Handling Undiscovered Vulnerabilities Using a Provenance Network

Amrit'anshu Thakur(1,2), Dr. Rayford Vaughn(1) and Valentine Anantharaj(2)
1. Department of Computer Science & Engineering
2. GeoResource Institute, High Performance Computing Collaboratory
Mississippi State University
Starkville, MS 39759
amrit@gri.msstate.edu, vaughn@cse.msstate.edu, val@gri.msstate.edu

***Abstract*—This paper elaborates on a novel approach at preventing exploits from vulnerabilities which remain uncovered during the testing phase of a system's development lifecycle. The combination of predicted usage patterns, a Provenance network model and a clustering methodology provide a secure failure mechanism for both known and unknown security issues within the system. The paper also addresses of the requisite supporting infrastructure and deployment issues related to the model. The idea is to approach the growing problem of newer and more complex vulnerabilities in an ever more intricate and vast set of systems using a generic software state mapping procedure for recognizable (and thus the complementary unrecognizable) patterns to judge the stability at each step in an operation sequence. Thus abstracting these vulnerabilities at a higher level provides us a generic technique to classify and handle such concerns in the future and in turn prevent exploits before a corrective patch is released.**

I. Introduction

Most traditional software engineering lifecycles have a significantly large testing phase towards the end. From a conventional standpoint, this phase was primarily aimed at testing the 'acceptable' functionality of the system. Requirements were traced from design and development through atomic partitions of test cases. Analysis was limited to the deployment of requested services and their associated quality parameters. There was a distinctive lack of an early and in depth scrutiny of design, code or even test cases. Bug fixing commenced well into the testing phase and there was little or no attention directed towards potential security concerns. All that has changed in more recent years. There is a growing awareness of the criticality of considering security issues early on and integrating them as part of the software development lifecycle. Modern standards of development recommend security considerations not as a bolt on feature but as an integral part of the development process [1]. More emphasis is placed on the creation of standards, reviews, validation, practices, documentation and building of a secure component from scratch rather than worrying about it in the end [6][7]. The new paradigm of systems development has resulted in better security. Current requirement models which consider multiple perspectives [15] from the beginning have resulted in the creation of methods which represent different customizable viewpoints [14] such as the system's security, from the very onset of its development rather than a break fix approach later on. The functionality and pervasiveness of complicated interconnected systems is on the rise and unfortunately so are the safety issues. Expert evaluation of thousands of lines of interdependent code is a costly, tedious and often ineffective proposition. Automated procedures are either too specific or too unproductive without manual intervention. Another problem is the constantly evolving nature of vulnerabilities. It is extremely difficult (if not impossible) to derive a universal set of potential vulnerabilities and variants. Thus it is wise to devise methods of generic security applicability which are not only more effective but also mature in practice while maintaining a wider audience base. We suggest a method of addressing known and unknown vulnerabilities using concepts of provenance and pattern matching. Data Provenance is a methodology to capture the steps of derivation of an end data product and incorporating it with the final dataset thus making it more self describing [8] and semantically intelligent. This also helps in the appraisal of trust or quality of the information in the dataset [2][9]. A provenance based trust network created during a systematic testing process is used as a reference point for the system's usage. This enables handling of known and unknown exceptions which could be potential threats to the system. The concept of provenance has been successfully used in several complex domains (which require a comprehensive semantic traceability mechanism) such as health sciences, chemical industries and scientific computing [9]. Its use in a software engineering related security area could be somewhat of a novelty. Another idea used here and borrowed from an external domain is that of automated clustering based on individual cluster characteristics. The motivation is to put into place some form of clustering technique where 'most similar' candidates appear in the same group [3] and this is performed in a mechanized fashion based on the attributes these candidates possess. Another step is manually aided interpolation to fully define a cluster's elements given its upper and lower bounds. The cluster here represents the program statements. The attributes suggested in this paper are a recommendation and can be modified to suit the needs of the environment. They should however should contain enough information to differentiate between dissimilar clusters based on their attributes. This will become clearer with a case study presented towards the end of this paper.

