Past, Present and Future of Vulnerability Discovery

funnywei
Speech outline

• The development of vulnerability discovery technology
• The security of patches
The Development of Vulnerability Detection

- past → present
- future
Existing Paradigms

- Testing
- Formal verification

Cost

Assurance

(sweet spot?)

Extracted from D. Wagner’s ppt
Vulnerability discovery

- fuzzing
- API testing
- static analysis
Fuzzing targets

- File Formats
- Network Protocols
- Web applications
- Environment variables
- COM Objects
- IPCs
Fuzz

• **Advantage:**
  - Easily understand
  - No time gap between finding vulnerabilities and triggering vulnerabilities.
  - No false positive

• **Disadvantage:**
  - Disadvantages of Black box testing
  - The period of constructing test case such as complex protocol is tedious
  - Unable to test undocumented interfaces
Framework of File Format Fuzzing

1. **Generate fuzzed data**
2. **Start executing**
3. **Monitor for exception**

- **Patterns**
  - Brute force
  - Equivalent classes
  - Boundary values
  - Grouping of Compound fields
  - Record exceptions code
  - Analysis and finding out useful exceptions
  - Determine exploitability
API testing

• Test API security
  – For example, NetpwPathCanonicalize, the 303th export function of netapi32.dll

• Find out those dlls or exes that call the API.

• test the security of these dlls or exes when they call the API
Static analysis

• Attempt to find specific vulnerable code patterns, or detect a possible control-flow path on which the program reaches a vulnerable state.

• the question arose:
  Given a program and security property, the goal is to verify whether the program satisfies the property, and if not, identify why.
Static analysis framework

Program → program's model → analysis algorithm → result

vulnerability → vulnerability's model
Static analysis (1)

- Pattern matching
- AST analysis
- Lexical analysis
- bugscam
- metaC++
- FlawFinder
- Rats
- ITS4
Static analysis (2)

- dataflow analysis
- type system
- tainted analysis
- model checking
- static analysis
- Cqual
Model Vulnerability

- Temporal logic
- Constrain based model
- Type qualifier
- Dependence Graph based Model
Temporal logic model

- Describe the order of security-relevant operations as temporal safety logic
- Use FSA model to describe the properties
- Reference the article -- "Model Checking One Million Lines of C Code"
A simple Example of temporal safety properties

• After calling chroot, a process should immediately call chdir(“/”) to change its working directory to the root of the sub-filesystem.
Constraint model

David Wagner et al. formalize the detection of buffer overflow to integer constraint.

The generation of linear constraint condition

len(s) represent the size of usage, the range is [a, b].
Alloc(s) represent the size of allocated buffer, the range is [c, d].
If b<c, overflow will not happen
If a>d, overflow will happen
If d>b>c>a, overflow may happen
Type Qualifier Model

- Type qualifier framework developed by Jeffrey foster
- One approach: static *taint analysis*
  - Extend the C type system
  - Qualified types express annotations: e.g., `tainted char *` is an untrusted string
  - Type checking enforces safe usage
  - Type inference reduces annotation burden
Dependence Graph based analysis

- Dependence Graphs were first presented by Ottenstein et al. as an intermediate form in 1984.
- What is PDG (Program Dependence Graph)?
  - vertices—represent statements
  - edges—represent control and data flow
    - Dashed line represent data flow dependence
    - Real line represent control flow dependence
void func()
{
    int sum = 0, i = 1;
    while (i < 11)
    {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}
PDG simple examples(2)

entry func()

i=1 → printf(i) → while(i<11) → sum=0 → printf(sum)

i=i+1 → sum=sum+i
SDG(System Dependence Graph)

- Presented by Horwitz et al, 1990.
- A procedural call from procedural A to B is modeled with a call vertex in A, an entry vertex in B, and an introprocedural control dependence edge between them.
- Parameters are handled with actual-in and actual-out vertices in A, formal-in and formal-out vertices in B, and interprocedural data dependence edges connecting them.
- If a procedural uses a global variable, it is treated as a (hidden) input parameter, and is encoded as additional actual-in and formal-in vertices.
Detection of Integer flaws (1)

The input has to be validated before it is used.

- the use point has to be control dependent on the validation point.

