Modeling Class of Software Vulnerabilities with Vulnerability Cause Graphs

By

Rahul Hiran

LIU-IDA/LITH-EX-A--09/056--SE

2009-10-21
Master’s Thesis

Modeling Class of Software Vulnerabilities with Vulnerability Cause Graphs

by

Rahul Hiran

LIU-IDA/LITH-EX-A--09/056--SE

Supervisor: David Byers
IDA, Linköpings Universitet

 Examiner: Nahid Shahmehri
IDA, Linköpings Universitet
Abstract

Vulnerabilities discovered in software are not only due to programming errors but also due to design flaws. There are a number of methods to avoid design flaws which are all manual processes and need expertise. We believe that the study of models of classes of vulnerabilities would give developers sufficient knowledge in how to avoid these vulnerabilities. A model of class of vulnerability can also help in the decision making process during the software development process.

In this thesis, we present a procedure for modeling a class of vulnerabilities given instances of Vulnerability Cause Graphs (VCGs). Using VCGs will structure the representation of causes to vulnerabilities.

The approach presented in this thesis makes it possible to divide the work of modeling a class of vulnerability without any permanent dependence on any specific persons. The approach is also flexible enough to accommodate new causes of vulnerabilities in software when being discovered.
I would like to express my gratitude to my examiner, Professor Nahid Shahmehri, and supervisor, David Byers, for their guidance during the thesis. I would also like to thank Shanai Ardi for her help at the initial stages of my thesis. I have got a lot of feedback and guidance when it was needed.

It was great experience to work with the modeling of vulnerabilities. I had to be creative in suggesting solutions, as the problem I was addressing had very little documentation. I also had to be practical at the same time to make sure that the solution can be implemented.

I owe a lot to my friend, Catherine Farestad, for language proofreading.

Thanks to my friend, Jean-Sébastien Susset for proofreading and many interesting discussions.

Finally, I can’t forget to thanks my parents and brothers for supporting throughout my master studies and for never questioning the choices I make in life and trusting me.
Table of Contents

1. Introduction ..................................................................................................................................... 1
   1.1. Introduction ............................................................................................................................. 1
   1.2. Motivation ............................................................................................................................... 2
   1.3. Goals ........................................................................................................................................ 2
   1.4. Method of work ....................................................................................................................... 2
   1.5. The Outline of the thesis ......................................................................................................... 3

2. Background ...................................................................................................................................... 5
   2.1. Introduction ............................................................................................................................. 5
   2.2. Classes of vulnerabilities and Instance of vulnerabilities ........................................................ 5
   2.3. Vulnerability Cause Graphs ..................................................................................................... 5
   2.4. Types of nodes in VCG ............................................................................................................. 6
   2.5. The Semantics of the Vulnerability Cause Graphs .................................................................. 7
   2.6. Example of a Vulnerability Cause Graph ................................................................................. 8
   2.7. Ways to create class VCG ........................................................................................................ 8
   2.8. Summary .................................................................................................................................. 9

3. Analysis .......................................................................................................................................... 11
   3.1. Introduction ........................................................................................................................... 11
   3.2. Modeling instance VCGs of buffer overflow class of vulnerability ........................................ 11
      3.2.1. CVE-2009-0490 .............................................................................................................. 11
      3.2.2. CVE-2008-3252 .............................................................................................................. 15
      3.2.3. CVE-2005-2943 .............................................................................................................. 17
      3.2.4. CVE-2007-0063 .............................................................................................................. 19
      3.2.5. CVE-2002-0728 .............................................................................................................. 23
      3.2.6. The referenced VCGs from the published research papers .......................................... 24
   3.3. Inferences from the instance VCGs and the use of these inferences ................................... 26
   3.4. The generalization procedure ............................................................................................... 27
   3.5. Summary ................................................................................................................................ 28

4. Proposition .................................................................................................................................... 29
   4.1. Introduction ........................................................................................................................... 29
   4.2. Concept of merging of VCGs (The manual method) ............................................................. 29
   4.3. Merging of VCGs by edge processing .................................................................................... 31
4.4 The procedure for merging two VCGs ................................................................. 33
4.5 Semantics of the VCG at each step of the transformation ............................... 35
  4.5.1 Semantic preservation property ................................................................. 35
  4.5.2 Change in the semantics of a VCG when a node is added to the VCG .......... 36
  4.5.3 Change in the semantics when an edge is added to the VCG ...................... 37
4.6 Issues with the proposed merging procedure: ............................................... 41
4.7 Rule of ignoring edges (The two basic rules) ..................................................... 42
4.8 Cycles during the merging procedure ............................................................... 44
  4.8.1 An example of formation of a cycle by merging two VCGs ....................... 44
  4.8.2 The reasons for ambiguities and inconsistencies in the VCGs ................. 44
  4.8.3 Basic blocks basics ...................................................................................... 44
4.9 Basic block rules for resolving cycles during the merging procedure ............... 45
  4.9.1 Type 1: Source VCG containing basic block with two nodes and the destination VCG having basic block with more than two nodes ................................................................. 46
  4.9.2 Type 2: The source VCG and destination both having basic block with more than two nodes ............................................................................................................... 52
  4.9.3 Type 3: Source and destination VCG both having basic blocks with two nodes .......... 56
4.10 Procedure for resolving the cycles during the merging procedure .................. 59
4.11 Merging Procedure for the VCGs with conditions ......................................... 60
4.12 Summary ......................................................................................................... 64
5 Creating class VCG for Buffer Overflow class of vulnerability .......................... 65
  5.1 Introduction ...................................................................................................... 65
  5.2 Creating a class VCG for buffer overflow from instance VCGs .................... 65
  5.3 Merging of VCGs to apply the rule of ignoring edges ..................................... 69
  5.4 Summary ......................................................................................................... 70
6 Evaluation .......................................................................................................... 71
  6.1 Implementation .............................................................................................. 71
    6.1.1 Requirements for the basic merging procedure ...................................... 71
    6.1.2 Choice of language ................................................................................... 71
    6.1.3 Choice of Java library ............................................................................... 71
    6.1.4 Development environment ....................................................................... 72
    6.1.5 Version specifications .............................................................................. 72
    6.1.6 Future implementation ............................................................................ 72
  6.2 Using merging procedure in the detection of conjunction nodes .................... 73
1 Introduction

1.1 Introduction
We all know the risks associated with flying in airplanes. Still we are ready to accept the risk and are ready to fly the next time we have the opportunity. Why? Because it is convenient. Travelling from one country to another country or continent would not be short and easy if it was not for airplanes. We are sure of safe journey, because we believe that the system is secure. We trust that the airplane is not vulnerable to bad weather, lightning, or any catastrophe that may arise. Being vulnerable in itself is not dangerous. It will become dangerous and catastrophic only if the airplane is struck by a lightning. We might not be able to control if lightning strikes an airplane or not, but we can always make sure that the airplane is not affected in any catastrophic way when lightning does strike. It is better to reduce the vulnerability of an airplane before, than to take actions after an accident due to existence of the vulnerability in the airplane. The same thinking has brought the focus to the vulnerabilities in software. In the Software industry, vulnerability is often referred to a weakness in a computing system that can result in harm to the system or its operations, especially when the weakness is exploited by a hostile person or organization.\(^1\) Vulnerability can be defined as susceptibility to attack, injury etc.

The attacks and malicious threat code to exploit the vulnerabilities in software is rising at an alarming rate. According to Internet Security Threat Report (1), Symantec created 1,656,227 new malicious code signatures during 2008. That is a 165 percent increase compared to 2007. The Sophos Security threat report (2) suggests that the variety and number of attacks will continue to escalate in the future. The best way to encounter these ever increasing threats is by developing non-vulnerable software. According to (3) it costs 10-40 times more if a defect is found in the application testing stage and 600 times more if it is found post release, rather than in a coding phase. The report commissioned by McAfee Inc suggests that Intellectual property is the new currency for the Cybercriminals. Data loss in intellectual property rights was more than 1 trillion in the year 2008 due to Data theft and Cybercrime(4). Thus, there is a big incentive to detect the vulnerabilities right away with-in the software development process rather than finding them after the software has been deployed. The researchers at the Department of computer and information science (IDA) at Linköping University along with other partner Universities and research institutes are working on a project called “SHIELDS”. The main theme of the project is detecting known security vulnerabilities from within design and development tools.

At IDA, the researchers are working on defining validated software development process components that enhances security throughout the software lifecycle. The approach to defining process components is based on a thorough analysis of vulnerabilities and their causes. The results are represented in the form graph called Vulnerability cause graphs (VCG). A VCG organizes information about vulnerability and its causes, and is designed to promote the understanding of vulnerabilities and reuse of analysis results. A VCG consists of nodes and edges. Nodes represent how causes may contribute to the vulnerability and the edges represent the relations between different causes. Each cause is then analyzed individually to determine how it can be mitigated (5).

\(^1\) http://www.thefreedictionary.com/vulnerability
\(^2\) http://shields-project.eu/
1.2 Motivation

Vulnerabilities are not a new topic in the software field. Regardless, they still appear in software indicating that programmers and designers of software still do not know how to deal with the vulnerabilities. There are number of tools like RATS, Yasca, FindBugs, Splint, Cppcheck, Coverity, etc. for scanning of the code and finding bugs in software. However, bugs in the coding are not the only security threat in software. There are also design flaws. According to the article “Risk analysis in software design”(6) roughly 50 percent of security problems are due to design flaws. Finding these design flaws in the software is still a manual process. There are numbers of methods and principles suggested for a secure design. Threat modeling (7), Security design review(8), Following secure design guidelines and principles (9), Misuse cases (10) and Attack trees (11) are some of them. All these approaches need manual analysis and expertise.

The manual analysis can work well if the personnel involved know about the classes of vulnerabilities and their causes. If a person involved does not have enough experience then he can be trained to have this knowledge. For example, to learn about the causes and ways to avoid the web security problems, a document referred as OWASP testing guide (12) can be studied. Unfortunately, the document is too long and it cannot be assured that all the causes of the vulnerabilities and the ways to mitigate those vulnerabilities are enumerated in this document. This document targets only web applications. The people involved in secure design will need to study about other classes of vulnerabilities and different ways to avoid them as well. So, there will be more documents and books to study. The task of secure design may become difficult. Also, these documents and books will need constant updates as and when new causes and new ways to mitigate the causes are found.

We believe that a model for the complete class of vulnerability could be of an immense help to software designers in secure software development process. It can be an important reference during the manual analysis. The model can also be used in the automated analysis of software. The VCG for a class of vulnerability can also help to figure out common mitigation strategies to detect and avoid an entire class of vulnerability from software. Studying a complete class of vulnerability with single model will be easier than reading long documents.

Instances of VCGs can be modeled with the help of VCGs as described in “Modeling Software Vulnerabilities with Vulnerability Cause Graphs (13)”. Similarly a complete class of vulnerability can be modeled with the help of VCG. In this thesis, we intend to model complete class of vulnerability with Vulnerability cause graph.

I also intend to create a procedure where the VCG for a complete class of the vulnerability can be updated without any or very little manual intervention.

Our aim is to find the set of “Security activities” that if implemented will help people to avoid a complete class of vulnerability. “Security activities” are the software lifecycle activities, implemented to prevent vulnerabilities by addressing their causes (14). The first step to achieve this would be to create a VCG for the complete class of vulnerability.

1.3 Goals

The goals of the thesis are:
1. Modeling a complete class of vulnerabilities.
2. Derive and describe a procedure for creating model of class of vulnerability.
3. Describe how this model can be practically used.

1.4 Method of work

The initial phase was to study the classification of vulnerabilities and the research papers published on Vulnerability cause graphs. A number of instances of buffer overflow vulnerabilities were modeled to gain the practical experience in making VCGs. After the analysis, a procedure for modeling a
complete class of vulnerability was suggested. Every time a problem was encountered in the procedure, we went back and updated the procedure to solve the problem. One of the key aspects was to enumerate the different cases that could be encountered during modeling of class of vulnerability. The basic solution proposed at the beginning, was implemented and tested to demonstrate its applicability.

1.5 The Outline of the thesis
In the Background section, we present the details of Vulnerability cause graphs (VCGs) for the instances of the vulnerabilities in this section.

In the analysis section, we will analyze and model some instances of buffer overflow vulnerabilities. The modeling of the vulnerabilities will be done on the basis of the procedure explained in the paper titles “Modeling Software Vulnerabilities with Vulnerability Cause Graphs” (13). Some important conclusions will be also drawn by analyzing these instances of the VCGs. These observations will be important later on when we devise a procedure to create the VCG for the class of vulnerability in the proposition section.

In the proposition section, we will introduce the class of vulnerability can be modeled. We will also look into various conditions and problems that can be encountered during the creation of the class VCG.

In the evaluation section, we will see the implementation of the basic merging solution for modeling a class of vulnerability. We will see the evaluation based on the implementation and also analyze the solution proposed.

In the example section, we will create a VCG for buffer overflow class of vulnerability based on the solution proposed in the proposition section.

In the application section, the various ways in which the VCGs for class of vulnerability can be used will be discussed. Specifically we will see its use in Software development process, risk analysis and risk mitigation, various tools and the educational purposes.

In the final section, we will give conclusion and comments for future work.
2 Background

2.1 Introduction
In this chapter, we introduce the Vulnerability cause graphs in detail. The papers “Towards a structured unified process for software security”(5), “Modeling Software Vulnerabilities with Vulnerability Cause Graphs”(13) and “How can the developer benefit from security modeling?”(15) are referred to in this chapter.

2.2 Classes of vulnerabilities and Instance of vulnerabilities
There are a number of vulnerabilities being found in different software every day. As of at the time of writing this thesis, the number of publicly known vulnerabilities is 39,006 according to the NIST(16). These vulnerabilities are divided into different classes. The members of the class of vulnerability are referred to as instances of the vulnerabilities belonging to that particular class.

Buffer overflow, Cross-site scripting, Cryptographic issues, Cross-site request forgery, Path traversal are some examples of different classes of vulnerabilities.

Heap-based buffer overflow in LucVil PatPlayer 3.9 discovered in the year 2009 is an example of the instance of buffer overflow class of vulnerability.
Directory traversal vulnerability in exRecipe-Zee 91discovered in the year 2009 is an example of instance of path traversal class of vulnerability.
Vulnerability in Battle Blog 1.25 and 1.30 build 2 allows remote attackers to inject arbitrary web script or HTML via a comment. This is an example of instance of Cross-site scripting class of vulnerability. This vulnerability is also discovered in the year 2009.

The paper “Modeling Software Vulnerabilities with Vulnerability Cause Graphs” (13) describes a procedure for modeling an instance of the vulnerability. The results are presented in a form of graph called Vulnerability cause graph. This VCG represent the causes and relation between the causes that result in an instance of vulnerability. This VCG is referred to as instance VCG in the future discussion.

