belong to more than two classes, and 14 vulnerabilities are not thoroughly ambiguous. With PLOVER the 51 vulnerabilities fit into almost 50% of the whole taxonomy, because it includes a detailed and large classification structure from a diverse set of sources, including McGraw.

Commom Weaknesses Enumeration is an enhanced and improved effort for organizing vulnerability data that contributes with different perspectives (e.g. seven kingdoms, PLOVER, and other efforts), in a hierarchical fashion. CWE support multiple stakeholders with multiple views which serve to different purposes and audiences. We are going to move to the research view of the Commom

Buffer overflows, format strings, etc.	Structure and Validity Problems
Special Elements (Characters or Reserved Words)	Common Special Element Manipulations
Technology-Specific Special Elements	Path Traversal and Equivalence Errors
Channel and Path Errors	Cleansing, Canonicalization, and Comparison Errors
Information Management Errors	Race Conditions
Permissions, Privileges, and ACLs	Handler Errors
User Interface Errors	Interaction Errors
Initialization and Cleanup Errors	Resource Management Errors
Numeric Errors	Authentication Error
Cryptographic errors	Randomness and Predictability
Code Evaluation and Injection	Error Conditions, Return Values, Status Codes
Insufficient Verification of Data	Modification of Assumed-Immutable Data
Product-Embedded Malicious Code	Common Attack Mitigation Failures
Containment errors	Miscellaneous WIFFs

Table 2. PLOVER taxonomy

Weaknessess Enumeration (i.e. CWE-1000) because it is organized according to abstractions of software behaviors and the resources that are manipulated in those behaviors.

4.3 Knowledge Base

In our approach, we translate the combination of both the middleware and the vulnerability taxonomy characterization into a set of generic security configurations. Having previously defined key elements of VGA, we proceed to define a basic structure for the knowledge base (KB). A security configuration, can be built as follows:

Definition 4. Consider a set C of *security configurations*, then a configuration $c \in C$, can be:

- $c = m_i(a_j) \rightarrow t(a_j)$, simple.
- $c = m_1(a_1) \wedge m_2(a_2) \ldots \wedge m_i(a_j) \rightarrow t(a_1, a2, \ldots, a_j)$, compound.

Where $\forall m_i \in G : \{m_i \in V \vee m_i \in E\}$, and a_j is some attribute, property, or characteristic of m_i; and t is a vulnerability class (belonging to some known taxonomy T), that can be present in the system if c can be set.

5 Case Study: Through an Integer Overflow to a Denial of Service

This case study demonstrates that VGA concept, and its associates definitions, can be used to guide an analyst performing a source code inspection in finding vulnerabilities. In this case study based on CrossBroker, we assume that the vulnerability graph, and the knowledge base is already built. Let us proceed to analyze an attack vector from the CrossBroker vulnerability graph (Figure 3). The set V of components and resources in the vulnerability graph are {*SUBMIT, UAM, input_q, SA, RS, output_q, sandbox_dir, AL, CONDOR_G, LB, mysql, BDII, LRMS,*

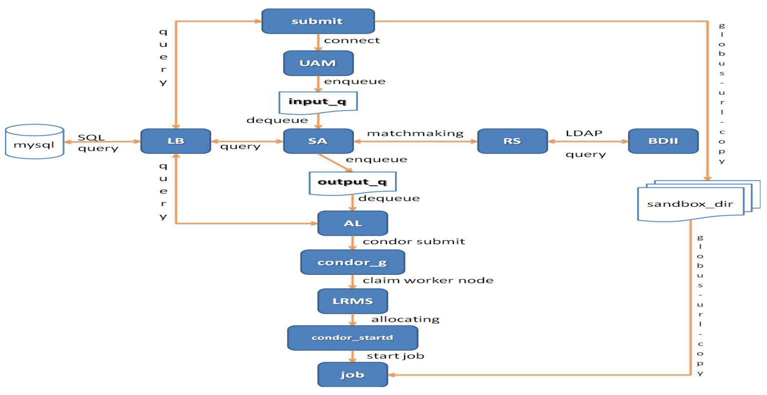

Fig. 3. CrossBroker vulnerability graph

CONDOR_STARTD, JOB}, and the set E of actions in the vulnerability graph are {*connect, globus-url-copy, enqueue, dequeue, matchmaking, ldap_query, enqueue, dequeue, query, query, query, sql_query, condor_submit, claim_worker_node, allocating, globus-url-copy, start_job*}, accordingly with the FPVA artifacts.

