

Modeling Software Vulnerabilities With Vulnerability Cause Graphs

David Byers Shanai Ardi Nahid Shahmehri Claudiu Duma

Department of computer and information science
Linköpings universitet, SE-58183 Linköping, Sweden
E-mail: {davby, shaar, nahsh, cladu}@ida.liu.se

Abstract

When vulnerabilities are discovered in software, which often happens after deployment, they must be addressed as part of ongoing software maintenance. A mature software development organization should analyze vulnerabilities in order to determine how they, and similar vulnerabilities, can be prevented in the future.

In this paper we present a structured method for analyzing and documenting the causes of software vulnerabilities. Applied during software maintenance, the method generates the information needed for improving the software development process, to prevent similar vulnerabilities in future releases.

Our approach is based on vulnerability cause graphs, a structured representation of causes of software vulnerabilities.

Keywords: software security, vulnerability modeling

1. Introduction

Vulnerabilities – security-related flaws – in software affect us almost daily, have forced us to change how we use computers and are at the center of some of the most spectacular and costly computer failures in recent years. For example, the total cost of the Code Red worm has been estimated at \$2.6 billion, and the Nachi worm affected operations at Air Canada and CSX railroad. Both exploited a class of vulnerabilities that has been known since at least 1988. Efforts are being made to reduce vulnerabilities in software, but the industry clearly has a long way to go.

In this paper we present a structured method for modeling software vulnerabilities, based on a formal graph representation called a vulnerability cause graph (VCG), introduced in our earlier work [1], and we demonstrate its application to a well-known vulnerability. A VCG organizes information about a vulnerability and its causes, and is designed to promote understanding of vulnerabilities and reuse of analysis results. Vulner-

ability modeling is performed throughout the software lifecycle, and is particularly important during maintenance, as vulnerabilities in deployed software are discovered.

Vulnerability modeling is a complement to efforts like the National Vulnerability Database [20] and the SecurityFocus vulnerability database [17]. They provide descriptions and catalogs of publicly available information about vulnerabilities, while vulnerability modeling provides in-depth analysis and understanding of vulnerabilities. This additional depth is important when determining how to prevent vulnerabilities. It is also a complement to efforts to analyze and classify vulnerabilities [2, 4, 8, 9, 18], and can implement parts of abstract process and lifecycle models for secure software [14, 15].

Our long-term goal is to develop methods and tools for systematically augmenting software development processes with activities that prevent vulnerabilities. We aim to be process agnostic: our results should be applicable to agile processes such as extreme programming [3] or feature-driven development [13] as well as in more conventional development processes [7, 10].

Vulnerability modeling with VCGs is the first stage of the process we are developing. Based on the vulnerability model, it is possible to determine which combinations of activities would prevent the vulnerability, allowing developers to select activities that fit their development process and software products, while also preventing vulnerabilities. Ultimately, we expect to be able to automatically optimize activity selection with respect to cost and effectiveness.

The major contributions of this paper are a complete methodology for vulnerability modeling that results in vulnerability cause graphs, as well as significant improvements to VCGs themselves.

2. Vulnerability cause graphs

Vulnerability cause graphs (VCGs) relate causes to vulnerabilities in the final software product, and are used as a starting point for improving the software develop-

ment process. Each vulnerability is modeled by a VCG, and based on the information present in the VCGs, we can derive combinations of software development activities that prevent the vulnerability [1]. This paper presents the second generation of our model.

VCGs must be well-defined so they can be used for automatic computation; the model and prevention semantics provide the requisite formalisms. VCGs must also be easy for humans to comprehend; the visual representation is designed with this in mind.

2.1. Model

A vulnerability cause graph is a directed acyclic graph in which all nodes but one represent causes and edges represent relationships between causes. There are four kinds of nodes in VCGs: simple nodes, compound nodes, conjunction nodes and exit nodes.

Every VCG has a designated *exit* node, which must be the only node in the graph without successors, and is the only node that does not represent a cause.

All other nodes are required to have at least one successor. Nodes can be *simple*, *compound* or *conjunctions*. Simple nodes represent conditions (causes) that may lead to vulnerabilities in the software being developed; they are the atoms of VCGs. Examples include “use of unsafe API” or “data file can contain executable code”. Compound nodes facilitate analysis reuse, maintenance of models and improve readability. They serve a similar purpose as procedures and functions in a programming language and represent entire VCGs that model reusable or complex analysis elements. Conjunctions represent the conjunction of two or more other nodes. A simplified UML model of the VCG is shown in figure 1.

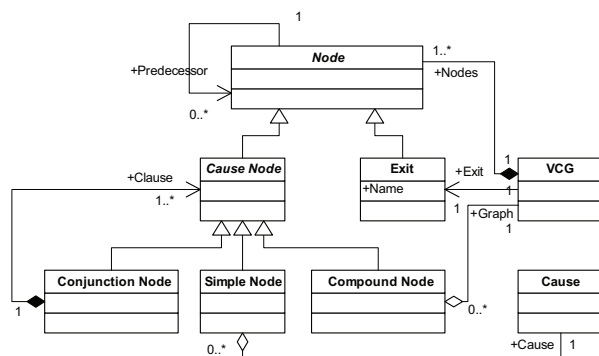


Figure 1. Simplified UML model of VCG

The predecessor-successor relationship between nodes in the VCG models how certain conditions cause other conditions to be a concern, with respect to preventing the vulnerability the VCG represents. If node *A* is a

predecessor of node *B*, then if *A* holds (i.e. is not mitigated during development), then *B* is a concern. This implies the following:

- If *A* and *B* are predecessors of *C*, then *C* is a concern if *A or B* hold (expresses “or” in the model).
- If *N* is a predecessor of *C*, and *N* is a conjunction consisting of $A_1 \dots A_n$, then *C* is a concern only if all of $A_1 \dots A_n$ hold (expresses “and” in the model).