VCGs are used to model instances of vulnerabilities. The same basics of modeling instances of vulnerabilities could be used to model complete class of vulnerability. The VCG for the complete class of vulnerability will be referred to as class VCG henceforth. It is important to note that, the objective of the thesis is to create class VCG with the help of instance VCGs. Similarly we can model a complete class of vulnerability with VCGs. VCG for complete class of vulnerability that we intend to create will be called as class VCG. This VCG will represent all the causes and relation between them that result into a vulnerability belonging to that class.

2.3 Vulnerability Cause Graphs
A Vulnerability cause graph (VCG) is a directed acyclic graph in which all nodes but one represents causes. Edges represent relationships between the causes. A single exit node in the VCG represents the instance of vulnerability that is modeled.

Vertices with no successor are known as vulnerabilities, and represent classes of potential vulnerabilities in software being developed. “Integer overflow” or “buffer overflow” are examples of 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. “*Missing range check*” or “*safe function used in incorrect way*” could be examples of causes.

**Security activity graphs:**

A *security activity graph* relates activities during software development to preventing potential vulnerabilities in the final product. Every vulnerability can have a corresponding SAG. SAGs show how security activities combine to prevent vulnerabilities or causes of vulnerabilities. SAGs also facilitate the selection of a set of security activities that is most suited. (E.g. cheapest, fastest, easiest)(14).

Security activity graphs are constructed from vulnerability cause graphs in a structured manner. More information about construction of the SAGs can be found in the paper titled “Towards a structured unified process for software security”(5).

### 2.4 Types of nodes in VCG

There are four types of nodes in the Vulnerability cause graphs. (13)

**Exit node:** Every VCG has one exit node and this is the only node in the graph without any successors. Normally an exit node is the *CVE identifier* indicating the vulnerability that is modeled.

*CVE identifiers* (also called "CVE-IDs," "CVE names," "CVE numbers," and "CVEs") are unique, common identifiers for publicly known information security vulnerabilities. Each *CVE Identifier* on the CVE List includes a CVE identifier number (i.e., "CVE-1999-0067"); indication of "entry" or "candidate" status; a brief description of the security vulnerability or exposure; and any pertinent references (i.e., vulnerability reports and advisories or OVAL-ID).

*CVE Identifiers* are used by information security product/service vendors and researchers as a standard method for identifying vulnerabilities and for cross-linking with other repositories that also use *CVE Identifiers*.³

The exit node identifies a specific instance of the vulnerability that is represented by the VCG. There is only one exit node for each VCG.

**Simple node:** The Simple node represents the simple cause of the vulnerabilities. They are the atoms of the VCGs.

**Compound node:** The Compound node represents the complex cause of the vulnerabilities. They facilitate analysis reuse, maintenance of models and improve readability. They represent entire VCGs that model reusable or complex analysis elements.

**Conjunction node:** The Conjunction node represents conjunction of two or more nodes. This node is used to represent the fact that two or more causes should be present simultaneously for the vulnerability to occur.

**Visual representation:**

![Visual representation of the nodes in VCG](http://cve.mitre.org/)

³ [http://cve.mitre.org/](http://cve.mitre.org/)
2.5 The Semantics of the Vulnerability Cause Graphs

The final goal of the Vulnerability modeling is to determine how to prevent the vulnerabilities. The semantics of VCGs are expressed in such terms and the semantics are derived from the mitigation of causes according to the article “Modeling Software Vulnerabilities with Vulnerability Cause Graphs”(13):

*A cause that is of concern during software development or maintenance is mitigated if actions are taken that result in the condition the cause represents being false.*

A simple node N is mitigated if the cause it represents is mitigated.

A node representing a cause that is not a concern during development or maintenance is considered blocked.(13)

We use the following notations to explain the semantics of the VCGs:

- N: N is a node in the VCG
- B (N): if B (N) is true, then cause N is blocked.
- M (N): if M (N) is true, then cause N is mitigated.

We will show the semantics of the VCGs by considering few basic cases:

**Fig. 2a**

- Cause D
  - Cause C
  - Vulnerability V

The fig.2a shows a VCG with two causes connected serially with the vulnerability V. If cause D is mitigated then cause C is of no concern as it cannot occur unless cause D is present. Also if cause C is mitigates then cause D is blocked. Blocking of cause C or cause D, will avoid the occurrence of vulnerability V.

\[ B (V) = M (V) \text{ or } (B (C) \text{ or } B (D)) \]

**Fig. 2b**

- Cause C
  - Cause D
  - Vulnerability V

The fig.2b shows a VCG with two causes connected to a vulnerability V. It shows that both the causes C and cause D can independently result in vulnerability V. It implies that to avoid the occurrence of vulnerability V it will be required to mitigate both the causes C and cause D.

\[ B (V) = M (V) \text{ or } (B (C) \text{ and } B (D)) \]

**Fig. 2c**

- Cause C
  - Cause D
  - Vulnerability V

The fig. 2c shows a VCG with the causes C and cause D in a conjunction node. It implies that for the vulnerability to occur, both the cause C and cause D must be present. So it is enough to just mitigate one of them to avoid the occurrence of the vulnerability.

\[ B (V) = M (V) \text{ or } (B (C) \text{ or } B (D)) \]
Background

**Generalized Semantic equation:**
Each of the predecessors of a node \( N \) in the VCG independently causes \( N \) to be a concern. Therefore, they must all be blocked, or \( N \) itself mitigated, for \( N \) to be blocked.

A node \( N \) in a VCG is said to be *blocked* if it is mitigated, or all of its immediate predecessors in the VCG are blocked (13).

\[
B(N) = M(N) \lor \left( \bigwedge_{p \in \mathcal{P}} B(p) \right)
\]

\( \mathcal{P}_N \): A predecessor set of nodes for node \( N \)

### 2.6 Example of a Vulnerability Cause Graph

Figure 2-2 shows the Vulnerability cause graph that models CVE-2009-0490(16). It models the stack based buffer overflow in Audacity 1.2.6 to 1.3.6. (Audacity is an open source sound editor and recording software.)

To mitigate CVE-2009-0490, we should mitigate the conjunction node consisting of two causes: “Missing range check” and “No use of safe functions for copying data”. It is enough to mitigate one of the two causes, since it is a conjunction node.

It can also be inferred from the Figure 2-2 that to block the conjunction node we can mitigate its immediate predecessor. In this case the immediate predecessor for the conjunction node is the cause “Use of non adaptive buffer”. From the VCG it can also be inferred that to mitigate the vulnerability it is enough to mitigate the cause “Copy of external data to internal buffer”.

Thus, to avoid the vulnerability CVE-2009-0490 we have to do the following:
Mitigate the cause “Missing range check” or “No use of safe functions for copying data” or “Use of non adaptive buffer” or “Copy of external data to internal buffer”.

### 2.7 Ways to create class VCG

We suggest two ways in which a class VCG can be created.

1. Make the analysis for the class of vulnerability and start making a class VCG right away.
2. Start with the VCGs for the instances of a class of vulnerabilities and then integrate these instance VCGs to make a class VCG i.e. merging existing instance VCGs to create a class VCG. (We will explain the merging procedure in depth in the beginning of chapter 4)

We have decided to follow the second way of making a class VCG. With the first method we may overlook the causes that are not that frequent and we could also miss important causes. Moreover, we want to avoid the manual analysis in the creation of the class VCG.

However, the problem with second method is that too many minor details could be modeled and class VCG might overgrow too large to be useful. We will devise a way to overcome this problem in the chapter of “Analysis”.

Figure 2-2: VCG for CVE-2009-0490

![VCG for CVE-2009-0490](image)
The next important step to create the class VCG is to make a merging procedure that can be implemented to create a class VCG from a number of instance VCGs. In chapter 3, we will analyze and create VCGs for some instances of buffer overflow vulnerabilities by the method described in (13). In chapter 4, we will use these VCGs to create to draw some inferences and devise a procedure for creating a class VCG. In chapter 5, we will use these instance VCGs and the proposed procedure for creating class VCG, to create a class VCG for buffer overflow.

It is important to note here that the instances VCGs that we tend to merge the VCGs for the instances of the vulnerability belonging to the same class of the vulnerability. will not merge instance VCGs for the instance of vulnerabilities belonging to different classes of the vulnerability.

2.8 Summary
In this chapter, we described the basics of VCGs for the instance of the vulnerabilities, their graphical representation, the semantic equations for the VCGs and the way they are intended to be inferred. We differentiated the terms “instance VCG” and the “class VCG” by defining them. We also discussed the ways in which class VCG for a particular class of vulnerability can be created. We will look into merging the instance VCGs to create a class VCG. The VCGs for the instances that will be merged will belong to the same class of the vulnerability.
3 Analysis

3.1 Introduction
In the previous section, we discussed the basics of instance VCGs. We also discussed the modeling of class VCG by merging the instance VCGs. However, at the time of writing the total number of instances of buffer overflow vulnerability is 1,629 according to National Vulnerability Database (17). Thus, if all these instances were to be modeled and merged to create a class VCG, it could grow very large. In this section we will model five instances of buffer overflow, analyze them and draw inferences to see if the size of the class VCG can be limited.

3.2 Modeling instance VCGs of buffer overflow class of vulnerability

3.2.1 CVE-2009-0490

Overview
Stack-based buffer overflow in the String_parse::get_nonspace_quoted function in library src/allegro/strparse.cpp in Audacity\(^4\) 1.2.6 and other versions before 1.3.6 allows remote attackers to cause a denial of service (crash) and possibly execute arbitrary code via a .gro file containing a long string (18).

Analysis (19)
Files referred: Audacity Version 1.2.6
1. strparse.cpp 2. allegrodd.cpp 3. allegrodd.h

Code Snippet 3.1 from the file /lib-src/allegro/strparse.cpp in Audacity 1.2.6

```cpp
void String_parse::get_nonspace_quoted(char *field) {
    skip_space();
    bool quoted = false;
    if (string[pos] == '"') {
        quoted = true;
        *field++ = '"';
        pos = pos + 1;
    }
    while (string[pos] && (quoted || !isspace(string[pos]))) {
        if (string[pos] == '"') {
            if (quoted) {
                *field++ = '"';
                pos = pos + 1;
            }
            *field = 0;
            return;
        }
        if (string[pos] == '\\') {
            pos = pos + 1;
        }
        if (string[pos]) {
            *field++ = string[pos];
            pos = pos + 1;
        }
    }
    *field = 0;
}
```

Code Snippet 3.1: From the file /lib-src/allegro/strparse.cpp in Audacity

\(^4\) http://audacity.sourceforge.net/
From code snippet it can be inferred that, the data is copied from the string to the char array variable named "field". The highlighted code in snippet 3.1 shows the data is being copied from the string to the field. However, there is no range check the copying is done. The max size of char array "field" is 80.

The code snippet 3.2 showing the use of function `get_nonspace_quoted()` in Audacity 1.2.6 from file `allegrord.cpp`.

```cpp
bool Allegro_reader::parse()
{
    int voice = 0;
    int key = 60;
    double loud = 100.0;
    double pitch = 60.0;
    double dur = 1.0;
    double time = 0.0;
    readline();
    bool valid = false; // ignore blank lines
    while (line_parser_flag) {
        bool time_flag = false;
        bool next_flag = false;
        double next;
        bool voice_flag = false;
        bool loud_flag = false;
        bool dur_flag = false;
        bool new_pitch_flag = false; // "P" syntax
        double new_pitch = 0.0;
        bool new_key_flag = false; // "K" syntax
        int new_key = 0;
        bool new_note_flag = false; // "A"-"G" syntax
        int new_note = 0;
        Parameters_ptr attributes = NULL;
        line_parser.get_nonspace_quoted(field);
        char pk = line_parser.peek();
        if (pk && !isspace(pk)) {
            line_parser.get_nonspace_quoted(field + strlen(field));
        }
        while (field[0]) {
            // print "field", "|";field;"|", "|";line_parser.string;"|", line_parser.pos
            char first = toupper(field[0]);
            if (strchr("ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ", first)) {
                valid = true; // it's a note or event
            }
            ...
```

Code Snippet 3.2: Showing the use of function `get_nonspace_quoted()` in Audacity 1.2.6
The declaration of the variable "field" is in the file "allegrord.h" shown in code snippet 3.3.

```cpp
#define field_max 80
class Allegro_reader {
public:
    FILE *file;
    int line_no;
    String_parse line_parser;
    bool line_parser_flag;
    char field[field_max];
    bool error_flag;
    Seq seq;
    double tsnum;
    double tsden;
    ...
};
```

Code Snippet 3.3: Showing declaration of variable “field” in Audacity 1.2.6

The problem is solved in the new version Audacity 1.3.7.

Code snippet 3.4 shows the code from `strparse.cpp` in the new version of the software.

In code snippet 3.4, the type of variable "field" has been changed from char array to string. Also, instead of direct data copy, the string operations now are used for data copying, which is safe to perform.

```cpp
void String_parse::get_nonspace_quoted(string &field) {
    field.clear();
    skip_space();
    bool quoted = false;
    if (((*str)[pos] == '"')) {
        quoted = true;
        field.append(1, '"');
        pos = pos + 1;
    }
    while (((*str)[pos] && (quoted || !isspace((*str)[pos]))) {
        if (((*str)[pos] == '"')) {
            if (quoted) {
                field.append(1, '"');
                pos = pos + 1;
            } else {
                return;
            }
        } else if (((*str)[pos] == '\')) {
            pos = pos + 1;
        } else if (((*str)[pos] == '"')) {
            field.append(1, (*str)[pos]);
            pos = pos + 1;
        }
    }
}
```

Code Snippet 3.4: Showing the changes in Audacity 1.3.7 to address the vulnerability
VCG for CVE-2009-0490:

- **Missing range check:**
  Applying range check while copying data to the variable “field” in the first code snippet would have prevented the vulnerability. The detection would have helped to take appropriate action so that the vulnerability would not exist.

- **No use of safe functions for copying:**
  Using safe functions for performing the function copy is secured than doing the data manipulations for data transfer. It leaves little room for bugs and vulnerabilities in the software.
3.2.2 CVE-2008-3252

Overview
Stack-based buffer overflow in the `read_article` function in `getarticle.c` in newsx 1.6\(^5\) allows remote attackers to execute arbitrary code via a news article containing a large number of lines starting with a period(20).

Analysis(21)
In code snippet 3.5, it can be seen that variable “`line`” grows every time a line starting with “.” is encountered. Hundreds of such lines can make `linebuf[]` to overflow and place and execute the arbitrary data on the stack. The line pointer needs to get initialized in the loop.

