Structuring a Vulnerability Description for Comprehensive Single System Security Analysis

Malgorzata Urbanska, Indrajit Ray, Adele E. Howe, Mark Roberts
Computer Science Department
Colorado State University
Fort Collins, CO 80523 U.S.A.
Email: {urbanska, indrajit, howe, mroberts}@CS.ColoState.EDU

***Abstract*—The National Vulnerability Database (NVD) provides unstructured descriptions of computer security vulnerabilities. These descriptions do not directly provide the information necessary to formally analyze how the user's and the attacker's actions lead to the exploit. Moreover, the descriptions vary in how they describe the vulnerabilities. In this paper, we describe a system for automatically extracting cause and effect information from a set of vulnerabilities. The result is a structured data set of vulnerability descriptions with pre- and post-condition relationships. We evaluate the system by comparing the output with a manually constructed representation for security analysis called the Personalized Attack Graph (PAG).**

***Index Terms*—security risk modeling, attack graphs, system security, attacks and defenses**

I. Security analysis for a single computer system

To secure a computer system, we have to know which vulnerabilities exist on the system and how they can be exploited. Vulnerability Scanners (VS) perform a security audit of a network/system to identify their weak spots. However, VSs simply scan the network/system, identify the weaknesses and match them against known vulnerabilities. They create a report that groups identified vulnerabilities into sets and gives some advice about how to fix these vulnerabilities. This information is not enough because the existence of a vulnerability does not mean that it can be exploited. Hence, the VS report can be challenging to apply, even for experienced security managers.

Vulnerabilities are exploited via sequences of events (actions on the part of users and attackers); so to understand whether a vulnerability can be exploited for a given system, one needs to know the sequence. However, the standard resource for capturing vulnerability information, Vulnerability Databases , contain textual descriptions, which do not necessarily describe the chain of events that lead to an exploit. In this work, we show how to transform such a textual vulnerability description into a more structured one that clearly identifies the pre- and post-conditions of the vulnerability that are needed for an security analysis of a single host.

One of the common models used to reason about the security risk in a network/system is attack graphs (AGs) [1]–[6]. AGs graphically capture all the possible ways in which a network/system can be attacked. They help to analyze the possible attack scenarios by showing a relationship between the vulnerabilities and the network/system configuration. Most AGs are based on the assumption of connectivity between hosts in the network. Assuming a host A with vulnerability V_A is connected to host B with vulnerability V_B, then if host A is compromised it implies that V_B is exploitable and hence host B can be compromised. For constructing AGs, these connectivities are necessary conditions, but they are not sufficient. This means that AGs cannot be really employed for a single host security analysis. We have extended AGs to focus on a single system. Our Personalized Attack Graph (PAG) [7] model incorporates information about a single computer system that facilities analysis of the necessary and sufficient conditions for a vulnerability to be exploited on a single host: the software installed on a system and the cause and consequence of actions of both the user and the attacker.

Building PAGs manually can be time consuming and prone to errors. Thus our goal is automate the process by structuring vulnerability descriptions in terms of pre- and post-conditions of user and attacker actions, as well as system configuration. We must extract information such as: infected software, preconditions (user actions, attacker strategies, and system activities and configuration) and post-conditions of exploiting particular vulnerability. In this paper we describe a system for automatically extracting information from the National Vulnerability Database (NVD) [8] for a set of vulnerabilities identified by scanner OpenVAS [9]. We evaluate the results by comparing the output with a manually constructed representation.

II. Vulnerability Information

Jajodia et al. [2] built a system that automatically constructs an AG from the information gathered by Nessus, an open source network vulnerability scanner. This solution creates an easy to understand graph, which represents the network's infrastructure. Williams et al. [6] use the Nessus scanner to gain information about vulnerabilities present in hosts. The AG is created and analyzed by a program written in C++. However, both of these systems work only for network analysis and cannot be applied to a single system because they are based on host connectivities.

