Offset-Aware Mutation based Fuzzing for Buffer Overflow Vulnerabilities: Few Preliminary Results

Sanjay Rawat, Laurent Mounier

Verimag, University of Grenoble
Grenoble, France

Second International Workshop on Security Testing, Berlin
March 25, 2011
1 Overview
1 Overview

2 Tainted Path Generation and Code Instrumentation
   - Tainted Dependency Sequence (TDS)
1. Overview

2. Tainted Path Generation and Code Instrumentation
   - Tainted Dependency Sequence (TDS)

3. Evolutionary Algorithm (EA)
   - Why EA?
   - A Typical EA
1. Overview

2. Tainted Path Generation and Code Instrumentation
   - Tainted Dependency Sequence (TDS)

3. Evolutionary Algorithm (EA)
   - Why EA?
   - A Typical EA

4. String Inputs Generation
   - Input Generation
   - Crossover and Mutation
Outline

1. Overview
2. Tainted Path Generation and Code Instrumentation
   - Tainted Dependency Sequence (TDS)
3. Evolutionary Algorithm (EA)
   - Why EA?
   - A Typical EA
4. String Inputs Generation
   - Input Generation
   - Crossover and Mutation
5. Improved Mutation: Offset-aware Mutation
   - Again the maze analogy
   - Pseudo-code of the Algorithm
1. Overview

2. Tainted Path Generation and Code Instrumentation
   - Tainted Dependency Sequence (TDS)

3. Evolutionary Algorithm (EA)
   - Why EA?
   - A Typical EA

4. String Inputs Generation
   - Input Generation
   - Crossover and Mutation

5. Improved Mutation: Offset-aware Mutation
   - Again the maze analogy
   - Pseudo-code of the Algorithm

6. Experimental Results
   - Experimental Results
1. Overview

2. Tainted Path Generation and Code Instrumentation
   - Tainted Dependency Sequence (TDS)

3. Evolutionary Algorithm (EA)
   - Why EA?
   - A Typical EA

4. String Inputs Generation
   - Input Generation
   - Crossover and Mutation

5. Improved Mutation: Offset-aware Mutation
   - Again the maze analogy
   - Pseudo-code of the Algorithm

6. Experimental Results
   - Experimental Results

7. Conclusions and Future Work
   - Conclusions and Future Work
void buildfname(char *gecos, char *login, char *buf) {
    /* gecos - tainted; login - tainted; buf - untainted */
    char bp = buf; p = gecos;
    char* p;
    /* fill in buffer */
    while (*p != '\0' && *p != ',' && *p != ';' && *p != '%') {
        if (*p == '&') {
            (void) strcpy(bp, login); /* BAD */
            *bp = toupper(*bp);
            while (*bp != '\0')
                bp++;
        } else {
            *bp++ = *p;
        }
        p++;
    }
    *bp = '\0';
}
Overview of the Approach

- It is a light weight smart fuzzer with a focus on BO vulnerabilities.
Overview of the Approach

- It is a light weight *smart* fuzzer with a focus on BO vulnerabilities.
- Performs a static analysis of source code (C code) to calculate tainted path (slice: source to sink).
Overview of the Approach

- It is a light weight *smart* fuzzer with a focus on BO vulnerabilities.
- Performs a static analysis of source code (C code) to calculate tainted path (slice: source to sink).
- Makes use of evolutionary strategies to generate inputs (dynamic analysis by code instrumentation).
Overview of the Approach

- It is a light weight *smart* fuzzer with a focus on BO vulnerabilities.
- Performs a static analysis of source code (C code) to calculate tainted path (slice: source to sink).
- Makes use of evolutionary strategies to generate inputs (dynamic analysis by code instrumentation).
- Provides information on exploitability of the vulnerability (by inspecting stack).
Overview of the Approach

- It is a light weight *smart* fuzzer with a focus on BO vulnerabilities.
- Performs a static analysis of source code (C code) to calculate tainted path (slice: source to sink).
- Makes use of evolutionary strategies to generate inputs (dynamic analysis by code instrumentation).
- Provides information on exploitability of the vulnerability (by inspecting stack).
- Consider an analogy...
Maze Analogy
Maze Analogy
Maze Analogy
Maze Analogy
Maze Analogy
Maze Analogy
Maze Analogy
Maze Analogy
Maze Analogy
Maze Analogy
Maze Analogy