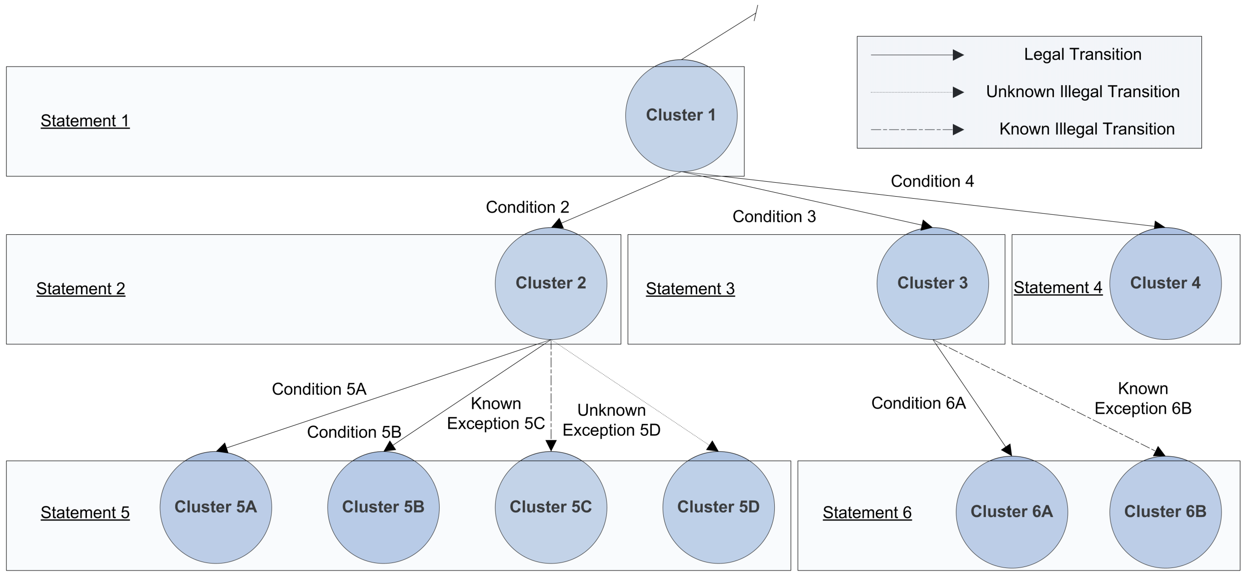


Fig. 1. Sample Provenance Network Instance

II. Process Components

The Provenance Network is based on the concept of establishing a traceable path of the usage in the application domain during the testing phases. Think of it as an electronic notebook [4] which provides an automated mechanism for recording the usage patterns after classification and filtering of noise. This kind of setup becomes critical when a distributed or shared environment is used in a process [4][5]. The network is built during the testing phases and later used to govern legal usage patterns for the system in its live operational environment. The network is modeled in the form of a tree (Fig.1). The nodes are program statements which could be potentially vulnerable to malicious intent. Legitimate examples of this could be expressions which use externally acquired variable values, memory allocation or handling procedures, statements which read, write, update, modify, delete external data entities, statements which deal with network or I/O ports, deal with security related issues such as authentication, publish or retrieve values to or from a web interface, run or relate to native OS systems commands or directory structures, compete for resource allocation etc. In case there exists an uncertainty on which statements to select, considering the entire program will also produce the same results but with a less optimized performance. The nodes are connected by links which in a downward traversal represent the sequence of statements (nodes) in their order of execution. The links have weights assigned to them which are integer values. The likelihood of traversal by a usage pattern which incorporates the link is directly proportional to the weight of the path i.e. a higher positive value for a link connecting two nodes shows that there exist usage patterns with the link in their path more often than other links. On the other hand a greater negative value indicates an identified threat or vulnerability in the programming confirmed by a greater number of usage patterns which identified the hazard. Please note that we will only concentrate on the identification of vulnerabilities and not on correctional procedures, therefore we will assume that the vulnerabilities remain within the system after installation. The other components required to implement the provenance oriented vulnerability model are the modified unsupervised attribute based clustering procedure and an assisted interpolation engine. As mentioned before, every node contains a set of attributes. Since the set of nodes represent expressions or program statements, attributes are naturally defined at the same granularity level. Also the characteristics elicited could be (should be) hierarchical for a better understanding of the attribute composition. A key point which we consider most important to cluster statements is the ’expanded form’ of the statement. This means that the expressions or statements which have notational variables or syntactical constructs which take form during runtime have to be captured for clustering analysis at runtime itself. This ensures that the clustering is based on real instances of those attributes and not just abstract ranges of usage. A simplified example of this could be considering the values of an integer variable type that is actually recorded and combined with an integer based Boundary Value Analysis [10] for clustering and test cases instead of the latter only. Even within the testing phase, the sources of the boundary values, real values and potential security threat inputs act as different sets of ’testers’. Other attributes such as relative stack position, timestamps, recursion calls etc. can be part of an the attribute based clustering scheme. The assisted interpolation engine helps generate missing values to fully populate