Le et al. [10] present Vulnerability Property Relationship Graphs (VPRG), a formal model of web-based vulnerabilities, represented as cause/consequence chains. However, the model is constructed manually. Subsequently, Le et al. [11] present an approach for automated extraction of the VPRG model

The EScript.api plugin in Adobe Reader and Acrobat 10.x before 10.0.1, 9.x before 9.4.1, and 8.x before 8.2.6 on Windows and Mac OS X allows remote attackers to execute arbitrary code or cause a denial of service (application crash) via a crafted PDF document that triggers memory corruption, involving the printSeps function. NOTE: some of these details are obtained from third party information.

Fig. 1. CVE-2010-4091 vulnerability description

from a plain text vulnerability description. The system relies on a Natural Language Processing Tools Kit supported by a "Dictionary of Terms and Relationships," which is responsible for providing the information about the relationships, terms and concepts in the vulnerability descriptions. However, constructing the dictionary requires significant effort and is prone to errors. Each sentence is partitioned into entities (noun, prepositional and verb phrases) and behavior (properties which describe the functionality of the web-application), and restated as a cause/consequence chain.

A. The National Vulnerability Database (NVD)

The NVD [8] is maintained by National Institute of Standards and Technology Computer Security Division, Information Technology's Laboratory and sponsored by the Department of Homeland Security's National Cyber Security Division. The NVD combines the information from all government and some commercial vulnerability resources. It offers a comprehensive search capability, delivers necessary statistics, and is updated hourly. The updates can be downloaded from its web page http://nvd.nist.gov/download.cfm

The NVD is based on Common Vulnerability and Exposure (CVE) names [12]. CVE names are distinct identifiers of publicly known security vulnerabilities. A unique identifier is assigned to each vulnerability or exposure by the CVE Numbering Authority, which also posts them on the CVE website. Figure 1 shows a plain text description taken from the NVD entry CVE-2010-4091.

B. The Open Vulnerability Assessment System (OpenVas)

OpenVas [9] is an open source vulnerability scanning and management system that provides a complete security analysis by combining services and security tools. The security scanner is supported by the daily updated feed of the Network Vulnerability Tests (NVTs), which is responsible for detection of known and potential security vulnerabilities. OpenVas offers over 25,000 NVTs (as of May 2012) and is available under the GNU General Public License (GNU GPL).

III. SYSTEM DESIGN

Our objectives are to extract information about the necessary conditions of exploiting a vulnerability and its consequences from plain text vulnerability description. Similarly to [11] we are using the dictionaries to guide the extraction process. However, we store only the keywords such as the name of the software and our approach is based on a set of rules.

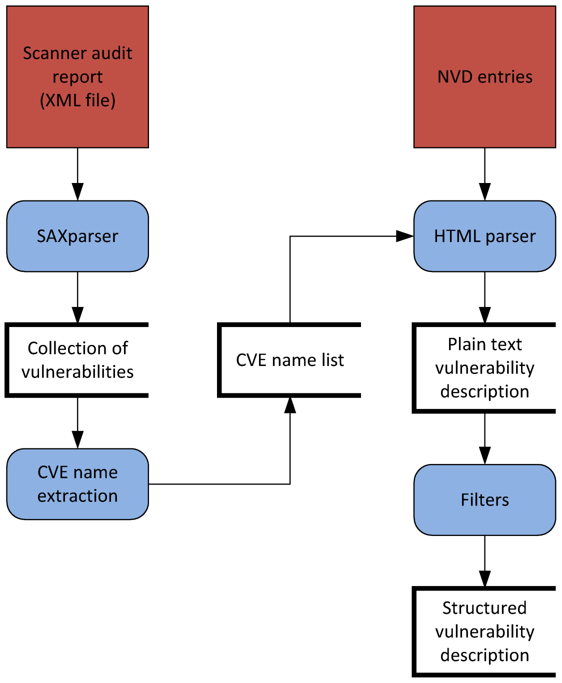

Scanner audit report (XML file)
NVD entries
SAXparser
HTML parser
Collection of vulnerabilities
CVE name list
Plain text vulnerability description
CVE name extraction
Filters
Structured vulnerability description

Fig. 2. Information flow; red squares denote outside system information, blue rounded rectangle are processes, and unclosed rectangles are data storage.