First, specific coordinates should be choosen from the middleware attack and impact surface, hence input and impact nodes are selected, in this case the "SUBMIT" and "MySQL" node. Second, having defined the input and impact nodes, the attack vector composition must be clearly depicted and recognized by the nodes and edges involved (Figure 4); In this case the SUBMIT, the LB, and the MySQL nodes, the "query" and "sql_query" edges compose the possible attack vector. Since nodes and edges were previously characterized accordingly to our proposal, the third step is to try instantiate the attributes, properties, and/or characteristics accordingly to the security configurations (generic attack vectors) described in the knowledge base.

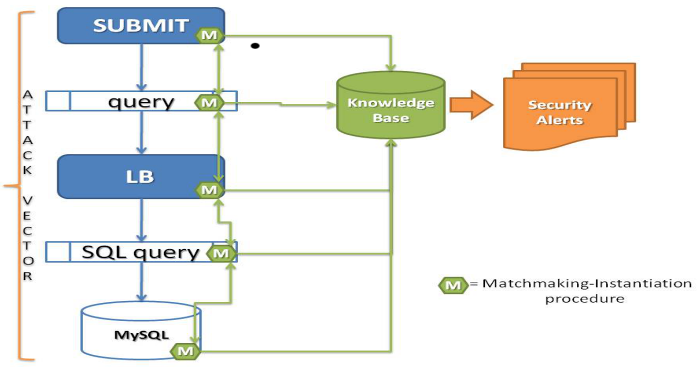

Fig. 4. CrossBroker attack vector

Instantiation process: for the CrossBroker attack vector,

- A) SUBMIT.[constraint] $\rightarrow$ [configuration]: Are big messages allowed?
- B) query.[state] $\rightarrow$ [configuration]: Is it a persistent connection?
- C) LB.[constraint] $\rightarrow$ [configuration]: Is the data in the correct format and size?
- D) sql_query.[state] $\rightarrow$ [configuration]: Is it a persistent connection?
- E) MySQL.[error_handling] $\rightarrow$ [configuration]: Are the error codes returned properly?
- A $\cap$ B = SUBMIT.[constraint] $\cap$ query.[state] $\rightarrow$ [configuration]: Has the user requested a timeout period to try to finish and release the connection even if the message has not been transmitted?
- B $\cap$ C = query.[state] $\cap$ LB.[constraint] $\rightarrow$ [configuration]: Were the data transmitted correctly and completely within the right time?
- C $\cap$ D = LB.[constraint] $\cap$ sql_query.[state] $\rightarrow$ [configuration]: Has the component requested a timeout period to try to finish and release the connection even if the message has not been transmitted?
- D $\cap$ E = sql_query.[state] $\cap$ MySQL.[error_handling] $\rightarrow$ [configuration]: Are the code and the query correctly returned and properly handled within the right time?

Fourth, the graph analyzer engine then should return a the set of alerts concerned to the security configurations which were instantiated by the different current values of the attributes, properties, and/or characteristics. In this case, when a submit request happens on CrossBroker, it is possible that *submit.[constraint]* allows either a big message or attachment, then *query.[state]* becomes persistent and the data starts transmitting to the LB component to save information about job status, but the *LB.[constraint]* trust that data is being properly transmitted based on the message header previously received. The LB component will try to register on MySQL component the job status, but the database returns an unexpected error due to an incorrect size of the data transmitted at the begin of all, hence *sql_query.[state]* remains established and the *MySQL.[error_handling]* contains an unexpected code because the LB component is still trying to write on the database, in addition to blocking the next incoming requests by not releasing the link, therefore becomes finally in a denial of service by an integer overflow in the size message difference and the improper handling of the unexpected errors.

6 Related Work

Vulnerability Assessment of middleware systems is a field that has attracted the interest of both research and commercial communities, due to the increasingly rapid growth of the use of distributed and high performance computing, as well as the increasingly rapid growth of threats. Our VGA approach is related to the Open Vulnerability and Assessment Language [8] project, and to the vulnerability cause graphs [11].

The Open Vulnerabilities and Assessment Language (OVAL) is an international, information security, community standard to promote open and publicly available security content, and to standardize the transfer of this information across the spectrum of security tools and services. OVAL includes a language used to encode system details, and an assortment of content repositories held throughout the community. The language standardizes the three main steps of an assessment process: 1) representing configuration information of systems for testing; 2) analyzing the system for the presence of the specified machine state (vulnerability, configuration, patch state, etc.); 3) and reporting the results of this assessment. The repositories are collections of publicly available and open content that utilize the language. OVAL is based primarily on known vulnerabilities identified in Common Vulnerabilities and Exposures (CVE) [3], a dictionary of standardized names and descriptions for publicly known information security vulnerabilities and exposures developed by the MITRE Corporation.