a cluster after it has been identified by a union of boundary and real instance values. The engine uses various techniques to enumerate all discrete values which can exist in a cluster and requires minimal manual intervention to do so. Fully automated engines are however not recommended until some maturity in the process is acquired such that all types of statements can be handled for interpolation. However to achieve this maturity, it is advised that that an interpolation repository be maintained which has an effective expression analysis and parsing scheme in place. A sample of expected results from such an interpolation are shown in the case study and judging by the customized instances (and their prospective values), an interpolation scheme can be devised by the user. This however can also be done manually by explicit specification of the range of acceptable or expected values and some interpolated instances in that range. A Provenance Network's instance is presented in Figure 1. The cluster names are generic for the purpose of a universal explanation of concepts. Consider the example to be a sub-network of a bigger Provenance Network. Cluster '1' represents the entry point for this partition. As can be seen from Figure 2, cluster formation is based on attributes of individual cluster instances. A clustering algorithm is used to execute the formation, preceding which a comparison scale based on the instance characteristics is devised (either empirically or heuristically) to ensure a fair comparison between the candidates or instance value. Another point to note is that though trees are conventionally used to represent hierarchical structures, the nodes here represent statements connected in the order of a top-down program flow. The attributes can be represented in a hierarchical structure as this facilitates a better understanding of the comparison algorithm to be used (Fig. 2). The Interpolation Engine executes each cluster to fully populate a Kleen's clousure for that instance.A usage pattern here is defined as a distinct path from one cluster to another. A recorded pattern thus satisfies all the intermediate transitions (legal known,illegal known or illegal unknown). An example should clarify this further. Let us suppose there exists a Usage Pattern recorded as UP_{1-5}=Clusters$\{1,2,5A\}$. Thus the existence of expanded expressions of statements 2 and 5 which satisfy clustering conditions 2 and 5A respectively is confirmed. If we consider single sequential transition usage patterns then $\mathrm{UP}_{1-2}=\{1,2\}$ transition between statement 1 and 2 is represented by clustering condition 2. Similarly there exists another usage pattern $\mathrm{UP}_{2-5}=\{2,5A\}$ where the transition is between statements 2 and 5 is also caused by an attribute based validation criteria of clustering specified in condition set 5A. Now let us consider the a use case [11] where the usage path takes half of UP (i.e. UP_{1-2}) by transforming statement 2 into cluster 2 but then it executes an attempt to exploit a known security vulnerability in statement 5 by the metamorphosis of the statement into cluster 5c instead of the normal usage pattern 5a. Since 5c was a known vulnerability which was uncovered and added to the provenance networks, it is easy to capture and prevent its execution. Now consider statement 5's transformation at runtime to an expression which exploits the vulnerability in it and is unknown to us. However the clustering algorithm is neutral to specificities of exploits and detects any attribute changes to form a new cluster altogether. As long as the metamorphosis of statement 5 doesn't match any existing cluster, a new cluster called 5D is formed to classify the attributes of the expanded expression as an unknown exploit or an untested expression - either way, an illegal unknown expression is prohibited from execution.