Code Snippet 3.5 from `getarticle.c`:

```c
static int
read_article(long where, char *group)
{
    char linebuf[MAX_HEADER_SIZE+1], *line;
    char path_line[MAX_HEADER_SIZE+1];
    int reject = 0;
    int is_newline = 1;
    int header = 1;
    int path_ok = -1;
    long len;
    long bytecount = 0L; /* BUG: */
    line=linebuf;
    line[MAX_HEADER_SIZE] = '\0'; /* better safe than sorry */
    path_line[0] = '\0';
    /* fetch the article, header and body */
    for (;;) {
        if (!get_server_msg(line, MAX_HEADER_SIZE)) {
            /* timeout: simply give up */
            return 0;
        }
        if (filter) filter_line(line); /* send to filter too */
        len = strlen(line);
        gross_bytecount += len;
        /* line starts with a period */
        if (is_newline && line[0]=='.') {
            if (line[1]=='\r' || line[1]=='\n') { /* single period, i.e. end of file */
                break;
            } else {
                if (line[1]=='\r') { /* escape-period, remove it */
                    line++;
                    len--;
                } else {
                    /* cannot happen */
                }
            }
        }
        ...
    }
}
```

Code Snippet 3.5: From Newsx 1.6

\(^5\) http://linux.wareseeker.com/Office/newsx-1.6.zip/323308
Changes made to solve the issue: https://bugzilla.redhat.com/attachment.cgi?id=311654
diff -up newsx-1.6/src/getarticle.c.stack newsx-1.6/src/getarticle.c

```c
long len;
long bytecount = 0L; /* BUG: */

line=linebuf;
line[MAX_HEADER_SIZE] = \0'; /* better safe than sorry */

path_line[0] = \0';

/* fetch the article, header and body */
x r ( ); { 
    line=linebuf;
    ( !get_server_msg(line, MAX_HEADER_SIZE)) { 
        timeout: simply give up */
        sturn 0;
    }
}
```

Here the 'line' pointer is initialized only once before the loop begins and then it is never re-initialized. Thus it can be incremented to result into buffer overflow.

**VCG for CVE-2008-3252:**

Incorrect variable initialization:
The software does not initialize or incorrectly initializes a resource, which might leave the resource in an unexpected state when it is accessed or used. This can have security implications when the associated resource is expected to have certain properties or values, such as a variable that determines whether a user has been authenticated or not(22).
Analysis of the node “Incorrect variable initialization”:
Looking further into the analysis of this cause we can suggest that the code paths are not well tested.

The code paths are not well tested:
This weakness can occur in code paths that are not well-tested, such as rare error conditions. This is because the use of uninitialized data would be noticed as a bug during testing of frequently-used functionality (22).

3.2.3 CVE-2005-2943
Overview
Stack-based buffer overflow in sendmail in XMail before 1.22 allows remote attackers to execute arbitrary code via a long -t command line option.

Analysis
The flaw is in SendMail.cpp file and in the function AddressFromAtPtr function shown in code snippet 3.7. XMail passes the user-supplied value without bounds checking to AddressFromAtPtr and attempts to store the hostname portion of the e-mail address in a 256-byte buffer. Although safe function strncpy is used, the third parameter passed to it is not checked for correct bounds and this can be exploited for buffer overflow.

Code Snippet from SendMail.cpp in Xmail:

```c
static char const *AddressFromAtPtr(char const *pszAt, char const *pszBase, char *pszAddress)
{
    char const *pszStart = pszAt;
    for (; (pszStart >= pszBase) && (strchr("<>:\t\n", *pszStart) == NULL); pszStart--);
    ++pszStart;
    char const *pszEnd = pszAt + 1;
    for (; (*pszEnd != '\0') && (strchr("<>:\t\n", *pszEnd) == NULL); pszEnd++);
    int iAddrLength = (int) (pszEnd - pszStart);
    strncpy(pszAddress, pszStart, iAddrLength);
    pszAddress[iAddrLength] = '\0';
    return (pszEnd);
}
```

Code Snippet 3.7: Showing the vulnerability in Sendmail

---

6 http://www.sendmail.org/
7 http://www.xmailserver.org/
VCG for CVE-2005-2943:

Incorrect use of safe function:
The format of the `strncpy()` function is:

```c
char * strncpy ( char * dest, const char * src, n );
```

The function “strncpy” copies the first `n` characters of `src` to `dest`. If the end of the source C string (which is signaled by a null-character) is found before `n` characters have been copied, `dest` is padded with zeros until a total of `n` characters have been written to it. No null-character is implicitly appended to the end of `dest`, so `dest` will only be null-terminated if the length of the C string in `src` is less than `n`.

If there is no null byte among the first `n` bytes of `src`, the result will not be null-terminated.

In a case where the length of `src` is less than that of `n`, the remainder of the `dest` will be padded with nulls.

Knowing that `strncpy()` will copy all of the buffer to the very last byte, we can essentially write `n` characters into buffer of size `n`, in turn overwriting where a NULL byte should be placed, thus making the code vulnerable to overflow buffer.
3.2.4 CVE-2007-0063

Overview

Integer underflow in the DHCP server in EMC VMware Workstation before 5.5.5 Build 56455 and 6.x before 6.0.1 Build 55017, Player before 1.0.5 Build 56455 and Player 2 before 2.0.1 Build 55017, ACE before 1.0.3 Build 54075 and ACE 2 before 2.0.1 Build 55017, and Server before 1.0.4 Build 56528 allows remote attackers to execute arbitrary code via a malformed DHCP packet that triggers a stack-based buffer overflow(26).

Analysis as explained in CoreLabs (27)

DHCP is built on a client-server model, where designated DHCP server hosts allocate network addresses and deliver configuration parameters to dynamically configured hosts. The term "server" refers to a host providing initialization parameters through DHCP, and the term "client" refers to a host requesting initialization parameters from a DHCP server. DCHP communication s on a local network use UDP ports 67 and 68.

The Dynamic Host Configuration Protocol (DHCP) specification indicates the requirements that a given DHCP implementation must fulfill. In summary, DHCP is designed to supply DHCP clients with the configuration parameters defined in the Host Requirements RFCs. After obtaining parameters via DHCP, a DHCP client should be able to exchange packets with any other host in the Internet. The TCP/IP stack parameters supplied by DHCP are listed in Appendix A of the corresponding RFC. Not all of these parameters are required for a newly initialized client. A client and server may negotiate for the transmission of only those parameters required by the client or specific to a particular subnet. DHCP allows but does not require the configuration of client parameters not directly related to the IP protocol. DHCP also does not address registration of newly configured clients with the Domain Name System (DNS).

The DHCP message definition includes a variable length field called “options” which are in turn indication of an additional variable length payload to the base DHCP message. The entire list of official DHCP options, also known as “vendor extensions” in BOOTP terminology, is provided in a companion RFC document to the protocol specification. One such option is the “maximum DHCP message size” option (MMS). The protocol specification indicates that “The client SHOULD include the ‘maximum DHCP message size’ option to let the server know how large the server may make its DHCP messages”.

DHCPD fails to properly validate the value provided in the “maximum message size” option by the DHCP client and thus allowing an attacker to specify MMS values that result in an integer underflow followed by a call to memcpy(3) with a negative third argument which in turns overwrites arbitrary portions of process memory.
The problem is found in the function responsible of processing the DHCP option received from the client:

In `src/usr.sbin/dhcpd/options.c`

```c
int cons_options(struct packet *inpacket, struct dhcp_packet *outpacket,
    int mms, struct tree_cache **options,
    int overload, /* Overload flags that may be set. */
    int terminate, int bootpp, u_int8_t *prl, int prl_len)
{
    unsigned char priority_list[300];
    int priority_len;
    unsigned char buffer[4096];
    int main_buffer_size;
    int mainbufix, bufix;
    int option_size;
    int length;

    //DHCP_FIXED_LEN is defined in dhcp.h

    if (!mms &&
        inpacket &&
        inpacket->options[DHO_DHCP_MAX_MESSAGE_SIZE].data &&
        (inpacket->options[DHO_DHCP_MAX_MESSAGE_SIZE].len >=
         sizeof(u_int16_t)))
        mms = getUShort(
            inpacket->options[DHO_DHCP_MAX_MESSAGE_SIZE].data);
    if (mms)
        main_buffer_size = mms - DHCP_FIXED_LEN;
    else if (bootpp)
        main_buffer_size = 64;
    else
        main_buffer_size = 576 - DHCP_FIXED_LEN;

    if (main_buffer_size > sizeof(buffer))
        main_buffer_size = sizeof(buffer);
}
```

Code Snippet 3.8: showing function `cons_options()` responsible for processing the DHCP options

`main_buffer_size` is signed and controlled by the attacker. As long as `main_buffer_size` is a small positive integer (<= 4096) execution flow will continue normally...
A small positive value of `main_buffer_size (<= 7)` will make `store_options` exit quickly and execution flow continues. Specifically, if the **Maximum Segment Size value** (mms) in the client packet satisfies the condition `(DHCP_FIXED_LEN < mms < DHCP_FIXED_LEN+4)` then `main_buffer_size` will be positive but less than 4.

```c
if (option_size <= main_buffer_size - mainbufix) {
    memcpy(&outpacket->options[mainbufix],
           buffer, option_size);
    mainbufix += option_size;
    if (mainbufix < main_buffer_size)
        outpacket->options[mainbufix++] = DHO_END;
    length = DHCP_FIXED_NON_UDP + mainbufix;
} else {
    outpacket->options[mainbufix++] = DHO_DHCP_OPTION_OVERLOAD;
    outpacket->options[mainbufix++] = 1;
    if (option_size >
        main_buffer_size - mainbufix + DHCP_FILE_LEN)
        outpacket->options[mainbufix++] = 3;
    else
        outpacket->options[mainbufix++] = 1;
    memcpy(&outpacket->options[mainbufix],
           buffer, main_buffer_size - mainbufix);
```
Analysis

The patch applied is shown in code snippet 3.11:

```c
if (!mms &&
    inpacket &&
    inpacket->options[DHO_DHCP_MAX_MESSAGE_SIZE].data &&
    (inpacket->options[DHO_DHCP_MAX_MESSAGE_SIZE].len >=
     sizeof(u_int16_t)))
    mms = getUShort(
        inpacket->options[DHO_DHCP_MAX_MESSAGE_SIZE].data);
+    if (mms < 576)
+        mms = 576;    /* mms must be >= minimum IP MTU */
+}
```

Code Snippet 3.11: Showing fix applied to solve the vulnerability

VCG for CVF-2007-0063:

![VCG Diagram](image)

Incorrect use of safe function `memcpy()`:

`memcpy` is not used safely because it is not made sure in the code that the value passed to the function is always positive.

**Format:**

```c
void * memcpy (void * destination, const void * source, size_t num);
```

The function copies the values of `num` bytes from the location pointed by `source` directly to the memory block pointed by `destination`.

The underlying types of the objects pointed by both the source and destination pointers are irrelevant for this function. The result is a binary copy of the data.

The function does not check for any terminating null character in `source` - it always copies exactly `num` bytes.

To avoid overflows, the size of the arrays pointed by both the `destination` and `source` parameters, shall be at least `num` bytes, and should not overlap (for overlapping memory blocks, `memmove` is a safer approach)(28).
3.2.5 CVE-2002-0728

**Overview**
Buffer overflow in the progressive reader for libpng\(^8\) 1.2.x before 1.2.4, and 1.0.x before 1.0.14, allows attackers to cause a denial of service (crash) via a PNG data stream that has more IDAT data than indicated by the IHDR chunk(29).

**Analysis**
Supplying a specially crafted PNG format file can cause heap corruption in Netscape and Mozilla. If the X size > \((2^{32} / \text{number\_bytes\_needed\_per\_pixel})\) then the number of bytes required for a row becomes greater than \(2^{32}\) and overflows.
This condition is checked out in the png library, but by default is not treated as fatal (to the image) in Mozilla/Netscape (30).

The faulty code was located in the file
/cvsroot/mozilla/modules/libimg/png/pngget.c

Fix applied: In the file pngget.c

```c
channels++;  
pixel_depth = *bit_depth * channels;  
rowbytes_per_pixel = (pixel_depth + 7) >> 3;  
- if ((*width > PNG_MAX_UINT/rowbytes_per_pixel))  
+ if (*width > PNG_MAX_UINT/rowbytes_per_pixel - 64)  
{  
- png_warning(png_ptr,  
+ png_error(png_ptr,  
"Width too large for libpng to process image data.");  
}  
return (1);
```

Code Snippet 3.12: Showing the vulnerable code and fix applied in libpng

**VCG for CVE-2002-0728:**

![Diagram of VCG for CVE-2002-0728]

---

\(^8\) [libpng.org](http://www.libpng.org)

---

23
Error detected but not raised:
When the check `(*width > PNG_MAX_UINT/rowbytes_per_pixel)` fails, the condition `png_error()` should have been invoked instead of `png_warning()` inside libpng. This will result the code to take a long jump and land safely.

Lacking design to implementation tracking:
This can be inferred because for the same kinds of check elsewhere, the condition raised is an error instead of a warning. There were thus some problems in keeping track of the implementation.

The developer does not understand the implication of the condition:
The developer is not trained or experienced enough to understand the implication of these conditions and thus fails to take appropriate actions.

3.2.6 The referenced VCGs from the published research papers
Here the VCGs from referenced papers are presented. These VCGs will be used in section 3.3 for drawing inferences by the analysis of the instance VCGs. The referred papers are “Modeling Software Vulnerabilities with Vulnerability Cause Graphs” (13), “How can the developer benefit from security modeling?” (15) and “Integrating a Security Plug-in with the OpenUP/Basic Development Process” (31).

![VCG Diagram](image-url)

Figure 3-6: VCG for CVE-2002-1337(13)
Analysis

CVE-2005-2558
Use of non-adaptive buffer
Copy of external data to internal buffer
Use of unsafe function for string copying
Range check problems
Wrong source size used
Use of C-like strings

CVE-2005-3192
Use of non adaptive buffer
External data influences buffer size
Unchecked integer arithmetic
Use of malloc
Unsafe use of malloc
Failed to check return value from malloc
Failed to check input parameter tp malloc

Figure 3-7: VCG for CVE-2005-2558(15)

Figure 3-8: VCG for CVE-2005-3192(31)
3.3 Inferences from the instance VCGs and the use of these inferences

In this section, general concepts will be formulated by abstracting common properties of the causes in the instance VCGs. It can be seen from the instance VCGs in section 3.2 above, that many of the causes for the various instances have common properties. These commonalities will be analyzed in this section. This will help in making class VCGs compact, yet keep the semantics of the VCG preserved.

