An Efficient Measurement of Object Oriented Design Vulnerability

Alka Agrawal   Shalini Chandra   Raees Ahmad Khan
Department of Information Technology
Babasaheb Bhimrao Ambedkar University
Lucknow, India
alka_csjmu@yahoo.co.in,{nupur_madhur, khanraees}@ yahoo.com

Abstract—As foundation is to building, design phase is to software because in this phase ideas start converting into actuality. Effects of vulnerabilities introduced in this phase manifest with ongoing software life cycle. Even detection and correction of these vulnerabilities seem to be difficult in absence of any efficient method or automated tool. This paper suggests an approach to identify vulnerable classes in object oriented design. The method proposed also investigates whether transitive nature of Inheritance contributes to propagation of vulnerabilities from one class to another or not. An algorithm for computing Vulnerability Propagation Factor (VPF) has been developed, which measures number of vulnerable classes because of the vulnerability in some classes of an object oriented design.

Keywords- Design Phase; Inheritance; Vulnerability Propagation; Vulnerability Propagation Factor;

I. INTRODUCTION

Inadequate requirement specification, insecure design, and poor coding practices may introduce vulnerabilities in software. Vulnerable software can be exploited at any point of time by attackers leading to undesired consequences [1]. Security of the software is measured in terms of number of existing vulnerabilities [2]. Gary McGraw characterizes vulnerabilities into two classes one is design time vulnerability called flaw and the other is implementation time vulnerability called bugs [3].

Literature survey reveals that detection and correction of vulnerabilities in design phase of development life cycle are important as making changes in the phase are much easier than to make them at the end [4]. Plenty of tools and approaches are available to detect and remove implementation time vulnerabilities. ITS4 [5], FLF Finder [6], MOPS [7], Penetration Testing [8] are to be named a few. But, approaches for detection and removal of vulnerabilities at design level [9, 4] are very few. In addition to it, almost negligible work has been done to address security issues in design phase of object oriented system [10].

Undoubtedly, vulnerability detection and correction processes in design phase are still almost manual which consume more time, resources and efforts [11]. Even tools to detect and remove design level vulnerabilities do not exist till date [12]. This leads the necessity to develop a tool to minimize vulnerabilities at design level. The work is a step towards the same. A vulnerability measure, Vulnerability Propagation (VP), has been proposed. The VP computes number of classes affected because of a vulnerable class in an Inheritance hierarchy of an object oriented design. On the basis of VP, Vulnerability Propagation Factor (VPF) for the design is calculated.

The rest of the paper is organized as follows: Next section discusses vulnerabilities and incidents due to vulnerabilities. Section 3 discusses background of the work. Section 4 suggests an approach to identify vulnerable classes. Section 5 defines Vulnerability Propagation. Section 6 discusses Inheritance and its role in propagation of vulnerability. Section 7 provides an algorithm to compute Vulnerability Propagation Factor (VPF) of an object oriented design with example. Succeeding sections discuss interpretation, advantages and limitations. Paper concludes in section 10.

II. SOFTWARE VULNERABILITY

Vulnerabilities are the weaknesses of a system which could be exploited at any point of time and tend to violate security policies. Many software failures causing loss in terms of money and reputation were reported because of the vulnerability. The loss due to Code Red Worm has been estimated to $2.6 billion and due to Nachi Worm, operations at Air Canada and CSX railroad were affected very badly [13]. Efforts in this direction have been started but the statistics shows that the problem is still growing [14]. Total 4129 vulnerabilities were reported to CERT Coordination Center in 2003, this was 70 percent increase over 2002 and almost fourfold increase since 2001 [15]. Currently, the rate of reporting vulnerabilities to CERT Coordination Center is 4,000 per year [16].

Though, most of the organizations are using antivirus and firewall, but these are also of little use. In 2004, CSI/FBI Computer Crime and Security Survey concluded that 98 percent of the participants were using firewall and 99 percent were using antivirus, still most of them suffered virus attack, system penetration and unauthorized access to information [17]. This shows that some other approach is also needed in addition to firewall and antivirus.