Tainted path generation
Maze Analogy

Overview

An Analogy

Input (string) generation

Tainted path generation
What is a tainted path (slice)?

```
1: gecos
2: login
6: bp = buf
8: p = gecos
9: while(*p != 0 && *p != ','
   && *p != ';' && p != '%')
23: *bp = '\0'
13: if(*p == '&')
19: bp++
20: *bp = *p
14: strcpy(bp, login)
15:*bp=toupper(*bp)
16: while(*bp != '\0')
11: p++
17: bp++
```
What is a tainted path (slice)?

```
1: gecos
2: login
6: bp = buf
8: p = gecos
9: while(*p != 0 && *p != ',' && *p != ';' && p != '%')
23: *bp = '\0'
13: if(*p == '&')
19: bp++
20: *bp = *p
14: strcpy(bp, login)
15: *bp = toupper(*bp)
16: while(*bp != '\0')
11: p++
17: bp++
```
What is a tainted path (slice)?

1: gecos

2: login

6: bp = buf

8: p = gecos

9: while(*p != 0 && *p != '\',
    && *p != ';')
    &
    && p != '％')

23: *bp = '0'

13: if(*p == '＆')

19: bp++

20: *bp = *p

14: strcpy(bp, login)

15:*bp=toupper(*bp)

16: while(*bp != '0')

11: p++

17: bp++
What is a tainted path (slice)?

1: gecos
2: login
6: bp = buf
8: p = gecos
9: while(*p != 0 && *p != ',' && *p != ';' && p != '%')

23: *bp = '\0'
13: if(*p == '&')
19: bp++
20: *bp = *p
14: strcpy(bp, login)
15: *bp = toupper(*bp)
16: while(*bp != '\0')
11: p++
17: bp++
What is a tainted path (slice)?
What is a tainted path (slice)?

1: gecos

2: login

6: bp = buf

8: p = gecos

9: while(*p != 0 && *p != ',' && *p != ';' && *p != '%')

13: if(*p == '&')

14: strcpy(bp, login)

15: *bp = toupper(*bp)

16: while(*bp != '\0')

11: p++

17: bp++

20: *bp = *p

19: bp++

23: *bp = '\0'
What is a tainted path (slice)?

```c
1: gecos
2: login
6: bp = buf
8: p = gecos
9: while(*p != 0 && *p != ',' && *p != ';' && p != '%')
23: *bp = '\0'
13: if(*p == '&')
14: strcpy(bp, login)
15: *bp=toupper(*bp)
16: while(*bp != '\0')
11: p++
17: bp++
19: bp++
20: *bp = *p
```
What is a tainted path (slice)?

1: gecos
2: login
6: bp = buf
8: p = gecos
9: while(*p != 0 && *p != ',' && *p != ';' && p != '%')

23: *bp = '0'
13: if(*p == '&')
19: bp++
20: *bp = *p

14: strcpy(bp, login)
15: *bp = toupper(*bp)
16: while(*bp != '0')
17: bp++
Tainted Path Generation and Code Instrumentation

- Makes use of a previous study on *Taint Dependency Sequences* (TDS) [4] to choose the paths to be exercised by the fuzzer.
- TDSs identify a sequence of instructions whose execution can be influenced by user inputs.
Taint Dependency Sequences (TDS) [4] to choose the paths to be exercised by the fuzzer.

TDSs identify a sequence of instructions whose execution can be influenced by user inputs.

Each TDS $t = < l_1, l_2, \ldots, l_n >$ associated to a variable $v$ is a sequence of program locations $l_i$ that a program execution path should traverse in order to reach $l_n$ with an input-dependent value assigned to $v$. 

Tainted Path Generation and Code Instrumentation

- Makes use of a previous study on *Taint Dependency Sequences* (TDS) [4] to choose the paths to be exercised by the fuzzer.
- TDSs identify a sequence of instructions whose execution can be influenced by user inputs.
- Each TDS $t = < l_1, l_2, \ldots, l_n >$ associated to a variable $v$ is a sequence of program locations $l_i$ that a program execution path should traverse in order to reach $l_n$ with an input-dependent value assigned to $v$. 
Tainted Path Generation and Code Instrumentation

- Makes use of a previous study on *Taint Dependency Sequences* (TDS) [4] to choose the paths to be exercised by the fuzzer.
- TDSs identify a sequence of instructions whose execution can be influenced by user inputs.
- Each TDS $t = \langle l_1, l_2, \ldots, l_n \rangle$ associated to a variable $v$ is a sequence of program locations $l_i$ that a program execution path should traverse in order to reach $l_n$ with an input-dependent value assigned to $v$.
- Based on the labels $\langle l_1, l_2, \ldots, l_n \rangle$, the source code is *instrumented* at $l_i$th line and compiled binary is used for further analysis.
Evolutionary Algorithm (EA)