Sequences in the graph represent conditions that are ordered by some form of causality (i.e. condition *A* causes *B* to be a concern). Conjunctions represent conditions that lack such a relationship, but jointly cause some other condition to be a concern.

Figure 2 shows an example of a small VCG. It shows, among other things, that if *Cause E* holds, then *Cause C* is a concern, as is the conjunction of *Cause A* and *Cause B*. It also shows that as long as *Cause C* holds (and is a concern), there is a risk of *Vulnerability V*.

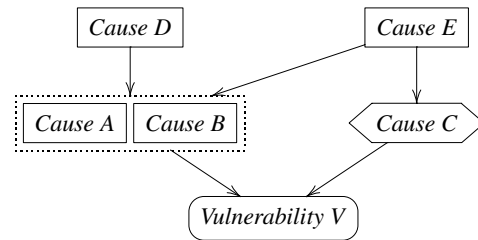


Figure 2. Second-generation vulnerability cause graph

2.2. Visual representation


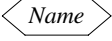


The preferred representation, which maps directly onto the VCGs model, is shown in figure 3.

Figure 2 contains all the visual elements. *Cause D* and *Cause E* are simple nodes; *Cause A* and *Cause B* are the clauses of a conjunction; *Cause C* is a compound node; and *Vulnerability V* is the exit node.

2.3. Prevention semantics

Since the ultimate goal of vulnerability modeling is to determine how to prevent vulnerabilities, the semantics of vulnerability cause graphs are expressed in such terms. The semantics of VCGs are derived from mitigation of causes.

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.

Visual representation	Model element
	Simple node
	Compound node
	Conjunction node ^a
	Exit node

^aA conjunction node must contain two or more simple and/or compound nodes; arbitrary combinations are permitted.

Figure 3. Visual representation of VCGs

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*. In the definitions below, N is a node; $C(N)$ is the set of clauses of a conjunction N ; if $M(N)$ is true, then N is mitigated; and if $B(N)$ is true, then N is blocked.

Since each of the predecessors of a node N in the VCG independently causes N to be a concern, they must all be blocked, or N itself mitigated, for N to be blocked. Hence,

A node N in a VCG is said to be *blocked* if it is mitigated, or all its immediate predecessors in the VCG are blocked ($B(N) = M(N) \vee (\bigwedge_{p \in P_N} B(p))$).

Since all the clauses of a conjunction must hold for the successor of a conjunction to hold, it is sufficient to block any one of them to mitigate the entire conjunction. Hence,

A conjunction is mitigated if any of its clauses are blocked ($M(N) = \bigvee_{c \in C(N)} B(c)$).

A compound node represents an entire graph, so its mitigation depends on the state of the exit node in that graph. Hence,

A compound node is mitigated if the exit node of the VCG it is associated with is blocked ($M(N) = B(E)$, where E is the exit node of the VCG represented by N).

By definition, the exit node of a VCG can never be mitigated as it always represents the consequences of other conditions. It can, however, be blocked. If the exit node of the VCG is blocked, then the vulnerability the VCG

models will be prevented.

From the point of view of preventing vulnerabilities, the semantics of a VCG is the function $B(E)$, where E is the exit node. Two VCGs V_1 (with exit node E_1) and V_2 (with exit node E_2) are equivalent if all sets of $M(C_i)$, where C_i are causes, that satisfy $B(E_1)$ also satisfy $B(E_2)$, and all sets of $M(C_i)$, that satisfy $B(E_2)$ also satisfy $B(E_1)$.

For the VCG in figure 2, with V denoting *Vulnerability* V , A denoting *Cause* A and so on: $B(V) = (M(C) \vee M(E)) \wedge ((M(A) \vee M(B)) \vee (M(D) \wedge M(E)))$.

3. Vulnerability cause analysis

The goal of vulnerability cause analysis is to produce a vulnerability cause graph for a specific vulnerability. The VCG will then be used to improve software development practices to ensure that similar vulnerabilities are prevented in the future. Classes of vulnerabilities can be modeled as well, simply by combining the models of several specific vulnerabilities. E.g., if V_1 , V_2 and V_3 are three vulnerabilities of the same class V_C , then a VCG for V_C can be constructed by creating compound nodes representing each vulnerability, and making them predecessors of a single exit node (see figure 4).

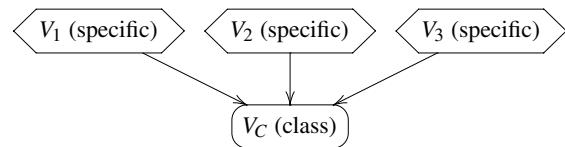


Figure 4. Vulnerability cause graph for a class of vulnerabilities

Vulnerability cause analysis is similar to root cause analysis (RCA), and could even be seen as an RCA method for security-related software failures. There are, however, some differences between our method and most RCA methods, discussed in section 6.