From the VCGs analyzed in the section 3.2 above, following inferences can be drawn:

3.3.1 Use of Unsafe functions: (CVE-2005-2558)

Functions like `strcpy()`, `strmov()` have been termed unsafe as there is no specification on the length of data being copied from the source to the destination.

Whenever causes like use of unsafe functions like `strcpy()`, `gets()` are encountered they can be generalized to “Use of unsafe functions”.

3.3.2 Range checks problems

Range checks may avoid buffer overflows. The problems with range checks can be:
1. Missing range check: (CVE-2009-0490)
   There is no range check at all when data is copied from the external source to the internal buffer
2. Unsafe Conditional range check: (CVE-2002-1337)
   There is a range check, but it is conditional. The Attacker might succeed in escaping the condition and overflow the buffer.
3. Range check separated from the copy location: (CVE-2002-1337)
   It is important that the range check is performed at the location where the data is being copied to the internal buffer. The Attacker may succeed to overflow the buffer between the locations of copying of data and the locations of the range check.
4. Incorrect range check applied:
   The problem arises when incorrect range checks are applied. This could be added as one of the causes in the VCG for “range check problems” VCG.
Whenever a cause related to a range check appears, we will generalize it as “range check problems” and add that cause to the VCG in Figure 3-10.

3.3.3. A Safe function used in an incorrect way
This generalization can be applied whenever a cause related to the use of a safe function in an incorrect way is encountered. Consider the VCG for the instances of CVE-2005-2943 and CVE-2007-0063.

The cause for the vulnerability CVE-2005-2943 is “Incorrect use of safe function strncpy()” and the cause for the vulnerability CVE-2007-0063 is “Incorrect use of safe function memcpy()”. These causes are related by a fact that a safe function is used in an incorrect way.

If any cause is encountered having the common property of “safe function being used in incorrect way” then it can be generalized to the general cause of “Safe function used in an incorrect way”.

3.4 The generalization procedure
In the section 3.3 above, we presented what generalization means and the basis for the generalization.

The generalization procedure will be discussed in this section.

Let current_node be a node that is being processed during the procedure of creating a class VCG and found that it can be generalized into generalize_node.

Take the following steps for generalizing the current_node.

1. Check if current_node is present in the VCG for the generalize_node.
2. If yes, go to step 3 else add current_node as one of the causes for the generalize_node VCG.
3. For further processing generalize_node will be treated as current_node. (This can be done by performing current_node=generalize_node. Any transformations meant for current_node will be done with the generalize_node)
In this chapter, we showed VCGs for a number of instances of the buffer overflow. We also explained how different causes having common properties can be generalized to a single cause in the class VCG. Generalization will help to limit the size of the class VCGs so that they do not over grow. At the end of this chapter the generalization procedure of the node was presented. The main outcomes of this chapter are the concept of generalization and the generalization procedure of a node.

Figure 3-12: Flowchart for the Generalization procedure

3.5 Summary
In this chapter, we showed VCGs for a number of instances of the buffer overflow. We also explained how different causes having common properties can be generalized to a single cause in the class VCG. Generalization will help to limit the size of the class VCGs so that they do not over grow. At the end of this chapter the generalization procedure of the node was presented. The main outcomes of this chapter are the concept of generalization and the generalization procedure of a node.
4 Proposition

4.1 Introduction
In this section, the method of merging of instance VCGs to create a class VCG will be explored as described in section 2.7.

In this chapter, we will also show how the merging procedure is devised. This procedure will be used to merge the instance VCGs to create a class VCG:

**Definitions:**
Before the beginning of the section, I want to define some terms that we will be used in the following discussion.

**Transformation:** Transformation refers to the change in the graph after a new edge or new node is added to the graph. That is, adding a new edge or node is the process of applying transformation.

**Results/Expressions:** Result refers to the semantics of the VCG. It is an expression that gives information about different possibilities in which the particular vulnerability can be blocked.

**Instance VCG:** Refers to the Vulnerability Cause Graph created for the instance of particular class of vulnerability.

**Class VCG:** Refers to the Vulnerability Cause Graph that represents the complete class of vulnerabilities. We will Class VCG from number of instance VCGs.

**Source VCG:** The VCG that is to be merged. It is usually an instance VCG.

**Destination VCG:** The VCG in to which an instance VCG is merged. It is usually a class VCG.

4.2 Concept of merging of VCGs (The manual method)
Consider the following two VCGs to be merged:

![Fig. 4a: The two VCGs to be merged](image)

Theoretically, to merge these two VCGs just remove exit nodes (here CVE-1 and CVE-2) and have a new exit node (say CVE-12). The edges connecting to the exit node in both the graphs in fig. 4a, will now point to the new common exit node. See fig. 4b

![Fig. 4b: Creating a common exit node](image)
The final merged VCG is shown in fig. 4f.
4.3 Merging of VCGs by edge processing

Instead of merging the two VCGs directly as described in section 4.2, each “edge” in an instance VCG (the VCG which is to be merged) will be processed in this method. Based on processing outcomes new edges/nodes will be added/deleted in the destination VCG.

Consider the same example CVE-2 to be merged into CVE-1. CVE-2 is the source VCG and CVE-1 is the destination VCG.

**Step 1:** Edge from A to CVE-2
Consider the edge connecting A to CVE-2, an exit node. Remove the exit node CVE-2 and rename the exit node CVE-1 to CVE-12. Node A is already present in CVE-1. So no new nodes will be added in the destination CVE.

**Step 2:** Edge EA from source VCG
It is seen that node E is not present in the destination graph. Therefore add node E to the destination graph. Now the newly added node should be connected to some other node. It can be seen that node E is connected to node A. Therefore in the destination VCG connect nodes E and A, as shown in the fig. 4h.

**Step 3:** Edge DA from source VCG
It is seen that node D is not present in the destination graph. Therefore add node D and then add an edge DA to the destination graph. The final VCG is shown in fig. 4i.

The VCG is same as VCG shown fig. 4c in section 4.2.

Now merge the VCG CVE-3 into VCG CVE-12. CVE-3 will be our source VCG and CVE-12 will be our destination VCG.

**Step 4:** Consider edge connecting F to exit node CVE-3
As usual first remove the exit node from the source VCG. In this case remove the exit node CVE-3 and rename the exit node in destination VCG to CVE-123.
Step 5: After processing edge GF. The destination VCG will look as shown in fig. 4k.

![Fig. 4k](image)

Step 6: Now process edge DG. It can be seen that nodes D and G are already present in the destination VCG. So connect these nodes in the destination VCG.

![Fig. 4l](image)

It can be observed that the final VCG obtained (fig. 4l) is the same as the one obtained from the basic method (fig. 4f) described in section 4.2. This process can be generalized to merge one VCG in another VCG. The generalized merging procedure is discussed in the next section.

Note:

**Renaming of the exit nodes after merging**

As stated in the introduction chapter, the use of exit node is that it helps in identifying which specific vulnerability is represented by a VCG. After merging of the VCGs, we would like to identify which instance VCGs were merged to create it. The exit node can be labeled accordingly. The label should be able to give information about the instance VCGs that were merged to create a merged VCG. If the entire instance VCGs belonging to particular class of vulnerability are merged, the exit node can be labeled as CWE identifier.

CWE identifier is a community-developed formal list of common software weaknesses. It serves as a common language for describing software security weaknesses, a standard measuring stick for software security tools targeting these vulnerabilities, and as a baseline standard for weakness identification, mitigation, and prevention efforts. Leveraging the diverse thinking on this topic from academia, the commercial sector, and government, CWE unites the most valuable breadth and depth of content and structure to serve as a unified standard.
4.4 The procedure for merging two VCGs
This section presents the edge by edge merging procedure for the two VCGs. (Refer Flowchart for the basic procedure of merging VCGs)
In the following procedure, instance VCG and class VCG are referred to as source VCG and destination VCG respectively.
1. Remove the exit node from the instance VCG (source VCG)
2. For each edge in the instance VCG.
   
   3. if(srcNode is present in the destination VCG)
      
      goto step 4
   
   else{if (srcNode can be generalized)
      
      generalize the srcNode. Go to step 4.
   
   else
   
   if not; add the srcNode to the class VCG (destination VCG)
   
   }
4. if(destNode is present in the destination VCG)
   
   goto step 5
   
   else{if (destNode can be generalized)
      
      generalize the destNode. Go to step 5.
   
   else
   
   if not; add the destNode to the class VCG (destination VCG)
   
   }
5. if(current edge is present in the class VCG)
   
   ignore and go to next edge
   
   else
   
   add the edge connecting the srcNode and the destNode in the class VCG.
   
   }
Figure 4-1: Flowchart for the basic procedure of merging VCGs
4.5 Semantics of the VCG at each step of the transformation

One of the final goals of the modeling of the vulnerabilities is to determine how to prevent the vulnerabilities. The semantics for the VCGs are described in such terms and the semantics are derived from the mitigation of causes. In section 2.5, the semantics of the instance VCG is described.

One of the goals of modeling a class of vulnerability is to determine the set of security activities to prevent the complete class of vulnerability or at least the set of vulnerabilities the developer wishes to avoid.

To achieve this purpose, it is necessary to make sure that the semantics of the VCGs are preserved for all transformations that are applied during merging of the instance VCGs to create class VCG. That is, the semantic of the new class VCG should still satisfy the semantic that was present before applying the transformation. This property will be called Semantic preservation property henceforth.

Notation used:
P\textsuperscript{number}:
   The superscript signifies that the node connects with N_i.
The subscript signifies the serial number of the node that is connected to N_i.

4.5.1 Semantic preservation property

The following statement should be true after any kind of transformation is applied to a VCG:

“If $B (N)_{\text{new}}$ is satisfied, so is $B (N)_{\text{old}}$”

Where,

$B (N)_{\text{old}}$: Semantics for a node N before the transformation is applied

$B (N)_{\text{new}}$: Semantics for a node N after the transformation is applied

In the merging procedure described in section 4.4, there are two distinct steps on how the semantics of the VCG can be affected.

1. A node is added to the class VCG
2. An edge is added to the class VCG

I will first show how the semantics are affected when we add a node to the graph.
4.5.2 Change in the semantics of a VCG when a node is added to the VCG

1. Consider a VCG:

\[ P_1^i \rightarrow P_2^i \rightarrow P_3^i \rightarrow P_k^i \rightarrow P_1^j \]

The semantic expression for the node \( N_i \) in the above graph is:

\[ B(N_i) = M(N_i) \lor \left( \land_{p \in P_{1-k}^i} B(p) \right) \] (1)

2. Add a node to the VCG as shown:

\[ P_{new}^i \rightarrow P_1^i \rightarrow P_2^i \rightarrow P_3^i \rightarrow P_k^i \rightarrow P_1^j \]

The semantic expression for the node \( N_i \) is:

\[ B(N_i) = M(N_i) \lor \left( \land_{p \in P_{1-k}^i} B(p) \right) \] (2)

It can be observed that the expressions (1) and (2) are both same.

We draw following conclusions:
1. Adding a new node does not affect the semantics of the graph. Since it is not connected to any other node, it cannot appear in the expression and implies that adding a node does not affect the semantic of the VCG in any way.
2. For the node to have any effect on the semantic of the graph, it is important that it is connected to some other node by an edge.
Thus the question is transformed from “How do the semantics change when a node is added to the VCG?” to “How do the semantics change when an edge is added to the VCG?”
4.5.3 Change in the semantics when an edge is added to the VCG

We will now show how adding an edge to the VCG satisfies the semantic preservation property.

For the case of adding an edge, there are two sub-cases:

a. An edge is added to a node whose predecessor node set is not empty
b. An edge is added to a node whose predecessor node set is empty

4.5.3.1 Case 1: An edge is added to a node whose predecessor node set is not empty

Semantic preservation property holds true if a new edge is added to a node whose predecessor node set is not empty.

We will prove the statement in three steps by the method of induction.

1. The base case.
2. The induction step for $k$
3. The final step for the case of $k+1$.

Consider the base case first:

In the base case, let node $P_1$ is connected to the node $N_1$.

The expression for this case is:

$$ B(N_1) = M(N_1) \lor (B(P_1^1)) \ldots \quad (3) $$

Add a new edge as shown in the diagram. Then the new expression is:

$$ B(N_1) = M(N_1) \lor ((B(P_1^1) \land (B(P_2^1))) $$

$$ B(N_1) = (M(N_1) \lor (B(P_1^1))) \land ((M(N_1) \lor (B(P_2^1))) \ldots \text{Distributive law for logical Expressions} $$

$$ B(N_1) = (1) \land ((M(N_1) \lor (B(P_2^1))) $$

Thus the statement is satisfied for the base case.
**Proposition**

**Induction step:**
Assume that the statement is true for \( k \) number of nodes \( (N_1, N_2, \ldots, N_k) \)

If an edge is added by connecting any node \( P_{\text{new}}^i \) to \( N_i \) the statement holds true.

Before adding an edge:

\[
B(N_i) = M(N_i) \lor \left( \land_{p \in p_{1..k}} B(p) \right) \quad (4)
\]

Add a new edge connecting \( N_i \) to \( P_{\text{new}}^i \)

\[
B(N_i) = M(N_i) \lor \left( \land_{p \in p_{1..k}} B(p) \land B(P_{\text{new}}^i) \right)
\]

The new equation will be:

\[
B(N_i) = (4) \land \left( M(N_i) \lor B(P_{\text{new}}^i) \right) \quad (5)
\]

Induction step: Assume that above statement holds true (That is, the expression before applying transformation is preserved)

**Final step:**
We want to prove the statement for \( k+1 \) node. \( (N_1, N_2, \ldots, N_k, N_{k+1}) \)
Proposition

For the above graph with k+1 nodes,
Before adding an edge:

\[ B(N_i) = M(N_i) \lor (\bigwedge_{p \in P_1^{i,k+1}} B(p)) \] \hspace{1cm} (6)

After adding a new edge:

\[ B(N_i) = M(N_i) \lor (\bigwedge_{p \in P_1^{i,k}} B(p) \land B(P_{new}^i)) \]

\[ B(N_i) = M(N_i) \lor (\bigwedge_{p \in P_{k+1}} B(p) \land B(P_{new}^i)) \]....Here range “1 to k+1” are divided into two parts: as “1 to k” and as (k+1)

\[ B(N_i) = (M(N_i) \lor (\bigwedge_{p \in P_1^{i,k}} B(p))) \land (M(N_i) \lor B(P_{k+1}^i)) \land (M(N_i) \lor B(P_{new}^i)) \]...Distribution law of

Term1 and term2 in the above expression can be replaced by expression (6) ....... (This proves that the expression before applying the transformation is preserved)

Term1 and term3 represents (5). This expression in turn satisfies the previous expressions- Only to emphasize that semantic preservation property holds true.