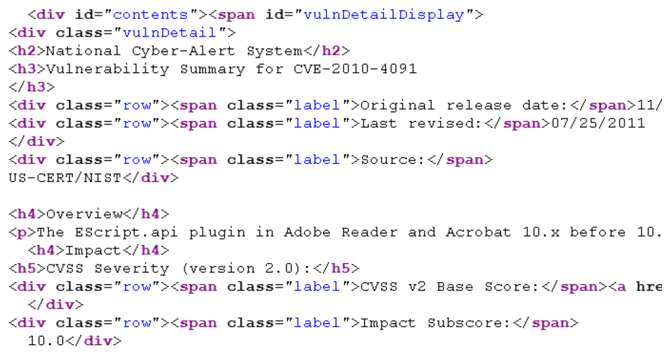
Figure 2 presents the information flow for our system to convert NVD entries to a structured form. We use the NVD as our source of textual vulnerability descriptions and the OpenVAS scanner to identify vulnerabilities in a single host. Similar to [2], [6], a computer is scanned, and an audit report is created and parsed by the SAXparser. The output of this step is a list of vulnerabilities. The CVE names are extracted from this list; the vulnerability records for each are obtained from the NVD in HTML form. The relevant vulnerability records are extracted by the HTML parser. The result at this point is a plain text vulnerability description. Further text processing operations are done on this description by applying a set of filters. The main goal of the filters is to extract necessary causal information such as infected software names and action pre-conditions. The new structured vulnerability description is constructed by applying all the filters.

For the purpose of extraction we use two different parsers:
SAXparser (Simple API for XML) [13] is an event-driven, serial-access mechanism for accessing XML documents.
Jsoup [14] is an open source Java library that provides an API for extraction and management of HTML.

A. CVE name extraction

The vulnerabilities identified by OpenVas are grouped into sets with the same NVT. The XML report created by the scanner contains a lot of XML elements, many of which are not relevant for our analysis (such as scanned port, names of the tasks, etc.). This makes searching more complex.

The parser uses sets of rules to parse the XML tree by looking for the relevant XML element (⟨result⟩, ⟨nvt⟩ and ⟨cve⟩).



```
  <div id="contents"><span id="vulnDetailDisplay">
<div class="vulnDetail">
<h2>National Cyber-Alert System</h2>
<h3>Vulnerability Summary for CVE-2010-4091
</h3>
<div class="row"><span class="label">Original release date:</span>11/
<div class="row"><span class="label">Last revised:</span>07/25/2011
</div>
<div class="row"><span class="label">Source:</span>
US-CERT/NIST</div>

<h4>Overview</h4>
<p>The EScript.api plugin in Adobe Reader and Acrobat 10.x before 10.
  <h4>Impact</h4>
<h5>CVSS Severity (version 2.0):</h5>
<div class="row"><span class="label">CVSS v2 Base Score:</span><a hre
  </div>
<div class="row"><span class="label">Impact Subscore:</span>
  10.0</div>
```

Fig. 3. Example input for HTML parser.

For each ⟨cve⟩ element, the list of CVE names associated with the same NVT is extracted. Finally, a single list with all extracted CVEs is created.

B. HTML Parser

Each CVE name is submitted to the NVD search engine. Using the jsoup library, the vulnerability description is extracted from NVD website HTML code and converted to plain text. Figure 3 presents the example HTML source code for CVE-2010-4091. Algorithm 1 shows the extraction process.

Algorithm 1: Extraction of Vulnerability Description from the HTML Code

```
Require: $list$ {list of *items* with *CVEnames* }
    *NVDaddress* {NVD url address}
Ensure: $text$ {plain text vulnerability description as a String}
 1: for $item \in list$ do
 2:   if item from $list$ is *CVEName* then
 3:     $cvefile \leftarrow$ create.file(*CVEName.txt*)
 4:     $url \leftarrow$ *NVDaddress* + *CVEname*
 5:     $html \leftarrow$ connect($url$)
 6:     $vulDetail \leftarrow html$.getElementByID(”contents”)
 7:     $overview \leftarrow vulDetail$.getElementByTag(”p”)
 8:     $text \leftarrow$ convertToText($overview$)
 9:   else
10:     print “no CVE name”
11:   end if
12: end for
```

C. Filters

To better understand how the extraction process works, we use the CVE-2010-4091 from Figure 1 as an example. Five filter groups are responsible for identifying distinct information from a vulnerability description.