Vulnerability cause analysis is a creative process supported by a systematic approach. Although it is impossible to entirely remove the creative element, the systematic approach is designed to maximize the probability of a successful outcome. The process described below is based on our experience from analyzing a number of known software vulnerabilities.

3.1. The vulnerability analysis database

The vulnerability analysis database (VAD) is a database containing knowledge about vulnerabilities, causes, security activities, VCGs and related information. All information generated during analysis and

modeling is entered into the database, successively improving the quality and quantity of information. The VAD ensures that all uses of the same cause or VCG are linked; provides search mechanisms essential for analysis reuse; documentation needed for using the models; and so forth. It is an essential tool for effective practical application of the vulnerability modeling methodology we present. In particular, analysis reuse is very difficult in practice without the VAD. We have implemented a rudimentary version of the VAD, sufficient to support our research.

3.2. Initial analysis

The first step of vulnerability cause analysis is developing a thorough understanding of the vulnerability in question. We typically perform this step using code review, static analysis tools, visualization tools, execution traces, and live debugging. Initial analysis is considered complete when we know what types of conditions and/or input would expose a vulnerability. Developing a working exploit is sometimes also helpful.

The results of the analysis is entered into the VAD.

3.3. Vulnerability cause graph construction

After completing the initial analysis, a base VCG is created consisting of an exit node only. The VCG is entered into the VAD and associated with the vulnerability. The complete VCG is constructed through a process of iterative refinement. Any node in the VCG that has not been completely analyzed is picked for further analysis.

Analysis of a node consists of the following, which may be performed repeatedly:

- Determine the validity of the node
- Determine if the node needs to be split
- Determine if the node needs to be converted to a compound node
- Find candidates for predecessors in the VCG
- Organize predecessor candidates in the VCG

These steps are iterated until no more changes or additions to the VCG can be found (this exit criterion is similar to that in many root cause analysis methods).

3.3.1. Determine node validity, splitting and conversion Simple nodes entered into the VCG should always represent simple conditions, not combinations or sequences of conditions, and the conditions represented by different nodes in the graph should not overlap.

When a complex condition is identified, it should be separated into several simpler nodes and possibly turned

into a compound node or conjunction (sections 4.1 and 4.4). For example, a node representing copying data to a buffer without appropriate bounds checks could be split into two nodes (copy to buffer and missing bounds checks), and converted to a compound node.

If part of the condition represented by one node is the same as part of the condition represented by another node, this indicates that graph transformations should be applied to convert conjunctions and combine identical nodes (sections 4.1 and 4.3).

These guidelines tend to improve the quality of analysis since it increases the depth of analysis, and tends to promote reusability of nodes, since simple conditions tend to repeat more often than complex ones. Furthermore, analysis of mitigation techniques is easier for simple conditions than for complex ones, an important consideration in the application of VCGs.

3.3.2. Find and organize predecessor candidates

The VCG is extended by finding new nodes to place into the VCG, based on an existing node. The predecessors of a node all represent conditions that, independently of any other conditions, might cause the condition the node represents to be a concern. For example, if a node represents the use of error values in the same range as data values, then a predecessor might represent the use of a single variable for both data and error values.

Finding the predecessors of a node starts with answering the question “under what circumstances is this cause a concern?”. This produces a combination of conditions that can be expressed as a propositional logic formula. For example, assume we determine that for node N to be a concern, condition A must hold true and at least one of conditions B , C and D must hold true, which can be expressed as $A \wedge (B \vee C \vee D)$. Note that if the node being analyzed is already present in some other VCG, then its predecessors in that VCG may be suitable as predecessors in the current VCG.

Conjunctions tend to obscure the structure of the model. By preferring disjunctions over conjunctions, the graph becomes fairly flat, and it becomes easy to trace the causes of vulnerabilities through paths in the graph. Therefore, in order to improve readability, expressions should be converted to disjunctive normal form. For example, $A \wedge (B \vee C \vee D)$ should be converted to $(A \wedge B) \vee (A \wedge C) \vee (A \wedge D)$. After converting to disjunctive normal form, each term of the expression can be used as a predecessor of the node being analyzed.

Our experience shows that disjunctions are naturally more common than conjunctions, and that the expressions arrived at in this step are typically even simpler than the examples shown here. Complex expressions are an indication that modeling is progressing in

too large steps. Nevertheless, it is possible to convert an arbitrarily complex expression to a VCG by observing that any parenthesized expression can be expressed as a compound node; a conjunction can be expressed as a conjunction node or sequence of nodes; and a disjunction is merely a set of predecessors of another node.

At any time during VCG construction, particularly at this stage, graph transformations (see section 4) may be applied to simplify the graph. Common transformations at this stage will include converting nodes to compound nodes, converting conjunctions to sequences and combining nodes representing the same cause.

For example, $(A \wedge B) \vee (A \wedge C) \vee (A \wedge D)$ could be transformed by converting all conjunctions to sequences, then combining the three resulting nodes that represent A , giving a final total of four simple nodes.

3.4. Graph validation and optimization

When the VCG is complete, it should be optimized for reuse and clarity. Following optimization, it is validated by a second analyst or team of analysts.