About other graph elements:
The same arguments can be applied for other node \( N_j \).

4.5.3.2 Case 2: An edge is added to a node whose predecessor node set is empty

Semantic preservation property does not hold true if an edge is added to a node whose predecessor node set is empty.

We will prove the statement by the method of contradiction.
For this, assume that the Semantic preservation property holds true if an edge is added to a node whose predecessor node set is empty.

By method of induction the basic case should also be true. Now we will prove that the basic case does not hold true. This will contradict the assumption.

Basic case:

![Diagram](attachment:image.png)
Proposition

The semantic for the base case is:
\[ B(N_1) = M(N_1) \lor B(P_1^1) \ldots (7) \]

The new semantics after a node \( X_1^1 \) is added to the VCG is:
\[ B(N_1) = M(N_1) \lor B(P_1^1) \lor B(X_1^1) \]

From the expression we can infer that mitigating the cause \( X_1^1 \) will be enough to mitigate the cause \( N_1 \).

However, in equation (7), we see that the cause \( X_1^1 \) is not present and mitigating it will not satisfy the previous semantics. The semantics are not preserved. Thus, our assumption is wrong. We can infer that:
Semantic preservation property does not hold true if an edge is added to a node whose predecessor node set is empty.

It can be inferred that during the merging procedure, if a condition arises where it is required to add an edge to a node whose predecessor node set is empty, this edge should be ignored. This will be referred to as “Rule of ignoring edges” henceforth.

The conditions and actions to be taken for these cases will be presented in the part that follows.
4.6 Issues with the proposed merging procedure:
I will first discuss the implication of the statement proved in the section 4.5.3.2. The implication of the statement presented in section 4.5.3.2 can be explained with the following example.
Condition 1:
Consider two VCGs (shown in fig. 4m) to be merged.

To prevent the vulnerability in VCG V1, *cause A* or *cause B*. should be mitigated.
To prevent the vulnerability in VCG V2, *cause A*. should be mitigated.
After integration by the procedure described in section 4.4, the final merged VCG will look as shown in fig. 4n.

Fig. 4m: VCG V1      VCG V2

From the VCG, it can be inferred that to prevent both the vulnerabilities either the *cause A* or the *cause B*. should be mitigated.
However, mitigating the *cause B* alone will not avoid vulnerability presented by VCG V2. The *cause B* has no role in causing the vulnerability represented by VCG V2, as it does not even appear in the VCG for CVE-1.

Thus, Semantic preservation property is not adhered if the merging procedure described in section 4.4.
The only possible solution to avoid both the vulnerabilities is mitigating the *cause A*. Thus, the final VCG should look as shown in fig. 4o:

Fig. 4o: Final VCG

Thus, the merging procedure needs update to handle these conditions. We will discuss the solution in section 4.7.
Condition 2: Formation of cycle during the merging procedure
The merging of the VCGs might result in the formation of the cycles in the final VCG. As defined VCGs are directed acyclic graphs and the formation of the cycles is a problem. We will discuss about the cycles during merging procedure in detail in section 4.8 and 4.9.
4.7 Rule of ignoring edges (The two basic rules)

### Notations Used in the following section:
1. \( V(G) \): Set of Vertices of a graph \( G \).
2. \( E(G) \): Set of edges for a graph \( G \).
3. Edge \((x, y)\) is considered to be directed from \( x \) to \( y \). Following can be said:
   a. \( y \) is called the head of \( x \)
   b. \( x \) is called the tail of \( y \)
   c. \( y \) is the direct successor of \( x \)
   d. \( x \) is the direct predecessor of \( y \)
4. Edge \( A \rightarrow B \): A directed edge going from node \( A \) to \( B \) i.e. edge \((A,B)\)
5. \( N'(X) \): Successor node set for the node \( X \)
6. \( N(X) \): Predecessor node set for the node \( X \)
7. \( E'(X) \): Edge set going out of node \( X \).
8. \( E(X) \): Edge set going into node \( X \).
9. \( P(X) \): Direct predecessor set of \( X \).
10. \( PD(X) \): Post Dominator set of node \( X \)

### Rule 1:
Consider a case where we are processing an edge \( A \rightarrow B \) from a source VCG. If node \( A \) (source node) has no predecessors in VCG V2 and is present in the destination VCG V1, remove all the edges that come into node \( A \) in the destination VCG and all the nodes to which the node \( A \) post-dominates in the destination VCG.

Check for the following conditions:
- In V1, (Edge \( B \rightarrow A \))
- In V1, none of the nodes belonging to the post-dominator set of node \( A \) should be present in VCG V2.
  This condition mathematically can be represented as:
  \[ \text{Check } PD \in (A) : X \notin V(V2) \]
- In V2, there should be no edges coming into node \( A \).
  This condition can be mathematically represented as:
  \[ \text{In } V2 \ (E(A) = \emptyset) \]

If these conditions are satisfied then perform the following transformation:
- Remove all the nodes belonging to the post-dominator set of node \( A \) in VCG V1.
  Mathematically the outcome can be represented as:
  \[ \text{In } V1 \ PD(A) \notin V(V1) \]
Here, node A in CVE-1 has no predecessors and it is present in the destination VCG CVE-2. Node A post-dominates node B in the destination VCG. After we apply the above rule, node B and the edge B-A will be removed from the final merged graph.

Rule 2:

Consider that we are processing the edge B->A from the source VCG. Node A has no predecessors in the destination VCG and node A post dominates the node B in the source VCG. So during merging, do not process the nodes to which the node A post-dominates in the source VCG and all the edges that come into the node A in the source VCG.

Mathematically:
Check for the following conditions:
   a. In V2, (Edge B->A)
   b. In V2, none of the nodes belonging to the post-dominator set of node A should be present in VCG V1.
      Mathematically this condition can be represented as:
      Check X ∈ PD (A): X∉V (V1)
   c. In V1, There should be no edges coming into node A.
      This condition can be mathematically represented as:
      In V1, (E (A) ==Ø)
If these conditions are satisfied then perform the following transformation:
   a. Remove all the nodes belonging to the post-dominator set of node A in VCG V2.
      Mathematically the outcome can be represented as:
      In V2, PD (A) ∉V (V2)

In the example above, node A post-dominates node B and node A has no predecessors in the destination VCG. Therefore we will ignore node B and the edge B-A during the merging process. The final VCG for the above example will look as shown here:
4.8 Cycles during the merging procedure
VCGs are directed acyclic graphs. During the merging procedure conditions may arise where cycles may be created in the VCGs. Obviously we wish to avoid the occurrence of the cycles in the VCGs. First, we will discuss why a cycle might arise in the VCG and see a basic example for the same. Then, we will discuss the various types and cases of the cycles that can be encountered and the actions needed to be taken when those conditions arise.

4.8.1 An example of formation of a cycle by merging two VCGs
First, it is important to understand how cycles might be encountered during the merging procedure. This will be explained with a basic example below:
Consider the following two VCGs to be merged.

![Diagram of two VCGs](image)

Figure 4-2: Cycle formation during merging

On processing edge A->B from VCG V2, it will be found that edge A->B is not present in VCG V1. Following the merging procedure described in section 4.4, edge A-B will be added to VCG V1. VCG V1 will look as below:

![Diagram of VCG V1 with cycle](image)

From the diagram above, we see that this results into a cycle in the merged VCG. There is an edge from node A to B and an edge from node B to A.

After having understood what we mean by the cycles in the VCG, we now look to why the VCGs V1 and V2 are modeled in such way that it results into formation of cycles when they are merged.

4.8.2 The reasons for ambiguities and inconsistencies in the VCGs
The VCGs are the product of manual analysis by different people. Different people think in different ways. The way a person sees a particular vulnerability-cause or cause-cause relationship might differ from how other person sees it. Vulnerability modeling is a creative process. This may result in the apparent ambiguities in the VCGs. The placement of causes as the ones shown in VCGs V1 and V2 in Figure 4-2 is possible because different people are involved in the creation of these VCGs.

Once the cycles are detected, they need to be resolved. As cycles always involve two or more edges, they cannot be resolved with edge-by-edge processing. We will explain and use the concept of basic blocks to resolve the cycles detected during the merging procedure.

4.8.3 Basic blocks basics
Basic block: A basic block is a sequence of nodes with a single entry point and a single exit point.
Entry point: An entry point of the basic block.
Exit point: An exit point of the basic block.
4.9 Basic block rules for resolving cycles during the merging procedure

I have formulated the different cases of the formation of the cycles that can be encountered during the merging procedure. In this section, we will discuss each one of them.

I have divided the different cases based on the number of nodes involved in the formation of the cycle in the VCGs.

The following three types can arise on the basis of the number of nodes involved in the formation of the cycles in VCGs. Each type has three sub-cases based on the presence or absence of the predecessor or successor nodes for the basic block in the source VCG.

Type 1: The source VCG containing a basic block with two nodes and the destination VCG containing basic blocks with more than two nodes.
   1) Case 1: Basic block in source VCG has no predecessors and successors nodes (section 4.9.1.1)
   2) Case 2: Basic block in source VCG has predecessor nodes but no successor nodes (section 4.9.1.2)
   3) Case 3: Basic block in source VCG has both predecessor and successor nodes (section 4.9.1.3)

Type 2: The source VCG and destination both containing basic blocks with more than two nodes.
   1) Case 1: Basic block in source VCG has no predecessors and successors nodes (section 4.9.2.1)
   2) Case 2: Basic block in source VCG has predecessor nodes but no successor nodes (section 4.9.2.2)
   3) Case 3: Basic block in source VCG has both predecessor and successor nodes. (section 4.9.2.3)

Type 3: The source VCG and destination both containing basic blocks with two nodes.
   1) Case 1: Basic block in source VCG has no predecessors and successors nodes (section 4.9.3.1)
   2) Case 2: Basic block in source VCG has predecessor nodes but no successor nodes (section 4.9.3.2)
   3) Case 3: Basic block in source VCG has both predecessor and successor nodes (section 4.9.3.3)
4.9.1 Type 1: Source VCG containing basic block with two nodes and the destination VCG having basic block with more than two nodes.

4.9.1.1 Case 1: B2 has no predecessors and successors node.

Take actions to satisfy following conditions
In VCG V1:

a. Create a conjunction of node A and B that replaces the node B. Remove both nodes A and B. The reason to create a conjunction of these two nodes is because they are the entry point and exit point of the basic blocks in both the source and the destination VCG.

Mathematically this step can be represented as follow:
Node AB: AB \in V(V1) and A, B \notin V(V1)

b. Connect the newly created conjunction node to the immediate successor of node B.

In this case, the immediate successor is the exit node CVE-2. Thus we add the edge AB->CVE2 Mathematically it can be represented as follow:
Edge (AB-> CVE2) \in E (V1)

c. All other edges and nodes are deleted. In this case we will delete the nodes C, D and G.

We have to delete these nodes to adhere to the Semantic preservation property.
These causes do not appear the source VCG. We can of course mitigate these causes to avoid vulnerability CVE-2, but not to avoid the vulnerability CVE-1. Thus, we will ignore the causes C, D and G.
In VCG V2:
   a. Create a conjunction node A and B. Remove both nodes A and B. The reason to create a
      conjunction of these two nodes is because they are the entry point and exit point of the basic
      blocks in both the source and the destination VCG.
      Mathematically this step can be represented as follow:
      \[ \text{Node AB: } AB \in V (V2) \text{ and } A, B \notin V (V2) \]
   b. Connect the newly created conjunction node to the immediate successor of node B.
      In this case, the immediate successor is the exit node CVE-2. Thus we add the edge AB->CVE2
      Mathematically it can be represented as follow:
      \[ \text{Edge (AB-> CVE1) } \in E (V2) \]

Final VCG

Steps in VCG V1
   a. Create the conjunction of node A and B which replaces node B. Remove node A.
   b. Add edge AB->CVE2
   c. Ignore all other nodes and edges. (A nodes in V2 have no predecessors, thus we have to ignore all other edges on V1)

Steps in VCG V2
   a. Create the conjunction of node A and B which replaces the node A. Remove node B.
   b. Add edge AB->CVE1

Merging
The normal merging procedure can be continued after this step.
Proposition

4.9.1.2 Case 2: B2 has predecessor nodes but no successor nodes

The Basic blocks of concern
Basic block B1: (A-C-D-B) in V1
Basic block B2: (B-A) in V2

Conditions (Shown by dotted arrows in the diagram)
a. Entry node A of B1 in V1 is exit node B of B2 in V2.
b. Exit node B of B1 in V1 is entry node of B2 in V2.

Perform the following transformation in the Graph:
In VCG V1:
   a. Create a conjunction node A and B. Remove the both node A and B. The reason to create a conjunction of these two nodes is because they are the entry point and exit point of the basic blocks in both the source and the destination VCG.
      Mathematically this step can be represented as follow:
      Node AB: AB \in V (V1) and A, B \notin V (V1)
   b. Connect the newly created conjunction node to the immediate successor of node B.
      In this case, the immediate successor is the exit node CVE-2. Thus we add the edge AB -> CVE2
      Mathematically it can be represented as follow:
      Edge (AB -> CVE2) \in E (V1)
   c. Remove all the edges going out of the node A. In this case, we will remove edges AC and AD.
      Mathematically it can be represented as follow:
      E^- (A) \notin E (V1)
   d. All the edges coming into node B should connect to the new conjunction node AB.
      In this case, we will create the edges C->AB and D->AB.
      It can be represented as follow:
      E (B) connects to node AB
   e. All the incoming edges to the node A will connect to all the immediate successor of the node A. In this case, node G will connect to the nodes C and D.
      Mathematically it can be represented as follow:
      E (A) connects to N^- (A)

In VCG V2:
   a. Create a conjunction node A and B. Remove the both node A and B. The reason to create a conjunction of these two nodes is because they are the entry point and exit point of the basic blocks in both the source and the destination VCG.
      Mathematically this step can be represented as follow:
      Node AB: AB \in V (V2) and A, B \notin V (V2)
   b. Connect the newly created conjunction node to the immediate successor of node B.
      In this case, the immediate successor is the exit node CVE-2. Thus we add the edge AB -> CVE2
      Mathematically it can be represented as follow:
      Edge (AB -> CVE1) \in E (V2)
   c. All the edges coming into node B will connect to the new conjunction node AB.
      In this case, we will create the edges E->AB.
      Mathematically it can be represented as follow:
      E (B) connects to node AB
**Proposition**

**Final VCG**

![Diagram of VCGs]

**Steps in VCG V1**
- a. Create the conjunction of node A and B which replaces node B. Remove node A.
- b. Add edge AB→CVE2
- c. Edge AC and AD removed.
- d. Connect CB and DB to node AB.
- e. Connect G to nodes C and D.