Infected software filters identify software (by name and version) is involved in a particular vulnerability. Algorithm 2 presents how they work. The infected software filters from Figure 1 are described in the phrase: “Adobe Reader and Acrobat 10.x before 10.0.1, 9.x before 9.4.1, and 8.x before 8.2.6.” This extraction is very challenging due to the variety of ways software can be specified, which results in very complex rules. First, a name, e.g., “Adobe Reader”, is found in the SoftwareList, and the occurrence of this name is checked in the vulnerability description. The string after the first occurrence of the name in the text is tokenized by the whitespace characters. Next, the tokens are checked against the rules for what follows (a number or a keyword). The order of occurrence of keywords and numbers is also important. The process iterates as long as software names are found in the plain text description.

Algorithm 2: Extracting Infected Software

```
Require: $text$ {vulnerability description as a String}
    $softwareList$ {file with $softwareNames$}
    $keyWord$ {word which supports extraction}
Ensure: $list$ {list of InfectedSoftware}
 1: for $softwareName \in softwareList$ do
 2:   $indexOfName \leftarrow text$.find($softwareName$)
 3:   $stringToLook \leftarrow text$.substring($indexOfName$)
 4:   $tokens \leftarrow$ tokenize($stringToLook$)
 5:   for $token \in tokens$ do
 6:     if $token$ is number then
 7:       getNext($token$)
 8:       if $token$ is $keyWord$ then
 9:         getNext($token$)
10:       else
11:         $list$.append($infectedSoftware$)
12:       end if
13:     else if ... then
14:       ...
15:     else
16:       return $list$
17:     end if
18:   end for
19: end for
```

Attacker action pre-condition filters are responsible for extracting information about the attacker’s required involvement in exploiting a vulnerability (as shown in algorithm 3). For Figure 1, the phrase: “*a crafted PDF document that triggers memory corruption*” captures an attacker pre-condition; the *keyWordsStart* is “via”, and *keyWordsStop* is a comma. However, in some cases, parsing this description is not straightforward; consider a phrase from CVE-2010-0483: “*by referencing a (1) local pathname, (2) UNC share pathname, or (3) WebDAV server with a crafted .hlp file in the fourth argument (aka helpfile argument) to the MsgBox function,*” the *keyWordsStart* is “by” but we cannot use the first comma for a stop because we would lose the rest of the information in the list.

System pre-condition filters find the system conditions needed for a successful attack, e.g., the phrase “*The EScript.api plugin*”. These filters work similarly to Algorithm 3 with a different set of keywords adapted to this specific

Algorithm 3: Extracting Attacker Action Pre-conditions

```
Require: $text$ {vulnerability description as a String}
    $keyWordStart, keyWordStop$ {word or phrase which support extraction}
Ensure: $list$ {list of Attacker Action Pre-conditions }
 1: scan $text$ from left to right
 2: $indexToStart \leftarrow text$.find($keyWordStart$)
 3: if $indexToStart \geq 0$ then
 4:     $stringToLook \leftarrow text$.substring($indexToStart$)
 5:     $indexToStop \leftarrow stringToLook$.find($keyWordStop$)
 6:     if $indexToStop \geq 0$ then
 7:         $list \leftarrow$ substring($keyWordStart, keyWordStop$)
 8:         return $list$
 9:     end if
10: else
11:     return null
12: end if
```