Learning from the different incidents of security failure, it is now believed that instead of wondering for
nothing, root cause of the security problem, the vulnerability, should be mitigated early in the development [17, 4].

III. BACKGROUND

By the occurrence of various security related incidents, it has been realized that application security is not enough to fight against vulnerabilities and threats [15, 17]. Because of this, researchers and practitioners are repeatedly advocating the need for vulnerability detection and correction at each phase of Software Development Life Cycle (SDLC) [18, 15, 19]. Wide range of tools/techniques/approaches for detection and correction of vulnerabilities have been developed [5-8, 20]. But unfortunately, they all are of later stage tools especially of coding and implementation phases. Literature on software vulnerability reveals that a little amount of effort has been made for detection and correction of design vulnerabilities. Moreover, automated tools for vulnerability detection in design phase are missing [12]. This strengthens the need for developing a tool to minimize vulnerabilities in design.

It has been noticed that researchers are developing Vulnerability Discovery Models (VDM). These models use the fact that vulnerabilities are the subset of defects and are based upon existing software reliability growth models [21]. Some of the vulnerability models include Anderson Thermodynamic Model (AT), Alhazmi-Malaiya Logistic Model (AML), Rascola Linear Model (RL), Rascola Exponential Model (RE), and Logarithmic Poisson Model (LP) [22]. These models operate on historical data including systems release date, system usage information and the date on which vulnerability is discovered [21]. These existing models are applicable in implementation stage and are estimating discovery of future vulnerabilities on the basis of historical data. This gives an idea of developing a model to predict vulnerabilities early before implementation. Moreover, the prediction mechanism should not be dependent on the previous history of data. The fact motivated the authors developing a methodology to be implemented in design phase to discover and mitigate vulnerabilities there itself.

Similar to defect density, a measure termed Vulnerability Density is introduced by researchers [23]. It is calculated by the number of vulnerabilities per unit of code size. Also, another run time metric, an attackability metric [24], measuring attack surface of software, is developed. With the help of these metrics vulnerability and attackability of two versions of the same software can be compared. This fact gives strength to the authors developing such a measure to be used to compare two design hierarchies early in the development.

Further, the following facts strengthen the idea of developing efficient measures for object oriented design vulnerabilities for which the authors have approached:

- Some mechanism is required on urgent basis to compare and comprehend software project in order to identify less attack prone software.
- It is almost impossible to completely safeguard the software; the only effort one can make to secure software is to reduce its vulnerability [25].

IV. IDENTIFYING VULNERABLE CLASSES IN DESIGN PHASE

In broader perspectives, software vulnerabilities can be classified into two basic categories namely design time vulnerability and implementation time vulnerability. There are many tools available to detect implementation time vulnerability. On the other hand, detecting design time vulnerability is mostly carried out manually [3]. The fact validates the reason detecting and correcting design level vulnerabilities are difficult. The process is time consuming, resource consuming and error prone too [11]. Researchers and practitioners emphasize on the need of enhancing security in design phase itself. Their suggestions, when aggregated [12, 26] with respect to object oriented design perspective, may yield the steps as shown in “fig. 1”.

At the first step, risk analysis is performed in design phase to uncover assets, threats, attacks and risks [3]. This helps to extract sensitive classes i.e. classes having both data members and methods attack prone. Hence these classes are suspected to be attacked. Second step attempts to secure these sensitive classes using security mechanisms. At the end, classes are assumed to be secure merely just applying security mechanisms. But, as a matter of fact, sensitive classes even after shielded with security mechanisms may not be completely secure. The simple reason is that no mechanism can guarantee absolute security [12].

Sensitive classes are secure till vulnerabilities in security mechanisms applied, are not targeted for attack. The proposed work depends on the fact that sensitive classes will always be in the eyes of attackers even though they are shielded with security mechanisms.

