TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection

Tielei Wang¹, Tao Wei¹, Guofei Gu², Wei Zou¹
¹Peking University, China
²Texas A&M University, US

Outline
- Introduction
  - Background
  - Motivation
- TaintScope
  - Intuition
  - System Design
  - Evaluation
- Conclusion

Fuzzing/Fuzz Testing
- Feed target applications with malformed inputs e.g., invalid, unexpected, or random test cases
  - Proven to be remarkably successful
  - E.g., randomly mutate well-formed inputs and runs the target application with the "mutations"

Fuzzing is great
In the best case, malformed inputs will explore different program paths, and trigger security vulnerabilities

However...

A quick example
```c
void decode_image(FILE* fd){
  ... 1
  int length = get_length(fd);
  int recomputed_chksum = checksum(fd, length);
  int chksum_in_file = get_checksum(fd); //line 6 is used to check the integrity of inputs
  if(chksum_in_file != recomputed_chksum) 2
    error();
  int Width = get_width(fd);
  int Height = get_height(fd);
  int size = Width*Height*sizeof(int); //integer overflow
  int* p = malloc(size);
  ... 3
}
```

Checksum: the bottleneck
Checksum is a common way to test the integrity of input data
Most mutations are blocked at the checksum test point
Our motivation

- Penetrate checksum checks!

Intuition

- Disable checksum checks by control flow alteration
- Fuzz the modified program
- Repair the checksum fields in malformed inputs that can crash the modified program

Key Questions

- Q1: How to locate the checksum test instructions in a binary program?
- Q2: How to effectively and efficiently fuzz for security vulnerability detection?
- Q3: How to generate the correct checksum value for the invalid inputs that can crash the modified program?

A1: Locate the checksum test instruction

Key Observation 1

**Checksum is usually used to protect a large number of input bytes**

```
if(checksum(Data) != Chksum)
```  

- Based on fine-grained taint analysis, we first find the conditional jump instructions (e.g., jz, je) that depend on more than a certain number of input bytes
- Take these conditional jump instructions as candidates

Key Observation 2

**Well-formed inputs can pass the checksum test, but most malformed inputs cannot**

- We log the behaviors of candidate conditional jump instructions

TaintScope Overview

- Checksum Locator
- Directed Fuzzer
- Checksum Repairer
- Execution Monitor
- Reports
A1: Locate the checksum test instruction

Key Observation 2

Well-formed inputs can pass the checksum test, but most malformed inputs cannot

- We log the behaviors of candidate conditional jump instructions
  1. Run well-formed inputs, identify the always-taken and always-not-taken insts
  2. Run malformed inputs, also identify the always-taken and always-not-taken insts

A2: Effective and efficient fuzzing

- Blindly mutating will create huge amount of redundant test cases --- ineffective and inefficient

```c
1 void decode_image(FILE* fd){
2 ... 
6 if(chksum_in_file != recomputed_chksu
7 goto 8;
8 int Width = get_width(fd);
9 int Height = get_height(fd);
10 int size = Width*Height*sizeof(int);//integer overflow
11 int* p = malloc(size);
12 ... 
```

- Directed fuzzing: focus on modifying the "hot bytes" that refer to the input bytes flow into critical system/library calls
  - Memory allocation, string operation...

A3: Generate the correct checksum

- The classical solution is symbolic execution and constraint solving
  - Solving $\text{checksum}(D_{in}) = C_{in}$ is hard or impossible, if both $D_{in}$ and $C_{in}$ are symbolic values

- We use combined concrete/symbolic execution
  - Only leave the bytes in the checksum field as symbolic values
  - Collect and solve the trace constraints on $C_{in}$ when reaching the checksum test inst.
  - Note that:
    - $\text{checksum}(D_{in})$ is a runtime determinable constant value.
    - $C_{in}$ originates from the checksum field, but may be transformed, such as from hex/oct to dec number, from little-endian to big-endian.
Evaluation

- **Component evaluation**
  - E1: Whether TaintScope can locate checksum points and checksum fields?
  - E2: How many hot byte in a valid input?
  - E3: Whether TaintScope can generate a correct checksum field?