III. Usage Patterns

There are three distinctive disadvantages to this approach. First and the perhaps the most trivial of the three is the incorrect classification of untested expression patterns (i.e. clusters) as a new illegal one. This can be solved by ensuring a rigorous testing procedure which incorporates all patterns (if not all instances of the pattern) for each expression specific to the application domain. A counter logic to this is if there is a specific range or type of expression we have not tested, then it is best to prevent its execution in a live environment until sufficient test coverage has been achieved (with the help of feedback logs and post installation tests). The second disadvantage is the classification of a known vulnerability as a known legal expression. The strength of the model lies in the fact that both solutions are known and therefore fine tuning of the classification should be able to differentiate them. Last and perhaps the only relevant drawback is the identification of an unknown vulnerability as a known legal expression. A possible solution to this is to construct the patterns for the legal expressions as tight as possible to ensure minimal room for unwarranted use. A general solution to all of the above and to ensure maximum probability of a vulnerability being caught is to study all the newly discovered unknown vulnerability clusters and classify them as legal or illegal in a timely manner, to update the usage network.The provenance network based security model (Fig. 3) begins with usage testing by the developer group. The choices of Usage Patterns this group makes will initiate the construction of the provenance network. There is also the assumption that the developers will test and use the system in accordance to the specifications, which is a formal document produced in the lifecycle and is closest to the funct ional (and non-functional) requirements expected by the user. Therefore the weight assigned to the links (i.e. conditions which lead to the formation of different clusters for a statement) will be highest when the developer group creates or reinforces it by exercising the path one or more times to reach the common usage points in the program flow. This weight is common for all developer actions and is proportionally higher to the other single action weights of the non-developer groups which test the system afterwards. Assigning weights to links will not have any impact on the binary decisions of whether to execute a certain link or not. A positive value will indicate execution (legal expressions) and a negative value will prevent the same (illegal known expressions). Execution is also prevented for a zero value link (discovered unknown illegal expression) till the trust on that link is ascertained at a later stage through testing. The values on the link only play a part in establishing trust on its usage. Thus higher the value on a link, higher the trust. The only links we don't know much about are the links which represent unknown expressions and are therefore of zero value. We prefer to treat them as unsafe, unnecessary or untested thus prevent execution.