Optimization consists of applying graph transformations (see section 4) to the VCG until the desired end result is achieved. The order of every sequence in the graph should be verified to ensure that it is a natural order (e.g. cause-effect or temporal order). Sequences that lack natural order should be considered for conversion to conjunction nodes. Any compound nodes that have been introduced at any stage in the process need to be analyzed completely (if this has not already been done). The process for doing so is identical to the process of analyzing vulnerabilities. Common subgraphs and single nodes should be eliminated where possible.

4. Graph transformations

Graph transformations can be applied at any time during the analysis to make the graph easier to read, more concise or more intuitive. All transformations must preserve the semantics of the VCG. For node N_i , if $B(N_i)$ is true, then N_i is blocked and if $M(N_i)$ is true, then N_i is mitigated.

Equivalence of graphs Two VCGs V_1 (with exit node E_1) and V_2 (with exit node E_2) are equivalent if all sets of $M(C_i)$, where C_i are causes, that satisfy $B(E_1)$ also satisfy $B(E_2)$, and all sets of $M(C_i)$, that satisfy $B(E_2)$ also satisfy $B(E_1)$.

Due the limited space available, we have only included sketches of the proofs of equivalence for each transformation.

To show equivalence of a transformation involving a set of nodes N , it is sufficient to show that there exists a

set postdominator S of N , such that for every node $S_i \in S$, $B(S_i)$ is unaffected by the transformation. A node S_i postdominates a node N_i if S_i is on every path from N_i to the exit node. S is a set postdominator of N if for every $N_i \in N$ there is a $S_i \in S$ that postdominates N_i .

If N is the set of nodes involved in a transformation and S is a set postdominator of N , then when computing $B(E)$ for an exit node E , then neither $B(N_i)$, nor $M(N_i)$ will appear in the function $B(E)$ other than as the result of expanding some $B(S_i)$. Hence, if $B(S_i)$ for all $S_i \in S$ are unaffected by the transformation, then $B(E)$ will also be unaffected by the transformation. This observation is the basis for the proof sketches given below.

4.1. Conversion of conjunctions

If C is a conjunction of $N_1 \dots N_n$ (the clauses in arbitrary order), P is the set of predecessors of C and S is the set of successors of C , then C can be replaced with a sequence of $N_1 \dots N_n$. Specifically, C is removed from the graph; N_i is made the sole predecessor of N_{i+1} for $i < n$; P is made the set of predecessors of N_1 ; and S is made the set of successors of N_n . This transformation can be reversed. Figure 5 shows an example of this transformation.

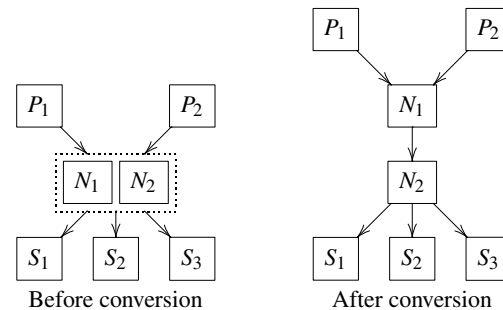


Figure 5. Conversion of conjunctions

Equivalence In the original graph, for $s \in S, B(s) = M(s) \vee M(C) \vee (\bigwedge_{p \in P} B(p)); M(C) = \bigvee_{i=1 \dots n} M(N_i)$. In the new graph, for $s \in S, B(s) = M(s) \vee \bigvee_{i=1 \dots n} M(N_i) \vee (\bigwedge_{p \in P} B(p))$. The graphs are equivalent as the two expressions are equal.

4.2. Reordering of sequences

If N_1 is the sole predecessor of N_2 , then N_1 and N_2 can be reordered so that the direct predecessors P of N_1 become the direct predecessors of N_2 , the direct successors S of N_2 become the direct successors of N_1 , and N_2 becomes the sole predecessor of N_1 . This transformation can be trivially reversed. Figure 6 shows an example of this transformation.

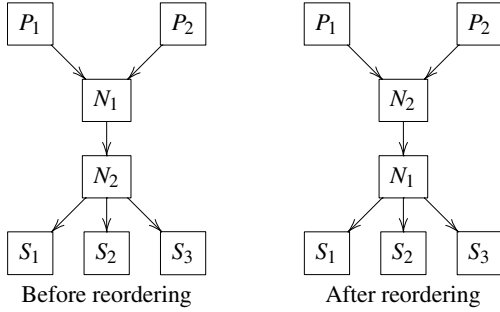


Figure 6. Reordering of sequences

Equivalence In the original graph, for every $s \in S$, $B(s) = M(s) \vee M(N_2) \vee M(N_1) \vee (\bigwedge_{p \in P} M(p))$. In the new graph, for every $s \in S$, $B(s) = M(s) \vee M(N_1) \vee M(N_2) \vee (\bigwedge_{p \in P} M(p))$. The graphs are equivalent as the two expressions are equal.

4.3. Combination of nodes

If N_1 and N_2 denote the same cause C , and from every predecessor of N_1 (denoted P^1) or N_2 (denoted P^2), it is possible to reach every successor of N_1 (denoted S^1) or N_2 (denoted S^2), then N_1 and N_2 can be combined to a single node X with predecessors set to $P^1 \cup P^2$ and successors set to $S^1 \cup S^2$. Figure 7 shows an example of this transformation.