The usage must be defined firstly

- the use point has to be data flow control on the define

Correct code pattern for integer input validation
Detection of Integer flaws(2)

Incorrect code patterns (partial)
Double free flaw

Incorrect code pattern

```c
char * buf = (char*)malloc(1024);
...
free(buf);
...
free(buf);
```
Static analysis

- **Disadvantages**
  - Rely on static rules of Predetermined code patterns
  - Any sufficient complex analysis of a program is equivalent to solving the halting problem, which provability cannot be decided.
  - Infeasible path $\Rightarrow$ False positive.
  - Time Gap between finding vulnerabilities and exploiting

- **Advantages**
  - Universal
  - Vulnerability easily located

- **Other purpose**
  - Assist patches analysis
Source-level analysis vs executables analysis

- why we could not apply the techniques of source code analysis to executables
  - No variables
  - No notion of type

- Reasons why we need to analyse executables
  - “what you see is not what you execute” phenomenon
  - Source level tools are only applicable when source is available
  - Source-level analysis tool typically skip over inlined assembly code
  - Make use of libraries which may not be available in source code.
Abstract interpretation

- **Analysis in the past**
  - Tracked only registers
  - If a register is assigned a value from memory, it is assumed to take on any value.

- **Recovery variables**
  - Abstract the runtime address space

- **Recovery Structures**
  - Aggregate structures identification

- **Design an intermediate representation**
  - High level code $\rightarrow$ IR $\rightarrow$ low level code
  - Low level code $\rightarrow$ IR $\rightarrow$ high level code
Principles of static analysis algorithm’s design

• Major goals:
  – scalability
  – Soundness
  – precision

• algorithm
  – Flow sensitive—dependent on control flow
  – Context sensitive
Current status of vulnerability detection tools

• characteristics
  – Partial automation
  – Linear program analysis

• disadvantages:
  – false positive
  – Lack of excellent graphical illustration
  – Few research in multi-threaded program
Partial Automation

• Automatically Testing
  – automatically construct test instances
  – automatically monitor executions

• Automatically analysis
  – Ranking of potential vulnerabilities
    • So we can put emphasis on analyzing the more important vulnerabilities.
  – Record execution path of error and locate vulnerable functions more quickly
  – Record stack information.
  – Record contents of registers
Ranking of potential vulnerabilities

- Generate severity ranking by
  - The more pointer analysis involved in finding a flaw, the lower the ranking
  - The more aliasing involved in finding a flaw, the lower the ranking
  - Flaws involving inter-procedural analysis are ranked lower than intra-procedural ones.

- Recognize type of
  - Overflows—heap? or Stack?
  - Integer handling—Overflows or Signedness?
  - DoS--Out of bounds reads? Infinite loops? NULL pointer dereferences?
  - Logic errors
  - Format strings
  - Race conditions
Data flow analysis

• Equations set—iteration method
  – Classic method

• Graph reachability—CFL reachability problems
Declare g:int

Procedure main
Begin
    declare x:int
    read(x)
    call P(x)
End

Procedure P(value a:int)
Begin
    if (a > 0) then
        read (g)
        a:= a-g
        call P(a)
        print(a, g)
    fi
end
CFL reachability problem

- The path \( \text{start}_{\text{main}} \rightarrow n1 \rightarrow n2 \rightarrow \text{start}_{\text{p}} \rightarrow n4 \rightarrow \text{exit}_{\text{p}} \rightarrow n3 \) \( \rightarrow \) \( \text{ee}(1\text{ee})_1 \)
  - Both a matched path and a realizable path
- The path \( \text{start}_{\text{main}} \rightarrow n1 \rightarrow n2 \rightarrow \text{start}_{\text{p}} \rightarrow n4 \) \( \rightarrow \) \( \text{ee}(1\text{e}) \)
  - A realizable path but not a matched path
- The path \( \text{start}_{\text{main}} \rightarrow n1 \rightarrow n2 \rightarrow \text{start}_{\text{p}} \rightarrow n4 \rightarrow \text{exit}_{\text{p}} \rightarrow n8 \) \( \rightarrow \) \( \text{ee}(1\text{ee})_2 \)
  - Neither a matched path nor a realizable path
Future