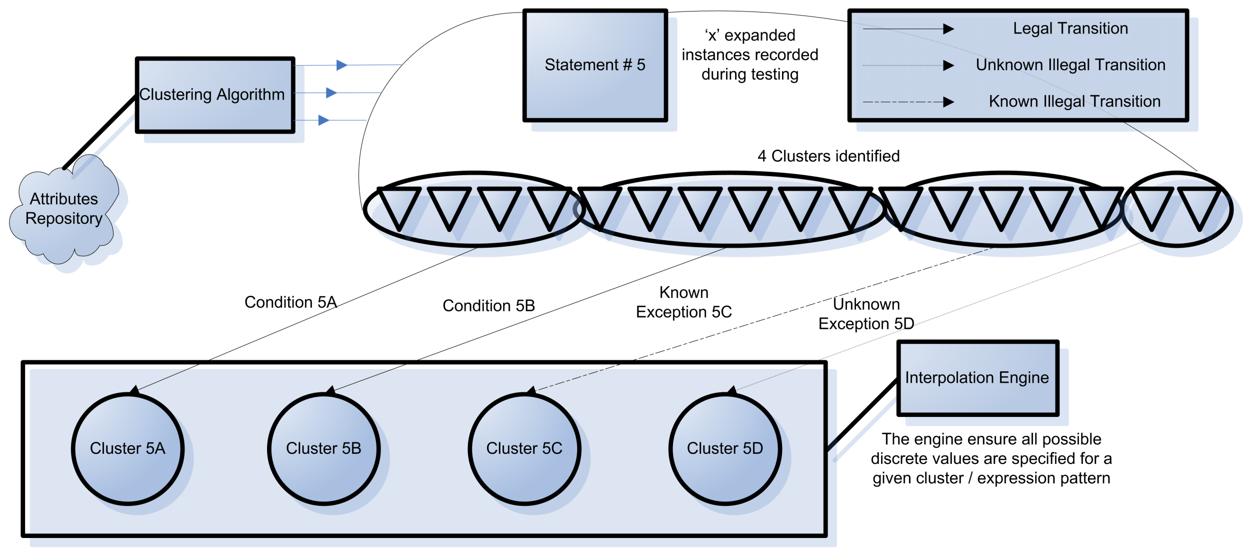


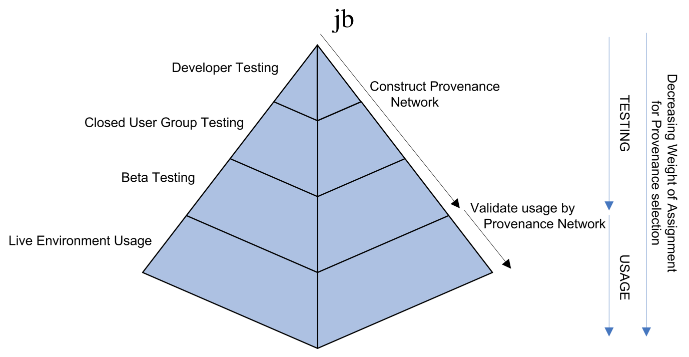
Fig. 2. Clustering for Statement No5

IV. Process Flow

The second partition of the testing phase is repeating the usage based testing of the system to develop the provenance network further. This time it's done by a closed user group and under supervision. The group may perform the tests from various perspectives [12] to bring out Usage Patterns of different sets of target audiences for the system. There is also the option of forming 'n' teams [13] and performing a specialized role such as end user, security expert, maintenance engineers or QA personnel which develop the provenance network further on (from where the developers left off). A point to note here is that theoretically the provenance network can never achieve a universal completion of all possible usage. The idea is to reduce the number of unknown expressions despite the fact that we should be able to handle them given their unique clustering signature. The next phase of testing is performed by a larger tester group such as an Alpha or Beta testing. Feedback from this sub-phase should be most helpful to form new cluster nodes or links and reaffirm the ones already made. Another aspect of this form of testing is that its probably the closest and therefore the last stage of testing before final deployment. Therefore fine tuning cluster expression recognition techniques are performed in a decisive and extensive manner during this sub-phase. The three sub-phases of testing are complete after this and the network can be frozen for a live environment usage hereafter. The final Provenance network will thus not only tell us what to run and what not to but should also give us a quantified comparison of trust in a specific usage pattern. The idea is to bu