Algorithm 4: Extracting User Action Pre-conditions

```
Require: $text$ {vulnerability description as a String}
    $keyWord$ {word or phrase which support extraction}
    $helpFile$ {file with $keyWords$}
Ensure: $userAction$ {user actions associated with exploit }
 1: $token \leftarrow$ tokenize($text$)
 2: while $text$ hasNext $token$ do
 3:     get($token$)
 4:     $keyWordToCheck \leftarrow helpFile$.take($keyWord$)
 5:     if $token$ = $keyWordToCheck$ then
 6:         return $userAction$
 7:         break
 8:     else
 9:         getNext($token$)
10:     end if
11: end while
```

extraction. As with the previous filters, the lack of consistency in vulnerability descriptions leads to added complexity. For instance, in (CVE-2008-3107) "*Unspecified vulnerability in the Virtual Machine in Sun Java Runtime Environment (JRE)*", the keyword "in" is not useful as a cue; therefore we have to keep checking contiguous word-tokens until the name of the software is found. Then the extraction rule stops and returns the whole string.

User action pre-condition filters identify conditions necessary for the user's involvement in exploiting a vulnerability. In most cases, the descriptions do not include that information explicitly. So, our system has to reason from what is there, i.e., check a set of conditions and infer (as shown in algorithm 4). From our example, the phrase "*open pdf document*" will be returned.

Post-condition filters extract the consequences of exploiting a particular vulnerability. In our example, filters will find the phrase "*execute arbitrary code or cause a denial of service*" which describes two ways that exploiting this vulnerability affects the system.

The last step is writing the strings returned from filters into a text file which forms the new structured VD. The output file for our example is shown in Figure 4.

```
Infected Software:
    Adobe Reader 10.x before 10.0.1
    Adobe Reader
    Acrobat before 9.4.1
    Acrobat 8.x before 8.2.6
    Acrobat
Precondition:
  attackerActions
    a crafted PDF document that triggers
       memory corruption
   userActions
     user opens pdf file
    systemAtributes
       The EScript.api plugin
Postcondition:
       execute arbitrary code
       cause a denial of service
              (application crash)
```

Fig. 4. Structured CVE-2010-4091 description produced by our system

IV. Evaluation

We evaluate the accuracy of our program by comparing its output to a PAG previously built by experts. Our criteria for this evaluation are: Does the output overlap with that found in the PAG? What information is in the PAG but not in the automatically generated description?

To construct the PAG, we identified vulnerabilities on a computer running Microsoft Windows XP Professional SP3 with common configurations. Before collecting the data, the system was secured and updated. Subsequently, the machine was disconnected from the Internet, and automatic updates were disabled. After three months, the machine was plugged into the Internet and scanned. The 216 vulnerabilities were identified among which 133 were critical, 74 severe and 9 moderate.

We chose a representative set of 11 vulnerabilities that cover a variety of attack scenarios resulting in a compromised system [7]. We constructed a PAG manually from the VDs for the 11 vulnerabilities and for the computer used for the data collection (as shown in Figure 5). To evaluate the automatic construction of the PAG, we ran our program on this set and compared the output with the PAG.

A. Description Accuracy

The output from the program is presented in Figure 4, and the PAG representation is shown in figure 6. Starting from the bottom of the PAG (Figure 5) on the highlighted path in Box1, the infected software is represented by only one node "Adobe Reader 9.4.1." The program output lists five names of infected