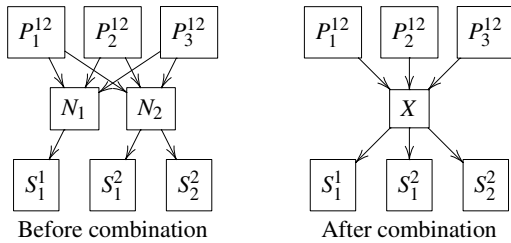


Figure 7. Combination of nodes

Equivalence In the original graph, for each $s \in S_i$, $B(s) = M(s) \vee M(N_i) \vee (\bigwedge_{p \in P^1} B(p) \wedge \bigwedge_{p \in P^2} B(p))$. In the new graph, $s \in S^1 \cup S^2$, $B(s) = M(s) \vee M(X) \vee (\bigwedge_{p \in P^1 \cup P^2} B(p))$, which is equivalent, showing that the graphs are equivalent.

4.4. Conversion to compound nodes

Let N be a set of nodes $N_1 \dots N_n$. Let P^i be the set of predecessors of N_i and S^i the set of successors of N_i . Let N_{\top} be the set of nodes in N with predecessors outside N . Let N_{\perp} be the set of nodes in N with successors outside N . N can be converted to a compound node if the following holds:

1. Every node in N must be reachable from some node in N_{\top} .
2. No node in N_{\top} has a predecessor in N , and all nodes in N_{\top} have the same set of predecessors (denoted P^{\top}).
3. No node in N_{\perp} has a successor in N , and all nodes in N_{\perp} have the same set of successors (denoted S^{\perp}).

This implies that nodes not in N_{\top} only have predecessors in N and nodes not in N_{\perp} only have successors in N .

To convert N to a compound node, N is removed from the VCG. A new compound node X is introduced, with predecessors set to P^{\top} and successors set to S^{\perp} . The VCG for X is created by making a new exit node the sole successor of all nodes in N_{\perp} and removing all predecessors from all nodes in N_{\top} . This transformation can be reversed. Figure 8 shows an example of this transformation.

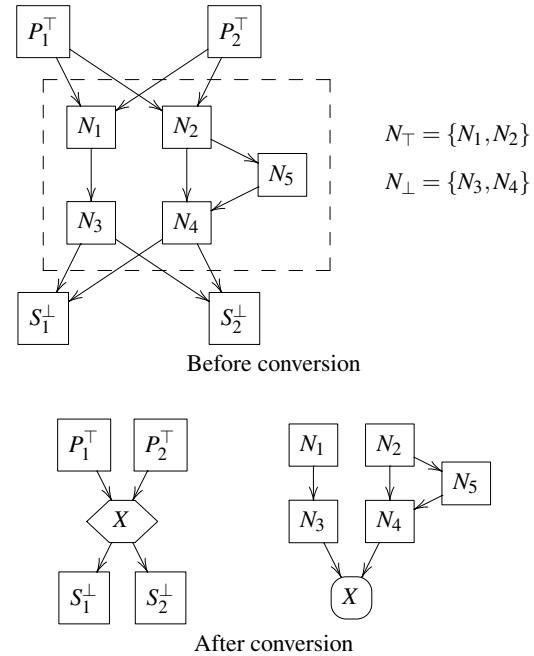


Figure 8. Conversion to compound nodes

Equivalence In the original graph, if $B(n)$, for any $n \in N$ is computed recursively without “expanding” $B(p)$ for any $p \notin N$, then the resulting expression is $R(n) \vee (\bigwedge_{p \in P^{\top}} B(p))$ ($R(n)$ can be thought of as a term representing the contribution of the nodes in N to $B(n)$, the semantic function of n), since every node in N can be reached from N_{\top} , and every node in N_{\top} has the same set of predecessors, P^{\top} . This further implies that for every $s \in S^{\perp}$, $B(s) = (\bigwedge_{n \in N_{\perp}} R(n)) \vee (\bigwedge_{p \in P^{\top}} B(p))$

In the new graph, N is replaced by X , with predecessors P^\top and successors S^\perp . Hence, for every $s \in S$, $B(s) = M(X) \vee (\bigwedge_{p \in P^\top} B(p))$. Furthermore, $M(X) = B(E)$, where E is the exit node of the new VCG. $B(E) = \bigwedge_{n \in N_\perp} R(n)$, since the new VCG consists only of nodes in N with the same structure as in the original graph. Hence, in the new graph, for every $s \in S^\perp$, $B(s) = (\bigwedge_{n \in N_\perp} R(n)) \vee (\bigwedge_{p \in P^\top} B(p))$, showing that the transformation preserves the semantics of the graph.

4.5. Derived transformations

It is possible to derive a number of transformations from the ones given above. For example, by successively rearranging pairs of nodes, the order of a sequence can be rearrange arbitrarily. It is also possible to eliminate common subgraphs from the VCG either by applying conversion to compound nodes, resulting in two identical VCGs, which can trivially be combined as one, then combining the compound nodes, and finally reversing the transformation to compound node; or by iteratively applying the “combination of nodes” transformations. Common terms can be factored out from conjunctions and made predecessors or successors of the conjunction by converting the conjunctions, then combining nodes (see figure 9).