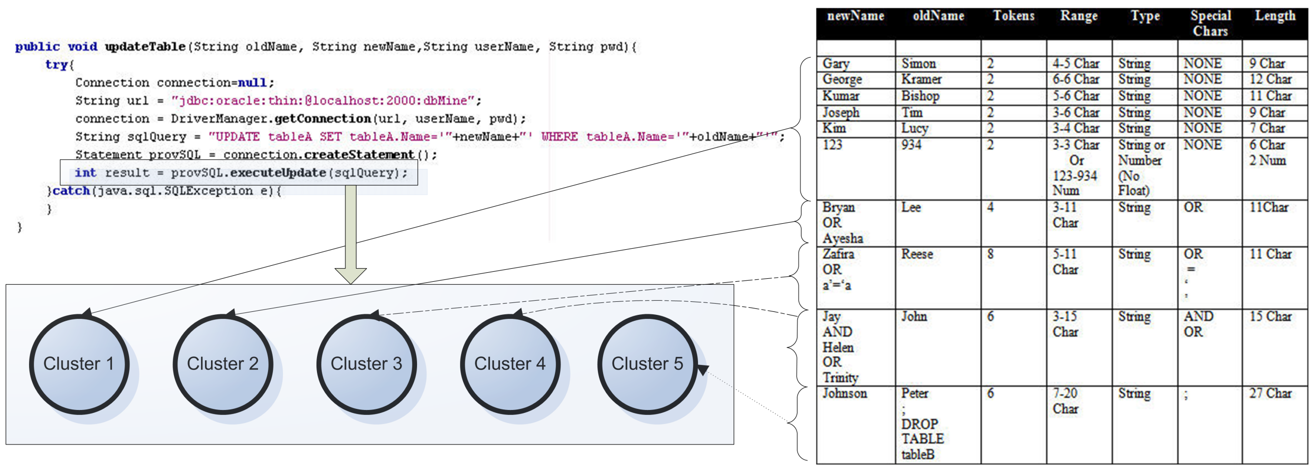
V. Case Study

We will look at a sample clustering procedure with real instances of input for a potential SQL injection attack. Before we proceed any further, a few points should be noted. The cluster scheme represented is only a subset of a much larger and eventually mature scheme that can be devised for a real provenance network. The number and type of attributes considered here might not be optimum and fine tuning will require better classification schemes. The sample taken here is a set of ten expanded statements whereas in a real system there would be many more i.e. the number (and thus the weight) of patterns which represent normal and intended usage will be much higher for complex provenance networks. The network's ability to capture unknown patterns depends on its ability to understand the known patterns very precisely. The example (Fig. 4) represents a sample of Java code which is susceptible to an SQL injection attack. Though this kind of threat can be neutralized using regular expressions, the idea is to build a generic pattern sensitive infrastructure which cannot only go beyond a specific attack type but also tackle ones we do not know enough (or anything) about. As mentioned before, while constructing the network we have an option to either consider all statements or only those which could have a security impact. This is an example of a statement from the latter kind's set. The table represents ten instances of expanded expressions at runtime values provided in ten distinct test



jb
Developer Testing
Closed User Group Testing
Beta Testing
Live Environment Usage
Construct Provenance Network
Validate usage by Provenance Network
TESTING
USAGE
Decreasing Weight of Assignment for Provenance selection

Fig. 3. Progression Pyramid



```
public void updateTable(String oldName, String newName,String userName, String pwd){
    try{
        Connection connection=null;
        String url = "jdbc:oracle:thin:@localhost:2000:dbMine";
        connection = DriverManager.getConnection(url, userName, pwd);
        String sqlQuery = "UPDATE tableA SET tableA.Name='"+newName+"' WHERE tableA.Name='"+oldName+"'";
        Statement provSQL = connection.createStatement();
        int result = provSQL.executeUpdate(sqlQuery);
    }catch(java.sql.SQLException e){
    }
}
```

newName	oldName	Tokens	Range	Type	Special Chars	Length
Gary	Simon	2	4-5 Char	String	NONE	9 Char
George	Kramer	2	6-6 Char	String	NONE	12 Char
Kumar	Bishop	2	5-6 Char	String	NONE	11 Char
Joseph	Tim	2	3-6 Char	String	NONE	9 Char
Kim	Lucy	2	3-4 Char	String	NONE	7 Char
123	934	2	3-3 Char Or 123-934 Num	String or Number (No Float)	NONE	6 Char 2 Num
Bryan OR Ayesha	Lee	4	3-11 Char	String	OR	11Char
Zafira OR a'='a	Reese	8	5-11 Char	String	OR = ' ;	11 Char
Jay AND Helen OR Trinity	John	6	3-15 Char	String	AND OR	15 Char
Johnson	Peter ; DROP TABLE tableB	6	7-20 Char	String	;	27 Char

Fig. 4. Case Study

cases. To simplify the process , we have omitted the inclusion of groups and corresponding weights to form links between the clusters. If observed carefully, one will note that the statement prone to the attack is similar in all instances at runtime minus the user input. Thus it makes logical sense to analyze the user inputs for variables 'oldName' and 'newName' only. The table represents ten such values. Cluster '1' identifies all input with two tokens of 'String' type each and no special characters or delimiters. The upper and lower bounds on the string sizes will be six and twelve characters respectively. Do note that boundary value analysis [10] and a dramatic increase in the number of times this cluster is tested will identify a more truthful pair of boundaries. An example of how this technique can be generic is that in an extensive testing scheme the boundaries should incorporate all 'realistic' values. Thus a potential buffer overflow attack expression will be recognized as a new cluster and prevented from execution. Cluster '2' on the other hand identifies four tokens of 'String' type with an 'OR' delimiter. Suppose while testing this is declared legal and using the interpolation engine any number of 'OR' instances are declared legal with alternating unique values. Thus the network identifies this as a query terminates the first and only intended query and starts a new one to remove an entire table from the database). However the change in pattern with the inflation of the size and the presence of a new type of special character makes the algorithm suspicious enough to classify this in a new cluster. The cluster may not essentially be an exploit in reality but is marked as an illegal expression and prevented from being executed. Cluster '5' can be further analyzed in the post installation feedback logs and classified as a known attack in the evolution of the provenance network. The use of provenance network is thus heavily dependent on its semantic depth and scale. The more the number of test cases, the more fine tuned the network becomes for the system. An automated infrastructure such as the one mentioned above should ensure the capture of relevant elements at the right granularity. Not only does it permit us to run the system in a tried and tested set of channels but it also generates confidence levels in system components. It also indicates when the testing of a system should cease. When the network stops growing and all of cluster recognition becomes a redundant exercise then it can be safely assumed that almost all uses of the system are tested and documented to tally against in an installed environment. The results from the examples indicate that the maturity of the network and all supporting components is critical in this technique. Given the above infrastructure, provenance networks of usage formed during the various testing phases could become critical in future vulnerability prevention and discovery.