- **Overall evaluation**
  - E4: Whether TaintScope can detect previous unknown vulnerabilities in real-world applications?

### Evaluation 1: locate checksum points
- We test several common checksum algorithms, including CRC32, MD5, Adler32. TaintScope accurately located the check statements.

### Evaluation 2: identify hot bytes
- We measured the number of bytes could affect the size arguments in memory allocation functions.

### Evaluation 3: generate correct checksum fields
- We test malformed inputs in four kinds of file formats.
- TaintScope is able to generate correct checksum fields.

### Evaluation 4: 27 previous unknown vulns
- MS Paint
- Google Picasa
- Adobe Acrobat
- ImageMagick
- irfanview
- gstreamer
- Winamp
- XEmacs
- Amaya
- dillo
- wxWidgets
- PDFlib

Acknowledgments:
Microsoft kindly supplied the following for working with us to help protect customers:
- Deepen Fates of Cyber Security Technologies for reporting an issue described in MS10-042
- Dominique Creteau of Secure Software for reporting an issue described in MS10-051
- Brian L. Shive of Black Dog Security Labs for reporting an issue described in MS10-064
- NBS, working with Superscan 4.10 pentest suite, for reporting an issue described in MS10-065
- Cody Peace of Treadstone Labs for reporting an issue described in MS10-068

See the blog post [Secure Windows](https://docs.microsoft.com/en-us/security/windows/security/protection/security-policy-settings) for more details on security issues described in MS10-042.
**Evaluation 4: 27 previous unknown vulns**

<table>
<thead>
<tr>
<th>Package</th>
<th>Type</th>
<th>Value</th>
<th>Checksum-aware</th>
<th>Severity Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Paint</td>
<td>Mox</td>
<td>1</td>
<td>N</td>
<td>CVE-2010-2523</td>
</tr>
<tr>
<td>Google Chrome</td>
<td>Mox</td>
<td>1</td>
<td>N</td>
<td>CVE-2010-2523</td>
</tr>
<tr>
<td>Adobe Acrobat</td>
<td>Mox</td>
<td>1</td>
<td>N</td>
<td>CVE-2010-2523</td>
</tr>
<tr>
<td>Apache Abode</td>
<td>Mox</td>
<td>1</td>
<td>N</td>
<td>CVE-2010-2523</td>
</tr>
<tr>
<td>WebEx</td>
<td>Mox</td>
<td>1</td>
<td>N</td>
<td>CVE-2010-2523</td>
</tr>
<tr>
<td>Oracle</td>
<td>Mox</td>
<td>1</td>
<td>N</td>
<td>CVE-2010-2523</td>
</tr>
<tr>
<td>MySQL</td>
<td>Mox</td>
<td>1</td>
<td>N</td>
<td>CVE-2010-2523</td>
</tr>
<tr>
<td>PHP</td>
<td>Mox</td>
<td>1</td>
<td>N</td>
<td>CVE-2010-2523</td>
</tr>
<tr>
<td>Nginx</td>
<td>Mox</td>
<td>1</td>
<td>N</td>
<td>CVE-2010-2523</td>
</tr>
<tr>
<td>Fitis</td>
<td>Mox</td>
<td>1</td>
<td>N</td>
<td>CVE-2010-2523</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>Mox</td>
<td>1</td>
<td>N</td>
<td>CVE-2010-2523</td>
</tr>
<tr>
<td>TaintScope</td>
<td>Mox</td>
<td>1</td>
<td>N</td>
<td>CVE-2010-2523</td>
</tr>
</tbody>
</table>

**Conclusion**

- Checksum is a big challenge for fuzzing tools
- TaintScope can perform:
  - Directed fuzzing
    - Identify which bytes flow into system/library calls.
    - Dramatically reduce the mutation space.
  - Checksum-aware fuzzing
    - Disable checksum checks by control flow alternation.
    - Generate correct checksum fields in invalid inputs.
- TaintScope detected dozens of serious previous unknown vulnerabilities.

Thanks for your attention!