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

Tielei Wang Tao Wei Guofei Gu Wei Zou

March 12, 2014

Taintscope is:

- A Fuzzing tool
- Checksum-Aware
- Directed

Why a new fuzzing tool?

Fuzzing tools already exist. They can be sorted in two categories:

- Mutation based

Why a new fuzzing tool?

Fuzzing tools already exist. They can be sorted in two categories:

- Mutation based
  - Not very efficient

Why a new fuzzing tool?

Fuzzing tools already exist. They can be sorted in two categories:

- Mutation based
  - Not very efficient
  - Cannot generate valid input if a **checksum** mechanism is used

Why a new fuzzing tool?

Fuzzing tools already exist. They can be sorted in two categories:

- Mutation based
  - Not very efficient
  - Cannot generate valid input if a **checksum** mechanism is used
- Generation based

Why a new fuzzing tool?

Fuzzing tools already exist. They can be sorted in two categories:

- Mutation based
  - Not very efficient
  - Cannot generate valid input if a **checksum** mechanism is used
- Generation based
  - Better performances

Why a new fuzzing tool?

Fuzzing tools already exist. They can be sorted in two categories:

- Mutation based
  - Not very efficient
  - Cannot generate valid input if a **checksum** mechanism is used
- Generation based
  - Better performances
  - Often implies having input specification, or source code.
  - Tools exist to automatically get input format, but cannot reverse engineer **checksum** algorithms.

Why a new fuzzing tool?

Example of input using checksum:

int format
int fileSize
int width
int height
...
int checksum

```
void decode_input(File * f){
    int recomputed_checksum = checksum(f);
    int checksum_in_file = get_checksum(f);
    if (recomputed_checksum != checksum_in_file)
        exit();
    width = get_width(f);
    ...
```

Contributions

Taintscope offers several major contributions:

- Checksum-aware
  - Detect checksum test in tested program
  - Bypass checksum test when fuzz-testing
  - Reconstruct input with valid checksum

Contributions

Taintscope offers several major contributions:

- Checksum-aware
  - Detect checksum test in tested program
  - Bypass checksum test when fuzz-testing
  - Reconstruct input with valid checksum
- Directed
  - Reduces the space of parameters to mutate

Checksum-awareness

Checksum-aware fuzz-testing is done in 3 steps:

Checksum-awareness

Checksum-aware fuzz-testing is done in 3 steps:

1. Pre-processing: locate checksum check points in the program

Checksum-awareness

Checksum-aware fuzz-testing is done in 3 steps:

1. Pre-processing: locate checksum check points in the program
2. Fuzz-testing: mutate input without touching the checksum data

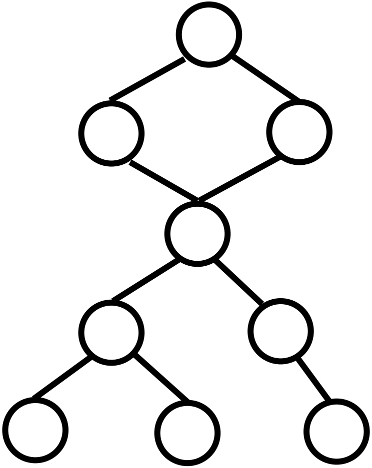
Checksum-awareness

Checksum-aware fuzz-testing is done in 3 steps:

1. Pre-processing: locate checksum check points in the program
2. Fuzz-testing: mutate input without touching the checksum data
3. Post-processing: for a crashing input, rebuild valid checksum

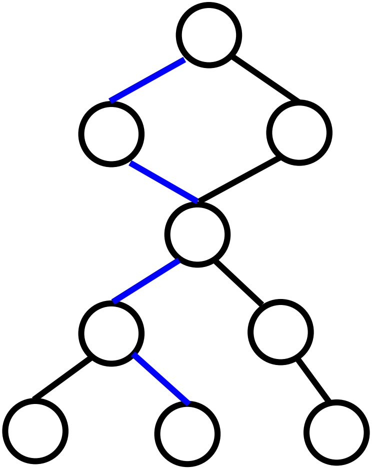
Checksum-awareness

How to locate checksum test in program?



