Getting started with vulnerability discovery using Machine Learning

Gustavo Grieco
Hack In The Box Lab 2016
CIFASIS - CONICET / VERIMAG
Motivation
program + input $\rightarrow$ security issue?
They are expensive and we want to discover more vulnerabilities, using less resources (time/money).

Program Behaviors
We should focus on programs and inputs that could do something “bad”.
They are expensive and we want to discover more vulnerabilities, using less resources (time/money).

**Program Behaviors**

We should focus on programs and inputs that could do something “bad”.
Overview and Applications

How?

program and inputs $\rightarrow$ traces $\rightarrow$ machine learning $\rightarrow$ program behaviors

Why?

Vulnerability Detection: $\rightarrow$ extrapolation and prediction of vulnerable inputs.

Seed selection: $\rightarrow$ reduction of the set of inputs to “cover” all the program behaviors.
Programs, traces and behaviors
Let’s start with..

1. A binary program: **gifflip**:
   
   A program to flip (mirror) GIF file along X or Y axes, or rotate the GIF file 90 degrees to the left or to the right.

2. A large number of inputs: hundreds or thousands gif files.
Graphics Interchange Format

The input space of *gifflip* can be specified using the following structure:

Extracting this information using the binary and some inputs is a very challenging task!
where similar gif structures are close together.
Input File Space

where similar files are close together.
Trace Space

where similar traces are close together.

Clusters of traces represent a program behavior
where similar traces are close together.

Clusters of traces represent a **program behavior**
What are traces anyway?
Developed by Intel and used in many projects.
Every instruction and its operands are recorded.
Traces are sequences of instructions with all its operands values.
- Developed by Google but only used in AFL.
- Every jump in a binary is instrumented to have a label using afl-gcc/g++ or QEMU.
- Traces are sequences of labels representing transitions between basic blocks.
- For instance: 1 − 3 − 4 − 3 − 4 − 2
Every call to the standard C library is captured and augmented with dynamic information of its arguments using ptrace.

Traces are sequences of events corresponding to such calls.
Remember:
Machine Learning algorithms cannot deal with values like strings, pointers, integers, that's why replace them with meaningful labels.
Traces Representations

Unfortunately..

Traces needs to be normalized since longer traces are likely to contain more information than short ones.

- Bag of words: a trace is represented as the bag (multiset) of its events, disregarding grammar and even event order but keeping multiplicity.
- Subtraces of maximum length: a trace is represented as the set of subtraces sampled from the original (long) trace.
For instance

Remember:
A trace and its representation can be completely different things.
Visual Explorations of Trace Space
• Parsing of simple regex expressions (pcre).
• Detection of file types using file (libmagic).
• Display of information of PNG files from pnginfo (libpng 1.2)
regex (pcre) - AFL - BOW
file (libmagic) - VD - BOW
Vulnerability Prediction
Overview

Vulnerability Detection Procedure

testcase output

dataset

✓ | ✗
Overview

Vulnerability Detection Procedure

testcase output

dataset

✓ | ✗

VDiscover

features train target

20
Overview

Vulnerability Detection Procedure

new testcase output ✓ | ✗

VDiscover features prediction

features prediction

20
1. **No source-code required**: Our features are extracted using static and dynamic analysis for binaries programs, allowing our technique to be used in proprietary operating systems.

2. **Automation**: No human intervention is needed to select features to predict, we focused only on feature sets that can be extracted and selected automatically, given a large enough dataset.

3. **Scalability**: Since we want to focus on scalable techniques, we only use lightweight static and dynamic analysis. Costly operations like instruction per instruction reasoning are avoided by design.
xa is a small cross-assembler for the 65xx series of 8-bit processors (i.e. Commodore 64). We can easily crash it:

```
$ gdb --args env -i /usr/bin/xa '\bo@e\0' '@o' '-o'
...
Program received signal SIGSEGV, Segmentation fault.
(gdb) x/i
$eip => 0x8049788: movzbl (%ecx),%eax
(gdb) info registers
eax 0x0  0
cecx 0x0  0
...
```