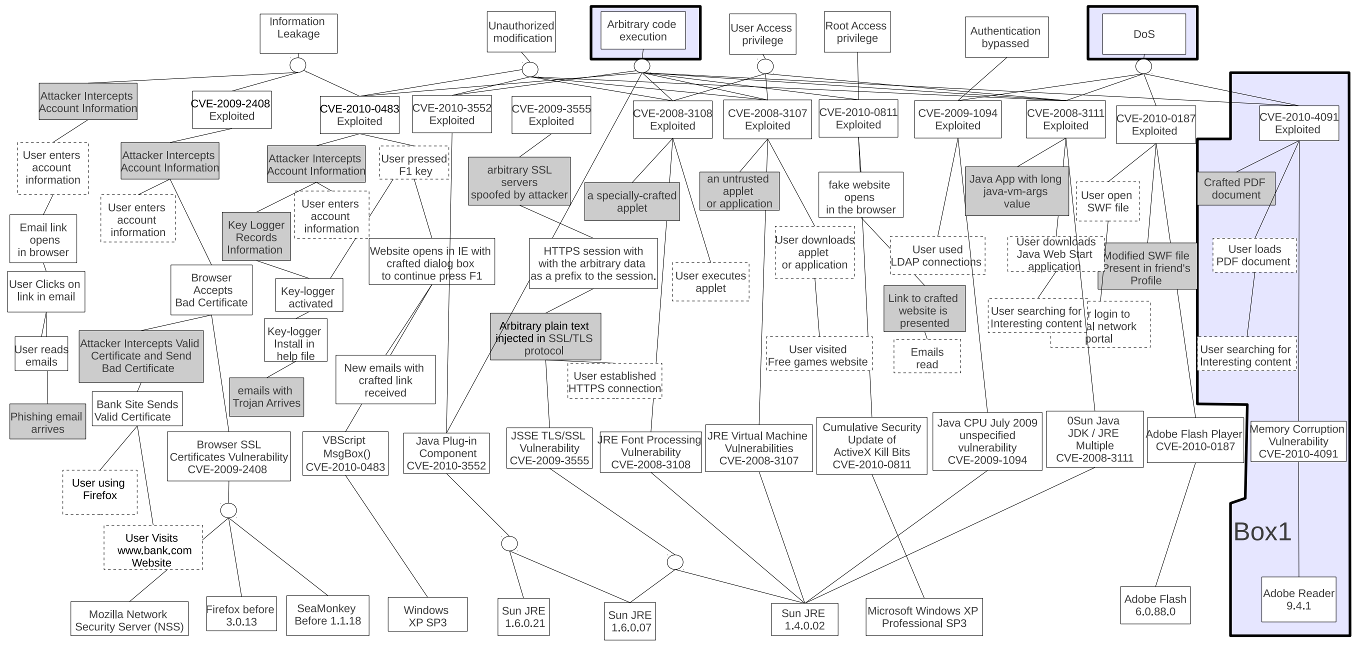

Information Leakage
Unauthorized modification
Arbitrary code execution
User Access privilege
Root Access privilege
Authentication bypassed
DoS
Attacker Intercepts Account Information
CVE-2009-2408 Exploited
CVE-2010-0483 Exploited
CVE-2010-3552 Exploited
CVE-2009-3555 Exploited
CVE-2008-3108 Exploited
CVE-2008-3107 Exploited
CVE-2010-0811 Exploited
CVE-2009-1094 Exploited
CVE-2008-3111 Exploited
CVE-2010-0187 Exploited
CVE-2010-4091 Exploited
User enters account information
Attacker Intercepts Account Information
Attacker Intercepts Account Information
User pressed F1 key
arbitrary SSL servers spoofed by attacker
a specially-crafted applet
an untrusted applet or application
fake website opens in the browser
Java App with long java-vm-args value
User open SWF file
Crafted PDF document
Email link opens in browser
User enters account information
Key Logger Records Information
User enters account information
Website opens in IE with crafted dialog box to continue press F1
HTTPS session with with the arbitrary data as a prefix to the session.
User executes applet
User downloads applet or application
User used LDAP connections
User downloads Java Web Start application
Modified SWF file Present in friend's Profile
User loads PDF document
User Clicks on link in email
Browser Accepts Bad Certificate
Key-logger activated
Link to crafted website is presented
User searching for Interesting content
r login to al network portal
User reads emails
Attacker Intercepts Valid Certificate and Send Bad Certificate
Key-logger Install in help file
Arbitrary plain text injected in SSL/TLS protocol
User visited Free games website
Emails read
User searching for Interesting content
New emails with crafted link received
User established HTTPS connection
Phishing email arrives
Bank Site Sends Valid Certificate
emails with Trojan Arrives
Browser SSL Certificates Vulnerability CVE-2009-2408
VBScript MsgBox() CVE-2010-0483
Java Plug-in Component CVE-2010-3552
JSSE TLS/SSL Vulnerability CVE-2009-3555
JRE Font Processing Vulnerability CVE-2008-3108
JRE Virtual Machine Vulnerabilities CVE-2008-3107
Cumulative Security Update of ActiveX Kill Bits CVE-2010-0811
Java CPU July 2009 unspecified vulnerability CVE-2009-1094
0Sun Java JDK / JRE Multiple CVE-2008-3111
Adobe Flash Player CVE-2010-0187
Memory Corruption Vulnerability CVE-2010-4091
User using Firefox
User Visits www.bank.com Website
Box1
Mozilla Network Security Server (NSS)
Firefox before 3.0.13
SeaMonkey Before 1.1.18
Windows XP SP3
Sun JRE 1.6.0.21
Sun JRE 1.6.0.07
Sun JRE 1.4.0.02
Microsoft Windows XP Professional SP3
Adobe Flash 6.0.88.0
Adobe Reader 9.4.1