Some of the facts behind vulnerability [27] of sensitive classes (when no security mechanism applied) are listed below:

- It provides entry points for external application
- It processes confidential information
- It involves in internal network communication
- It allows user authentication and control
Similarly, some of the facts behind vulnerability of secure classes (even after security mechanisms are applied to sensitive classes) [28] are:
- Cryptographic protocol design errors
- Reliance on non secure abstractions
- Security / usability trade off
- Absence of strict logging scheme

From the foregoing discussion and with the evidences from literature survey, it has been concluded that a class would be vulnerable if both of its data members and methods are sensitive irrespective of the implementation of security mechanism. Therefore, presence of vulnerability is always there. The only effort one can make is to reduce its propagation through different artifacts including Inheritance, Encapsulation, Coupling and Cohesion. Authors have made an effort to minimize its propagation through Inheritance.

V. VULNERABILITY PROPAGATION (VP)

In general, propagation means flow of something from one place to another. In object oriented design perspective, spreading of vulnerability from one vulnerable class to another is termed as Vulnerability Propagation.

Mathematically, Vulnerability Propagation (VP) of a class C, denoted by VP(C), is set containing classes in the hierarchy which directly or indirectly inherit class C. Therefore,

$$\text{VP}(C) = \{C_i: C_i \in \text{set of classes in hierarchy}\}$$

Cardinality of set VP(C) represents the number of classes which have become vulnerable due to class C. Therefore,

Number of vulnerable classes due to class C = |VP(C)|

Alternatively, Vulnerability Propagation of a class C is the total number of classes which directly or indirectly inherit class C.

Since a design may be considered as the collection of modules which are again collection of class hierarchies. So, the motive behind calculating Vulnerability Propagation of a class in Inheritance hierarchy is to use the parameter to calculate Vulnerability Propagation Factor (VPF) of design. If M is the number of vulnerable classes in a design and VP (1), VP (2)….VP(M) are the Vulnerability Propagations of these classes respectively, then VPF may be defined mathematically as follows:

$$\text{VPF} = \frac{\text{VP}(1) \cup \text{VP}(2) \ldots \cup \text{VP}(M)}{M} \quad (1)$$

VI. VULNERABILITY PROPAGATION THROUGH INHERITANCE

It is well established fact that object oriented paradigm reduces software cost and increases software extensibility, flexibility and reusability. Inheritance is a prominent feature of object oriented paradigm which provides reusability [29]. There are mainly two types of Inheritance; Single Inheritance and Multiple Inheritance. When a class is allowed to have only one super class, it is called Single Inheritance. If a class is allowed to have multiple super classes, it is called Multiple Inheritance.

Inheritance graph of a class hierarchy H of n classes is a directed graph H = (V, E) where V corresponds to the set of total number of classes and E is the set of total number of Inheritances in the class hierarchy. So, mathematically, Inheritance Graph G = (V, E) such that V = \{C_1, C_2, C_3, C_4, \ldots \ldots \ldots \ C_n\}, and E = \{C_i \rightarrow C_j\} where i, j are arbitrary numbers and i, j \in \{1, 2, 3, 4, \ldots \}

Graphical representation of Single, multiple and hybrid (follow both rules i.e. single & multiple) Inheritances are given in “fig. 2”.

Inheritance plays an important role in increasing reusability of design and code or of overall software; hence it also reduces effort and cost of software development. But at the same time, it has been noticed that Inheritance is also responsible for spreading vulnerability from one class to another, because of its transitivity property [29]. If a design supports Inheritance then all of the methods and variables defined for the parent class automatically become available for all of its subclasses, as well as for any subclass of these classes recursively down the hierarchy [10]. This characteristic of Inheritance easily spreads vulnerability among the classes.

VII. VULNERABILITY PROPAGATION FACTOR COMPUTATION

An Inheritance hierarchy may contain no class or one or more vulnerable classes. Also, there are more than one Inheritance hierarchies present in a design. So, calculation of Vulnerability Propagation Factor (VPF) of a design requires calculation of Vulnerability Propagation due to each vulnerable class C_i in Inheritance hierarchies present in the design. Then, union of Vulnerability Propagation due to each vulnerable class will give overall Vulnerability Propagation in design.

Now, using “(1)” vulnerability propagation factor of design can be calculated through proposed algorithm.