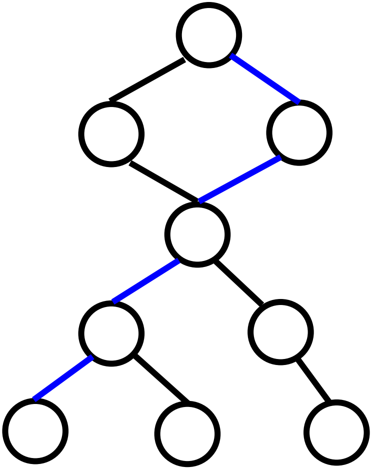
Checksum-awareness

How to locate checksum test in program?



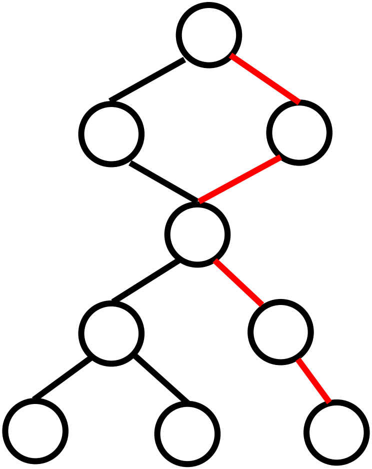
Checksum-awareness

How to locate checksum test in program?



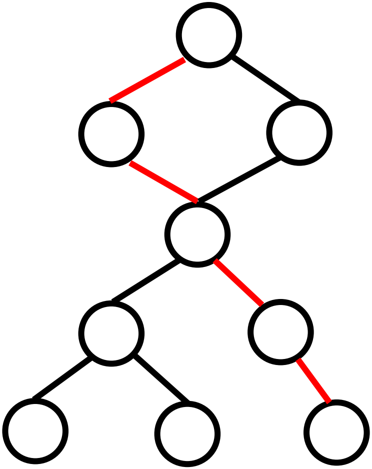
Checksum-awareness

How to locate checksum test in program?



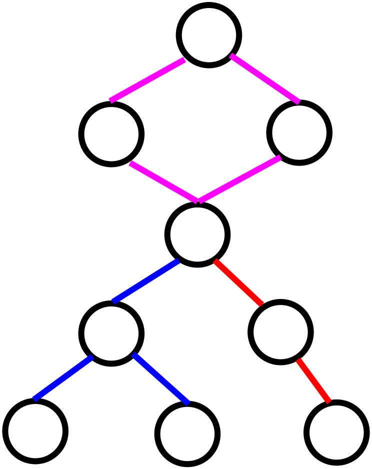
Checksum-awareness

How to locate checksum test in program?



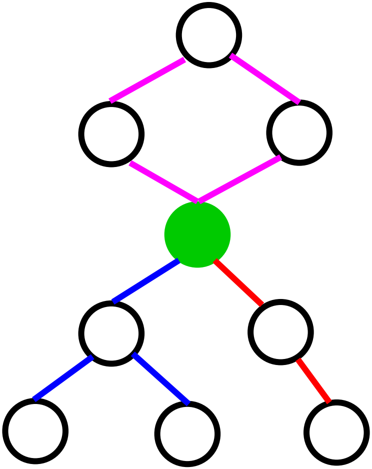
Checksum-awareness

How to locate checksum test in program?



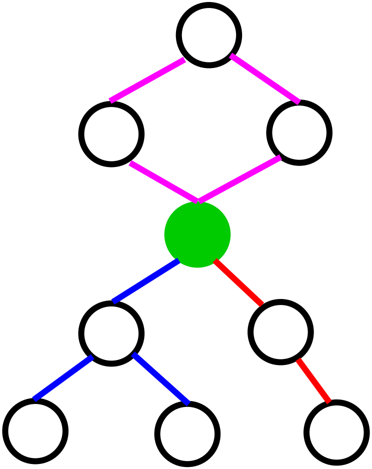
Checksum-awareness

How to locate checksum test in program?



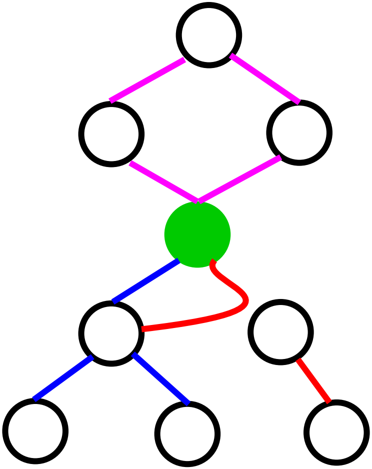
Checksum-awareness

How to fuzz-test knowing the checksum-point?