Fig. 5. Custom made graph, PAG, that captures the interplay between these vulnerabilities, user actions, attacker strategies, and system activities; Box1 highlights the exploit path equivalent to figure 1 .

```
Infected Software:
  Adobe Reader 9.4.1
Precondition:
  attackerActions
    Crafted PDF document
  userActions
    User searching for interesting content
    User loads PDF document
  systemAtributes
    Memory Corruption Vulnerability
       CVE-2010-4091
Postcondition:
       arbitrary code execution
       DoS
```

Fig. 6. CVE-2010-4091 description from PAG

software; one of them, "Adobe Reader," generally corresponds to "Adobe Reader 9.4.1" from the PAG. However, according to our goal (extracting precise information), this is not quite correct. The difference here occurred because the NVD entry has been updated since the PAG was created.

The next node up in Box1 states the existence of the vulnerability. The output further specifies the name of the vulnerable element in the infected software: *systemAtributes* of EScript.api. The PAG node "User searching for interesting content" does not have any equivalent in the output file. In this case, the output does not include this level of granularity.

The PAG node "User loads PDF document" corresponds to *userActions* from the output: "user opens pdf file." This information is not provided explicitly in the vulnerability description from NVD, but was inferred. The PAG node "Crafted PDF document" corresponds to "a crafted PDF document that triggers memory corruption" in *attackerActions*. In this case, the output delivers more information than is obtained from the PAG. The "Postcondition" from the output is equivalent to the top nodes in the PAG.

A similar evaluation is done for the rest of the vulnerabilities; the results are presented in Table I. The system has the best performance in the post-condition and infected software columns because it extracts the newest information directly from the website. The user action column shows the worst system performance because the PAG offers a better level of granularity than the system, which includes only the user actions that are directly associated with exploiting a vulnerability. The accuracy of the system strictly depends on the quality of the rules for extraction and their ability to infer what is implied.

V. Conclusions and Future Work

The vulnerability descriptions which are available from current VDs are not in a form that expedites security analyses, especially when the system being analyzed is a single computer. The descriptions do not directly provide the necessary information about user, and attacker involvement and the causal relationships need to be inferred. Moreover, the variability in the descriptions significantly complicates the process of constructing a structured description. Yet, automatically extracting the structured representation to build the PAG offers the promise of significantly reducing errors (due to out of date representations and missing information).

TABLE I
COMPARISON BETWEEN PAG AND SYSTEM; "O" OVERLAPS, "S"-AUTOMATIC EXTRACTION BY SYSTEM IS BETTER, "P"-MANUALLY BUILT PAG PERFORMS BETTER, AND "(P,S)/G"- BETTER LEVEL OF GRANULARITY

CVE Name	Post-condition	infected software	Precondition		
			Attacker Action	System	User Action
CVE-2010-4091	o	s	s	s	p/g
CVE-2009-2408	s	s	o	o	p
CVE-2010-0483	s	s	p	s	p
CVE-2010-3552	s	s	s	o	o
CVE-2009-3555	s	s	p	s	p
CVE-2010-0811	s	s	s	s	p/g
CVE-2009-1094	s	s	s	s	o
CVE-2008-3111	s	s	s	o	p/g
CVE-2008-3108	s	s	s	o	o
CVE-2008-3107	s	s	o	o	p/g
CVE-2010-0187	o	s	o	p	p/g

We demonstrated a method for automatically the extracting vulnerabilities set and evaluated that method on 11 vulnerabilities that form the core of a previously developed formal model. The current version of our program is able to construct descriptions of this set of actual vulnerabilities that are even sometimes more complete than those in a hand-constructed PAG. We should note that the program's filters were developed from our experience in building the PAG in the first place.