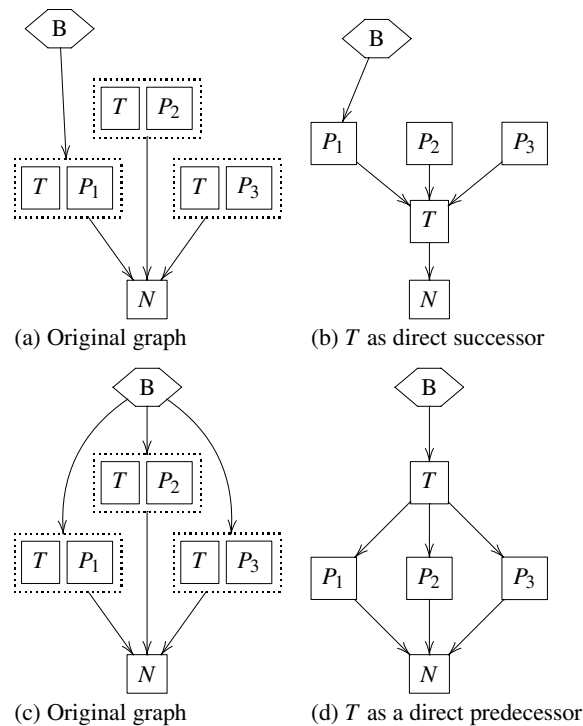


Figure 9. Factoring out common conjunction terms

5. Case study: CVE-2003-0161

We have applied vulnerability modeling to a number of well-known vulnerabilities. In general, this has resulted in a comprehensive understanding of the vulnerabilities and the measures required to prevent them. Here, we show the majority of the analysis of CVE-2003-0161 [21] (as designated in the Common Vulnerabilities and Exposures list [19]). This is a well-known severe vulnerability in versions before 8.12.9 of the `sendmail` mail server. This vulnerability is described as follows:

The `prescan()` function in the address parser (`parseaddr.c`) in `Sendmail` before 8.12.9 does not properly handle certain conversions from `char` and `int` types, which can cause a length check to be disabled when `Sendmail` misinterprets an input value as a special "NOCHAR" control value, allowing attackers to cause a denial of service and possibly execute arbitrary code via a buffer overflow attack using messages, a different vulnerability than CVE-2002-1337.

5.1. Initial analysis

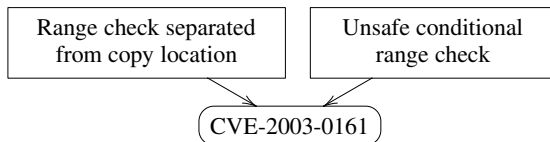
We performed a detailed analysis of the vulnerability by code inspection and analysis of known exploits in a debugger. This provided us with additional details:

- The `prescan` function tokenizes e-mail addresses by copying the modified input to an output buffer.
- The `int` variable `c` holds the last character read.
- When copying, `-1` is used as a sentinel value to indicate that no character is to be copied into the target buffer. This is stored in `c` as well.
- The assignment to `c` is from a signed character, so sign extension occurs. Thus, the input character `0xff` will be interpreted as “don’t copy”.
- Copying occurs in two places: one to copy characters and one to insert backslashes. The latter is not protected by a range check, and the range check of the former is not performed when `c` is `-1`.
- The lack of a range check when copying a backslash seems to not be a problem since a range check will be triggered when copying the following character, but that range check will be skipped (and no character copied) if `c` is `-1`.

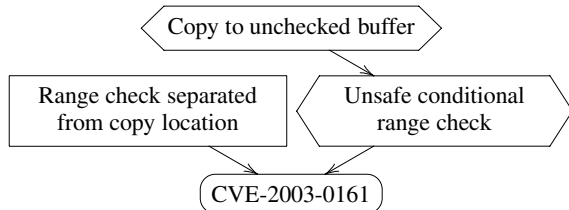
5.2. Vulnerability cause graph construction

VCG construction starts with a single exit node labeled “CVE-2003-0161”, representing the vulnerability being modeled.

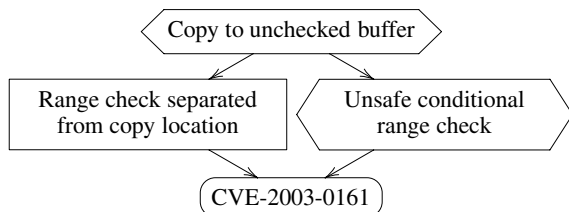
Iteration 1 The exit node is picked for further analysis. It is valid, cannot be split or converted to a compound node. The predecessor candidates of the node are its immediate causes. In this case we have determined that a buffer overflow can occur because a conditional range check can be bypassed and because the range check was not performed at the copy location. These two causes are entered as predecessors of the exit node. In our estimation either could have caused the vulnerability independently of the other.



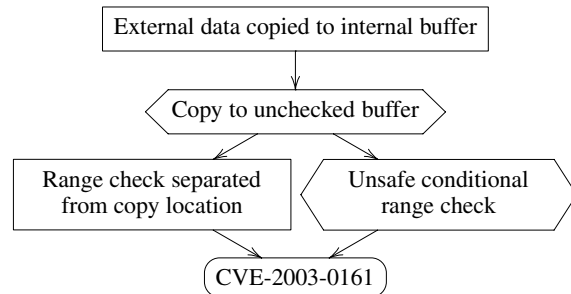
Iteration 2 The “unsafe conditional range check” is picked for further analysis. This does not represent a simple cause, but a concept, so it is converted to a compound node, the internals of which will be analyzed later. We next ask the question “why is this a cause for concern”. The answer is “because we are copying data into an unchecked buffer of indeterminate size”. This is a quite complicated cause, so it is entered into the graph as a compound cause (if entered as a simple cause, it would have been converted later).