In contrast to OVAL, our effort is not based on the specific CVE vulnerabilities, instead we claim that VGA approach works with CWE classification and with nonspecific software vulnerabilities, also VGA approach is based on FPVA stages, thereby gathering more meaningful information about the assessment process.

Vulnerability Cause Graphs is based on a thorough analysis of vulnerabilities and their causes, similar to root cause analysis. The results are represented as a graph, which Byers et al. [11] called vulnerability cause graph. Vulnerability cause graphs provide the basis for improving software development best practices in a structured manner. The structure of the vulnerability cause graph and the analysis of each individual cause are used to determine which activities need to be present in the development process in order to prevent specific vulnerabilities. In a vulnerability cause graph, vertices with no successors are known as vulnerabilities, and represent classes of potential vulnerabilities in software being developed (analysis may start with specific instances of known vulnerabilities). Vertices with successors are known as causes, and represent conditions or events that may lead to vulnerabilities being present in the software being developed. In our case, the most noticeable difference is that we want to know whether a vulnerability may exist and why, instead Byers' work knows the vulnerabilities and looks for their causes.

7 Future Work & Conclusions

In this paper we have described the vulnerability graph structure and the vulnerability graph analyzer to guide security practitioners during a source code assessment to identify effectively where and why vulnerabilities might be possible. There is a lot of tasks which have to be done before vulnerability graphs and VGA can be applied as effectively as we claim to grid security. The most relevant tasks we have noticed are the following:

Graph Representation: A vulnerability graph must be able to depict the middleware composition in a suitable and easy way. *Vulnerabilities Characterization:* A complete characterization of a set of vulnerabilities is required in order to check if the knowledge base is good enough to provide the vulnerability graph analyzer with the proper security configurations. *Attack Vectors:* With the improvements on the vulnerability graph and the middleware characterization, the vulnerability graph analyzer engine has to be able to construct meaningful attack vectors with a well-defined algorithm. *Instantiation process:* In the graph analyzer engine the instantiation is the most important process, it has to be clear and easy to deploy, indeed it must be based on a kind of weighted value for the middleware elements. Because all the security configurations (the knowledge base) can not be applied to all middleware elements in the same way. In addition to the vulnerability graph and VGA definitions, we have proposed a middleware characterization along with a formal definition of a knowledge base of security configurations, having improved our previous work with a meaningful approach. Finally, a case study has been introduced, where all definitions and elements have been applied, showing that it is possible to reduce the gap between the first stages of FPVA and the Component Analysis one.

References

1. Condor Project. http://www.cs.wisc.edu/condor.
2. Coverity Prevent. http://www.coverity.com.
3. CVE - Common Vulnerability and Exposures. The Mitre Corporation. http://cve.mitre.org/.
4. CWE - common Weakness Enumeration. The Mitre Corporation. http://cwe.mitre.org/.
5. Fortify Source Code Analyzer. http://www.fortify.com.
6. MIST Group: Middleware security and testing web site. http://www.cs.wisc.edu/mist.
7. Mitre - The Mitre Corporation. http://www.mitre.org/.
8. OVAL - Open Vulnerability and Assessment Language. http://oval.mitre.org/.
9. SRB - Storage Resource Broker. http://www.sdsc.edu/srb/.
10. PLOVER - Preliminary list of vulnerability examples for researchers. http://cwe.mitre.org/documents/sources/plover.pdf, March, 2006.
11. D. Byers, S. Ardi, N. Shahmehri, and C. Duma. Modeling software vulnerabilities with vulnerability cause graphs. In *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, pages 411 –422, 2006.
12. E. Fernandez del Castillo. *Scheduling for Interactive and Parallel Applications on Grid.* PhD thesis, Universitat Autònoma de Barcelona, 2008.
13. J. Kupsch and B. Miller. Manual vs. automated vulnerability assessment: A case study. *International Workshop on Managing Insider Security Threats*, 469:83–97, June 2009.
14. J. Kupsch, B. Miller, E. Heymann, and E. Cesar. First principles vulnerability assessment, mist project. Technical report, UAB & UW. http://www.cs.wisc.edu/mist/papers/VA.pdf, September 2009.
15. G. McGraw, K. Tsipenyuk, and B. Chess. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security and Privacy*, 3:81–84, 2005.
16. J. D. Serrano Latorre, E. Heymann, and E. Cesar. Developing new automatic vulnerability strategies for hpc systems. In *Latinamerican Conference on High Performance Computing - CLCAR*, pages 166–173, August 2010.
17. J. D. Serrano Latorre, E. Heymann, and E Cesar. Manual vs automated vulnerability assessment on grid middleware. In *Actas del XXI Jornadas De Paralelismo, JP2010. Celebrado en el marco del III Congreso Espanol de Informatica - CEDI.*, Sep 2010.