Thus, given the variability in vulnerability descriptions, the program's accuracy is likely to decrease as new vulnerability descriptions are added. New patterns will need to be added to the filters to accommodate the lack of consistency in the prose for the vulnerability descriptions and the lack of explicitly included information. Therefore, correct extraction requires an expansion in the number of rules which grows quickly with the number of cases. For real examples, with approximately hundreds of vulnerabilities in one computer system, we need to generate the extraction patterns automatically.

In our future work we plan to address most of problems stated above. Especially, we will investigate the time in which the extracting rules can be created and validated. We will also reduce the human factors that can cause errors in the whole process of extraction. To accomplish that, we are going to use one of the existing information extraction systems with a machine learning algorithm to generate extraction patterns.

VI. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0905232. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] K. Ingols, M. Chu, R. Lippmann, S. Webster, and S. Boyer, "Modeling modern network attacks and countermeasures using attack graphs," in *Proc. of the 25th Annual Computer Security Applications Conference*, Honolulu, HI, USA, Dec. 2009, pp. 117–126.

[2] S. Jajodia, S. Noel, and B. OBerry, "Topological analysis of network attack vulnerability," in *Managing Cyber Threats*, ser. Massive Computing, vol. 5, 2003, pp. 247–266.

[3] S. Jha, O. Sheyner, and J. Wing, "Two formal analyses of attack graphs," in *Proc. of the 15th IEEE workshop on Computer Security Foundations*, Cape Breton, Nova Scotia, Canada, June 2002, pp. 49–63.

[4] N. Poolsappasit, R. Dewri, and I. Ray, "Dynamic security risk management using bayesian attack graphs," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 1, pp. 61–74, Jan. 2011.

[5] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing, "Automated generation and analysis of attack graphs," in *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2002, pp. 273–284.

[6] L. Williams, R. Lippmann, and K. Ingols, "An interactive attack graph cascade and reachability display," in *Proc. of the Workshop on Visualization for Computer Security*, Sacramento, CA, USA, Oct. 2007, pp. 221–236.

[7] M. Roberts, A. E. Howe, I. Ray, and M. Urbanska, "Using planning for a personalized security agent," in *Workshop on Problem Solving using Classical Planners in Working Notes of the 26th AAAI Conference on Artificial Intelligence*, Toronto, Ontario, Canada, July 2012.

[8] National Vulnerability Database, "NVD XML Feed Documentation," Available from http://nvd.nist.gov, NVD, http://nvd.nist.gov, September 2012.

[9] The Open Vulnerability Assessment System (OpenVAS), "Community project carried out by volunteers," Available from http://www.openvas.org/, Sept. 2012.

[10] H.-T. Le, D. Subramanian, W.-J. Hsu, and P. K. K. Loh, "An empirical property-based model for vulnerability analysis and evaluation," in *Proc. of IEEE Asia-Pacific Services Computing Conference*, Dec. 2009, pp. 40–45.

[11] H.-T. Le and P. K. K. Loh, "Using natural language tool to assist vprg automated extraction from textual vulnerability description," in *Proc. of IEEE Workshops of International Conference on Advanced Information Networking and Applications*, Mar. 2011, pp. 586 –592.

[12] Common Vulnerabilities and Exposure, "The dictionary of common names (CVE identifiers)," http://cve.mitre.org/, Sept. 2012.

[13] Simple API for XML, "David megginson," Available from http://www.saxproject.org/, Sept. 2012.

[14] jsoup, " Java HTML Parser," Available from http://jsoup.org/, Sept. 2012.