**Steps in VCG V2**
- a. Create the conjunction of node A and B which replaces node B. Remove node B.
- b. Add edge AB→CVE1
- c. Edges going into node B will now go into node AB. (E connects into node AB)

**Merging**

The normal merging procedure can be continued after this step.

**Note:**

**Are the transformations applied justified and do we lose information during merging procedure?**

With the above steps, we are basically trying to resolve the ambiguities and inconsistencies that were present in the VCGs that were being merged. Do these steps and actions make sense?

Yes, our objective is to adhere to the Semantic preservation property. With the above steps we make sure that this property is adhered to in whatever transformations we apply.

Aren’t we loosing vital information about the causes during the merging procedure?

Yes, we do loose the information during the merging procedure. However, our objective is to find common sets of activities that if performed will mitigate all the concerned vulnerabilities that were merged. Again, in a bid to adhere to Semantic preservation property, we have to ignore some of the nodes and causes during the merging procedure.
**4.9.1.3 Case 3: B2 has both predecessor and successor nodes.**

**Basic Blocks in VCG V1 and V2 of concern**

a. Basic Block B1: (A-C-D-B) in V1

b. Basic Block B2: (B-A) in V2

**Conditions** (Shown by dotted arrows in the diagram)

a. Entry node A of B1 in V1 is exit node of B2 in V2.

b. Exit node B of B1 in V1 is entry node of B2 in V2.

**Take actions to satisfy following conditions**

**In VCG V1:**

a. Create a conjunction node A and B. Remove the both node A and B. The reason to create a conjunction of these two nodes is because they are the entry point and exit point of the basic blocks in both the source and the destination VCG.

   Mathematically this step can be represented as follow:

   Node AB: $AB \in V(V1)$ and $A, B \notin V(V1)$

b. Connect the newly created conjunction node to the immediate successor of node B.

   In this case, the immediate successor is the exit node CVE-2. Thus we add the edge $AB \rightarrow CVE2$

   Mathematically it can be represented as follow:

   Edge $(AB \rightarrow CVE2) \in E(V1)$

c. Remove all the edges going out of the node A. In this case, we will remove edges AC and AD.

   Mathematically it can be represented as follow:

   $E^+(A) \notin E(V1)$

d. All the edges coming into node B should connect to the new conjunction node AB.

   In this case, we will create the edges C$\rightarrow$AB and D$\rightarrow$AB.

   It can be represented as follow:

   $E(B)$ connects to node AB

f. All the incoming edges to the node A will connect to all the immediate successor of the node A.

   In this case, node G will connect to the nodes C and D.

   Mathematically it can be represented as follow:

   $E(A)$ connects to $N^+(A)$

**In VCG V2:**

a. Create a conjunction node A and B. Remove the both node A and B. The reason to create a conjunction of these two nodes is because they are the entry point and exit point of the basic blocks in both the source and the destination VCG.

   Mathematically this step can be represented as follow:

   Node AB: $AB \in V(V2)$ and $A, B \notin V(V2)$

b. Connect the newly created conjunction node to the immediate successor of node B.

   In this case, the immediate successor is the exit node CVE-2. Thus we add the edge $AB \rightarrow CVE2$

   Mathematically it can be represented as follow:

   Edge $(AB \rightarrow CVE1) \in E(V2)$
c. Make conjunction node AB a direct predecessor of the node to which the node E connects to. In this case the node E connects to exit node CVE-1. So node AB will be made a direct predecessor of the exit node CVE-1. Mathematically this step can be represented as:
\[ AB \in P (N' (E)) \]
d. Make direct successors of node A, a direct predecessor of the conjunction node AB. In this case, a direct successor of node A is the node E. Thus, E will be made a predecessor of the new conjunction node AB. Mathematically this step can be represented as:
\[ S (A) \in N' (AB) \]
e. The nodes connected to the node B will now connect to the node E. In this case node F will connect to the node E. Mathematically this step can be represented as:
\[ N (B) \in N (E) \]

Based on these action steps, the new VCGs will look as follows:

![Diagram](image)

**Steps in VCG V1**
- a. Create the conjunction of node A and B which replaces node B. Remove node A.
- b. Add edge AB->CVE1
- c. Edge AC and AD removed.
- d. Connect CB and DB to node AB.
- e. Connect G to nodes C and D.

**Steps in VCG V2**
- a. Create the conjunction of node A and B which replaces node B. Remove node B.
- b. Add edge AB->CVE1
- c. Make the conjunction node AB, a direct predecessor of the exit node.
- d. Make E a predecessor of node AB.
- e. Node F will be now predecessor of node E.

**Merging**
The normal merging procedure can be continued after this step.

**Why are the positions of the nodes interchanged during merging procedure?**
I will specifically discuss the transformation steps c, d and e for the VCG V2 here. Above, we suggested the transformation:

\[ AB \in P (N' (E)) \] (Step “c” in the transformations for the VCG V2)
\[ S (A)\in N' (AB) \] (Step “d” in the transformations for the VCG V2)
\[ N (B) \in N (E) \] (Step “e” in the transformations for the VCG V2)

Basically, with these transformation steps what we achieve is the interchange in the placement of the node AB and the node E.

Why do we need this transformation?
Let us see how the VCGs will look if these transformations are not applied.
In VCG V1, the conjunction node AB will connect to the exit node CVE-2. In VCG V2, the conjunction node AB will connect to node E which in turn will connect to the exit node CVE-1. So, we assume here that the destination VCG is correct and the source VCG has inconsistency in the placement of the causes.
Thus, we will interchange the positions of conjunction node AB and the node E in the source VCG so that we will get the semantically correct VCGs at the end of the merging procedure.
4.9.2 Type 2: The source VCG and destination both having basic block with more than two nodes

4.9.2.1 Case 1: B2 has no predecessor and successor node.

The basic Blocks in VCG V1 and V2 of concern
a. Basic Block B1: (A-C-D-B) in V1
b. Basic Block B2: (B-C-D-A) in V2

Conditions (Shown by dotted arrows in the diagram)

a. Entry node A of B1 in V1 is exit node of B2 in V2.
b. Exit node B of B1 in V1 is entry node of B2 in V2.

Take actions to satisfy following conditions

In VCG V1: (For the description of the transformations applied here, please refer to the transformations applied for VCG V1 in section 4.9.1.2)
a. Node AB : AB ∈ V(V1) and A, B ∉ V(V1)
b. Edge (AB -> CVE2) ∈ E(V1)
c. E^+(A) ∉ E(V1)
d. E (B) connects to node AB

e. E (A) connects to N^+(A)

In VCG V2:

a. Create a conjunction node A and B. Remove the both node A and B. The reason to create a conjunction of these two nodes is because they are the entry point and exit point of the basic blocks in both the source and the destination VCG.

Mathematically this step can be represented as follow:

Node AB: AB ∈ V(V2) and A, B ∉ V(V2)

b. Connect the newly created conjunction node to the immediate successor of node B.

In this case, the immediate successor is the exit node CVE-2. Thus we add the edge AB -> CVE2

Mathematically it can be represented as follow:

Edge (AB -> CVE1) ∈ E(V2)

c. Remove all the edges going out of node B. In this case we will remove the edges BE and BF.

This step can be represented as follows:

E^+(B) ∉ E(V2)

d. Edges going into node A will now connect to the new conjunction node AB. In this example, the nodes E and F will connect to the node AB.

This step can be represented as:

E (A) connects to node AB

e. Edges coming into node B will connect to the node where B is connected. Here, there are no edges coming into node B. Thus, there are no transformations applied here.

E (B) connects to N^+(B)
Proposition

The new VCGs after this action will look as below:

Steps in VCG V1
a. Create the conjunction of node A and B which replaces node B. Remove node A.
b. Add edge AB->CVE2
c. Remove edge AC and AD.
d. Connect CB and DB to new node AB.
e. Incoming edges to node A, if any, will be connected to the nodes where A is connected. (Here edges GC and GD will be created)

Steps in VCG V2
a. Create the conjunction of node A and B which replaces node B. Remove node B.
b. Add edge AB->CVE1
c. Remove edges BE and BF.
d. Connect edges from node E and F to new node AB.
e. Incoming edges to node B, if any, will be connected to the nodes where B is connected. (There are no incoming edges coming into node B).

Merging
The normal merging procedure can be continued after this step.
4.9.2.2  Case 2: B2 has a successor node.

The basic blocks in VCG V1 and V2 of concern
a. Basic Block B1: (A-C-D-B) in V1
b. Basic Block B2: (B-C-D-A) in V2

Conditions (Shown by dotted arrows in the diagram)
a. Entry node A of B1 in V1 is exit node of B2 in V2.
b. Exit node B of B1 in V1 is entry node of B2 in V2.

Take actions to satisfy following conditions
In VCG V1: Same as in the case “B2 has no predecessor and successor node” in section 4.9.1.1.
In VCG V2: (For the description of the transformations here, please refer to transformations applied for VCG V2 in the section 4.9.2.1)
a. Node AB : AB \( \not\in \) V(V2) and A, B \( \not\in \) V(V2)
b. Edge (AB-> CVE1) \( \not\in \) E(V2)
c. \( E^+(B) \) \( \not\in \) E(V2)
d. Interchange node H and AB
f. \( E(B) \), if any, connects to \( N^+(B) \)

The new VCGs will look as below:

Take actions to satisfy following conditions

Steps in VCG V1
a. Create the conjunction of node A and B which replaces node B. Remove node A.
b. Add edge AB->CVE2
c. Remove edge AC and AD.
d. Connect CB and DB to new node AB.
e. Incoming edges to node A, if any, will be connected to the nodes where A is connected.
   (Here edges GC and GD will be created)

Steps in VCG V2
a. Create the conjunction of node A and B which replaces node B. Remove node B.
b. Add edge AB->CVE1
c. Remove edges BE and BF.
d. Interchange nodes H and BF.
e. Create edges from node H to node E and node F.
f. Incoming edges to node B, if any, will be connected to nodes to which node B is connected.

Merging
The normal merging procedure can be continued after this step.
### Proposition

#### 4.9.2.3 Case 3: B2 has a predecessor node.

The Basic Blocks in VCG V1 and V2 of concern
- a. Basic Block B1: (A-C-D-B) in V1
- b. Basic Block B2: (B-C-D-A) in V2

**Conditions** (Shown by dotted arrows in the diagram)
- a. Entry node A of B1 in V1 is exit node of B2 in V2.
- b. Exit node B of B1 in V1 is entry node of B2 in V2.

**Take actions to satisfy following conditions**

In VCG V1: (For the description of the transformations applied here, please refer transformations applied for VCG V1 in section 4.9.2.1)
- a. Node AB : AB ∈ V(V1) and A, B ∉ V(V1)
- b. Edge (AB-> CVE2) ∈ E(V1)
- c. E'(A) ∉ E(V1)
- d. E (B) connects to node AB
- e. E (A) connects to N(A)

In VCG V2: (For the description of the transformations applied here, please refer transformations applied for VCG V1 in section 4.9.2.1)
- a. Node AB : AB ∈ V(V2) and A, B ∉ V(V2)
- b. Edge (AB-> CVE1) ∈ E(V2)
- c. E'(B) ∉ E(V2)
- d. E (A) connects to node AB
- e. Node we will connect to the node where node B is connected. Thus, we will add the edges IE and IF in this case. Mathematically, the transformation can be represented as:
  E (B) connects to N*(B)

**Steps in VCG V1**
- a. Create the conjunction of node A and B which replaces node B. Remove node A.
- b. Add edge AB->CVE2
- c. Remove edge AC and AD.
- d. Connect CB and DB to new node AB.
- e. Incoming edges to node A, if any, will be connected to the nodes where A is connected.
  (Here edges GC and GD will be created)

**Steps in VCG V2**
- a. Create the conjunction of node A and B which replaces node B. Remove node B.
- b. Add edge AB->CVE1
- c. Remove edges BE and BF.
- d. Connect edges from node E and F to new node AB.
- e. Incoming edges to node B, if any, will be connected to nodes where B is connected.
  (Here edges IE and IF will be created)

**Merging**

The normal merging procedure can be continued after this step.
4.9.3 Type 3: Source and destination VCG both having basic blocks with two nodes

4.9.3.1 Case 1: Without any successor or predecessor in the basic block of source VCG.

Basic Blocks in VCG V1 and V2 of concern
a. Basic Block B1: (A-B) in V1
b. Basic Block B2: (B-A) in V2

Conditions
a. Entry node A of B1 in V1 is exit node of B2 in V2.
b. Exit node B of B1 in V1 is entry node of B2 in V2.

Take actions to satisfy following conditions
In VCG V1:

a. Create a conjunction node A and B. Remove both node A and B. The reason to create a conjunction of these two nodes is because they are the entry point and exit point of the basic blocks in both the source and the destination VCG. Mathematically this step can be represented as follow:
   Node AB: AB \in V(V1) and A, B \notin V(V1)
b. Connect the newly created conjunction node to the immediate successor of node B.
   In this case, the immediate successor is the exit node CVE-2. Thus we add the edge AB->CVE2
   Mathematically it can be represented as follow:
   Edge (AB-> CVE2) \in E (V1)

In VCG V2:

a. Create a conjunction node A and B. Remove both node A and B. The reason to create a conjunction of these two nodes is because they are the entry point and exit point of the basic blocks in both the source and the destination VCG. Mathematically this step can be represented as follow:
   Node AB: AB \in V(V2) and A, B \notin V(V2)
b. Connect the newly created conjunction node to the immediate successor of node B.
   In this case, the immediate successor is the exit node CVE-2. Thus we add the edge AB->CVE2
   Mathematically it can be represented as follow:
   Edge (AB-> CVE1) \in E (V2)

Steps in VCG V1
a. Create the conjunction of node A and B which replaces node B. Remove node A.
b. Add edge AB->CVE2

Steps in VCG V2
a. Create the conjunction of node A and B which replaces node B. Remove node B.
b. Add edge AB->CVE1

Merging
The normal merging procedure can be continued after this step.
**4.9.3.2 Case 2: Source VCG with a predecessor to the basic block**

The Basic Blocks in VCG V1 and V2 of concern
a. Basic Block B1: (A-B) in V1
b. Basic Block B2: (B-A) in V2

Conditions
a. Entry node A of B1 in V1 is exit node of B2 in V2.
b. Exit node B of B1 in V1 is entry node of B2 in V2.

