Dowsing for overflows:
a guided fuzzer to find buffer boundary violations

István Haller, Asia Slowinska, Matthias Neugschwandtner, Herbert Bos

Usenix Security 2013
August 14, 2013
Bugs, bugs everywhere

- Buffer overflows still represent a top 3 threat (after 40 years)
- Applications grow at a rapid pace, testing cannot keep up
- Containment of software faults?
- Solve the root cause via automated testing!
Possibility of automated testing

**Static analysis**
- Deployed in practice
- Difficult to make path-sensitive and inter-procedural
- Lack of accuracy makes for many FPs/FNs

**Symbolic execution**
- Observations only relevant for given execution path
- Core focus is on input generation
- Goal is to achieve significant code coverage
- Exponential in nature (input/code)
Testing model

- Search for buffer overflows
  - Dowser focuses on complex loops
  - Other approaches for simple pointer computation
- Source code available: Typical in testing
- Existing test inputs to reach every complex loop
Example

- Nginx web server, buffer overflow in URI parser
- Application too complex for traditional tools
- Complete code coverage may not even the trigger bug!

```c
while (p <= r->uri_end) // >300 lines of code
    switch (state)
    case sw_usual:  *u++ = ch; ...
    case sw_slash:  *u++ = ch; ...
    ...
    case sw_dot:    *u++ = ch; ...
        if (ch == '/') u--; ...
    case sw_dot_dot: *u++ = ch; ...
        if (ch == '/' ) u -= 4; ...
    ...
```
Testing with Dowser

- **Objective**: focus the testing effort around specific high-priority code fragments
- **Spot-checking** instead of looking at general picture
- Builds on symbolic execution, guided by in-depth analysis
- End-to-end solution starting from source-code:
Dowsing

Identify and rank loops based on bug probability
Dowsing in a nutshell

- Static analysis during compilation process
- Search for loops containing pointer dereference
- Analyze data-flow graph to infer complexity measure
Applied to real software

- Compare the ranking efficiency of the proposed heuristic to instruction counting and random order
- Buffer overflows reported in CVE for: nginx, ffmpeg, inspircd, libexif, poppler, snort, sendmail
Input tracking

Only sub-set of input is relevant for spot-checking
Infer relationships between inputs and candidate loops
Example input: HTTP Request

Long input with multiple tokens.

GET /long/path/file HTTP/1.1
Host: thisisthehost.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 1337
Highlight of HTTP Request

Only small part influences given loop

GET /long/path/file HTTP/1.1
Host: thisisthehost.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 1337

- Dynamic information flow tracking
- Track the influence of input on variables
- Can be performed at different granularities (details in paper)
Benefits of input reduction

- Symbolic execution is input driven in nature
- Provides implicit fine-grained modularization
- Enables symbolic execution for applications with large input
  - Conversion table in movie file for ffmpeg
  - Font description in PDF file for poppler
Bug search

Guide symbolic execution towards potential bug
Basics of symbolic execution

- "White-box fuzzing"
- Avoid generating input that replicates execution path
- Run-time feedback about possible execution paths
- Aimed at test-case generation
Snippet of symbolic execution

Constraint solver used to check for possible divergence

if (a < 4)
    do_something1;
else
    do_something2;

if (a < 2)
    do_something3;
else
    do_something4;

Input: 1 3 5
In practice input reduction was found to be insufficient

Large number of conditional branches still to be covered

Only some conditional statements are relevant

```c
if (a[i] == 'A')
    printf(...);
```

Focus on the branches influencing pointer value

**Value Coverage** search strategy
Value Coverage vs traditional search strategies

![Graph showing search time vs symbolic input bytes for Depth First Search, Code Coverage, and Value Coverage. The graph demonstrates the efficiency of Value Coverage in comparison to traditional search strategies.]
Details behind Value Coverage Search

- Only some execution paths are relevant to pointer arithmetic
- Learn the general behavior of conditionals using small inputs
- Result: 66% of conditionals eliminated
- Influence on example:

```c
while (p <= r->uri_end)
    switch (state)
    case sw_usual:  *u++ = ch;
    case sw_slash:  *u++ = ch;
    case sw_dot:    *u++ = ch;
        if (ch == '/') u--;  
    case sw_dot_dot: *u++ = ch;
        if (ch == '/') u -= 4;
```
Details behind Value Coverage Search

- Only some execution paths are relevant to pointer arithmetic
- Learn the general behavior of conditionals using small inputs
- Result: 66% of conditionals eliminated
- Influence on example:

```c
while (p <= r->uri_end)
    switch (state)
        case sw_usual:  *u++  = ch;

        case sw_dot:
            u--;  

        case sw_dot_dot:
            u -= 4;
```
### Evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th>LoC</th>
<th>Symbolic Input</th>
<th>Symbolic execution</th>
<th>Dowser</th>
</tr>
</thead>
<tbody>
<tr>
<td>nginx 0.6.32</td>
<td>66k</td>
<td>URI field</td>
<td>&gt; 8 h</td>
<td>&gt; 8 h</td>
</tr>
<tr>
<td>ffmpeg 0.5</td>
<td>300k</td>
<td>Huffman table</td>
<td>&gt; 8 h</td>
<td>&gt; 8 h</td>
</tr>
<tr>
<td>inspircd 1.1.22</td>
<td>45k</td>
<td>DNS response</td>
<td>200 sec</td>
<td>200 sec</td>
</tr>
<tr>
<td>poppler 0.15.0</td>
<td>120k</td>
<td>JPEG image</td>
<td>&gt; 8 h</td>
<td>&gt; 8 h</td>
</tr>
<tr>
<td>poppler 0.15.0</td>
<td>120k</td>
<td>Embedded font</td>
<td>&gt; 8 h</td>
<td>&gt; 8 h</td>
</tr>
<tr>
<td>libexif 0.6.20</td>
<td>10k</td>
<td>EXIF tag/length</td>
<td>&gt; 8 h</td>
<td>652 sec</td>
</tr>
<tr>
<td>libexif 0.6.20</td>
<td>10k</td>
<td>EXIF tag/length</td>
<td>&gt; 8 h</td>
<td>347 sec</td>
</tr>
<tr>
<td>libexif 0.6.20</td>
<td>10k</td>
<td>EXIF tag/length</td>
<td>&gt; 8 h</td>
<td>277 sec</td>
</tr>
<tr>
<td>snort 2.4.0</td>
<td>75k</td>
<td>UDP packet</td>
<td>&gt; 8 h</td>
<td>&gt; 8 h</td>
</tr>
</tbody>
</table>

**Table:** Bugs detected with Dowser.
Conclusions

- End-to-end solution for guided symbolic execution
- The spot-check approach enables focused search
- Built-in prioritization mechanism to optimize testing effort
- Heuristics geared towards buffer overflow type bugs
- Dowser shows scalability beyond traditional tools