- Visualization in all stages
- higher automation
- parallelization
- Exploitability determination
- integrate static analysis with dynamic detection
Bottle-neck—phenomena

- Fuzzer Tech
  - Analysis generated exceptions—time consuming
- Static Analysis
  - large computation
    - NP problem
    - Halting problem
  - Trade off between scalability and precision
Bottle-neck

• analysis of Exceptions
  – Exploitability determined by people
    • Need for Graphical illustration
    • Need for automatic analysis

• Static analysis
  – require higher precision
    • More computation
visualization

- visualization
- vulnerability discovery process
- exploitation process
- debugging and tracing process
- testing process
- vulnerability models
- development of Exploit
- progress of testing
- data flow
- reason for vulnerability
- debug process
Parallization

(1) J. Wilander’s experiment
- Wu-ftpdl 2.6-4 consists of app 20,000 lines of code and produce a dependence graph with approx 130,000 vertices.
- An analysis for integer input validation flaws took 15h on a 2.66 GHz Pentium 4.

(2) Balakrishnan’s experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>Procedures</th>
<th>Instructions</th>
<th>Malloc sites</th>
<th>Indirect jumps</th>
<th>Calls</th>
<th>Indirect calls</th>
<th>Memory usage (MB)</th>
<th>Value-set analysis (sec.)</th>
<th>Affine-relation analysis (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>javac</td>
<td>36</td>
<td>3555</td>
<td>1</td>
<td>0</td>
<td>133</td>
<td>79</td>
<td>51</td>
<td>76</td>
<td>29</td>
</tr>
<tr>
<td>cat (2.0.14)</td>
<td>123</td>
<td>3892</td>
<td>1</td>
<td>3</td>
<td>138</td>
<td>4</td>
<td>42</td>
<td>9</td>
<td>26</td>
</tr>
<tr>
<td>cut (2.0.14)</td>
<td>129</td>
<td>4329</td>
<td>2</td>
<td>3</td>
<td>182</td>
<td>4</td>
<td>48</td>
<td>7</td>
<td>42</td>
</tr>
<tr>
<td>grep (2.4.2)</td>
<td>245</td>
<td>16808</td>
<td>18</td>
<td>4</td>
<td>654</td>
<td>6</td>
<td>102</td>
<td>117</td>
<td>75</td>
</tr>
<tr>
<td>gcc (2.96)</td>
<td>252</td>
<td>22984</td>
<td>8</td>
<td>3</td>
<td>1048</td>
<td>4</td>
<td>232</td>
<td>108</td>
<td>295</td>
</tr>
<tr>
<td>tar (1.13.19)</td>
<td>581</td>
<td>47739</td>
<td>11</td>
<td>21</td>
<td>2553</td>
<td>29</td>
<td>258</td>
<td>220</td>
<td>156</td>
</tr>
<tr>
<td>awk (3.1.0)</td>
<td>595</td>
<td>69927</td>
<td>84</td>
<td>33</td>
<td>3669</td>
<td>152</td>
<td>623</td>
<td>1017</td>
<td>1011</td>
</tr>
<tr>
<td>winhlp32 (5.0.2195.2014)</td>
<td>1018</td>
<td>108380</td>
<td>0</td>
<td>10</td>
<td>6002</td>
<td>1005</td>
<td>737</td>
<td>1712</td>
<td>1290</td>
</tr>
</tbody>
</table>
automation

• Automation of testing process
• Automation of analysis process
Automatic Determination of exploitability

- It is hard!
- We need to know the privilege
  - Privilege inference
    - Finding all privileges reaching a given program point
- Static analysis
  - Whether integer overflow is related to the operation of array
  - The precision of exploitability is determined by the precision of algorithm
  - Etc.
- Dynamic Detection
  - analysis of overflowed stack
  - Analysis of registers
  - Integrated static analysis, can determine the vulnerability type of error function
The Security of Patches

- Why the need to research the security of patches
- Theory of structural comparison
- Improved comparison algorithm
- Some new methods of comparison
Why the need to research the security of patches

- Anti bin-diff
- Anti 0-day
# Examples of Patches (using FunnyDiff)