**Objective:** Generate input that produces an execution sequence corresponding to the chosen path (TDS).

- *Smart* fuzzing can be considered as a *search problem*.
- We make use of *evolutionary strategies* (ES) to generate candidate inputs.
A Typical EA

Algorithm 1 Pseudo-code of a typical evolutionary algorithm

```
INITIALIZE population with random candidates
repeat
   SELECT parents
   RECOMBINE parents to generate children (crossover)
   MUTATE offsprings
   EVALUATE new candidates with fitness function
   SELECT fittest candidates for the next population
until TERMINATION CONDITION is met
return BEST Solution, if found
```

ES differs from GA mainly because of the introduction of strategies parameters, i.e. each individual $x$ is mutated to $x'$ by performing following operation:

$$x' = x + N(0, \sigma)$$
Initial Population

- Considers string inputs.
- Regular expression is used to generate initial set of inputs.
- We start with a set of inputs $I := \langle i_1, i_2, \ldots, i_m \rangle$.

We construct a frequency matrix $freq = \begin{pmatrix} f_{ij} \end{pmatrix}$, where $f_{ij}$ is the frequency of $l_j$ for the input $i \in I$. This matrix is used later while evaluating fitness.
Fitness Function:

- For a given TDS $t = \langle l_1, l_2, \ldots, l_n \rangle$, fitness value of each $i \in I$ (set of inputs), we define fitness function $F_i$ as follows:

$$F_i = \sum_{j=1}^{n} w_j \times f_{ij}$$  \hspace{1cm} (1)

where $w_j \in W$ (set of weights) corresponds to weight associated with $l_j \in t$ and $f_{ij}$ is the frequency of $l_j$ for $i \in I$.

- $F_i$ is affected by the structure of the program i.e. rare statements, nested conditional statements etc.
Fitness Function: Effect of nested statements
Fitness Function: Effect of nested statements

Frequency Spectrum Analysis

A careful analysis reveals a lot about execution trace of the program by means of frequencies corresponding to $t_j$s (Thomas Ball et. al. [2][3][6]).
String generation: an example

if s == 'A'

if s == 'B'

...
String generation: an example

```plaintext
if s == 'A'

if s == 'B'

s1 = abc A xy
```
String generation: an example

- If $s = 'A'$
  - $s1 = abcAxy$
- If $s = 'B'$
  - $s2 = xyBjka$
String generation: an example

- If \( s = 'A' \):
  - \( s1 = abcAxy \)
  - \( s2 = xyBjka \)
  - Desired string \( = f(s1 + s2) \supseteq \{A, B\} \)

- If \( s = 'B' \)
String generation: an example

- \(s1 = \text{abcAxy}\)
- \(s2 = \text{xyBjka}\)
- Desired string = \(f(s1 + s2) \supseteq \{A, B\}\)
- Set-theory is used for such a purpose, especially *Symmetric Difference*. 
String generation: an example

- **s1** = abc\textcolor{red}{A}\textcolor{green}{xy}
- **s2** = xy\textcolor{red}{B}jka
- Desired string = \( f(s1 + s2) \supseteq \{ A, B \} \)
- Set-theory is used for such a purpose, especially \textit{Symmetric Difference}.
- For characters (constraints) learning, we maintain a set \( M \) which is used to generate new strings.
Crossover and Mutation

Crossover

Crossover is performed by interchanging substrings of parent string (using n-point crossover).
Crossover and Mutation

**Crossover**

Crossover is performed by interchanging substrings of parent string (using n-point crossover).

**Mutation**

- Mutation is based on *set-theoretic* definitions. We try to learn from bad inputs too!!!
- This is the main step which is responsible for learning (light) constraints automatically.
Crossover and Mutation

Crossover

Crossover is performed by interchanging substrings of parent string (using n-point crossover).

Mutation

- Mutation is based on set-theoretic definitions. We try to learn from bad inputs too!!!
- This is the main step which is responsible for learning (light) constraints automatically.
- Under mutation, each input is manipulated either by adding or deleting characters from a set $M$ (at pseudo-random positions!).
Maze example, again!
Maze example, again!
Maze example, again!
Offset-Aware Mutation based Fuzzing for Buffer Overflow Vulnerabilities: Few Preliminary Results

- Improved Mutation: Offset-aware Mutation
- Again the maze analogy

Maze example, again!
Maze example, again!
Maze example, again!
Maze example, again!

\[ s_1 = \text{abcAxy}; s_2 = \text{xyBjka} \]
Maze example, again!

