Towards Automatic Generation of Vulnerability-Based Signatures

David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha
(presented by Boniface Hicks)
Beware the LION

• New year 2001
• 10,000 systems affected
• invades Linux systems through a network exploit
• infiltrates BIND DNS through TCP or UDP protocol
• allows infiltration through a legit request, but then can execute arbitrary commands through additional string of characters.
• incident report March 30 by CERT
Vulnerability

• **vulnerability** - type of bug that can be used by an attacker to alter the intended operation of software in a malicious way

• **exploit** - an actual input that triggers a devastating consequence (these can be polymorphic)
Overview

• **Goal**: *automatic* signature generation

• **Challenges**:  
  ‣ polymorphism  
  ‣ vulnerability-based (*not exploit-based*)

• **Approach**:  
  ‣ *vulnerability signature*: whether executing an input potentially results in an unsafe program state  
  ‣ *vulnerability condition*:  
    • representation (how to express a vulnerability as a signature)  
    • coverage (measured by false positive rate)
Contributions

• requires single sample exploit to identify vulnerability
• formal definition of vulnerability signature
• expose trade-off between matching time and accuracy
• notion of coverage introduced
• manual control of introducing imprecision
• new static analysis techniques and novel application
• prototype implementation handles COTS binaries
Vulnerability signature

- vulnerability signature - representation for set of inputs that define a specified vulnerability condition

- trade-offs:
  - representation: matching accuracy vs. efficiency
  - signature creation: creation time vs. coverage

- \{P, T, x, c\} = binary program (P), instruction trace (T), exploit string (x), vulnerability condition (c).
Example

• P given in box
• x = g/AAAA
• T =
  {1, 2, 3, 4, 6, 7, 8, 9, 8, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11}
• c = heap overflow (on 5th iteration of line 11)

```c
1 char *geturl (char inp[10]) {
2   char *url = malloc(4);
3   int c = 0;
4   if (inp[c] != 'g' && inp[c] != 'G')
5     return NULL;
6   inp[c] = 'G';
7   c++;
8   while (inp[c] == ' ')
9     c++;
10  while (inp[c] != ' ')
11     url = inp[c]; c++; url++;
12 }
13 printf("%s", url);
14 return url;
15}
```
Vulnerability sig notation

- \( (P, c) = (< i_1, \ldots, i_k >, c) \)
- \( T(P, x) \) is the execution trace of running \( P \) with input \( x \).
- \( T \models c \) means \( T \) satisfies vulnerability condition \( c \)
- \( L_{P,c} \) consists of the set of all inputs \( x \) to a program \( P \) such that \( T(P, x) \models c \)
- Formally: \( L_{P,c} = \{ x \in \Sigma^* | T(P, x) \models c \} \)
- An exploit for a vulnerability \((P,c)\) is an input \( x \in L_{P,c} \)
Vulnerability sig definition

• A vulnerability signature is a matching function MATCH which for an input $x$ returns either EXPLOIT or BENIGN for a program $P$ without running the program.

• A perfect vulnerability signature satisfies:

$$\text{MATCH}(x) = \begin{cases} \text{EXPLOIT} & \text{when } x \in L_{P,c} \\ \text{BENIGN} & \text{when } x \notin L_{P,c} \end{cases}$$

• completeness: $\forall x : x \in L_{P,c} \Rightarrow \text{MATCH}(x) = \text{EXPLOIT}$

• soundness: $\forall x : x \notin L_{P,c} \Rightarrow \text{MATCH}(x) = \text{BENIGN}$
Vulnerability Condition

\[ c : \Gamma \times D \times M \times K \times I \rightarrow \{\text{BENIGN, EXPLOIT}\} \]

- \( \Gamma \) is a memory
- \( D \) is the set of variables defined
- \( M \) is the program’s map from memory to values
- \( K \) is the continuation stack
- \( I \) is the next instruction to execute
Example

• Formal operational semantics definition of the vulnerability in the example:

\[ \Gamma, D, M, K \vdash *\text{exp} \rightsquigarrow D, M, K \triangleright *\Box \vdash \text{exp} \]

\[ \Gamma[n \rightarrow \text{SafePtr}(m, s)] , D, M : [n \rightarrow v_n], K \triangleright *\Box \vdash n \]

\[ \rightsquigarrow \begin{cases} 
\text{BENIGN if } m \leq n < m + s \\
\text{EXPLOIT} 
\end{cases} \]

1. In order to dereference a pointer *\text{exp}, \text{exp} must first be be evaluated.
2. Once \text{exp} is resolved to address \text{n}, get a safe pointer to \text{n}, (with base address \text{m}, size \text{s}) and check whether it’s in the specified range.
Sig representation classes

• Turing machine signatures
  ‣ precise (no false positive or negatives)
  ‣ may not terminate (in presence of loops, e.g.)