<table>
<thead>
<tr>
<th>patch</th>
<th>Files</th>
<th>version</th>
<th>size</th>
<th>Num of functions</th>
<th>Matched number</th>
<th>Changed number</th>
<th>Running time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS06-040</td>
<td>Netapi32.dll</td>
<td>5.0.2195.7105</td>
<td>309,520</td>
<td>1,172</td>
<td>1,168</td>
<td>7</td>
<td>7,153</td>
</tr>
<tr>
<td></td>
<td>Netapi32.dll</td>
<td>5.0.2195.7038</td>
<td>326,928</td>
<td>1,168</td>
<td>1,168</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS06-020</td>
<td>Msdtcprx.dll</td>
<td>2001.12.4414.311</td>
<td>426,496</td>
<td>2,623</td>
<td>2,621</td>
<td>4</td>
<td>10,828</td>
</tr>
<tr>
<td></td>
<td>Msdtcprx.dll</td>
<td>2001.12.4414.308</td>
<td>425,472</td>
<td>2,621</td>
<td>2,621</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS05-047</td>
<td>Umpnpmgr.dll</td>
<td>5.1.2600.2744</td>
<td>121,344</td>
<td>268</td>
<td>249</td>
<td>19</td>
<td>1,062</td>
</tr>
<tr>
<td></td>
<td>Umpnpmgr.dll</td>
<td>5.1.2600.2710</td>
<td>116,224</td>
<td>249</td>
<td>249</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS06-014</td>
<td>Msadco.dll</td>
<td>2.81.1124.0</td>
<td>143,360</td>
<td>896</td>
<td>894</td>
<td>3</td>
<td>2,594</td>
</tr>
<tr>
<td></td>
<td>Msadco.dll</td>
<td>2.81.1117.0</td>
<td>143,360</td>
<td>894</td>
<td>894</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS06-002</td>
<td>Fontsub.dll</td>
<td>5.0.2195.7071</td>
<td>79,632</td>
<td>204</td>
<td>183</td>
<td>35</td>
<td>968</td>
</tr>
<tr>
<td></td>
<td>Fontsub.dll</td>
<td>5.0.2180.1</td>
<td>78,096</td>
<td>183</td>
<td>183</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M05-039</td>
<td>Umpnpmgr.dll</td>
<td>5.0.2600.2710</td>
<td>116,224</td>
<td>249</td>
<td>249</td>
<td>1</td>
<td>1,000</td>
</tr>
<tr>
<td></td>
<td>Umpnpmgr.dll</td>
<td>5.0.2600.2180</td>
<td>116,224</td>
<td>249</td>
<td>249</td>
<td></td>
<td></td>
</tr>
<tr>
<td>patch</td>
<td>Same name</td>
<td>Unique name</td>
<td>Reference the same strings</td>
<td>The same prime product</td>
<td>Same indegree</td>
<td>recursive</td>
<td>Heuristic matching</td>
</tr>
<tr>
<td>--------</td>
<td>-----------</td>
<td>-------------</td>
<td>----------------------------</td>
<td>------------------------</td>
<td>---------------</td>
<td>-----------</td>
<td>-------------------</td>
</tr>
<tr>
<td>06-040</td>
<td>686</td>
<td>243</td>
<td>72</td>
<td>13</td>
<td>34</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>06-020</td>
<td>2529</td>
<td>12</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>05-047</td>
<td>237</td>
<td>3</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>06-014</td>
<td>873</td>
<td>14</td>
<td></td>
<td>4</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>06-002</td>
<td>99</td>
<td>38</td>
<td></td>
<td>9</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>
Theory of structural comparison

- Control flow based
- Partial order based
Improved comparison algorithm

- Initialize trusted fixed points
  - fuzzy clustering based
- Heuristic verification during fixed point propagation
How to improve patches’ security

• Reduction of function names dumped from symbol file (PDB)
• Change the structural signature of function
  – Add meaningless if-else structure to generate incorrect signature.
• Handling referenced strings
  – Such as encoded strings
• Change of Prime product
  – Replacement of equivalent instructions
  – Add random meaningless (redundant) instructions
some new methods of comparison

• Import/export table functions as fixed points
• The same variables
• RPC interface
• The same stack layout
• The same parameters
• Loop’s property
• The end