**Question:**

It is just a NULL pointer dereference, should we spend our resources trying to fuzz this test case?
Smashing the stack..

```
$ gdb --args env -i /usr/bin/xa \bo@e\0\ 'o' 'AAAA...AAAA-o'

Copyright (C) 1989-2009 Andre Fachat, Jolse Maginnis, David Weinehall and Cameron Kaiser.

 Couldn’t open source file '@o’!
 Couldn’t open source file ’o@’!

 *** buffer overflow detected ***: /usr/bin/xa terminated

...

vulnerability detection procedure

We used a simple fuzzer producing 10,000 mutation for each test case.
Smashing the stack..

$ gdb --args env -i /usr/bin/xa '\bo@e\0' '@o' 'AAAA...AAAA-o'

Copyright (C) 1989-2009 Andre Fachat, Jolse Maginnis, David Weinehall
and Cameron Kaiser.

o@e:line 1: 1000: Syntax error
o@e:line 2: 1000: Syntax error
o@e:line 3: 1000: Syntax error

Couldn’t open source file '@o'!
Couldn’t open source file 'o@'!

*** buffer overflow detected ***: /usr/bin/xa terminated

...

vulnerability detection procedure

We used a simple fuzzer producing 10,000 mutation for each test case.
• A total of 1039 bugs in 496 packages.
• Every bug is packed with a crash report and the required inputs to reproduce it.
For instance

vulnerability detection procedure

Around 8% was found vulnerable to interesting memory corruptions.
Model training/inference
Training and Testing
<table>
<thead>
<tr>
<th></th>
<th>Flagged</th>
<th>Not Flagged</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flagged</td>
<td>55%</td>
<td>17%</td>
</tr>
<tr>
<td>Not Flagged</td>
<td>45%</td>
<td>83%</td>
</tr>
</tbody>
</table>

These results are obtained using Random Forest (scikit-learn) in 1-3 grams representation.

Not flagged cases are slower, because the fuzzer will not find vulnerabilities.
<table>
<thead>
<tr>
<th></th>
<th>Flagged</th>
<th>Not Flagged</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flagged</td>
<td>55%</td>
<td>17%</td>
</tr>
<tr>
<td>Not Flagged</td>
<td>45%</td>
<td>83%</td>
</tr>
</tbody>
</table>

These results are obtained using Random Forest (scikit-learn) in 1-3 grams representation.

Not flagged cases are slower, because the fuzzer will not find vulnerabilities.
Seed Selection for fuzzing [WIP]
Overview

Seed selection in mutational fuzzing for a program P:

1. Collect a very large number of input files (seeds).
2. Select a subset of seeds according to some criteria.
3. Start fuzzing with selected seeds checking if P fails.

Observation:
Seed selection should avoid redundancy in the initial selection.
Collecting seeds


- HTML and CSS files obtained randomly sampling from the first 10k most visited pages (Alexa)
- Files are randomly cut in fragments of certain max sizes (128b, 1k)
- All kinds of languages, encoding and types of websites were retrieved!
Targets

- libxml2 (2.7.2): "xmllint –html @@"
- w3m (0.5.3): "w3m -dump -T text/html @@"
- gumbo-parser (0.9.0): "clean_text @@"
- html2text (1.3.2a): "html2text @@"
- htmlcxx (0.85): "htmlcxx @@"
- htmldoc (1.8.27): "htmldoc @@"
- html-xml-utils (6.5): "hxnormalize @@"
- tidy (20091223cvs): "tidy @@"

All these programs were recompiled using ASAN in order to detect invalid memory reads/writes.
Targets

- libxml2 (2.7.2): “xmllint –html @@”
- w3m (0.5.3): “w3m -dump -T text/html @@”
- gumbo-parser (0.9.0): “clean_text @@”
- html2text (1.3.2a): “html2text @@”
- htmlcxx (0.85): “htmlcxx @@”
- htmldoc (1.8.27): “htmldoc @@”
- html-xml-utils (6.5): “hxnormalize @@”
- tidy (20091223cvs): “tidy @@”

All these programs were recompiled using ASAN in order to detect invalid memory reads/writes.
General settings:

- AFL 1.94b was used instrumenting the target programs (recompiled using afl-gcc/g++).
- For each experiment, we fuzzed at least 48hs in a dedicated core using “quick and dirty” mode (-d).

Selecting seeds:

- AFL includes its own seed selection (called corpus minimization) based on afl-traces and implemented in afl-cmin.
- VDiscover includes a pattern based seed selection algorithm.
Fuzzing time!

General settings:

- AFL 1.94b was used instrumenting the target programs (recompiled using afl-gcc/g++).
- For each experiment, we fuzzed at least 48hs in a dedicated core using “quick and dirty” mode (-d).

Selecting seeds:

- AFL includes its own seed selection (called corpus minimization) based on afl-traces and implemented in afl-cmin.
- VDiscover includes a pattern based seed selection algorithm.
From traces to vectors

trace extraction

\$ \text{vd -i seeds -o program.traces -c "/program @@"}

⇓

complete trace

\[ \ldots \text{read(Num32B8,HPtr32,Num32B24)} \quad \text{free(HPtr32)} \quad \text{calloc(Num32B8,Num32B24)} \quad \ldots \]

⇓

fixed size subtrace

\[
\begin{array}{cccc}
\text{read(Num32B8,HPtr32,Num32B24)} & \text{free(HPtr32)} & \text{calloc(Num32B8,Num32B24)}
\end{array}
\]

⇓

fixed size real vector