A. Algorithm

The algorithm for calculating Vulnerability Propagation Factor for a design uses the principle of Breadth First Search (BFS) algorithm. It takes identified vulnerable classes in a queue VUL [1...M] and an array H [N] of Inheritance graphs as input, where M is the total number of identified vulnerable classes and N is the number of Inheritance hierarchies in the design.

![Figure 2. Single, Multiple, Hybrid Inheritance](image)
For an Inheritance graph \( H = (V, E) \) and a distinguished vulnerable vertex \( vul \), algorithm explores the edges of \( H \) to obtain every vertex that is reachable from \( vul \). The algorithm creates trees corresponding to each vulnerable vertex \( vul \) and at the same time counts all vertices of the trees. Finally, it produces Vulnerability Propagation Factor of design, using \( count \).

To verify the progress, the algorithm shows status of each vertex as \( unknown \), \( known \) and \( traversed \). Initially the vertex is unknown and becomes \( known \) when it is discovered. Later, it becomes \( traversed \) when all of its adjacent vertices become \( known \). The algorithm assumes that Inheritance graphs \( H = (V, E) \) are represented using adjacency lists. It maintains several additional data structures with each vertex in Inheritance graphs. The status of each vertex \( u \in V \) is stored in the variable \( status[u] \), predecessor of \( u \) is stored in the variable \( p[u] \). The algorithm uses two FIFO queues, \( VUL \) containing all vulnerable vertices and \( Q \) managing a set of all known vertices. The algorithm is described as:

\[
\text{VPF (VUL [1...M], H [N])}
\]

1. \( vp = 0 \)
2. for \( [i = 1; \ i \leq M; \ i++] \)
3. \( status[VUL[i]] = known \)
4. for \( [i = 1; \ i \leq M; \ i++] \)
5. \( vul \leftarrow \text{Head (VUL)} \)
6. \( count = 0 \)
7. for \( [j = 1; \ j \leq N; \ j++] \)
8. if \( vul \in H[j] \)
9. \( \text{Call VP (H[j], vul)} \)
10. \( \text{Exit} \)
11. \( \text{dequeue (VUL)} \)
12. \( \text{vp} = \text{vp} + (\text{count}-1) \)
13. \( \text{vpf_design} = \frac{\text{vp}}{M} \)

Where \( \text{VPF (H[j], vul)} \) is defined as follows:

\[
\text{VP (H[j], vul)}
\]

9.1 for each vertex \( u \in V (H[j]) - vul \)
9.2 if \( status[u] = Nil \)
9.3 then do \( status[u] \leftarrow unknown \)
9.4 \( p[u] \leftarrow Nil \)
9.5 \( p[vul] \leftarrow Nil \)
9.6 \( Q \leftarrow vul \)
9.7 while \( Q \neq \emptyset \)
9.8 do \( u \leftarrow \text{head (Q)} \)
9.9 for each \( v \in \text{adj}[u] \)
9.10 do if \( status[v] \leftarrow unknown \)
9.11 then \( status[v] \leftarrow known \)
9.12 \( p[v] \leftarrow u \)
9.13 \( \text{enqueue (Q, v)} \)
9.14 \( \text{dequeue (Q)} \)
9.15 \( status[u] \leftarrow traversed \)
9.16 \( \text{count} \leftarrow \text{count} + 1 \)

**B. Implementation**

Since, a vertex becomes vulnerable whether it inherits one vulnerable class or more. So, it does not matter how many vulnerable classes are there at the root in making a class vulnerable. Hence in cases where one class inherits two or more vulnerable classes, it will appear in tree corresponding to only one of those vulnerable vertices. This will help to avoid repetition. The status variable keeps track of these vertices so that they can not be counted twice or more.

The algorithm proceeds as follows: for each vulnerable vertex, \( status \) is set as \( known \). Then it is checked to which hierarchy it belongs to. Further \( \text{VP (H[j], vul)} \) algorithm is called to find out all of the vertices directly or indirectly inheriting this vulnerable vertex. The \( \text{VP (H[j], vul)} \) algorithm produces trees corresponding to each vulnerable vertex as root node and number of vertices in each tree. Finally, \( \text{VPF of design} \) is computed.