Take actions to satisfy following conditions (For the description of the transformations applied here, please refer the description for transformations in VCG V1 and V2 in section 4.9.3.1)

In V1:
- a. Node AB : AB ∈ V(V1) and A, B ∉ V(V1)
- b. Edge (AB -> CVE2) ∈ E(V1)
- c. Edges coming into node A will connect to the newly formed conjunction node AB.
   We can represent this step as:
   $E(A)$ connects to node AB

In V2:
- a. Node AB : AB ∈ V(V2) and A, B ∉ V(V2)
- b. Edge (AB -> CVE1) ∈ E(V2)
- c. Edges coming into node B will connect to the newly formed conjunction node AB.
   We can represent this step as:
   $E(B)$ connects to node AB

New VCGs will look as below:

**Merging**
The normal merging procedure can be continued after this step.
4.9.3.3 Case 3: A Source VCG with an successor to the basic block

The Basic Blocks in VCG V1 and V2 of concern
a. Basic Block B1: (A-B) in V1
b. Basic Block B2: (B-A) in V2

Conditions
a. Entry node A of B1 in V1 is exit node of B2 in V2.
b. Exit node B of B1 in V1 is entry node of B2 in V2.

VCG V1 VCG V2

Take actions to satisfy following conditions (For the description of the transformations applied here, please refer the description for transformations in VCG V1 and V2 in section 4.9.3.1 and section 4.9.3.2)

In V1:
- Node AB : ABєV(V1) and A,B \notin V(V1)
- Edge (AB-> CVE2)єE(V1)
- E(A) connects to node AB

In V2:
- Node AB : ABєV(V2) and A,B \notin V(V2)
- Edge (AB-> CVE1)єE(V2)
- Connect E'(B) to node AB
- Interchange the position of node AB and K.

We will describe why we need to interchange the nodes here.
In VCG V1, the conjunction node AB will connect to the exit node CVE-2.
In VCG V2, the conjunction node AB will connect to the node K which in turn will connect to the exit node CVE-1.

So we assume here that the destination VCG is correct and the source VCG has inconsistency in the placement of the causes.

Thus, we will interchange the positions of conjunction node AB and the node K in the source VCG so that we will get the semantically correct VCGs at the end of the merging procedure.

The new VCGs will look as below:

Steps in VCG V1
a. Create the conjunction of node A and B which replaces node B. Remove node A.
b. Add edge AB->CVE2
c. Connect node D to node AB

Steps in VCG V2
a. Create the conjunction of node A and B which replaces node B. Remove node B.
b. Add edge AB->CVE1
c. Connect node J to node AB. (Predecessors of the basic block)
d. Interchange the position of node AB and K. (Successor of the basic block)
Merging
The normal merging procedure can be continued after this step.

4.10 Procedure for resolving the cycles during the merging procedure
I will update the merging procedure, so that a check is performed for the formation of the cycle every time a new edge is added to the VCG. If it is found that a cycle is created while applying the transformations during merging procedure, the merging procedure should be stopped and perform the following steps to resolve the cycle.
1. Find the basic blocks of concern.
2. Find the type of the basic block.
3. Find the subtype of the basic block.
4. Perform the actions for the particular case of the basic block detected so that the cycles can be avoided.
5. Restart the merging procedure.

Each of the steps is explained below:
1. Find the basic blocks of concern.
The first step to resolving a cycle is finding the basic blocks of concern. For us the basic blocks of concern are the ones that result into the formation of a cycle.

2. Find the type of the basic block.
Once the basic blocks of concern are detected, the next step is to find the type of the basic block involved in the formation of the cycle on the basis of number of nodes involved in source VCG and the destination VCG. The number of nodes involved can be two or more than two in source VCG and the destination VCG.

3. Find the subtype of the basic block.
In this step, find the subtype of the basic block on the basis of presence or absence of the predecessor or successor nodes in the basic blocks involved.

4. Perform the actions for the particular type and subtype of basic block detected.
Perform the actions as described in various cases and sub cases described in section 4.9, so that the cycle is resolved.

5. Restart the merging procedure.
The final step is to restart the merging procedure. It will also involve the making of a log of the information related to the cycles and reporting the analysts about these conditions and the actions taken.
4.11 Merging Procedure for the VCGs with conditions

1. Remove the exit node from the instance VCG (source VCG).
2. For each edge in the instance VCG.
   
   3. if (srcNode is present in the destination VCG)
      
      goto step 4
   
      else if (srcNode can be generalized)
      
      generalize the srcNode. Go to step 4.
   
      else
      
      if not; add the srcNode to the class VCG (destination VCG)
   
   4. if (destNode is present in the destination VCG)
      
      goto step 5
   
      else if (destNode can be generalized)
      
      generalize the destNode. Go to step 5.
   
      else
      
      if not; add the destNode to the class VCG (destination VCG)
   
   5. if (current edge is present in the class VCG)
      
      ignore and go to next edge
   
      else if (the “Rule of ignoring edges” can be applied.)
      
      apply the “Rule of ignoring edges” and go to processing of the next edge
   
      Else if (the addition of the edge results in the formation of a cycle.)
      
      apply the “Procedure for resolving the cycles”.
   
      else
      
      add the edge connecting the srcNode and the destNode in the class VCG.
Proposition

Figure 4-3: Flowchart of procedure for merging of VCGs with the conditions
Proposition

Continued from previous page: Flowchart of procedure for merging of VCGs with the conditions
Proposition

Generalization procedure for the srcNode

Start

Check if srcNode is present in the VCG for src_Generalized_Node?

Yes

For further processing src_Generalized_Node will be treated as srcNode

No

Add srcNode as a cause in the VCG for src_Generalized_Node

Exit from the generalization process

Generalization procedure for the destNode

Start

Check if destNode is present in the VCG for dest_Generalized_Node?

Yes

For further processing dest_Generalized_Node will be treated as destNode

No

Add destNode as a cause in the VCG for dest_Generalized_Node

Exit from the generalization process
4.12 Summary
At the start of the chapter, we explained the manual merging procedure for the VCGs. Then we presented the example of the merging of the VCGs based on the basis of individual edge processing. This was followed by the discussion on semantics of the VCG and the effect of the various transformations applied during the merging procedure on the semantics of the VCG. That led us to the rule of ignoring edges.

Another condition that may arise during merging procedure was the creation of the cycles in the VCGs. We enumerated various cases of the cycles based on the number of nodes involved. We also presented solution in case cycles are encountered during the merging procedure.

The main outcome of the chapter was the procedure for resolving the cycles and the complete procedure of merging VCGs as well as handling of various conditions that can arise in the process of merging the VCGs.
5 Creating class VCG for Buffer Overflow class of vulnerability

5.1 Introduction
In this chapter, we will create a class VCG for buffer overflow class of vulnerability by applying merging procedure devised in chapter 4. We will use the VCGs that we created in chapter 3.

Notation Used:

\[ N \]

- Used to denote the edge from the node bearing “N” to the node where the circle connects.

5.2 Creating a class VCG for buffer overflow from instance VCGs
Consider the VCGs for CVE-2008-3252 (from section 3.2.2) and CVE-2005-3192. (from section 3.2.6) VCG for CVE-2005-3192 (Source VCG) will be merged into VCG for CVE-2008-3252 (Destination VCG).

Step1: Remove the exit node CVE-2008-3252 and label it as “Buffer overflow” as below.

Fig. 5a: VCG for CVE-2008-3252

Fig. 5b: VCG for CVE-2005-3192
Creating class VCG for Buffer Overflow class of vulnerability

Step 2:
Consider edge connecting the cause “Unsafe use of malloc” and exit node CVE-2005-3192 in the source VCG. This edge is not present in the destination VCG. Take following steps based on the merging procedure.
- Remove the exit node CVE-2005-3192
- Add the node “Unsafe use of malloc” to the destination VCG
- Add an edge connecting to this node and the exit node in the destination VCG.

**Fig. 5c: Destination VCG after step 1 and 2**

Step 3:
Consider the edge “External data influences buffer size – Unsafe use of malloc” from the source VCG. Take the following actions based on merging procedure described in section 4.11.
- Add the node “External data influences buffer size” to the destination VCG
- As the node “Unsafe use of malloc” is already present in the destination VCG, we need not add any other node to the destination VCG.
- The edge “External data influences buffer size – Unsafe use of malloc” is not present in the destination graph, so we add this edge to the destination VCG.

**Fig. 5d: Destination VCG after step 3**
Creating class VCG for Buffer Overflow class of vulnerability

Step 4:
Consider the edge “Unchecked integer arithmetic- Unsafe use of malloc function” from the source VCG. After following the merging procedure, following VCG will be derived. The actions taken are the same as in step 3.

![Diagram of VCG after step 4]

Step 5:
Consider the edge “Use of non-adaptive buffers- External data influences buffer size” from the source VCG. Take the following actions based on the merging procedure.

- Check if the node “Use of non-adaptive buffer” is present in the destination VCG. As the node is already present, we do not take any action.
- Check if the node “External data influences buffer size” is already present in the destination VCG. As the node is already present, we do not take any action.
- Check if a edge connecting these nodes is present in the destination VCG. As this edge is absent, we add this edge as shown below.

![Diagram of VCG after step 5]
Creating class VCG for Buffer Overflow class of vulnerability

Step 6:
Consider the edge “Use of non adaptive buffer- Unchecked integer arithmetic” from the source VCG. Take the same actions as in step 5 and the new VCG will look as shown in Figure 5-1.

![Figure 5-1: VCG after merging VCG for CVE-2005-3192 with VCG for CVE-2008-3252](image)

Applying the merging procedure for rest of the instance VCGs described in chapter 3, a VCG shown in Figure 5-2 will be derived.

![Figure 5-2: Merged VCGs for CVE-2009-0490, CVE-2008-3252, CVE-2005-2943, CVE-2002-0728, CVE-2002-1337 and CVE-2005-3192](image)
5.3 Merging of VCGs to apply the rule of ignoring edges

Now merge the VCG for CVE-2005-2558 (referred as V2) in the VCG shown in Figure 5-2 (referred as V1).

Start with the conjunction node connecting to the exit node in the V2. The conjunction node is not present in the VCG V1. We will add the conjunction node to V1 and also connect it to the exit node “Buffer overflow”.

The next edge that will be processed is the edge going out from the node “Use of non-adaptive buffer” node. The node is already present in VCG V1. We will just create an edge between this node and the conjunction node added in the above step.

The edge “copy of external data to internal buffer -> Use of non-adaptive buffer” from V2 is already present in the VCG V1.

The next edge from V2 to be processed is “Use of C-like strings-> Copy of external data to internal buffer”. Here the condition for applying “Rule of ignoring edges can be applied”. Thus the edge has to be ignored based on the rule for ignoring edges.

The final class VCG for buffer overflow based on the instances VCG in chapter 3 will be as shown in Figure 5-4.
Creating class VCG for Buffer Overflow class of vulnerability

(Merging of Figure 5-2 and Figure 5-3)

5.4 Summary
In this chapter, we applied the proposed merging procedure to the instance VCGs that we created in chapter 3. A class VCG for buffer overflow based on the instance VCG created and cited in chapter 3 is the final outcome in this chapter. We have been able to incorporate all the causes irrespective of their frequent or rare occurrence. This is one of the important outcomes of the merging procedure that we have proposed in section 4.1.11 From Figure 5-4, the decisions can be taken at the initial stages of the software development lifecycle, about the security activities needed to be implemented that will prevent the set of vulnerabilities that were used in the creation of the VCG in Figure 5-4. It can be inferred from the Figure 5-4 that mitigating the cause “Use of non-adaptive buffers” will completely remove the buffer overflow vulnerability.

The blocking of the following set of causes could also avoid the buffer overflow vulnerability.

- Range check problems
- Incorrect variable Initialization
- Incorrect use of safe function
- Error detected but not raised
- Malloc function used in wrong way

This is just one of many sets of causes that can be mitigated with the help of security activities in the software development process to avoid the buffer overflow vulnerability.

The optimal set of activities can be selected on the lines of the work in paper titled “A Cause-Based Approach to Preventing Software Vulnerabilities”(32).
6 Evaluation

In this chapter, we will evaluate the proposed merging procedure for creating a class VCG. I will first evaluate implementation of simple merging procedure given in section 4.4. We will also discuss the other uses of the merging procedure that were discovered during devising the procedure. At the end we will some of the cases that cannot be solved by the procedure suggested.

6.1 Implementation

In order to evaluate the merging procedure, I implemented basic merging procedure. The implementation accepts the VCGs and merges them by edge by edge processing. The procedure of generalization was incorporated in the implementation. Rule of ignoring edges and the procedure for resolving the cycles were not implemented. From the experience of the implementation of the basic procedure it can be concluded that complete procedure can be implemented. The section 6.1.6 discusses the requirements for the complete implementation of the merging procedure to create class VCGs.

6.1.1 Requirements for the basic merging procedure

Based on the solution that I proposed, the following requirements can be formulated for the basic merging procedure described in section 4.4.

1. Creating acyclic directed graphs
2. Addition and deletion of the nodes and edges in the graph
3. Making the mathematical representation for the graphs

6.1.2 Choice of language

Java was the choice of programming language since that could let us focus completely on the problem we had and not get lost in the implementation details compared to if we had chosen C or C++. We give more information about Java below.

Java:
Java 9 is a programming language originally developed by James Gosling at Sun Microsystems. It was released in 1995 as a core component of the Java platform. Java being object oriented allows the creation of modular programs and reusable code. Several other advantages are platform independence, security, multithreading and robustness. The most important advantage of Java is that it is an open source. Sun Microsystems relicensed most of the Java technologies under the GNU General Public License. The disadvantage Java has is that it is significantly slower and more memory-consuming than natively compiled languages such as C and C++. However, that is not much of a concern as our application is not a time-critical application.

6.1.3 Choice of Java library

It was important to at least test the basic concept of merging of the VCGs. The graph theory is used extensively in the merging procedure. For the basic testing initially, we thought that we would implement the graph theory concepts. However, it was not feasible to go into details of implementing the graph theory concepts. During the research, we came across open source java graph libraries like JGraphT 10, Annas, GraphsJ. Implementing the basic graph theory would have distracted me from the problem that we were solving and so it was decided that I will use one of the open source libraries. JGraphT is a free java class library that provides mathematical graph-theory objects and algorithms.

9 http://www.java.com/en/
10 http://www.jgrapht.org/
Upon comparing it was found \textit{JGraphT} suited best for our requirements of basic implementation purpose due to following reasons:

1. It is free
2. It has good documentation and full javadoc is available
3. It has implementation for the mathematical graph theory and the algorithms
4. It can be extended to have visualization with the help of another graph layout package called \textit{GraphViz} \footnote{http://www.graphviz.org/}, which is a package of open source tools for drawing graphs.

The fact that it was an open source was attractive. It provided the functions and services that were best suited for our implementation and testing purpose. It can also be powered with the visualization capabilities with the help of other library called JGraph.