\[
\begin{pmatrix}
0.12 \\
0.31 \\
0.06 \\
0.91 \\
0.42
\end{pmatrix}
\]
libxml2 traces and results
libxml2 traces and results

Paths explored using AFL
libxml2 traces and results

Crashes discovered using AFL
libxml2 traces and results

Unique crashes discovered using AFL
Give me a break!
Workshop Time!
Overview

1. Installing VDiscover.
2. Creating test cases and extracting traces.
3. Trace visualization and seed selection.
4. Training and predicting with ZZUF dataset.
Installing VDiscover

Make sure you install a recent version, not the ancient version from the Ubuntu repositories (you can download packages here)

1. Setup a VM:

   ```
   vagrant init ubuntu/trusty32
   vagrant up --provider virtualbox
   vagrant ssh -- -X
   ```

2. Take some minutes to update and install basic stuff (git, python-setuptools, python-matplotlib, python-scipy ..)

   ```
   git clone https://github.com/CIFASIS/vdiscover-workshop
   git clone https://github.com/CIFASIS/VDiscover
   cd VDiscover
   ./setup.py install --user
   ```

   (don’t forget to append “PATH=$PATH:~/local/bin” to your .bashrc)
VDiscover

- Open source (GPL3) and available here: http://www.vdiscover.org/
- Written in Python 2:
  - python-ptrace
  - scikit-learn (and dependencies)
- Composed by:
  - tcreator: test case creation
  - fextractor: feature extraction
  - vpredictor: trainer and predictor
  - vd: a high level script to save time extracting data
- Trace should be collected in x86 (because i’m lazy!)
Setting up a test case

$ printf '<b>Hello!</b>' > test.html

$ tcreator --name test-html --cmd "/usr/bin/html2text file:$(pwd)/test.html" out

Workshop Time!

Experiment adding and removing arguments and files to check how test cases are created.
$ printf \'<b>Hello!</b>\' > test.html
$ tcreator --name test-html --cmd "/usr/bin/html2text file:$(pwd)/test.html" out

**Workshop Time!**

Experiment adding and removing arguments and files to check how test cases are created.
Collecting my first trace (1)

$ fextractor --dynamic out/test-html/ > trace1.csv
$ cat trace1.csv

out/test-html/ strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32
strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32
strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32
strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32
strcmp:0=GxPtr32 strcmp:1=GxPtr32 ...

Workshop Time!
Take a few minutes to extract traces from other programs and how to include/exclude events from different modules (–inc-mods/–ign-mods)
Collecting my first trace (1)

$ fextractor --dynamic out/test-html/ > trace1.csv
$ cat trace1.csv

out/test-html/ strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32 ...

Workshop Time!
Take a few minutes to extract traces from other programs and how to include/exclude events from different modules (–inc-mods/–ign-mods)
Collecting my first trace (2)

$ printf ‘<baaa>Bye!’ > test.html
$ fextractor --dynamic out/test-html/ > trace2.csv
$ cat trace2.csv

out/test-html/ strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32
  strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32
  strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32
  strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32
  strcmp:0=GxPtr32 strcmp:1=GxPtr32 ..

It looks exactly the same!!

.. but in fact, they are not. Later, we are going to show how to easily visualize traces..
Visualizing test cases

- Collecting data:
  
  ```
  $ tar -xf bmpsuite-2.4.tar.gz
  $ vd -m netpbm -i bmps "/usr/bin/bmptopnm @@" -o bmptopnm-traces.csv
  ```

- Clustering using bag of words and display:
  
  ```
  $ vpredictor --cluster-bow --dynamic bmptopnm-traces.csv
  ```

- After the clustering, a file (bmptopnm-traces.csv.clusters) will be written.

**Exercise:**

Using the source code of bmptopnm, try to understand why test cases are clustered like this.
Seed Selection

$ tseeder bmptopnm-traces.csv.clusters seeds
Copying seeds..
bmps/badbitcount.bmp
bmps/pal4gs.bmp
bmps/rgba32-61754.bmp
bmps/pal4.bmp
bmps/shortfile.bmp
bmps/baddens2.bmp

Question
You can adjust how many test cases per cluster are selected using -n.
A detailed explanation of this dataset is available here:
http://www.vdiscover.org/OS-fuzzing.html
ZZUF dataset (2)

- cmds.csv.gz: 64k command-line to fuzz
- traces.csv.gz: sampled and balanced traces ready to be trained and tested
- zzuf.csv.gz: output from zzuf after fuzzing

To split the data in train and test sets:

```
$ ./split.py dataset/traces.csv.gz 42
```
Training and testing a bug predictor

- Training:
  
  $ vpredictor --dynamic --train-rf data/42/train.csv --out-file model.pklz

- Testing:
  
  $ vpredictor --test --dynamic --model model.pklz data/42/test.csv --out-file predicted.out

...  
Accuracy per class: 0.72 0.78

Average accuracy: 0.75