Acknowledgment

The authors would like to thank Dr. Edward B. Allen for his continuous guidance during the class 'CSE 8990 Secure Coding - Spring 2007' at the Department of Computer Science and Engineering, Mississippi State University. I would also like to thank the GeoResource Institute for its support in this endeavor. This research was supported in part by the Applied Sciences Program of the National Aeronautical and Space Administration via the Mississippi Research Consortium grant USMMRCSSC1216200565D.

References

[1] G. McGraw and J. Viega, *Building Secure Software: How to Avoid Security Problems the Right Way,* Addison-Wesley Professional, Boston, MA, Sep 2005.

[2] N. Aamir and A. Pervaiz, Study of Data Provenance and Annotation Model for Information Reliability Suggested for Pathological Laboratory Environment in Pakistan, *Proceedings of the First International Conference on Information and Communication Technologies*, Karachi, Pakistan, Aug. 2005.

[3] A. Wai-Ho, C.C. Keith, K.C. Andrew and W. Yang, Supporting multi-perspective requirements engineering, *Attribute Clustering for Grouping, Selection, and Classification of Gene Expression Data*, Issue 2, vol. 2, Apr. 2005.

[4] T. Talbott, M. Peterson, J. Schwidder and J.D. Myers, "Requirements for Requirements Engineering Technique," *Adapting the electronic laboratory notebook for the semantic era*, Proceedings on the 2005 International Symposium on Collaborative Technologies and Systems, Saint Louis, Missouri, May. 2005.

[5] S. Rajbhandari and D.W. Walker, *Incorporating Provenance in Service Oriented Architecture*, International Conference on Next Generation Web Services Practices, Seoul, Korea, Jul. 2006.
[6] P. Devanbu and S. Stubblebine, *Software Engineering for security: a Roadmap*, Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, Jun. 2000,
[7] M.D. Bryans, *Security Engineering in an Evolutionary Acquisition Environment*, Proceedings of the 1998 workshop on New security paradigms, Charlottesvillage VA, Sep. 1998,
[8] S. Munroe, S. Miles, L. Moreau, J. Salceda, *New architectural paradigms: PrIMe: a software engineering methodology for developing provenance-aware applications*, Proceedings of the 6th international workshop on Software engineering and middleware, Portland, Oregon, Nov. 2006.
[9] R. Bose and J.Frew, *Lineage Retrieval for Scientific Data Processing: A Survey*, ACM Computing Surveys, issue 1, vol. 37, Mar. 2005,
[10] M. Ramachandran, *Testing Components Using Boundary Value Analysis*, Proceedings on the 29th EuroMicro Conference, Belek-Antalya, Turkey, Belek-Antalya, Sep. 2003.
[11] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd Edition,* Addison-Wesley Professional, Boston MA, Sep. 2003.
[12] V. Basili, F. Schul and I. Rus, *How Perspective Based Reading can Improve Requirements Inspection*, IEEE Computer, issue 7 vol. 33, July 2000, pp.73 - 79
[13] E.H. Sibley and E. Editor, *N-Fold Inspection, a Requirements Analysis Technique*, Communications of the ACM, issue 2, vol. 33, Feb. 1990
[14] A. Thakur, R. Vaughn and V. Anantharaj, *On the Same Page : Building Stakeholder Consensus on Requirements*, Common Ground Publishing, Design Principles and Practices: An International Journal, 2008
[15] O. Laitenberger, K. El Emam and T.G. Harbich, *An internally replicated quasi-experimental comparison of checklist and perspective based reading of code documents*, IEEE Computing, IEEE Transactions on Software Engineering, Volume 25, Issue 5, pp. 387-421, May 2001