- $s_1 = abc \text{A} xy; s_2 = xy \text{B} jka$
- $M = s_1 \triangle s_1 = bcABjk$
Offset-Aware Mutation based Fuzzing for Buffer Overflow Vulnerabilities: Few Preliminary Results

- Improved Mutation: Offset-aware Mutation
- Again the maze analogy

Maze example, again!

\[
\begin{align*}
  s_1 &= \text{abc} \text{A} \text{xy}; \\
  s_2 &= \text{xy} \text{B} \text{jka} \\
  M &= s_1 \triangle s_1 = \text{bcABjk} \\
  \text{offset-mutation}(s_1, M, 4)
\end{align*}
\]
Maze example, again!

- $s_1 = \text{abc}\ A\ xy; \ s_2 = \text{xy}\ B\ jka$
- $M = s_1 \triangle s_1 = \text{bcABjk}$
- offset-mutation($s_1, M, 4$)
- $\text{abcAbc}...; \ \text{abcAk}...; \ \text{abcAB} ...$
Algorithm 2: Pseudo-code of the proposed EA based approach

INITIALIZE population $I$ with $\lambda$ random candidates
TERMINATION CONDITION: program crashed or 1000 iterations are over
$M \leftarrow \{\}$

repeat
    EXECUTE program with $I$, RECORD offset
    CALCULATE fitness
    CALCULATE $diff_1$, $diff_2$ and $notImp$ (till 2 generations)
    UPDATE $M$ as $M \leftarrow M \cup (diff_1 \triangle diff_2)$
    if $N(0, \sigma) > large\_number$ then
        $M \leftarrow M \setminus notImp$
    end if
    if $N(0, \sigma) > large\_number$ then
        $M \leftarrow M \cup new\_chars$
    end if
    SELECT $\mu$ fittest candidates for the next population
    for each of $\mu$ candidates do
        if probability_mutation then
            for $i = 1$ to $\lambda/\mu$ do
                MUTATE candidate using $M$ and offset
            end for
        else
            for $i = 1$ to $\lambda/\mu$ do
                SELECT two parents randomly
                RECOMBINE parents to generate child
            end for
        end if
    end for
    UPDATE $I$ with newly generated candidates
until TERMINATION CONDITION is met
Offset-Aware Mutation based Fuzzing for Buffer Overflow Vulnerabilities: Few Preliminary Results

Improved Mutation: Offset-aware Mutation

High Level Flow Diagram

High Level Flow Diagram

- program slice e.g. tainted data path of the c program
- source instrumentation based on program slice
- fitness function
crossover/mutation selection
- execution trace
- Evolutionary algorithm module
- compiled binary
- new generated inputs
- malicious input
- after-crash data collection e.g. input length stack registers...
- Debugger

Evolutionary algorithm module
Experimental Results

- We experiment on Verisec dataset [1]. We choose two (representative!) programs which do string manipulations on inputs.

- We compare results of our algorithm with blind fuzzing approach and a GA based approach without offset based mutation.

**Table:** Experimental Results. Legend: A1 (resp. A2)- evolutionary strategy with (resp. without) offset-aware mutation, A3- random fuzzing

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Application</th>
<th>Name</th>
<th>Constraints</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sendmail</td>
<td>mime_fromqp</td>
<td>‘=n’</td>
<td>370</td>
<td>154</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>edbrowse</td>
<td>ftps</td>
<td>‘--’</td>
<td>*</td>
<td>*</td>
<td>290</td>
</tr>
</tbody>
</table>
Conclusions

This work:

- presents preliminary results on our ongoing work [5].
- approximates constraints (characters) on inputs based on execution trace.
- proposes a new mutation technique for generating strings based inputs, called \textit{offset-aware mutation}.
- obtains experimental results which are significantly better than the previous results.
Future work

A lot!!

- Static analysis on executables to generate tainted path.
- During static analysis, learn dependency on inputs and critical nodes in tainted path to enhance mutation.
- Do a rigorous analysis on the crash to infer exploitability of the vulnerability.
Verisec.

Thomas Ball.
What's in a region? or computing control dependence regions in near-linear time for reducible control flow.
*ACM Letters on Programming Languages and Systems, 2(1-4):1–16, 1993.*

Thomas Ball.
The concept of dynamic analysis.

Dumitru Ceara, Laurent Mounier, and Marie-Laure Potet.
Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences.

Sanjay Rawat and Laurent Mounier.
An evolutionary computing approach for hunting buffer overflow vulnerabilities: A case of aiming in dim light.

Thomas W. Reps.
The use of program profiling for software testing.
Thank You!!
&

Sanjay.Rawat@imag.fr
Laurent.Mounier@imag.fr
thanks