• symbolic constraint signatures
  ‣ approximates looping, aliasing
  ‣ guaranteed to terminate

• regular expression signatures
  ‣ approximates elementary constructs (counting)
  ‣ very efficient
• Can provide a precise, even exact, characterization of the vulnerability condition in a particular program.

• A TM that exactly emulates the program has no error rate.

```c
char *url = malloc(4);
int c = 0;
if (inp[c] != 'g' && inp[c] != 'G')
    return BENIGN;
c++;
while (inp[c] == ' ') c++;
while (inp[c] != ' ') {
    if (c >= 4) return EXPLOIT;
    *url = inp[c]; c++; url++;
}
return BENIGN;
```
Example

- **symbolic constraint** says that for 10-char input, the first char is ‘g’ or ‘G’, up to four of the next chars may be spaces and at least 5 chars are non-spaces.

- **regexp:** `[g|G][ ][ ]*[^ ]{5,}` says ‘g’ or ‘G’ followed by 0 or more spaces and at least 5 non-spaces.

    symbolic constraint sig for example

    \[
    (\text{inp}[0] = 'g' \lor \text{inp}[0] = 'G') \land \big[ (\text{inp}[1:5] \neq ' ') \lor \\
    (\text{inp}[1] = '' \land \text{inp}[2:6] \neq '') \lor \\
    (\text{inp}[1:2] = '' \land \text{inp}[3:7] \neq '') \lor \\
    (\text{inp}[1:3] = '' \land \text{inp}[4:8] \neq '') \lor \\
    (\text{inp}[1:4] = '' \land \text{inp}[5:9] \neq '') \big]
    \]
Accuracy vs. Efficiency

<table>
<thead>
<tr>
<th>Representation</th>
<th>Creation</th>
<th>Signature Size</th>
<th>Matching</th>
<th>Minimization</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turing machine Sig.</td>
<td>poly(N)</td>
<td>poly(N)</td>
<td>Undecidable</td>
<td>Undecidable</td>
<td>Undecidable</td>
</tr>
<tr>
<td>Symbolic Constraint Sig.</td>
<td>poly(N)</td>
<td>poly(N)</td>
<td>poly(S)</td>
<td>exp(S)</td>
<td>exp(S)</td>
</tr>
<tr>
<td>Regular Expression Sig.</td>
<td>poly(N) - exp(N)</td>
<td>exp(N)</td>
<td>O(S)</td>
<td>O(S^2)</td>
<td>O(S^2)</td>
</tr>
</tbody>
</table>

- TM - inlining vulnerability condition takes poly time
- Symb. Constraint - poly-time transformations on TM
- Regexp - solve constraint (exp time; PSPACE-complete) or data-flow on TM (poly time)
MEP and PEP coverage

- MEP is a straight-line program -- e.g. the path that the exploit took to reach the vulnerability
- PEP includes different paths to the vulnerability
- A complete PEP coverage signature accepts all inputs in $L_{P,c}$
- Complete coverage through a chop of the program includes all paths from the input read ($v_{init}$) to the vulnerability point ($v_{final}$).
Procedure

• Get MEP for exploit (e.g. T in example)
• Compute chop for MEP to get complete coverage
• Compute initial signature S
• refine S by adding alternative MEPs from chop
• Example:
  ‣ initial MEP is T
  ‣ PEP is lines {1-5, 7-12}
Algorithm

1. Pre-process
   1. Disassemble binary
   2. Convert to an intermediate representation (IR)

2. Chop P wrt trace T, develop complete PEP

3. Compute the signature
   1. Compute TM signature
   2. Develop symbolic constraint signature
   3. Compute regular expression
Evaluation

• 9000 lines C++ code

• CBMC model checker to build/solve symbolic constraints, generate regexp’s

• disassembler based on Kruegel; IR new

• ATPhttpd
  ▸ various vulnerabilities; sprintf-style string too long
  ▸ 10 distinct subpaths to regexp in 0.1216sec

• BIND
  ▸ stack overflow vulnerability; TSIG vulnerability
  ▸ 10 distinct graphs in symbolic constraint
  ▸ 30 micro-secs for chop
  ▸ 88% of functions were reachable between entry and vulnerability