Checksum-awareness

How to fuzz-test knowing the checksum-point?



Checksum-awareness

Using the checksum locator it is possible to:

- Bypass checksum test by modifying the program
- Test input on the modified program to find crashing cases

Checksum-awareness

Using the checksum locator it is possible to:

- Bypass checksum test by modifying the program
- Test input on the modified program to find crashing cases

But how to use those inputs on the real program?

- Need to reconstruct valid checksum

Checksum-awareness

Using our previous example:

int format
int fileSize
int width
int height
...
int checksum

```
void decode_input(File * f){
    int recomputed_checksum = checksum(f);
    int checksum_in_file = get_checksum(f);
    if (recomputed_checksum != checksum_in_file)
        exit();
    width = get_width(f);
    ...
```

Checksum-awareness

Using our previous example:

int format
int fileSize
int width
int height
...
int checksum

```
void decode_input(File * f){
    int recomputed_checksum = checksum(f);
    int checksum_in_file = get_checksum(f);
    if (recomputed_checksum != checksum_in_file)
        exit();
    width = get_width(f);
    ...
```

Checksum-awareness

Why not use that checksum everytime instead of modifying the program?

- In practice, finding back the checksum is more complicated
- That step is too expensive to do it thousands of time

Checksum-awareness

So Taintscope is a checksum-aware fuzzing tool:

- Detects checksum tests
- Bypasses them for fuzz-testing
- Corrects input so they can work on original program

Directed fuzzing

Fuzz-testing is expensive

- Large size of input
- Hundreds or thousands of bytes to mutate
- Very likely to generate rejected input

Directed fuzzing

Directed fuzzing allows to find **hot bytes** in the input, which are:

- Are more likely to trigger bugs or crashes
- Are less likely to be the cause of rejected input

What is a **hot byte**?

- An input byte that will be used in a security-sensitive call (such as `malloc` or `strcpy`)

Directed fuzzing

How to find hot bytes?

- Start from a valid input
- Give all byte in the input a unique label
- Use a taint-tracer to see where the input bytes are used

If an input byte is used (directly or indirectly) in a sensitive function call, this byte is a hot byte.

Directed fuzzing

Taintscope finds those hot bytes and focuses on them for fuzz-testing.

The hot-byte detection can be done simultaneously with the checksum pre-processing step, leading to less overhead.

Evaluation and results

Taintscope was evaluated on real-world applications such as:

- Image viewer
  - Google Picasa
  - Adobe Acrobat
  - Image magick
- Media Players
  - MPlayer
  - Winamp
- Web Browsers
- libtiff
- XEmacs

Evaluation and results

First test on hot bytes identification.

Application	Input size	# Hot bytes	Run time
ImageMagick (png)	5149	9	1m54s
ImageMagick (jpg)	6617	11	1m13s
Picasa (png)	2730	18	5m16s
Acrobat (png)	770	21	3m8s
Acrobat (jpg	1012	13	4m14s

Evaluation and results

Second test on Checksum localization

Application	# points (1^{st})	# points (2^{nd})	Detected
Picasa (png)	830	1	Yes
Acrobat (png)	5805	1	Yes
TCPDump (pcap)	5	2	Yes
Tar	9	1	Yes

In practice : Around twenty runs to find the checksum location.
Done in tens of minutes.

Evaluation and results

Third test on checksum reconstruction:

Format	# checksum	size	time
PNG	4	4	271.9
PCAP	8	2	455.6
TAR	3	8	572.8

Evaluation and results

Out of those tests, Taintscope has found 27 severe vulnerabilities in common applications caused by:

- Buffer overflow
- Integer overflow
- Double free
- Null pointer dereference
- Infinite loop

Conclusion

- Only few bytes are **hot** in most input files, making directed fuzzing essential in fuzz testing
- Taintscope is able to detect checksum check points in programs, and checksum fields in input
- Taintscope is able to automatically create valid input passing the checksum check
- Taintscope can be used on real-world application to find serious vulnerabilities

Conclusion

However:

- Taintscope cannot handle signed inputs.
  - It can bypass the check and find vulnerabilities
  - But cannot recreate valid input afterwards
- All benefits of directed fuzzing are lost when data is encrypted, as every input byte will be detected as hot.
- Checksum location depends heavily on the availability of well-formed and malformed inputs

Questions?