6.1.4 Development environment

Once the programming language was decided to be \textit{Java}, the choice of the development platform was quite obvious. In our case it was Eclipse SDK platform\footnote{http://www.eclipse.org/}. Again, Eclipse is free, open source platform with great community support and plug-ins which makes Eclipse extremely attractive. Eclipse employs plug-ins in order to provide all of its functionality on top of (and including) the runtime system, in contrast to some other applications where functionality is typically hard coded. (33)

6.1.5 Version specifications

In this section I will give the information of the software that was used for implementation purpose.

Development platform:
Eclipse SDK: Version 3.4.2 (Eclipse Ganymede) as a development platform.

Implementation language:
Java: Version 1.6.0_11 from Sun Microsystems Inc.

Java library for Graph theory concepts:
\textit{JGraphT}: Version 0.8.1 for the Windows

6.1.6 Future implementation

For the implementation of a complete merging procedure from section 4.10, following requirements need to be implemented:

1. Detection of cycles in the graph
2. Creating a visualization (diagram) for the graph from its mathematical representation
3. Saving and restoring of the graph in a file
4. Finding a post dominator node set for each node
5. Finding predecessor and successor node set for each node
6. Accessing the repository (Vulnerability analysis database) and using the data accessed to create the VCGs
7. Finding the basic blocks the VCG
8. Maintaining logs for all the significant steps in the merging procedure. The conditions (ignoring edges, detection of the cycles and actions taken, etc.) that are encountered during the merging procedure should be logged and reported.
6.2 Using merging procedure in the detection of conjunction nodes

Sequences of causes in the VCGs represent conditions that are ordered by some form of causality. Conjunctions represent conditions that lack such a relationship, but jointly cause other conditions to be a concern. (13)

If a cycle is detected during the processing of edges/nodes during the merging process, we can safely deduce that there exists no specific form of causality among the causes and that they can be converted together into a conjunction node. We can infer from these cycles that these sequences lack natural order (e.g. cause-effect order). Thus, merging procedure can be helpful in detecting the conjunction nodes in the instance VCGs.

6.3 Using the merging procedure to find inconsistencies and ambiguities in the instance VCGs

A condition where a cycle is detected during the merging procedure implies that there is inconsistency and ambiguity in the construction of the VCG. Following argument can be presented:

Generally, causes at the top of the VCGs are harder to mitigate and represent inherent drawback in the way a particular software/system works. In the example shown in Fig. 6a, cause A is at the top of the VCG for CVE-2. The same cause appears as a direct cause in the VCG for CVE-1. Thus, it can be concluded that some of the causes have been missed in the construction of the VCG. It could also imply that the causes are not in an incorrect order.

The VCGs are the product of manual analysis by different people. Different people think in different ways. The way a person sees a particular vulnerability-cause or cause-cause relationship might differ from how another person sees it. This may result in ambiguities in the VCGs. A cycle involving more than two nodes in a merged VCG can be one of these ambiguities. Thus, merging of these VCGs can be a good way to find and address these issues.
6.4 Unresolved cases
I have found cases that cannot be resolved by the merging procedure. These cases require manual intervention to be solved. In this following example shown in Figure 6-1, a cycle can be detected during a merging procedure. However, the cycle cannot be resolved by any of the Basic block rules. The only way to resolve this ambiguity is by manual intervention. A solution for these kinds of cases can be devised as a part of future work.

![Figure 6-1: The example of the VCGs that needs manual intervention for merging](image)

The issue in the example is basic block that cannot be detected in the source VCG. Due to the presence of edge GF, the set of nodes A-E-B-F cannot be termed as a basic block. Thus, the cycle cannot be resolved by the basic block rules presented in the section 4.9. The only way to resolve the issue is by manual intervention.
7 Applications

7.1 The Software development process
The class VCGs can be used as a support for the decision making during the software development process. Especially, class VCG scan be useful during the design phase of the software.

Example 1:
Class VCs can be used during the design phase of the software.
For example, for a web based application, it is discovered from the class VCG that not parsing the input data is the main cause that leads to Cross Site Scripting attacks. Therefore, it can be decided during the design phase that a special module will be implemented that will do the checking of the input data and all the external data has to pass through this module.

Example 2:
A class VCG for cross site scripting reveals that not parsing input from the user is one of the important causes for the cross site scripting vulnerabilities. From security point of view, the team can make the decision that all developers should be given proper training and understanding about parsing of any input based on a security point of view.

7.2 Risk analysis and risk mitigation
Risk analysis in Software engineering is the process of identifying and analyzing the potential threat to a software development project.
Following concepts are involved in the risk analysis: (34)
1. Asset: Object of protection efforts.
2. Risk: The probability that the asset will suffer an event of negative impact.
3. Threat: A source of attack.
4. Vulnerability: Defect or weakness in the system.
5. Countermeasures: Controls prescribed for the protection of asset.
6. Impact: Impact can be tied to monetary losses or to a reputation.
7. Probability: A likelihood that a given event will trigger.

The risk analysis can be done with the help of class VCGs. We will now give a short introduction about how to use class VCGs in risk analysis.

Prioritizing the risks is important in risk analysis. It helps allocating the resources to the most critical risks. Class VCGs could be used in the prioritization of risks.

We can tag following data to the causes that appear in a class VCG
a. Impact
b. Frequency of occurrence
c. Threats

There could be more data tagged to the causes.
Therefore the risks can be prioritized on the basis of these metrics.
Class VCGs can help in one more aspect.

Consider the VCG in Fig. 7a:
Let, the cost of mitigating the cause C is high. At the same time, the cost of mitigating causes B and cause F below.
Therefore it could be decided that security activities that mitigate the causes $B$ and $F$ would be given priority. The security activities to mitigate cause $C$ are no longer needed to be implemented. Thus, class VCGs with different information can help prioritizing the risks and activities during risk analysis.

Consider the example of a class VCG for buffer overflow in Figure 5-4. It can be inferred that “range check problem” is one of the frequent causes to buffer overflow vulnerability. Thus, the security activities to mitigate these vulnerabilities can be given priority and implemented.

How can a metric be calculated?
The metric could be calculated by the analysis of different causes during the integration procedure. E.g. the metric of frequency of occurrence is easy to count. Every time a cause appears that is already present in the class VCG during the merging procedure, the count associated with the cause can be incremented.

Thus, knowledge about previous vulnerabilities can be used in calculating the various metrics for the causes in the class VCG.

7.3.2 Vulnerabilities and risks detection:
The personnel in charge of risk analysis can use class VCGs as a reference to unearth all vulnerabilities that can occur for a given application. As class VCGs give a complete description of which causes there are that lead to vulnerabilities, all these causes can form the basis for formulation of the risks to the application being developed. In this way, there will be no possibility of any risk to be missed.

7.3 Tools
In this section, we will present some of the ways in which class VCGs can be used by the software tools:

a. A software tool could be developed for risk analysis. The tool will generate its analysis based on the class VCGs it is supplied with. If we are to complete the software project successfully, it is important that the correct risks are detected and addressed. Class VCGs can help in detecting the risks.

For the VCG CVE-2002-0728 in section 3.2.5, one of the causes that appears is “Developer does not understand the implication of the condition detected” and for the VCG CVE-2002-1337 in section 3.2.6 one of the causes for the vulnerability is “Developer does not understand the documentation”.

If the occurrence of causes related to “developer knowledge and abilities” appears frequently, this can form the basis for the formulation of the risk for future software development.
Example: For the causes mentioned above, the risk could be formulated as “*Competent developers not available*” or “*developers not trained enough*”. The use of class VCGs can help enlisting most important risks (e.g. based on the basis of frequency of occurrence and its impact) for the project, based on the metrics associated with a class VCG.

b. A software tool can be developed for detecting any unsecure settings and connections in a system. For this purpose we can create and use the class VCG of “*configuration and setting problems*”. The tool can use class VCG as a basis for detecting any security vulnerabilities due to configuration and settings in a system.

I will describe the system configuration vulnerability in the mail notification program *Comsat*. This program waits for reports from incoming mail for any users and prints the first few lines of the message on the terminal. This terminal is described in the file `/etc/utmp`, which was configured to be world-writable. A malicious user could modify the terminal file with the `/etc/passwd`. The user then sends the mail to itself containing the line that starts with `root::0:0`. The second field that corresponds to the password field is empty. This implies that the superuser has no password. The *Comsat* program overwrites the password file with the message. The user can now login as *root* without providing a password (35).

Here, I will show the Vulnerability cause graph for this vulnerability based on the analysis presented above.

![Vulnerability cause graph](image)

**Figure 7-1: Instance VCG for "Comsat /etc/utmp vulnerability"

The exit node:
The vulnerability has no CVE assigned to it. Instead of having the CVE identifier as an exit node, we use the “*Comsat /etc/utmp vulnerability*” as an identifier.

**/etc/utmp writable:**
File `/etc/utmp` was world-writable. Anyone could influence the contents of the file `/etc/utmp`.

User terminal determined from `/etc/utmp`:
The user terminal is determined from the contents of the file which is world-writable.

External data accepted and written to terminal:
External data is accepted and written to the terminal. The terminal is determined from the contents of the `/etc/utmp` file.
If the above VCG was to be converted and generalized in a class VCG it would look as shown in Figure 7-2:

The software tool can use this class VCG to check if the condition arises in the current configuration and settings of the system that would result in the vulnerability for the system. If the tool finds that above causes are present, it can inform the user about the vulnerability in the settings of the system.

This is a very simple example of the configuration and setting problems. The class VCG can get more complex and comprehensive if more and more VCGs for instances of the vulnerability are merged.

The tool will require analyzing of the statements and determining what it actually means in the context of the system. E.g. the cause “writable ‘file’” implies that the tool should have rights to determine the access rights of the files in the systems and determine which of these files are writable.

This kind of tool can be particularly useful in the case of mobile phones. The mobile phones do not have enough memory to save entire instance VCGs. The class VCG could be easily stored and used for detecting any vulnerability in the configuration and settings in the mobile phones.

### 7.4 Training purpose

Most of the vulnerabilities that occur in software are due to unsafe coding and design practices of developers that are due to their lack of knowledge. Imparting right training to the developers will significantly help reducing the vulnerabilities in the software.

Class VCGs can be used for study purposes to students and employees. The main advantage of the class VCG is that it will not have too minute details for the particular class of vulnerability but still have enough information to give the learner a complete picture of the particular class of vulnerability.
Having seen the application of the class VCGs in chapter 7, it can be inferred that modeling a class of vulnerability is useful. The traditional way of modeling would be to start from scratch and try to model the class of vulnerability right away. However, this approach has many drawbacks:

We may miss out on not so significant and frequent causes in the modeling process. However, modeling only significant and frequent causes would still leave the software vulnerable. Also, people in charge of the modeling will need to have a deep knowledge about the complete class of vulnerability that is being modeled. This will make the modeling dependent on a particular person/group of persons. This method will also not be flexible enough to incorporate the newly found causes that may result into vulnerabilities.

The approach presented in this thesis of integrating instances of VCGs to create a model for a class of vulnerability will be able to overcome all these problems. The different instance of VCGs can be created by a large number of different analysts and integrated by our merging procedure. These analysts need not belong to same team and also need not communicate with each other. Thus, the dependence on the team or any person is completely removed. The uniqueness of this approach is that many people can contribute in the modeling of a class of vulnerabilities without any permanent dependence on each other. The ambiguities and inconsistencies are obvious with this approach. However, the issues can also be resolved during the merging procedure. This approach also makes sure that all the causes, either frequent or rare, will make their way into the merged class VCG. No causes would be left out of modeling. The approach we presented also offers the much-needed flexibility to incorporate new causes to the model when they are discovered. The only step needed will be to model the instance vulnerability and integrate it into the existing class VCG. If the instance VCGs has a new set of causes in its model, they will appear in the class VCG after applying the merging procedure.

The key contributions of this thesis are:

a) enumeration for the need of class VCG
b) procedure to create a class VCG along with solutions for various conditions that can arise during the procedure
c) the use of class VCGs.
d) the use of the merging procedure in detecting ambiguities, inconsistencies and conjunctions in the instance VCGs.

For the future, the implementation of the proposed procedure for merging of the VCGs can be done. Future work may also concern finding more cases and actions to be taken to resolve them during the merging procedure.

Analysis regarding selecting a set of security activities on the basis of class VCG that would mitigate the complete class of vulnerability, similar to work presented in the paper titled “A Cause-Based Approach to Preventing Software Vulnerabilities”(32) will be one of the important part of future work.


List of Figures

Figure 2-1: Visual representation of the nodes in VCG ................................................................. 6
Figure 2-2: VCG for CVE-2009-0490 ......................................................................................... 8
Figure 3-1: VCG for CVE-2009-0490 ......................................................................................... 14
Figure 3-2: VCG for CVE-2008-3252 ......................................................................................... 16
Figure 3-3: VCG for CVE-2005-2943 ......................................................................................... 18
Figure 3-4: VCG for CVE-2007-0063 ......................................................................................... 22
Figure 3-5: VCG for CVE-2002-0728 ......................................................................................... 23
Figure 3-6: VCG for CVE-2002-1337(13) .................................................................................. 24
Figure 3-7: VCG for CVE-2005-2558(15) .................................................................................. 25
Figure 3-8: VCG for CVE-2005-3192(31) .................................................................................. 25
Figure 3-9: Generalization to "Use of unsafe functions" ............................................................. 26
Figure 3-10: Generalization to "Range check problems" .............................................................. 26
Figure 3-11: Generalization for the "Safe functions used in incorrect way" ............................... 27
Figure 3-12: Flowchart for the Generalization procedure ........................................................... 28
Figure 4-1: Flowchart for the basic procedure of merging VCGs ................................................ 34
Figure 4-2: Cycle formation during merging ................................................................................ 44
Figure 4-3: Flowchart of procedure for merging of VCGs with the conditions ......................... 45
Figure 5-1: VCG after merging VCG for CVE-2005-3192 with VCG for CVE-2008-3252 ......... 68
Figure 5-2: Merged VCGs for CVE-2009-0490, CVE-2008-3252, CVE-2005-2943, CVE-2002-0728, CVE-2002-1337 and CVE-2005-3192 ................................................................. 68
Figure 5-3: VCG for CVE-2005-2558 ......................................................................................... 69
Figure 5-4: Class VCG for Buffer Overflow ................................................................................ 70
Figure 6-1: The example of the VCGs that needs manual intervention for merging .................. 74
Figure 7-1: Instance VCG for "Comsat /etc/utmp vulnerability" .................................................. 77
Figure 7-2: Class VCG for "configuration and setting problems" based on single instance VCG .... 78
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtidiga ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådant form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstympännärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Rahul Hiran

Linköping, October 21, 2009