Iteration 3 In the third iteration “range check separated [...]” is picked for further analysis. This node does not need to be split or converted. The answer to the question “why is this a cause for concern” is again that we are copying data into an unchecked buffer, so “copy to unchecked buffer” is made a predecessor of “range check separated [...]”.



Iteration 4 In the fourth iteration, we pick “copy to unchecked buffer”, which represents data copied to a buffer with certain dangerous properties, for further analysis. The answer to the question “why is this a cause for concern” is that data is being copied from an external source to an internal buffer, which is a simple cause.



Further iterations Further iterations are not included in this example; they identify product-specific causes of this specific vulnerability related to the design of sendmail.

5.3. Graph validation and optimization

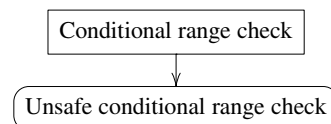
Order of sequences The graph contains one sequence that could be reordered (“external data copied [...]” to “copy to unchecked buffer”), but it is already ordered correctly: the decision to use an unchecked buffer is made after the decision to copy data.

Other graph transformations Two nodes were converted to compound nodes during the analysis. The two remaining simple nodes cannot reasonably be split into multiple causes at this time.

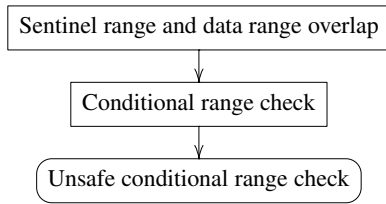
5.4. Analysis of compound nodes

Both compound nodes must be further analyzed. In this example we show only the analysis of “unsafe conditional range check”.

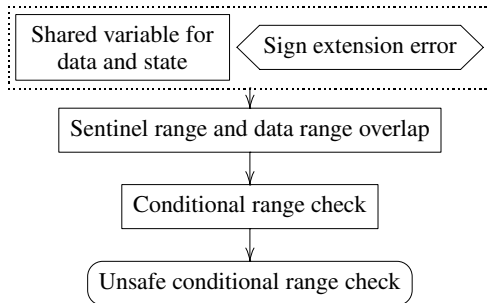
Iteration 1 The direct cause of an unsafe conditional range check is that there is a conditional range check (a range check that is only performed if certain conditions are met).



Iteration 2 Next, “conditional range check node” is picked for further analysis. It is a concern because the range of sentinel values and range of data values overlap.



Iteration 3 Next, “sentinel range and [...]” is picked for further analysis. It is a concern because a single variable is used for sentinel and data values, and there is a sign extension error in the code. This is expressed using a conjunction. Sign extension error is a complex cause likely to be present in the VAD, and certainly reusable in other analyses.



At this point no further nodes are added to the graph. As with any other VCG, this graph is subject to validation and optimization, but this will not result in any changes to the graph.

5.5. Discussion

When we originally applied our modeling method to this and several other known vulnerabilities, we found that the results gave a much more detailed understanding of the vulnerabilities than was available in published sources. For example, for CVE-2003-0161 we discovered that the problem was not merely caused by having the character `0xff` in the input, but rather by having a backslash followed by `0xff`. That understanding is vital when attempting to prevent similar vulnerabilities in the future.

We also found striking similarities between vulnerabilities found at different times in the same software (e.g. CVE-2003-0694 and CVE-2003-0161 in sendmail), clearly demonstrating that mitigation of the original vulnerability was not performed with preventing other, similar, vulnerabilities firmly in mind. Had a deeper analysis been performed, and that used to guide maintenance activities, several vulnerabilities could have been eliminated at a much earlier stage than was the case.

6. Related work

6.1. Root cause analysis

The vulnerability modeling method presented here can be seen as a method for root cause analysis of security-related software failures. Indeed, our method meets many of the requirements of a root cause analysis method. There are, however, some points which should be highlighted.

We are concerned not only with what *did* cause the vulnerability, but with what *might have* caused vulnerability. One of the reasons for this is that in many situations, there will not be sufficient evidence available to determine the actual causes. Furthermore, we are not strongly focused on the root causes, since preventing certain vulnerabilities (or even classes of vulnerabilities) is sometimes more easily done by addressing contributing causes (e.g. through implementation-related, rather than design or requirements-related, activities).

We require a high degree of formalism in the representation of the analysis, as it will be used to automatically generate a framework for process improvement. Although some RCA methods use a formal representation, it is not a general requirement.

6.2. Analysis of vulnerabilities

Ours is not the only work aimed at analyzing and preventing vulnerabilities. Schumacher outlines a general lifecycle model (Software Improvement Feedback Loop, SIFR) for software security [15], but lacks sufficient detail to be applicable in practice. Vulnerability modeling can fill some of the gaps in SIFR and similar high-level models.