In detail; step1 of \( \text{VPF sets variable vp to 0. Steps 2 to 3 set status variable for each vulnerable (grayed in “Fig. 3”) vertex to known. In for loop from step 4 to12, each time a vulnerable vertex is assigned to variable vul from the head of the VUL, its count variable is set to 0. Algorithm VP is called for each vul (step 9), which generate a tree corresponding to each vul and count number of nodes in the tree using count variable. At the exit of the algorithm VP, VUL is dequeued so that the next vertex in Q may occupy the head of the VUL; count is decremented by 1 and is added to vp to yield cumulative Vulnerability Propagation due to each vulnerable class. Finally, step13 results as a Vulnerability Propagation Factor.}

The algorithm \( \text{VP (H[j], vul)} \) called in step 9 of \( \text{VPF works as follows: In the for loop from step 9.1 to 9.4, the status variable of each vertex except vul is examined to know whether it has been already included in a tree corresponding to some other vulnerable vertex vul of that Inheritance graph. If it is not the case (status variable is Nil), status variable of vertex is set as unknown and parent p as Nil. In step 9.5, parent of vertex vul is set to Nil as now it becomes root vertex. Step 9.6 puts vul in Q. The while loop from step 9.7 to 9.16 iterates till Q is not empty. Each time, head of the Q is assigned to a variable u. Status of all vertices adjacent to u are set known and parent p is u (if the status variable is unknown) and these adjacent vertices are inserted into Q. The Q is now dequeued so that the next vertex in Q may occupy the head position. Status of u is set as traversed to indicate that all of its adjacent vertices are explored. Finally, count is incremented by 1 to show increment in the Vulnerability Propagation.}

C. A try out example

In order to understand better the \( \text{VPF algorithm} \), an example has been taken to explain. Three Inheritance hierarchies in a design are given, whose graphical representation is shown in “Fig. 3a” as H [1], H [2], and H [3]. The vulnerable vertices are grayed to distinguish them from normal vertices. The VUL in “Fig. 3b” contains these vulnerable vertices. This forms input to the algorithm. “Fig. 4” contains trees corresponding to each vulnerable vertex \( C_2, C_6, C_{10} \) and \( C_{19} \).

Now, \( \text{VPF of the design is calculated as:} \)

\[
\text{vpf_design} = \frac{\text{vp}}{M} = \frac{7}{4} = 1.75
\]
(1)\) would lie between 0 and \( P \). i.e. \( VPF \) will always be greater than or equal to 1. i.e.

\[
0 \leq |VP (1) U VP (2) U ... U VP (M)| \leq P
\]

So using above inequality the value of VPF of a design will always be greater than or equal to 1 i.e.

\[
VPF \geq 0
\]

Considering above value of VPF, an inference can be drawn for vulnerability of a design as:

(Higher the VPF, higher the vulnerability in design)

This may serves as an aid to remediate vulnerabilities and achieve target level of security.

IX. ADVANTAGE AND LIMITATION

Advantages of VPF of an object oriented design are three fold:

- It may establish comparison between design of different versions of software or designs of different software.
- It may help to choose some alternative design decisions to improve existing design.

On the other hand, effort of measuring vulnerability of an Inheritance hierarchy and hence an object oriented design is at very young stage. So, lacunas are obvious. One of the major limitations of the work is its applicability to only object oriented software. Also, only one aspect of object oriented design has been considered when calculating VPF i.e. Inheritance, while other aspects including Encapsulation, Polymorphism, and Coupling etc. should also be taken into account.

X. CONCLUSION

Since design is called skeleton of software, it is comparatively easy to implement security at this phase. At the same time, if security aspects are ignored at this phase, the resulting design might become vulnerable \([30, 31]\). Even a single flaw in design will manifest in next phases of software life cycle and it will become harder and harder to diagnose this vulnerability as the software life cycle proceeds. The work investigates the role of Inheritance in vulnerability propagation. A measure called VPF has been introduced as the major contribution to the work. VPF reveals the fact that how a single vulnerable class can manifest itself in class hierarchy, as well as in design to make whole software vulnerable.

REFERENCES


Available at: www.cs.kuleuven.be/publicaties/rapporten/cw/CW346.ps.gz