Many researchers have attempted to classify software vulnerabilities [2, 4, 8]. Vulnerability modeling could benefit from classification efforts, as vulnerabilities in the same class are likely to have similar causes. Classification efforts might benefit from vulnerability analysis, as vulnerabilities with similar causes are likely to be related in some way.

Others have published analysis of specific vulnerabilities [9, 18], which is the kind of work that can provide the information required at the initial step of vulnerability modeling, as it provides a thorough understanding of the vulnerability in question.

6.3. Experience-based prevention approaches

Currently, most approaches to software security are based on experience and the application of best practices [5, 11, 12, 14, 16]. There is no doubt that these meth-

ods are valuable, as evidence shows that they do prevent vulnerabilities. Nevertheless, these approaches usually have several drawbacks. The most important drawbacks, in our estimation, are lack of flexibility and evolution.

Best practice approaches tend to be quite difficult to adapt to other situations (e.g. software development processes, organizations, product types) than those for which they were conceived. One of the primary goals of our work is to overcome this limitation, and the applications of VCGs we have proposed [1] can be adapted to a wide range of situations.

Best practice approaches also tend to be static. Although they can be evolved to meet new challenges, they rarely include mechanisms to do so. As a result, these approaches are not very effective in meeting new threats.

Our approach is motivated by the need for continuous improvement. It is the reason we have designed a method for vulnerability in the first place, and the reason for why we address analysis reuse.

7. Conclusions

Comprehensive analysis of vulnerabilities should be an integral part of software maintenance. Effective response to, and long-term prevention of, vulnerabilities, depend on the ability to understand the causes of vulnerabilities, and the ability to address those causes.

In this paper we have presented a systematic method, suitable for use in the analysis phase [6] (or equivalent) of maintenance, for analysis of software vulnerabilities that results in a formal model of the vulnerability, expressed as a vulnerability cause graph. In addition to the method, we have extended our earlier work with significant improvements to VCGs (compound and conjunction nodes, and a formal model and semantics). Our method promotes reuse of analysis and over time produces a body of knowledge of vulnerabilities and their causes. Finally, we have applied the method to real vulnerabilities with good results.

The next step of our work will be to apply vulnerability modeling on software with high security requirements, developed by our industrial partners. To this end we will implement the supporting tools required for efficient application of vulnerability modeling.

Vulnerability modeling is one part of a more comprehensive method for preventing vulnerabilities. Based on the vulnerability model, it is possible to systematically (and to a large extent automatically) determine what actions must be taken in software development and maintenance to prevent vulnerabilities. The overall process has been outlined in previous work [1] and will be the subject of future articles.

References

- [1] S. Ardi, D. Byers, and N. Shahmehri. Towards a structured unified process for software security. In *Proceedings of the ICSE 2006 Workshop on Software Engineering for Secure Systems (SESS06)*, 2006.
- [2] T. Aslam, K. Ivan, and E. Spafford. Use of a taxonomy of security faults. In *Proceedings of the 19th National Computer Security Conference*, 1996.
- [3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
- [4] M. Bishop. Vulnerabilities analysis. In *Web Proceedings of the 2nd International Workshop on Recent Advances in Intrusion Detection*, 1999.
- [5] M. Howard. Building more secure software with improved development processes. *Security & Privacy Magazine*, 2(6):63–65, Nov-Dec 2004.
- [6] IEEE Std. 1219-1998. *Standard for Software Maintenance*. IEEE Computer Society Press, 1998.
- [7] I. Jacobson, G. Booch, and J. Rumbaugh. *Unified Software Development Process*. Addison-Wesley, 1999.
- [8] I. Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, 1998.
- [9] I. Krsul, E. Spafford, and M. Tripunitra. An analysis of some software vulnerabilities. In *Proceedings of the 21st NIST-NCSC National Information Systems Symposium*, pages 111–125, 1998.
- [10] R. C. Linger. Cleanroom process model. *IEEE Software*, 11(2):50–56, March 1994.
- [11] S. B. Lipner. The trustworthy computing security development lifecycle. In *Proceedings of the 20th Annual Computer Security Applications Conference*, pages 2–13. IEEE Computer Society, December 2004.
- [12] G. McGraw. Software security. *Security & Privacy Magazine*, 2(2):80–83, Mar-Apr 2004.
- [13] S. R. Palmer and J. M. Felsing. *A Practical Guide to Feature-Driven Development*. Prentice-Hall, 2002.
- [14] S. T. Redwine and N. Davis. *Processes to Produce Secure Software*, appendix B. Task Force on Security Across the Software Development Lifecycle, 2004.
- [15] M. Schumacher, R. Ackermann, and R. Steinmetz. Towards security at all stages of a system's life cycle. In *Proceedings of the International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2000.
- [16] Secure Software, Inc. The CLASP application security process. <http://www.securesoftware.com/> (accessed April 2006).
- [17] SecurityFocus. SecurityFocus vulnerability database. <http://www.securityfocus.com/vulnerabilities>.
- [18] E. Spafford. The internet worm program: An analysis. *Computer Communication Review*, 19(1), 1989.
- [19] The common vulnerabilities and exposures list. <http://cve.mitre.org/>.
- [20] US-CERT/NIST. National vulnerability database. <http://nvd.nist.gov/>.
- [21] US-CERT/NIST. Vulnerability summary CVE-2003-0161. <http://nvd.nist.gov/nvd.cfm?cvename=CVE-2003-0161>.